Ties between Parametrically Polymorphic Type Systems and Finite Control Automata

Extended Abstract

JOSEPH (YOSSI) GIL and ORI ROTH, The Technion

We present a correspondence and bisimulation between variants of parametrically polymorphic type systems and variants of finite control automata, such as FSA, PDA, tree automata and Turing machine. Within this correspondence we show that two recent celebrated results on automatic generation of fluent API are optimal in certain senses, present new results on the studied type systems, formulate open problems, and present potential software engineering applications, other than fluent API generation, which may benefit from judicious use of type theory.

CCS Concepts: • Software and its engineering → General programming languages; API languages; Polymorphism.

Additional Key Words and Phrases: type systems, automata, computational complexity, fluent API

1 INTRODUCTION

Computational complexity of type checking is a key aspect of any type system. Several classical results characterize this complexity in type systems where the main type constructor is function application: Type checking in the Simply Typed Lambda Calculus (STLC), in which function application is the sole type constructor, is carried out in linear time. In the Hindley-Milner (HM) type system [Damas and Milner 1982; Hindley 1969; Milner 1978], obtained by augmenting the STLC with parametric polymorphism with unconstrained type parameters, type checking is harder, and was found to be deterministic exponential (DEXP) time complete [Kfoury et al. 1990]. However, the Girard–Reynolds type system [Girard 1971, 1972; Reynolds 1974] (System-F) which generalizes HM is undecidable [Wells 1999].

In contrast, our work focuses in type systems where the main type constructor is pair (or tuple), i.e., no higher order functions. This type constructor models object based programming, including concepts such as records, classes and methods, but not inheritance. In particular, we investigate the computational complexity of such systems in the presence of parametric polymorphism, also called genericity, allowing generic classes and generic functions.

We acknowledge significant past work on more general systems modeling the combination of genericity with the object oriented programming paradigm, i.e., classes with single and even multiple inheritance. Type checking in these is particularly challenging, since inheritance may be used to place sub-type and super-type constraints on the parameters to generics. In fact, Kennedy and Pierce [2007] showed that, in general, such type systems are undecidable. Their work carefully analyzed the factors that may lead to undecidability, and identified three decidable fragments, but without analyzing their complexity. In fact, the presumed decidability of C#’s type system is a result of adopting one particular such fragment. Later, Grigore [2017] proved that the combination of inheritance and genericity in the Java type system makes its type system undecidable (after Amin and Tate [2016] showed that is unsound).

Recent work on the practical fluent API problem, drew our attention to these kind of systems. However, this work is mostly of theoretical nature. We present a correspondence and bisimulation between variants of these type systems, organized in a conceptual lattice \( \mathcal{G} \) of type systems, and variants of finite control automata, such as FSA, PDA, tree automata and Turing machine, organized in another conceptual lattice \( \mathcal{A} \).
With this correspondence we determine the exact computational complexity class of type checking of many, but not all, type systems in \( \mathcal{T} \); for other type systems, we provide upper and lower bounds, leaving the precise characterizations as open problems. We also show that two celebrated results on the fluent API problem, are optimal in certain senses. The research also has practical applications for language design, e.g., Thm. 5.1 below shows that introducing functions whose return type is declared by keyword `auto` to C#, would make its type system undecidable.

1.1 Background

Recall\(^1\) that a fluent API generator transforms \( t \), the formal language that specifies the API, into \( L = L(t) \), a library of type definitions in some target programming language, e.g., Java, Haskell, or C++. Library \( L(t) \) is the fluent API library of \( t \) if an expression \( e \) type checks (in the target language) against \( L(t) \) if and only if word \( w = w(e) \) is in the API language \( t \), \( w(e) \in t \Leftrightarrow e \) type checks against \( L(t) \). It is required that expression \( e \) is in the form of a chain of method invocations. The word \( w = w(e) \) is obtained by enumerating, in order, the names of methods in the chain of \( e \), e.g., a fluent API generator for C++ receives a language \( t \) over alphabet \( \Sigma = \{a, b\} \), i.e., \( t \subseteq \{a, b\}^* \) and generates as output a set of C++ definitions in which an expression such as

\[
\text{new Begin().a().b().a().a().end()} \tag{1.1}
\]

type checks if, and only if word \( abaa \) belongs in language \( t \).

The most recent such generator is TypelevelLR due to Yamazaki, Nakamaru, Ichikawa and Chiba [2019]. TypelevelLR compiles an LR language \(^2\) \( t \) into a fluent API library \( L(t) \) in either Scala, C++, or, Haskell (augmented with "four GHC extensions: MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, and UndecidableInstances"), but neither Java nor C#.

The architecture of TypelevelLR makes it possible to switch between different front-ends to translate a context free grammar specification of \( t \) into an intermediate representation. Different such front-ends are SLR, LALR, LR(1) grammar processors. Similarly to traditional multi-language compilers, the front-ends compile the input specification into a library in Fluent, an intermediate language invented for this purpose; the back-ends of TypelevelLR translates \( L_{\text{Fluent}} \) into an equivalent library \( L = L(L_{\text{Fluent}}) \) in the target languages.

TypelevelLR strikes a sweet spot in terms of front-ends: It is a common belief that most programming languages are LR, so there is no reason for a fluent API generator to support any wider class of formal languages for the purpose of the mini programming language of an API. On the other hand, TypelevelLR’s client may tune down the generality of TypelevelLR, by selecting the most efficient front-end for the grammar of the particular language of the fluent API.

We show that in terms of computational complexity, TypelevelLR strikes another sweet spot in selecting Fluent, specifically, that Fluent = LR (Thm. 5.3). Equality here is understood in terms of classes of computational complexity, i.e., for every set of definitions \( L \) in Fluent, there exists an equivalent formal language \( t = L(L) \in LR \), and for every \( t \in LR \) there exists equivalent library \( L = L(t) \). Also term Fluent in the equality refers to the computational complexity class defined by the Fluent language. This abuse of notation is freely used henceforth.

Is there a similar sweet spot in the back-ends of TypelevelLR? Why didn’t TypelevelLR include neither Fluent-into-Java nor Fluent into-C# back-ends? And, why were these GHC extensions selected and not others? These and similar questions made the practical motivation for this work.

1.2 A Taxonomy of Parametric Polymorphism

It follows from Thm. 5.3 that Fluent can be compiled into a type system \( T \) only if \( T \) is (computationally wise) sufficiently expressive, i.e., \( L \subseteq T \). But which are the features of \( T \) that make it this

\(^1\)Sect. B gives more precise definitions and motivation for the fluent API problem

\(^2\)Left-to-right, Rightmost derivation [Knuth 1965]
expressive? Motivated by questions such as theses, we offer in Sect. 3 a taxonomy, reminiscent of \(\lambda\)-cube [Barendregt 1991], for the classification of parametrically polymorphic type systems. The difference is that \(\lambda\)-cube is concerned with parametric polymorphism where the main type constructor is function application; our taxonomy classifies type system built around the pairing type constructor, as found in traditional imperative and object oriented languages.

The taxonomy is a partially ordered set, specifically a lattice, \(\mathfrak{T}\) of points spanned by six, mostly orthogonal characteristics. (See Table 3.1 below.) A point \(T \in \mathfrak{T}\) is a combination of features (values of a characteristic) that specify a type system, e.g., Fluent is defined by combination of three non-default features, monadic-polymorphism, deep-argument-type, and rudimentary-typeof of three characteristics; features of the three other characteristics take their default (lowest) value, linear-patterns, unary-functions, and, one-type.

We say that \(T_1\) is less potent than \(T_2\), if \(T_1\) is strictly smaller in the lattice order than \(T_2\), i.e., any program (including type definitions and optionally an expression to type check) of \(T_1\) is also a program of \(T_2\). In writing \(T_1 = T_2\) (\(T_1 \subseteq T_2\)) we mean that the computational complexity class of \(T_1\) is the same as (strictly contained in) that of \(T_2\).

The \(P_p\) Type System. We employ \(\mathfrak{T}\) to analyze Fling, yet another API generator [Gil and Roth 2019] (henceforth G&R), capable of producing output for Java and C#. Although Fling does not explicitly define an intermediate language equivalent to Fluent, type definitions produced by Fling belong to a very distinct fragment of type systems of Java and C#, which G&R call “unbounded unspecialized parametric polymorphism”, and we call \(P_p\) here.

In plain words, \(P_p\) refers to a type system in which genericity occurs only in no-parameters methods occurring in generic classes (or interfaces) that take one or more unconstrained type arguments, as in, e.g., List. 1.1. In terms of lattice \(\mathfrak{T}\), type system \(P_p\) is defined by feature polyadic-parametric-polymorphism of the "number of type arguments" characteristics (and default, least-potent feature value of all other characteristics).

Listing 1.1 An example non-sense program in type system \(P_p\)

```java
class Program {
    // Type definitions
    interface γ1 {} interface γ2 {}
    interface γ3<x1,x2> { x1 a(); x2 b(); γ4<γ2,γ3<x1,x2>> c(); }
    interface γ4<x1,x2> { γ4<x2,x1> a(); γ4<γ3<x1,x2>,γ3<x2,x1>> b(); γ3<x1,x2> c(); }

    // Initializer with expression(s) to check.
    (γ3<γ1,γ2>) null).c().a().b().b().a(); // Type checks
    (γ3<γ1,γ2>) null).c().a().b().b().a(); // Type check error
}
```

We prove that \(P_p = DCFL\) (Thm. 4.1), i.e., computational complexity of \(P_p\) is the same as Fluent. Further, we will see that type systems less potent than \(P_p\) reduce its computational complexity. Other results (e.g., Thm. 4.2 and Thm. 4.3) show that making it more potent would have increased its computational complexity.

Combining Theory and Practice: Fling+TypelevelLR architecture. As Yamazaki et al. noticed, translating of Fluent into mainstream programming language is not immediate. Curiously, the type systems of all target languages of TypelevelLR are undecidable. However, it follows from Thm. 5.3 that the target language, from the theoretical complexity perspective, is only required to be at least as expressive as DCFL, as is the case in language such as Java, ML, (vanilla) Haskell, and C#.

---

3read “plain parametric polymorphism”, or “polyadic parametric polymorphism” here
4Although not intended to be executable, Java (and C++) code in this examples can be copied and pasted as is (including Unicode characters such as \(γ\)) into-, and then compiled on- contemporary IDEs. Exceptions are expressions included for demonstrating type checking failure.
To bring theory into practice, notice that all these languages contain the \( \mathbb{P}_T \) type system. We envision a software artifact, whose architecture combines TypelevelLR and Fling, making it possible to compile the variety of LR grammars processors into any programming language which supports code such as List. 1.1. Front ends of this “ultimate fluent API generator” are the same as TypelevelLR. However, instead of directly translating Fluent into a (rather straightforward) implementation, e.g., in Java, of the algorithm behind the proof of Thm. 5.3, plugging it as a back end of TypelevelLR. Concretely, the artifact compiles Fluent into a specification of a DPDA (deterministic pushdown automaton) as in the said proof. We then invoke (a component of) Fling to translate the DPDA specification into a set of \( \mathbb{P}_T \) type definitions. The back-ends of Fling are then employed to translate these type definitions to the desired target programming language.

Outline. Sect. 2 presents the lattice \( \mathfrak{A} \) of finite control automata on strings and trees, ranging from FSAs to Turing machines, and reminds the reader of the computational complexity classes of automata in this lattice, e.g., in terms of families of formal languages in the Chomsky hierarchy. The lattice of parametrically polymorphic type systems \( \mathfrak{T} \) is then presented in Sect. 3. The presentation makes clear bisimulation between the runs of certain automata and type checking in certain type systems, whereby obtaining complexity results of these type systems.

Sect. 4, concentrating on parallels between real-time automata and type systems, derives further complexity results. In particular, this section shows that Fling is optimal in the sense that no wider class of formal languages is used by \( \mathbb{P}_T \), the type system it uses. Non real-time automata, and their relations to type systems which admit the typeof keywords are the subject of Sect. 5. In particular, this section sets the computational complexity of Fluent and several variants. Sect. 6 then turns to discussing the ties between non-deterministic automata and type systems that allow an expression to have multiple types.

Sect. 7 concludes in a discussion, open problems and directions for future research.

While reading this paper, readers should notice extensive overloading of notation, made in attempt to highlight the ties between automata and type systems. The list of symbols in Sect. A should help in obtaining a full grasp of this overloading. Appendices also include some of the more technical proofs and other supplementary material.

2 FINITE CONTROL AUTOMATA

This section presents a unifying framework of finite control automata and formal languages, intended to establish common terminology and foundation for the description in the forthcoming Sect. 3 of parametrically polymorphic type systems and their correspondence to automata.

Definitions here are largely self contained, but the discussion is brief; it is tacitly assumed that the reader is familiar with fundamental concepts of automata and formal languages, which we only re-present here.

2.1 Characteristics of Finite Control Automata

We think of automata of finite control as organized in a conceptual lattice \( \mathfrak{A} \). The lattice (strictly speaking, a Boolean algebra) is spanned by seven (mostly) orthogonal characteristics, such as the kind of input that an automaton expects, the kind of auxiliary storage it may use, etc. Overall, lattice \( \mathfrak{A} \) includes automata ranging from finite state automata to Turing machines, going through most automata studied in the classics of automata and formal languages (see, e.g., Hopcroft, Motwani and Ullman [2007]).

Concretely, Table 2.1 defines lattice \( \mathfrak{A} \), by row-wise enumeration of the said characteristics and the values that each may take. We call these values properties of the lattice.
Values of a certain characteristics are mutually exclusive: For example, the first row in the table states that the first characteristic, number of states, can be either stateless (the finite control of the automata does not include any internal states) or stateful (finite control may depend on and update an internal state). An automaton cannot be both stateful and stateless.

An automaton in \( \mathfrak{A} \) is specified by selecting a value for each of the characteristics.

The table enumerates properties in each characteristic in increasing potency order. For example, in "number of states" characteristic, stateful automata are more potent than stateless automata, in the sense that any computation carried out by \( A \), a certain stateless automaton in \( \mathfrak{A} \), can be carried out by automaton \( A' \in \mathfrak{A} \), where the only difference between \( A \) and \( A' \) is that \( A' \) is stateful.

Each automaton in the lattice might be fully specified as a set of seven properties (\( p_1, \ldots, p_7 \)). In the abbreviated notation we use, a property of a certain characteristic is mentioned only if it is not the weakest (least potent) in this characteristic. For example, the notation "\( \langle \rangle \)" is short for the automaton with least-potent property of all characteristics,

\[
\mathfrak{A}_\bot = \langle \rangle = \langle \text{stateless, no-store, language, real-time, determ, shallow, linear} \rangle,
\]

the bottom of lattice \( \mathfrak{A} \). Table 2.2 offers additional shorthand using acronyms of familiar kinds of automata and their mapping to lattice points. For example, the second table row maps FSAs to lattice point \( \langle \text{stateful, non-deter} \rangle \).

### Table 2.1. Seven characteristics and 18 properties spanning lattice \( \mathfrak{A} \) of finite control automata

| Characteristic                      | Values in increasing potency |
|-------------------------------------|------------------------------|
| No. states                          | 1 stateless                  |
| (Def. 2.2)                          | 2 stateful                   |
| FSA                                 | 1 no-store                   |
| (Sect. 2.3)                         | 2 pushdown-store             |
| Recognizer kind                     | 3 tree-store                 |
| (Def. 2.1, Def. 2.9)                | 4 linearly-bounded-tape-store|
| Determinism                         | 5 unbounded-tape-store       |
| (Def. 2.6)                          | 1 deterministic              |
| ε-transitions                       | 2 ε-transitions              |
| (Def. 2.5)                          | 1 real-time                  |
| Determinism                        | 2 ε-transitions              |
| (Def. 2.6)                          | 1 real-time                  |
| Determinism                        | 2 ε-transitions              |
| Rewrite multiplicity                | 1 linear-rewrite             |
| (Def. 2.7)                          | 2 non-linear-rewrite         |
| Rewrite depth                       | 1 shallow-rewrite            |
| (Def. 2.8)                          | 2 deep-rewrite               |

### Table 2.2. Selected automata in the lattice \( \mathfrak{A} \) and their computational complexity classes

| Acronym                          | Common name | Lattice point | Complexity |
|----------------------------------|-------------|---------------|------------|
| Deterministic Finite State Automaton | DFSA        | \langle \text{stateful} \rangle | REG        |
| Finite State Automaton           | FSA         | non-deterministic-DFSA = \langle \text{stateful, non-deterministic} \rangle | REG\(^2\) |
| Stateless Real-time Deterministic PushDown Automaton | SRDPDA | \langle \text{pushdown, stateless, } \varepsilon \text{-transitions, non-deterministic, shallow, linear} \rangle | \subseteq \text{DCFL} |
| Real-time Deterministic PushDown Automaton | RDPDA | stateful-SRDPDA = \langle \text{pushdown, stateful, } \varepsilon \text{-transitions, non-deterministic, shallow, linear} \rangle | \subseteq \text{DCFL}\(^3\) |
| Deterministic PushDown Automaton | DPDA        | \varepsilon \text{-transitions-RDPDA = } \langle \text{pushdown, stateful, } \varepsilon \text{-transitions, deterministic, shallow, linear} \rangle | \text{DCFL}\(^4\) |
| Tree Automaton                   | TA          | \langle \text{tree-store, stateless, real-time, shallow, linear} \rangle | \text{DCFL} |
| PushDown Automaton               | PDA         | non-deterministic-DPDA = \langle \text{pushdown, stateful, } \varepsilon \text{-transitions, non-deterministic, shallow, linear} \rangle | \text{CFL}\(^5\) |
| Real-time Turing Machine         | RTM         | \langle \text{linearly-bounded-tape, stateful, real-time, deterministic, shallow, linear} \rangle | \subseteq \text{CSL} |
| Linear Bounded Automaton         | LBA         | \langle \text{linearly-bounded-tape, shallow, linear} \rangle \vee \text{FSA} = \langle \text{linearly-bounded-tape, deterministic, shallow, linear} \rangle | \text{CSL} |
| Turing Machine                   | TM          | \langle \text{unbounded-tape-LBA = } \langle \text{unbounded-tape, stateful, deterministic, shallow, linear} \rangle \rangle | \text{RE}\(^6\) |

\(^1\text{REG} \subseteq \text{DCFL} \subseteq \text{CFL} \subseteq \text{CSL} \subseteq \text{PR} \subseteq \text{RE} \subseteq \text{RE}

\(^2\langle \varepsilon \text{-transitions, non-deterministic} \rangle \vee \text{FSA} = \text{FSA}

\(^3\text{Autebert et al. 1997, Example 5.3}\)

\(^4\text{deep-DPDA} = \text{DPDA}

\(^5\text{deep-DPDA} = \text{PDA}

\(^6\text{non-deterministic-TM} = \text{TM} \)

Observe that just as the term pushdown automaton refers to an automaton that employs a pushdown auxiliary storage, we use the term tree automaton for an automaton that employs a tree
auxiliary storage. Some authors use the term for automata that receive hierarchical tree rather than string as input. In our vocabulary, the distinction is found in the *language-recognizer vs. forest-recognizer* properties of the “recognizer kind” characteristic.

The final column of Table 2.2 also specifies the computational complexity class of the automaton defined in the respective row. In certain cases, this class is a set of formal languages found in the Chomsky hierarchy. From the first two rows of the table we learn that even though DFSAs are less potent than FSAs, they are able to recognize exactly the same set of formal languages, namely the set of regular languages denoted REG.

2.2 Finite Control Automata for Language Recognition

As usual, let \( \Sigma \) be a finite alphabet, and let \( \Sigma^* \) denote the set of all strings (also called words) over \( \Sigma \), including \( \varepsilon \), the empty string. A (formal) language \( \ell \) is a (typically infinite) set of such strings, i.e., \( \ell \subseteq \Sigma^* \).

**Definition 2.1.** A recognizer of language \( \ell \subseteq \Sigma^* \) is a device that takes as input a word \( w \in \Sigma^* \) and determines whether \( w \in \ell \).

Let \( A \) be a finite control automata for language recognition. (Automata for recognizing forests are discussed below in Sect. 2.4.) Then, \( A \) is specified by four finitely described components: states, storage, consuming transition function, and \( \varepsilon \)-transition function:

1. **States.** The specification of these includes (i) a finite set \( Q \) of internal states (or states), (ii) a designated initial state \( q_0 \in Q \), and, (iii) a set \( F \subseteq Q \) of accepting states.

   **Definition 2.2.** \( A \) is stateful if \( |Q| > 1 \); it is stateless if \( |Q| = 1 \), in which case \( F = Q = \{q_0\} \).

2. **Storage.** Unlike internal state, the amount of data in auxiliary storage is input dependent, hence unbounded. The pieces of information that can be stored is specified as a finite alphabet \( \Gamma \) of storage symbols, which is not necessarily disjoint from \( \Sigma \). The organization of these symbols depends on the auxiliary-storage characteristic of \( A \): In pushdown-store automata, \( \Gamma \) is known as the set of stack symbols, and the storage layout is sequential. In tree-store automata, the organization is hierarchical and \( \Gamma \) is a ranked-alphabet. In tape automata, \( \Gamma \) is called the set of tape symbols, and they are laid out sequentially in a uni-directional tape.

Let \( \Gamma \) denote the set of possible contents of the auxiliary storage. In pushdown automata \( \Gamma = \Gamma^\infty \); in tape automata, the storage contents includes the position of the head: Specifically, in unbounded-tape-store (employed by Turing machines), \( \Gamma = \mathbb{N} \times \Gamma^* \). We set \( \Gamma = \mathbb{N} \times \Gamma^* \) also for the case of linearly bounded automata. For tree-store automata, \( \Gamma = \Gamma^\Delta \), where \( \Gamma^\Delta \) is defined below as the set of trees whose internal nodes are drawn from \( \Gamma \).

**Definition 2.3.** An Instantaneous Description (ID, often denoted \( i \)) of \( A \) running on input word \( w \in \Sigma^* \) includes three components: (i) a string \( u \in \Sigma^* \), where \( u \) is a suffix of \( w \), specifying the remainder of input to read; (ii) the current state \( q \in Q \), and, (iii) \( y \in \Gamma \), the current contents of the auxiliary storage.

The auxiliary storage is initialized by a designated value \( y_0 \in \Gamma \). Any run of \( A \) on input \( w \in \Sigma^* \) begins with ID \( i_0 = \langle w, q_0, y_0 \rangle \), and then proceeds as dictated by the transitions functions.

**Definition 2.4.** \( A \) is no-store if \( |\Gamma| = 0 \), in which case \( \Gamma \) is degenerate, \( \Gamma = \{y_0\} \).

3. **Consuming transition function.** Denoted by \( \delta \), this partial, possibly multi-valued function, defines how \( A \) proceeds from a certain ID to a subsequent ID in response to a consumption of a single input letter.
• Function $\delta$ depends on (i) $\sigma \in \Sigma$, the current input symbol, being the first letter of $u$, i.e., $u = \sigma u'$, $u' \in \Sigma^*$ (ii) $q \in Q$ the current state, and, (iii) $\gamma \in \Gamma$, the current contents of the auxiliary storage.

• Given these, $\delta$ returns a new internal state $q' \in Q$ and the new storage contents $\gamma'$ for the subsequent ID. The "remaining input" component of the subsequent ID is set to $u'$.

(4) $\epsilon$-transition function. A partial, multi-valued function $\xi$ specifies how $A$ moves from a certain ID to a subsequent ID, without consuming any input. Function $\xi$ depends on the current state $q \in Q$ and $\gamma$, the storage’s contents, but not on the current input symbol. Just like $\delta$, function $\xi$ returns a new internal state $q' \in Q$ and storage contents $\gamma'$ for the subsequent ID. However, the remaining input component of IDs is unchanged by $\xi$.

Automaton $A$ accepts $w$ if there exists a run $t_0, t_1, \ldots, t_m$, that begins with the initial ID $t_0 = \langle w, q_0, y_0 \rangle$ and ends with an ID $t_m = \langle \epsilon, q, \alpha \rangle$ in which all the input was consumed, the internal state $q$ is accepting, i.e., $q \in F$, and no further $\epsilon$-transitions are possible, i.e., $\xi(t_m)$ is not defined.

On each input letter, automaton $A$ carries one transition defined by $\delta$, followed by any number of $\epsilon$-transitions defined by $\xi$, including none at all. A real-time automaton is one which carries precisely one transition for each input symbol.

**Definition 2.5.** $A$ is real-time if there is no id $i$ for which $\xi(i)$ is defined.

Real-time and non-real-time automata are, by the above definitions, non-deterministic. Since both $\delta$ and $\xi$ are multi-valued, an ID does not uniquely determine the subsequent ID.

**Definition 2.6.** $A$ is deterministic if (i) partial functions $\xi$ and $\delta$ are single valued, and, (ii) there is no ID $i$ for which both $\xi(i)$ and $\delta(i)$ are defined.

Both deterministic and non-deterministic automata may hang, i.e., they might reach an ID $i$ for which neither $\xi(i)$ nor $\delta(i)$ are defined. If all runs of a non-deterministic automaton $A$ on a given input $w$ either hang or reach a non-accepting state, $A$ rejects $w$. Alternatively, if the only run of a deterministic automaton $A$ on $w$ hangs, automaton $A$ rejects $w$. Hanging is the only way a stateless automaton can reject. A stateful automaton rejects also in the case it reaches a non-accepting state $q \in Q \setminus F$ after consuming all input.

### 2.3 Rewrites of Auxiliary Storage

Since functions $\delta$ and $\xi$ are finitely described, they are specified as two finite sets, $\Delta$ and $\Xi$ of input-to-output items, e.g., the requirement in Def. 2.5 can be written as $|\Xi| = 0$. Since the transformation of auxiliary storage $\gamma$ to $\gamma'$ by these functions must be finitely described, only a bounded portion of $\gamma$ can be examined by $A$. The transformation $\gamma$ to $\gamma'$, what we call rewrite of auxiliary storage, must be finitely described in terms of this portion.

**Tape initialization, rewrite, and head overflow.** The literature often defines tape automata with no consuming transitions, by making the assumption that they receive their input on the tape store which allows bidirectional movements. Our lattice $\mathcal{U}$ specifies that the input word $w$ is consumed one letter at a time. No generality is lost, since with the following definitions tape automaton $A \in \mathcal{U}$ may begin its run by consuming the input while copying it to the tape, and only then process it with as many $\epsilon$-transitions are necessary.

The contents $\gamma$ of tape auxiliary storage is a pair $(h, y_0 y_1 \cdots y_{m-1})$, where integer $h \geq 0$ is the head’s position and $y_0 y_1 \cdots y_{m-1} \in \Gamma^*$ is the tape’s content. Let $y_0 = \langle \epsilon, 0 \rangle$, i.e., the tape is initially empty and the head is at location 0. Rewrites of tape are the standard read and replace of symbol-under-head, along with the move-left and move-right instructions to the head: Tape rewrite $\gamma \rightarrow y'$ (respectively, tape rewrite $\gamma \rightarrow y'_{\uparrow}$ means that if $y_h = \gamma$ then replace it with, not necessarily distinct, symbol $y' \in \Gamma$ and increment (respectively, decrement) $h$. A third kind of rewrite is $\bot \rightarrow \gamma$, which means that if the current cell is undefined, i.e., $h \notin \{0, \ldots, m - 1\}$, replace it with $\gamma \in \Gamma$. 

$\bot$
The automaton hangs if $h$ becomes negative, or if $h$ exceeds $n$, the input’s length, in the case of a linear bounded automaton.

Rewrites of a pushdown. Rewrites of a pushdown auxiliary storage are the usual push and pop operations; we will see that these can be regarded as tree rewrites.

Trees. A finite alphabet $\Gamma$ is a signature if each $\gamma \in \Gamma$ is associated with an integer $r = r(\gamma) \geq 1$ (also called arity). A tree over $\Gamma$ is either a leaf, denoted by the symbol $e$, or a (finite) structure in the form $\gamma(t)$, where $\gamma \in \Gamma$ of arity $r$ is the root and $t = t_1, \ldots, t_r$ is a multi-tree, i.e., a sequence of $r$ (inductively constructed) trees over $\Gamma$. Let $\Gamma^\Delta$ denote the set of trees over $\Gamma$.

Let $\text{Depth}(t)$ be the depth of tree $t$ (for leaves let, $\text{Depth}(e) = 0$). We shall use freely a monadic tree abbreviation by which tree $\gamma_2(\gamma_1(e), \gamma_1(\gamma_1(e)))$ is written as $\gamma_2(\gamma_1, \gamma_1(\gamma_1))$, and tree $\gamma_1(\gamma_2(\ldots(\gamma_n(e)))$ is written as $\gamma_1(\gamma_2 \cdots \gamma_n)$. If the rank of all $\gamma \in \Gamma$ is 1, then $\Gamma^\Delta$ is essentially the set $\Gamma^*$, and every tree $t \in \Gamma^\Delta$ can be viewed as a stack whose top is the root of $t$ and depth is $\text{Depth}(t)$.

In this perspective, a pushdown automaton is a tree automaton in which the auxiliary tree is monadic. We set $\gamma_0$ in tree automata to the leaf $e \in \Gamma^\Delta$, i.e., the special case pushdown automaton starts with an empty stack.

Terms. Let $X = \{x_1, x_2, \ldots\}$ be an unbounded set of variables disjoint to all alphabets. Then, a pattern (also called term) over $\Gamma$ is either some variable $x \in X$ or a structure in the form $\gamma(t_1, \ldots, t_r)$, where the arity of $\gamma \in \Gamma$ is $r$ and each of $t_1, \ldots, t_r$ is, recursively, a term over $\Gamma$. Let $\Gamma^\Delta$ denote the set of terms over $\Gamma$. Thus, $\Gamma^\Delta \subseteq \Gamma^\Delta$, i.e., all trees are terms. Trees are also called grounded terms; ungrounded terms are members of $\Gamma^\Delta \setminus \Gamma^\Delta$. A term is linear if no $x \in X$ occurs in it more than once, e.g., $\gamma(x, x)$ is not linear while $\gamma(x_1, \gamma(x_2, x_3))$ is linear.

Terms match trees. Atomic term $x$ matches all trees in $\Gamma^\Delta$; a compound linear term $\tau = \gamma(t_1, \ldots, t_r)$ matches tree $t = \gamma(t_1, \ldots, t_r)$ if for all $i = 1, \ldots, r$, $t_i$ recursively matches $t_i$, e.g., $\gamma(x_1, \gamma(x_2, x_3))$ matches $\gamma(e, \gamma(e, \gamma(e, e)))$. To define matching of non-linear terms define tree substitution $s$ (substitution for short) as a mapping of variables to terms, $s = \{x_1 \rightarrow t_1, \ldots, x_r \rightarrow t_r\}$. Substitution $s$ is grounded if all terms $t_1, \ldots, t_r$ are grounded. An application of substitution $s$ to term $\tau$, denoted $\tau/s$, replaces each variable $x_i$ with term $t_i$ if and only if $x_i \rightarrow t_i \in s$. The notation $\tau' \sqsubseteq \tau$ is to say that term $\tau$ matches a term $\tau'$, which happens if there exists a substitution $s$ such that $\tau' = \tau/s$.

Tree rewrites. A tree rewrite rule $\rho$ (rewrite for short) is a pair of two terms written as $\rho = t_1 \rightarrow t_2$. Rewrite $\rho$ is applicable to (typically grounded) term $t_1'$ if $t_1' = t_1/s$, for some substitution $s$. If rewrite $\rho$ matches term $t_1'$ then $t_1'/\rho$, the application of $\rho$ to $t_1'$ (also written $t_1'/t_1 \rightarrow t_2$) yields the term $t_2' = t_2/s$.

The definition of rewrites does not exclude a rewrite $\gamma_1(x_1) \rightarrow \gamma_2(x_1, \gamma_1(x_2))$, whose right-hand-side term introduces variables that do not occur in the left-hand-side term. Applying such a rewrite to a tree will always convert it to a term. Since the primary intention of rewrites is the manipulation of trees, we tacitly assume here and henceforth that it is never the case; a rewrite $t_1 \rightarrow t_2$ is valid only if $\text{Vars}(t_2) \subseteq \text{Vars}(t_1)$.

Manipulation of tree and pushdown auxiliary storage is defined with rewrites. For example, the rewrite $\gamma_1(\gamma_2(x)) \rightarrow \gamma_3(x)$, or in abbreviated form $\gamma_1(\gamma_2x) \rightarrow \gamma_3x$, is, in terms of stack operations: if the top of the stack is symbol $\gamma_1$ followed by symbol $\gamma_2$, then pop these two symbols and then push symbol $\gamma_3$ onto the stack.

With these definitions:

• Each member of set $\Delta$ is in the form $\langle \sigma, q, \rho, q' \rangle$ meaning: if the current input symbol is $\sigma$, the current state is $q$ and auxiliary storage $t$ matches $\rho$, then, consume $\sigma$, move to state $q'$ and set the storage to $t/\rho$.

• Each member of set $\Xi$ is in the form $\langle q, \rho, q' \rangle$ meaning: if the current state is $q$ and auxiliary storage $t$ matches $\rho$, then, move to state $q'$ and set the storage to $t/\rho$.
A tree rewrite \( \rho = \tau_1 \rightarrow \tau_2 \) is linear if \( \tau_1 \) is linear, e.g., rewrites \( \gamma(x) \rightarrow \gamma'(x, x, x) \) and \( \gamma(x_1, x_2) \rightarrow \gamma'(x_2, x_1, \varepsilon) \) are linear, but \( \gamma(x, x) \rightarrow \varepsilon \) is not. Notice that rewrites of tape and pushdown auxiliary storage are linear: the transition functions of these do never depend on the equality of two tape or pushdown symbols.

**Definition 2.7.** A is linear-rewrite if all rewrites in \( \Xi \) and \( \Delta \) are linear.

Let \( \text{Depth}(\rho) \), \( \rho = \tau_1 \rightarrow \tau_2 \), be \( \text{Depth}(\tau_1) \), and where the depth of terms is defined like tree depth, a variable \( x \in X \) considered a leaf. A term (rewrite) is shallow if its depth is at most one, e.g., \( x, \gamma(x) \), and \( \gamma(x, x) \) are shallow, while \( \gamma(\gamma(x)) \) is not. Rewrite of tape storage are shallow by definition, since only the symbol under the head is inspected.

**Definition 2.8.** A is shallow-rewrite if all rewrites in \( \Xi \) and \( \Delta \) are shallow.

### 2.4 Finite Control Automata for Forest Recognition

In the case that the set of input symbols \( \Sigma \) is a signature rather than a plain alphabet, the input to a finite control automata is then a tree \( t \in \Sigma^\Delta \) rather than a plain word. We use the term forest for what some call tree language, i.e., a (typically infinite) set of trees. Generalizing Def. 2.1 we define:

**Definition 2.9.** A recognizer of forest \( \mathcal{F} \subseteq \Sigma^\Delta \) is a device that takes as input a tree \( t \in \Sigma^\Delta \) and determines whether \( t \in \mathcal{F} \).

As explained in Sect. 2.2 a language-recognizer automaton scans the input left-to-right. However, this order is not mandatory, and there is no essential difference between left-to-right and right-to-left automata. This symmetry does not necessarily apply to a forest-recognizer automaton—there is much research work on comparing and differentiating bottom-up and top-down traversal strategies of finite control automata (e.g., Coquidé et al. [1994] focus on bottom-up automata, Guessarian [1983] on top-down, while Comon et al. [2007] presents several cases in which the two traversal strategies are equivalent.)

Our interest in parametrically polymorphic type systems sets the focus here on the bottom-up traversal strategy only. Most of the description of language-recognizer automata above in Sect. 2.2 remains unchanged. The state and storage specification are the same in the two kinds of recognizers, just as the definitions of deterministic and real-time automata. Even the specification of \( \xi \), the \( \varepsilon \)-transition function is the same, since the automaton does not change its position on the input tree during an \( \varepsilon \)-transition.

However, input consumption in forest recognizers is different than in language recognizers, and can be thought of as visitation. A bottom-up forest-recognizer consumes an input tree node labeled \( \sigma \) of rank \( r \) by visiting it after its \( r \) children were visited. Let \( q_1, q_2, \ldots, q_r \) be the states of the automaton in the visit to these children, and let \( q \) be the multi-state of the \( r \) children, i.e., \( q = q_1, q_2, \ldots, q_r \). Then, the definition of \( \delta \) is modified by letting it depend on multi-state \( q \in Q^k \) rather than on a single state \( q \in Q \). More precisely, each input-to-output item in \( \Delta \) takes the form \( (\sigma, q, \rho, q') \), meaning, if (i) the automaton is in a node labeled \( \sigma \), and (ii) it has reached states \( q_1, q_2, \ldots, q_r \) in the \( r \) children of this node, and if storage rewrite rule \( \rho \) is applicable, then select state \( q' \) for the current node and apply rewrite \( \rho \).

Consider \( \rho \), the rewrite component of an input-output item. As it turns out, only tree auxiliary storage makes sense for bottom up forest recognizers.\(^5\) Let \( t_1, \ldots, t_r \) be the trees representing the contents of auxiliary storage in \( r \) children of the current node. Rewrite rule \( \rho \) should produce a new tree \( t \) of unbounded size from a finite inspection of the \( r \) trees, whose size is also unbounded.

We say that \( \rho \) is a many-input tree rewrite rule (for short, rewrite when the context is clear) if it is in the form \( \rho = \tau_1, \ldots, \tau_r \rightarrow \tau' \). Rule \( \rho = \tau_1, \ldots, \tau_r \rightarrow \tau' \) is applied to all children, with the straightforward generalization of the notions of matching and applicability of a single-input-rewrite:

\(^5\)In top-down forest recognizers pushdown auxiliary storage is also admissible.
A multi-term \( \tau \) is a sequence of terms \( \tau = \tau_1, \ldots, \tau_r \), and a multi-tree \( t \) is a sequence of trees, \( t = t_1, \ldots, t_r \). Then, rule \( \rho = \tau \rightarrow \tau' \) applies to (also, matches) \( t \) if there is a single substitution \( s \) such that \( \tau_i/s = t_i \) for all \( i = 1, \ldots, r \). The application of \( \rho \) to \( t \) is \( \tau/s \).

3 PARAMETRICALLY POLYMORPHIC TYPE SYSTEMS

This section offers a unifying framework for parametrically polymorphic type systems. Definitions reuse notations and symbols introduced in Sect. 2 in the definition of automata, but with different meaning. For example, the Greek letter \( \sigma \) above denoted an input letter, but will be used here to denote the name of a function defined in a certain type system. This, and all other cases of overloading of notation are intentional, with the purpose of highlighting the correspondence between the two unifying frameworks.

3.1 The Lattice of Parametrically Polymorphic Type Systems

Examine Table 3.1 describing \( \mathfrak{I} \), the lattice (Boolean algebra) of parametrically polymorphic type systems. This table is the equivalent of Table 2.1 depicting \( \mathfrak{A} \), the lattice of finitely controlled automata.

Table 3.1 give means for economic specification of different variations of parametrically polymorphic types systems. For example, inspecting Yamazaki et al.’s work we see that the type system of the Fluent intermediate language is

\[
Fluent = \langle \text{monadic-parametric-polymorphism}, \text{deep-type-pattern}, \text{rudimentary} \rangle,
\]

i.e., (i) it allows only one parameter generics, e.g.,

```
interface y1<> {} interface y2<> {} interface y3<> {}
```

(ii) it allows generic functions to be defined for deeply nested generic parameter type, such as

```java
static <> y1<y2<y3<>> f(y2<y2<>> e) {return null;}
```

and, (iii) it allows in the definition of function return type, a `typeof` clause, but restricted to use only one function invocation, e.g.,

```java
static <> typeof(f(e)) g(y3<> e) {return null;}
```

In contrast, the type system used by, e.g., G&R, is simply

\[
P_h = \langle \text{polyadic-parametric-polymorphism} \rangle.
\]

The remainder of this section describes in detail the characteristics in Table 3.1.

3.1.1 Object Based Type System. Type system \( \langle \rangle \), the bottom of \( \mathfrak{I} \), also denoted \( \mathfrak{I}_\perp \) models object based programming paradigm, i.e., a paradigm with objects and classes, but without inheritance nor parametric polymorphism. A good approximation of the paradigm is found in the Go programming language [Donovan and Kernighan 2015]. The essence of \( \mathfrak{I}_\perp \) is demonstrated in this (pseudo Java syntax) example:

```java
interface A { B a(); void b(); } interface B { B b(); A a(); } new A().a().b().b().a().b();
```

\[\text{For concreteness we exemplify abstract syntax with the concrete syntax of Java or C++}.
\]

\[\text{Code rendered in distinctive color as in abuses the host language syntax for the purpose of illustration.}\]
The example shows (i) definitions of two classes, (ii) methods in different classes have the same name, but different return type, (iii) an expression whose type correctness depends on these definitions.

Fig. 3.1 presents the abstract syntax, notational conventions and typing rules of $\mathcal{F}_\perp$. The subsequent description of type systems in $\mathcal{F}$ is by additions and modifications to the figure.

**Fig. 3.1** The bottom of lattice $\mathcal{F}$: the type system $\langle \rangle$ modeling the object-based paradigm

| $P$       | $\Delta$ | $\delta$              | $e$                 | \(\sigma\) |
|-----------|-----------|-----------------------|---------------------|-------------|
| Program   | Set of function definitions | Function name, drawn from alphabet $\Sigma$ |
| $e$       | $\Delta$ | $\delta$             | $\epsilon$         |
| Expression                                                     |

A type in $\mathcal{F}_{\perp}$ is either drawn from $\Gamma$, or is the designated bottom type $\epsilon$. The atomic expression, bootstrapping expression $\epsilon$, is denoted by $\epsilon$, and its type is $\epsilon$.

The figure defines program $P$ in $\langle \rangle$ as a set $\Delta$ of function definitions $\delta$ followed by an expression $e$ to type check. For $\sigma$ drawn from set $\Sigma$ of function names, and types names $\gamma_1, \gamma_2$ drawn from set $\Gamma$ of class names, we can think of a function definition of the form $\sigma : \gamma_1 \rightarrow \gamma_2$ as either

- a method named $\sigma$ in class $\gamma_1$ taking no parameters and returning class $\gamma_2$, or,
- an external function taking a single parameter of type $\gamma_1$, and returning a value of type $\gamma_2$.

With the first perspective, the recursive description of expressions is the Polish convention, $e ::= e.\sigma$, best suited for making APIs fluent. With the latter perspective, this recursive definition should be made in prefix notation, i.e., $e ::= \sigma(e)$. Fig. 3.1 uses both variants, and we will use these interchangeably. Indeed, the distinction between methods and functions is in our perspective only a syntactical matter.

The special case of a function taking the unit type as argument, $\sigma : \epsilon \rightarrow \gamma$, can be thought of as an instantiation of the return type, new $\gamma$. The function name, $\sigma$, is not essential in this case, but is kept for consistency. Also in the figure is the standard function application typing rule. Overloading on the parameter type is intentionally allowed, i.e., methods defined in different classes may use the same name. The one type only rule excludes overloading based on the return type.

3.1.2 Plain Parametric polymorphism. Let $\mathbb{P}_\delta$ be short for lattice point $\langle$polyadic-parametric-polymorphism$\rangle$, as demonstrated in List. 1.1 above. $\mathbb{P}_\delta$ is the type system behind LINQ, the first theoretical treatise of fluent API [Gil and Levy 2016], FlING and other fluent API generators, e.g., of [Xu 2010] and [Nakamaru et al. 2017].

The definition of $\mathbb{P}_\delta$ relies on the definitions of trees, terms and rewrites in Sect. 2.3. Notice that in $\mathcal{F}_{\perp}$, types were drawn from set $\Gamma$. In allowing generic types the type repertoire is extended to $\Gamma^\Delta$, the set of trees over signature $\Gamma$. A type $\gamma \in \Gamma$ of rank $r \geq 1$ is a generic with $r$ type parameters; the only leaf, of rank 0, is the unit type $\epsilon$. $\mathbb{P}_\delta$ also admits “terms”, i.e., trees including formal variables drawn from the set $\Gamma^\Delta$. We refer to terms of $\mathbb{P}_\delta$ as “ungrounded types”; an ungrounded type is also

---

\*ignore the somewhat idiosyncratic distinction between classes and interfaces

\*https://docs.microsoft.com/en-us/dotnet/api/system.linq
viewed in $\mathbb{P}_\mathfrak{f}$ as a *type pattern* that typically match “grounded types” (trees in $\Gamma^\Delta$), but can also be used for matching over ungrounded types.

Fig. 3.2 summarizes the changes in $\mathbb{P}_\mathfrak{f}$’s definitions with respect to those of $\mathfrak{X}_\perp$ in Fig. 3.1.

### Fig. 3.2 The type system $\mathbb{P}_\mathfrak{f}$

| (same as Fig. 3.1 (a) and...) | (same as Fig. 3.1 (b) and...) | (same as Fig. 3.1 (c) and...) |
|-------------------------------|-------------------------------|-------------------------------|
| $\delta ::= \sigma : y(x) \rightarrow \tau$ |
| $\vdash \sigma : \mathfrak{e} \rightarrow t$ |
| $x ::= x_1, \ldots, x_r$ |
| $\tau ::= y(\tau) \mid x \mid t$ |
| $\tau ::= \tau_1, \ldots, \tau_r$ |
| $t ::= y(t) \mid \epsilon$ |
| $t ::= t_1, \ldots, t_r$ |

(a) Abstract syntax

(b) Typing rules

(c) Variables and notations

The main addition of $\mathbb{P}_\mathfrak{f}$ to $\mathfrak{X}_\perp$ is allowing function definition $\delta$ to take also the form $\sigma : y(x) \rightarrow \tau$, where $x = x_1, \ldots, x_r$ here is a sequence of $r$ distinct type variables:

- The single parameter to functions is a multi-variable, yet shallow and linear, type pattern $y(x)$. This requirement models the definition of methods in List. 1.1, i.e., in generic classes with $r$ independent type variables. The structure of this pattern implicitly models the Java/C# decree (which is absent from C++) against specialization of generics for specific values of the parameters.
- Also, as demonstrated by List. 1.1, $\tau$, the return type of a function in this form, is a type pattern of any depth constructed from the variables that occur in $x$ but also from any other types in $\Gamma$.

The figure also shows how the *Function Application* typing rule is generalized by employing the notions of matching and tree substitution from Sect. 2.3.

The definition of a *dyadic-parametric-polymorphism* type system adds to Fig. 3.2 the requirement that $r(y) \leq 2$. In *monadic-parametric-polymorphism*, used for fluent API generation by Nakamaru et al. [2017] and Yamazaki et al. [2019], the requirement becomes $r(y) = 1$ which means abstract syntax rule $t ::= y(t)$ instead of $t ::= y(\tau)$ and $\delta ::= \sigma : y(x) \rightarrow \tau$ instead of $\delta ::= \sigma : y(x) \rightarrow \tau$.

#### 3.1.3 Type Pattern depth

Java, C#, C++ and other languages allow definitions of generic functions which are not methods. For example, static Java function $f$ defined by

```java
static <x1,x2,x3> y1<y2<x3,x2>,x1> f(y1<x1,y2<x2,x3>,e) {return null;}
```

is applicable only if the type of its single argument matches the deep type pattern $y_1(x_1, y_2(x_2, x_3))$. The corresponding lattice property is obtained by adding derivation rule

$$\delta ::= \sigma : \tau \rightarrow \tau' \quad \text{term } \tau \text{ is linear.}$$

(3.3)

along with the requirement that $\tau$ is linear to Fig. 3.2.

As we shall see, the *deep-type-pattern* property increases the expressive power of $\mathbb{P}_\mathfrak{f}$. However, the syntax of invoking generic, non-method functions in contemporary languages breaks the elegance of fluent API: Using functions instead of methods, (1.1) takes the more awkward form

```java
end(a(a(b(a(new Begin()))))))
```

(3.4)

The syntactic overhead of the above “reverse fluent API” can be lessened with a change to the host language; the case for making the change can be made by sorting out the expressive power added by the deep property.
3.1.4 Type Pattern Multiplicity. Recall the abstract syntax rule of \( \delta \) in type system \( \mathcal{P}_1 \) (Fig. 3.2),

\[
\delta ::= \sigma : \gamma(x) \rightarrow \tau \quad \text{term } \gamma(x) \text{ is linear}
\]  

(3.5)

The deep-type-pattern property generalized this abstract syntax rule by allowing functions whose argument type is not restricted to the flat form \( \gamma(x) \). Another orthogonal dimension in which (3.5) can be generalized is by removing the constraint that "term \( \gamma(x) \) is linear", i.e., allowing non-linear type patterns. Such patterns make it possible to define function \( \sigma : \gamma(x, x) \rightarrow x \) that type checks with expression parameter \( e : \gamma(t_1, t_2) \) if and only if \( t_1 = t_2 \). Noticing that \( t_1 \) and \( t_2 \) are trees whose size is unbounded, and may even be exponential in the size of the program, we understand why the term non-linear was coined. Non-linear type patterns may coerce the type-checker into doing non-linear amount of work, e.g., the little Java program in List 3.1 brings the Eclipse IDE and its command line compiler ecj to their knees.

Listing 3.1 Java proram in type system \( S_2 = \langle n\text{-ary,deep,non-linear} \rangle \) requiring over five minutes of compilation time by ecj executing on contemporary hardware

```java
class S2 {
  interface e() {
    interface C<r1, r2> {
      C<r1, r2> f();
    }
  }
}

Listing 3.2
```

Type system non-linear-\( \mathcal{P}_1 \) is defined by replacing (3.5) by its relaxed version,

\[
\delta ::= \sigma : \gamma(x) \rightarrow \tau \quad \text{term } \gamma(x) \text{ may be non-linear.}
\]  

(3.6)

Likewise, type system \( \langle \text{deep,non-linear} \rangle \cup \mathcal{P}_1 \) is obtained by replacing (3.3) by the relaxed version,

\[
\delta ::= \sigma : \tau \rightarrow \tau' \quad \text{term } \tau \text{ may be non-linear.}
\]  

(3.7)

3.1.5 Arity of functions. Yet a third orthogonal dimension of generalizing (3.5) is the number of arguments; so far, \( \sigma \) was thought of as unary function, i.e., either as a nullary method that takes no explicit parameters, or a generic unary, non-method function. The \( n\text{-ary-functions} \) property of polymorphic type systems allows binary, ternary, and in general \( n \)-ary functions, \( n \geq 1 \). The details are in Fig. 3.3.

Fig. 3.3  The type system \( \langle n\text{-ary-functions,deep} \rangle \)

Comparing the figure to Fig. 3.2 above we notice the introducing of notation \( e \) for a sequence of expressions. With this notation, a call to an \( n \)-ary function can be written \( e.\sigma \) (Polish, fluent API like, convention) or as \( \sigma(e) \) (traditional convention). As might be expected, the figure also extends the function application typing rule to non-unity functions.

Note that languages embedded in \( n\text{-ary-}\mathcal{P}_1 \) are no longer languages of words, but rather forests—languages of trees. Indeed, an expression in \( n\text{-ary-}\mathcal{P}_1 \) is a tree of method calls, and the set \( \Delta \) in an \( n\text{-ary-}\mathcal{P}_1 \) program defines the set of tree-like expressions that type-check against it.
3.1.6 Type capturing. A primary motivation for introducing keyword decltype to C++, was streamlining the definition of wrapper functions—functions whose return type is the same as the wrapped function, e.g.,

```cpp
template<typename x> auto wrap(x e) {return wrappee(e);} /*...auto $=wrapee(e);*/
```

As it turns out, keyword decltype dramatically changes the type system, by bringing about the undesired effect that type checking is undecidable. The predicament is due to the idiom of using the type of one function to declare the return type of another. Alternative, seemingly weaker techniques for piecemeal definition of the return type, e.g., allowing typedefs in classes do not alleviate the problem. Likewise, the idiomatic is possible even with the seemingly weaker feature, of allowing functions whose return type is auto, as in

```cpp
template<typename x> auto wrap(x e)(return wrappee(e));
```

Note that neither Java nor C# permit auto functions; it appears that the designers of the languages made a specific effort to block loopholes that permit piecemeal definition of functions return type.

Fig. 3.4 presents abstract modeling of C++’s decltype; for readability we use the more familiar typeof keyword. The figure describes n-ary-functions; for unary-functions let \( n = 1 \).

**Fig. 3.4** Type system \( \langle \text{full-typeof,deep,n-ary-functions} \rangle \)

| (same as Fig. 3.3 (a) and...) | (same as Fig. 3.3 (b) and...) | (same as Fig. 3.3 (c) and...) |
|-------------------------------|-------------------------------|-------------------------------|
| \( \text{P} \) := \( \Delta \Xi \mathcal{e} \) | \( \text{f} = \sigma \) or \( \text{f} = \varphi \) | \( \Xi \) := \( \delta \) |
| \( \Xi \) := \( \xi^* \) | \( \text{f} : \xi_1 \times \cdots \times \xi_r \rightarrow \text{typeof } \varrho \) | \( \xi \) := \( \varphi : \tau \rightarrow \text{typeof } \varrho \) |
| \( \xi := \varphi : \tau \rightarrow \text{typeof } \varrho \) | \( \text{t}_1 = \tau_1/s \cdots \tau_r = \tau_r/s \) | \( \varphi \) drawn from alphabet \( \Phi \) disjoint to \( \Sigma \) |
| \( \text{Typeof } \) Expression | \( \delta/\sigma : t \) | \( \varrho \) Pseudo expression, an expression whose type is not grounded |
| \( \xi := \varphi \) or \( \varrho \) | \( \delta := \delta, \xi \rightarrow \text{typeof } \varrho \) | \( \varrho \) Sequence of pseudo-expressions |
| \( \text{Typeof } \) Definition, used only in typeof clause | \( \delta := \delta, \varrho \mid \varrho, \delta \rightarrow \tau \) | \( \varrho \) Auxiliary function definition |
| \( \varrho \) Auxiliary function names, \( \sigma \) | \( \theta := \delta, \varrho \mid \varrho, \delta \rightarrow \tau \) | \( \varphi \) Auxiliary function names, \( \varrho \) |
| (same as Fig. 3.3 (c) and...) | \( \theta := \delta, \varrho \mid \varrho, \delta \rightarrow \tau \) | \( \delta := \varrho, \theta \mid \varrho, \theta \rightarrow \tau \) |
| \( \varrho \) Sequence of pseudo-expressions | \( \theta := \delta, \varrho \mid \varrho, \delta \rightarrow \tau \) | \( \delta := \delta, \varrho \mid \varrho, \delta \rightarrow \tau \) |
| \( \text{Sequence of } \) auxiliary functions (\( \varphi \)). | \( \theta := \delta, \varrho \mid \varrho, \delta \rightarrow \tau \) | \( \theta := \delta, \varrho \mid \varrho, \delta \rightarrow \tau \) |

The figure uses two syntactical categories for defining functions: \( \delta \in \Delta \), which as before, defines a function named \( \sigma \in \Sigma \) that may occur in expression \( \mathcal{e} \) (more generally \( \mathcal{e} \)); the similarly structured \( \xi \in \Xi \) uses distinct namespace \( \varrho \in \Phi \) for functions that may occur in a typeof clause.

**Pseudo-expressions.** Compare \( \tau \rightarrow \text{typeof } \varrho \) (the format of a definition of function named \( \sigma \) in the figure) with \( \tau \rightarrow \tau \) (the format of this definition in n-ary-function type system (Fig. 3.3)). Without type capturing, \( \sigma \)’s return type is determined by a tree rewrite of the argument type(s). With type capturing, the return type is determined by subjecting type \( \tau \) to other function(s). To see this, expand the recursive abstract syntax definition of \( \delta \), assuming for simplicity that \( n = 1 \),

\[
\delta := \sigma : \tau \rightarrow \text{typeof } \tau, \varrho_1, \ldots, \varrho_r, \tag{3.8}
\]

i.e., the pseudo-expression \( \varrho \) in this case is \( \varrho = \tau, \varrho_1, \cdots, \varrho_r \). If \( n > 1 \) the return type of a function defined with typeof is specified by hierarchical structure \( \delta \), for which the figure coins the term pseudo-expression. Notice that a plain expression is a tree whose leaves (type instantiations) are drawn from \( \Gamma \) and internal nodes (function calls) are drawn from \( \Sigma \). Pseudo expressions are more general in allowing type variables in their leaves. As emphasized in the figure, these variables must be drawn from \( \tau \), the multi-pattern defining the types of arguments to \( \sigma \).

A full-typeof type system allows any number of function calls in pseudo-expression \( \varrho \), as in (3.8). In contrast, a rudimentary-typeof type system allows at most one function symbol in pseudo-expressions. This restriction is obtained by replacing the abstract syntax rule for \( \varrho \) in Fig. 3.4 with a simpler, non-recursive variant, \( \theta := \tau, \sigma \mid \tau \).

To describe the semantics of typeof, we need to extend the notion of tree substitution to pseudo-expressions as well. The application of function \( \sigma \) of (3.8) to a multi-expression \( \mathcal{e} \) with multi-type \( t \)
requires first that \( t \subseteq \tau \), where the matching uses a grounded substitution \( s \). Then, \( \theta/s \), the application of \( s \) to pseudo-expression \( \theta \) is the plain-expression obtained by replacing the type variables in \( \theta \) with the ground types defined by \( s \).

**Type of Expression** typing rule employs this notion as follows: typing expression \( e.\sigma \) with function \( \sigma : \tau \rightarrow \text{type of} \ \theta \) and arguments \( e : t \), we (i) match the argument types with the parameter types, \( t = \tau/s \), deducing substitution \( s \), (ii) type \( \theta/s : t \) (using an appropriate typing rule), and finally (iii) type \( e.\sigma : t \). As an application of the **Type of Expression** rule requires an additional typing, of \( \theta \), its definition is recursive.

### 3.1.7 Overloading

The one-type property means that expressions must have exactly one type (as defined in Fig. 3.1). With the more potent, multi-type property, expressions are allowed multiple types, by disposing the One Type Only type inference rule of Fig. 3.1. With multi-type-overloading, expressions are allowed multiple types. With eventually-one-type, the semantics of the Ada programming language [Persch et al. 1980] apply: Sub-expressions are allowed to have multiple types. However, upper level expressions are still required to be singly typed. For example, while the upper level expression \( e = \sigma_1(\sigma_2(\sigma_1())) \) can be assigned at most one type, both \( \sigma_1() \) and \( \sigma_2(\sigma_1()) \) may have many types.

### 3.2 Bisimulation of Automata and Type Systems

The notation used in this section highlight ties between tree automata and type systems, e.g., a tree \( t = y_1(y_2(y_3), y_4) \) can be understood as an instantiated generic type, \( y_1<2<y_3>, y_4> \), to use Java syntax. Likewise the tree rewrite \( \rho = y_1(y_2(x_1), x_2) \rightarrow y_2(x_2) \) can be interpreted as a Java function \( \text{static}<x_1, x_2>y_2<2><2>	ext{foo}(y_1<y_2<x_1>, x_2>e)() \). Applying \( \rho \) to \( t \) yields the tree \( y_2(y_1) \), while the return type of the invocation \( \text{foo(new } y_1<y_2<y_3>, y_4)>() \) is \( y_2<y_4> \).

In fact, with the above definitions of type systems and finite control automata, we can now easily pair certain automata with type systems.

| Observation 1. | 1. | 2. | 3. | 4. | 5. |
|---------------|----|----|----|----|----|
|               | \( \mathfrak{I} \subseteq \text{FSAt} \) | \( \text{deep-PA} \subseteq \text{deep-TA} \) | \( \text{deep-P} \subseteq \text{TA} \) | \( \text{non-linear-P} \subseteq \text{non-linear-TA} \) | \( (\text{monadic}) = \text{SRDPA} \) |

To be convinced, notice the natural bisimulation of automata and type system, obtained by a one-to-one correspondence between, e.g.,

- a run of an automaton and the type checking process as dictated by the type checking rules,
- the hanging of an automaton, and failure of type checking,
- the input word or tree, and the type-checked expression,
- input-output items in \( \Lambda_A \) and function definitions in \( \Lambda_T \),
- the contents of auxiliary storage, and the type of checked expression.

Observe however that states of an automaton do not easily find their parallel in the typing world (except for \( \mathfrak{I} = \text{FSAt} \), in which classes correspond to states). Luckily, the expressive power of many of the automata we deal with does not depend on the presence of states, e.g., it is easy to see that \( \text{deep-TA} = (\text{deep.stateful}) \vee \text{TA} \).

### 4 PARAMETRIC POLYMORPHISM AND REAL-TIME AUTOMATA

The following result employs the type-automata correspondence to characterize the complexity class of type system \( \text{PA} \).

**Theorem 4.1.** \( \text{PA} = \text{DCFL} \)

Recalling the equivalence \( \text{PA} = \text{TA} \) (Obs. 1), the gist of the theorem is the claim \( \text{TA} = \text{DCFL} \). Towards the proof we draw attention to G&R’s “tree encoding”, which is essentially a reduction by which every DPDA is converted to an equivalent tree automaton. Their work then proceeds to show how this tree automaton is emulated in the \( \text{PA} \) type system they use (and that the emulation does not incur exponential space (and time) overhead). Hence, by G&R [2019]
DCFL = DPDA ⊆ TA = P₀.

A similar result is described by Guessarian [1983]. In fact, we note that Guessarian’s contribution is more general, specifically she achieves the result that augmenting tree automata with ε-transitions and multiple states does not increase their computational class.

**Fact 4.1 (Guessarian 1983, Corollary 1.(i)).** \((ε\text{-transitions, stateful}) \cup TA = TA\)

Fact 4.1 generalizes (4.1), since DPDA's are instances of \((ε\text{-transitions, stateful}) \cup TA\), where the tree store is linear. The proof of Thm. 4.1 is completed by showing the inverse of (4.1).

**Lemma 4.1.** \(TA \subseteq DPDA\).

**Proof.** The proof is constructed by employing Theorem 3 of Guessarian [1983]. (Notice that she uses the term "pushdown tree automaton" (PDTA) for top-down tree-automata. However, for the purpose of the reduction, we concentrate on input trees that are in the form of a string, i.e., the tree traversal order is immaterial.)

Observe that Lem. 4.1 means that G&R’s result is the best possible in the following sense: It is impossible to extend Fling to support any wider family of fluent API languages within the limits of the fragment of the Java type system that Fling uses. Moreover, as shown by Grigore [2017], allowing the fluent API generator a larger type system fragment, makes type-checking undecidable if the larger fragment includes the common Java idiom of employing `super` in signatures, as in e.g., method `boolean removeIf(Predicate<? super E> filter)` found in the standard java.util.Collection class.

Combining Obs. 1 (5), known results (Table 2.2) and Thm. 4.1, we have

\[
\langle \text{monadic} \rangle = \text{SRDPDA} \subseteq \text{DCFL} = P₀ = \langle \text{polyadic} \rangle, \tag{4.2}
\]

i.e., had \(P₀\) been weakened to allow only monadic generics, its expressive power would have been reduced. Conversely, we would like to check the changes to complexity when \(P₀\) is made more potent. Consider now allowing generic functions (on top of methods of generic classes) by adding the deep-type-pattern feature to \(P₀\).

**Theorem 4.2.** \(\text{DCFL} \subseteq \text{deep-TA} = \text{deep-}P₀\)

Again, recall the equivalence `deep-P₀ = deep-TA` from Obs. 1. The set containment, \(\text{DCFL} \subseteq \text{deep-TA}\) follows from (4.1). It remains to show that this containment is proper.

The proof of Thm. 4.2 in Sect. C.1 is by encoding the context sensitive language \(a^n b^n c^n \subseteq \{a, b, c\}^∗\) in type system `deep-P₀`, and relying on the following definition: For an integer \(k ≥ 0\), let \(U_k\), the unary type encoding of \(k\), be a grounded type in \(P₀\), \(U_0 = \text{Zero}\), and \(U_k = \text{Succ} <\text{Zero}, \ldots, \text{Zero}>\), with types (in Java syntax) `interface Zero()` and `interface Succ<T>()` (assumed implicitly henceforth). Thus, \(U_0 = \text{Zero}\), \(U_1 = \text{Succ<Zero>}, U_2 = \text{Succ<Succ<Zero>>, etc.}\)

Note that in type system \(P₀\) it is possible to `increment` and `decrement` integers,

```
1 static Zero zero() { return null; }
2 interace Zero { ...
3   Succ<Zero> inc(); ...
4 }
```

We have, e.g., that the type of expression `zero().inc().inc().inc().dec().inc()` is \(U_4\).

We now show that just like deep patterns, non-linear patterns, i.e., patterns in which the same type variable occurs more than once, increase the computational power of \(P₀\).

This increase is attributed here to the ability of non-linear patterns to compare nested types, in particular types that are unary encoding of the integers. For example, the Java generic function `static <X>void equal(X e1,X e2)()` in type system `\langle non-linear-patterns,n-ary-functions\rangle \cup P₀\)` type-checks if the types of its arguments are (say) \(U_0\) and \(U_0\), and does not type check if these are (say) \(U_8\) and \(U_7\). More importantly, type comparison is also possible if all functions are unary.

Consider, e.g., the two argument generic type \(γ\), `interface γ<X1,X2>()`, and the generic unary function `equal, static<X>void equal(γ<X,x,>ε)()`, in type system `non-linear-P₀`. Then, function `equal`
**type-checks** if the type of its single argument is (say) γ<ℓ₀,ℓ₂>, and does not type-check if this type is (say) γ<ℓ₁,ℓ₃>. With this observation, we can state.

**Theorem 4.3.** $\text{DCFL} \subseteq \text{non-linear-TA} = \text{non-linear-Pₗ}$

The proof of Thm. 4.3 in Sect. C.2 is again by encoding the context sensitive language $a^n b^n c^n \subseteq \{a,b,c\}^*$, but this time in type system non-linear-Pₗ. The ability of this type system to compare integers encoded as types is the gist of the proof.

Recall that a type system is dyadic if no generic takes more than two type parameters. Considering the shallow case, we claim no more than placing dyadic between monadic and polyadic in (4.2),

$$\langle\text{monadic}\rangle \subseteq \langle\text{dyadic}\rangle \subseteq \langle\text{polyadic}\rangle,$$

although we conjecture $\langle\text{monadic}\rangle \not\subseteq \langle\text{dyadic}\rangle$ can be shown relatively easily. In contrast, in deep type system, the expressive power does not increase by allowing more than two generic parameters.

**Theorem 4.4.** $\langle\text{deep,polyadic}\rangle - \langle\text{deep,dyadic}\rangle$

Proof. (sketch) Relying on the automata-type correspondence, we construct for every deep-TA automaton $A$, an equivalent binary deep-TA $A'$. Let $γ$ be a tree node in $A$ of rank $k > 2$: Replace $γ$ with nodes $γ_1,γ_2,\ldots,γ_k$ of rank two, and $γ_k$ of rank one. Tree nodes appear in both sides of tree rewrite rules, and in the initial auxiliary storage tree: Replace every occurrence of $γ$ in $A$, $γ(τ_1,τ_2,\ldots,τ_k)$, with $γ_1(τ_1,γ_2(τ_2,\ldots,γ_{k-1}(τ_{k-1},γ_k(τ_k)\ldots))$. $\square$

### 5 TYPE CAPTURING AND $ε$-TRANSITIONS

In the previous section we showed that the addition of deep-type-pattern property, as found in generic, non-method functions of (say) Java, to the Pₗ type system, increases its computational complexity, but does not render it undecidable. We now prove that the addition of even rudimentary typeof to Pₗ makes it undecidable.

**Theorem 5.1.** $\langle\text{deep,rudimentary-typeof}\rangle \in \text{RE}$.

The following reduction is pertinent to the proof of Thm. 5.1.

**Lemma 5.1.** A Turing machine $M$ can be simulated by a deep-rewrite, stateful tree automaton $A$ which is allowed $ε$-transitions.

Proof. As explained in Sect. 2, we can assume that $M$ accepts its input on the tape with the head on the first letter, and then engages in $ε$-transitions only. Also, w.l.o.g., $M$'s tape is extended infinitely in both directions by an infinite sequences of a designated blank symbol $♭$.

Fig. 5.1 is an example of such a machine $M$ with internal states $q_0$ through $q_4$, single accepting state $q_4$, and, tape alphabet $Γ = \{a, b, ♭\}$. The machine terminates in an accepting state if and only if the tape is initialized with a word $a^n b^n$, $n \geq 0$: To see this, notice that the machine repeatedly replaces $a$ from the beginning of the word and its counterpart letter $b$ from the word's end by $♭$, until no more $a$'s or $b$'s are left. The convention of depicting transitions over edges in the graph of states is standard, e.g., the arrow and label rendered in purple (going from state $q_1$ to state $q_2$) is the $ε$-transition item

$$\langle q_0, a \rightarrow ♭, q_1 \rangle,$$

which states that if the Turing machine is in internal state $q_0$, and, the symbol under head is $a$, then (i) replace $a$ by $♭$, (ii) increment $h$, and and, (iii) change internal state to $q_1$.

The encoding of $M$ in $A$ includes the following components:
(1) Adopting the set of states $Q$, set of accepting states $F$, and initial state $q_0$ of $M$.
(2) A rank-1 tree symbol for each of the tape symbols, including $b$.
(3) Employing the designated leaf symbol $\varepsilon \notin \Gamma$ to encode the infinite sequences of $b$ at the ends of the tape.
(4) Introducing a rank-3 tree symbol $\circ$ for encoding the tape itself. The center child of a node labeled $\circ$ encodes of a $\circ$ node encodes the cell under the head; its left (resp. right) child encodes the tape to the left (resp. to the right) of the head. For example, the tape contents $\cdots bbbqabbbbbb \cdots$ is encoded by a certain tree $t = \circ(b(\varepsilon), a(\varepsilon), a(b(b(\varepsilon))))$.
For the sake of readability we write $\circ$ nodes in infix notation, e.g., $t = b(\varepsilon)/a(\varepsilon)/a(b(b(\varepsilon)))$, or even more concisely $t = b/a/abb$.
(5) Setting $y_0 = \varepsilon/\sigma_1/\sigma_2 \cdots \sigma_n$, i.e., letting the initial state of auxiliary storage encode the input word $\sigma_1\sigma_2 \cdots \sigma_n$.
(6) Introducing $|\Sigma| + 1$ transitions in $A$ for each of $M$’s transitions: A single transition for dealing with the $\varepsilon$ leaf denoting an infinite sequence of blanks, and a transition for each tape symbol.
In demonstration, transition $\langle q_0, a \rightarrow b_+, q_1 \rangle$ (5.1) is encoded in four $\varepsilon$-transitions of $A$ which differ only in their tree rewrite rule.
\[
\begin{align*}
\langle q_0, x_1/a/ax_2 \rightarrow b/x_1/a/x_2, q_1 \rangle & \quad \langle q_0, x_1/a/bx_2 \rightarrow b/x_1/b/x_2, q_1 \rangle \\
\langle q_0, x_1/a/bx_2 \rightarrow b/x_1/b/x_2, q_1 \rangle & \quad \langle q_0, x_1/a/\varepsilon \rightarrow b/x_1/b/\varepsilon, q_1 \rangle.
\end{align*}
\] (5.2)

The rules above distinguish between the values the right child of node $\circ$, i.e., the symbol to the right of the head: For example, the first rule, $x_1/a/ax_2 \rightarrow b/x_1/a/x_2$, deals with the case this child is $a$ followed by some tape suffix captured in variable $x_2$. The rule rewrites the node, making $a$ the center child.

Notice that with the encoding, the input to $A$ is encoded in its transitions rules. □

Relying on Lem. 5.1, the proof of Thm. 5.1 is completed by encoding the automaton $A$ of the lemma in the appropriate type system.

Proof of Thm. 5.1. We encode automaton $A = A(M)$ as a program $P = \Delta\Xi e$ in type system $\langle \text{deep, rudimentary} \rangle \setminus \Psi \Phi$. In this encoding, set $\Delta$ is empty, and there is a function $\xi \in \Xi$ for every $\varepsilon$-transition item in set $\Xi$ of $A$. Expression $e$ type checks against $\Xi$, if, and only if, machine $M$ (automaton $A$) halts.

In the encoding, the tree vocabulary of $A$ incarnates as generic types: A three parameter generic type $\circ$, and generic one-parameter type $\gamma$ for each tape symbol, including $b$. Also the argument to every function $\xi \in \Xi$ function is a deep pattern over possible instantiations of $\circ$.

Also, introduce a function symbol $\phi_q$ for every $q \in Q$, and let every transition $\langle q, \tau \rightarrow \tau', q' \rangle$ of $A$ add an overloaded definition $\phi_q : \tau \rightarrow \text{typeof} \ \tau'.q'$ to this symbol. Thus, function $\phi_q$ emulates $A$ in state $q$ with tape $\tau$: It applies the rewrite $\tau \rightarrow \tau'$ to the type, and employs the resolution of $\text{typeof}$ to continue the computation in function $q'$ which corresponds to the destination state $q'$.

For example, the Turing machine transition shown in (5.1), encoded by the tree automaton transitions of (5.2), is embedded in C++ using $\text{decltype}$, as depicted in List. 5.1.

**Listing 5.1** Definitions in type system $\langle \text{rudimentary-typeof, deep} \rangle \setminus \Psi \Phi$ (using C++ syntax) encoding the tree automata transitions of (5.2)
Further, to encode the input word, set \( e = o(e, \sigma_1, \sigma_2(\cdots \sigma_n(e) \cdots)) \cdot \varphi_{q_0} \), or, in monadic abbreviation form, \( e = o(e, \sigma_1, \sigma_2(\cdots \sigma_n(e) \cdots)) \cdot \varphi_{q_0} \).

To terminate the typing process, further overload \( \varphi_e \) with definition \( \varphi_e : o(x_1, \gamma(e), x_2) \rightarrow e \) for every accepting state \( q \in F \) and cell symbol \( \gamma \in \Gamma \), for which a Turing machine transition is not defined. These definitions correspond to the situation of \( A \) reaching an accepting state—type checking succeeds if and only if \texttt{typeof} resolution reaches such a definition.

The full C++ encoding of the Turing machine of Fig. 5.1 is shown in List. D.1 in the appendices. □

Having examined the contribution of \texttt{deep} by itself, and the combination of \texttt{deep} and \texttt{rudimentary} to the computational complexity of \( \texttt{P}_k \), it is time to consider the contribution of \texttt{rudimentary by itself} to complexity. The following shows that there is no such contribution.

**Theorem 5.2.** \( \texttt{P}_k = \texttt{rudimentary}-\texttt{typeof}-\texttt{P}_k \)

**Proof.** The first direction \( \texttt{P}_k \subseteq \texttt{rudimentary}-\texttt{typeof}-\texttt{P}_k \) is immediate, as every \( \texttt{P}_k \) program is also a \texttt{rudimentary}-\texttt{typeof}-\texttt{P}_k program by definition. We prove \texttt{rudimentary}-\texttt{typeof}-\texttt{P}_k \subseteq \texttt{P}_k.

Given a program \( P = \Delta \Xi e \) in \texttt{rudimentary}-\texttt{P}_k, we need to convert it into equivalent program \( P' \) in type system \( \texttt{P}_k \). By Thm. 4.1 it is sufficient to convert \( P \) into a vanilla tree automaton, i.e., one with neither states nor \( \varepsilon \)-transitions. Instead, we convert \( P \) into a more potent tree automaton \( A \) which is allowed both \( \varepsilon \)-transitions and states, and then employ Guessarian’s observation \( \langle \varepsilon \text{-transitions}, \text{stateful} \rangle \sqrt{\text{TA}} = \text{TA} \) (see Fact 4.1 above) to complete the proof.

The set of internal states of \( A \) includes an initial and accepting state \( q_0 \) and a state \( q_\varphi \) for every auxiliary function name \( \varphi \) used in \( \Xi \).

Consider a definition in \( P = \Delta \Xi e \) of a (primary or auxiliary) function that employs a \texttt{typeof} clause \( \tau \rightarrow \texttt{typeof} \theta \). With rudimentary \texttt{typeof}, pseudo-expression \( \theta \) is either \( \tau' \) or \( \tau'.\varphi \). Therefore, every function definition is either in the direct form \( \tau \rightarrow \tau' \) or in the forwarding form \( \tau \rightarrow \texttt{typeof}\tau'.\varphi \).

There are four cases to consider:

1. **Primary function definitions,** found in \( \Delta \), are encoded as consuming transitions of \( A \):
   
   (a) Direct definition \( \sigma : \tau \rightarrow \tau' \) is encoded as transition \( \langle \sigma, q_0, \tau \rightarrow \tau', q_0 \rangle \).

   (b) Forwarding definition \( \sigma : \tau \rightarrow \texttt{typeof} \tau'.\varphi \) is encoded as transition \( \langle \sigma, q_0, \tau \rightarrow \tau', q_\varphi \rangle \).

2. **Auxiliary function definitions,** found in \( \Xi \), are encoded as \( \varepsilon \text{-transitions} \) of \( A \):
   
   (a) Direct definition \( \varphi : \tau \rightarrow \tau' \) is encoded as transition \( \langle q_\varphi, \tau \rightarrow \tau', q_0 \rangle \).

   (b) Forwarding definition \( \varphi : \tau \rightarrow \texttt{typeof} \tau'.\varphi \) is converted to \( \varepsilon \text{-transition} \) \( \langle q_\varphi, \tau \rightarrow \tau', q_\varphi \rangle \).

In all four cases, the change from input type to output type by a function is encoded as a rewrite of the tree auxiliary storage of \( A \). Direct definitions are encoded by \( A \) moving into state \( q_0 \) and \texttt{typeof} function \( \psi \) is encoded by \( A \) moving into state \( q_\varphi \).

Notice that state \( q_0 \), the only accepting state, is the only state with outgoing consuming transitions, and it is also the only one without outgoing \( \varepsilon \)-transitions. Therefore, the automaton consumes a letter in state \( q_0 \), and finishes conducting \( \varepsilon \)-transitions back in \( q_0 \), or otherwise it rejects the input.

With the above construction, expression \( e = \varepsilon.\sigma_1.\cdots.\sigma_n \) type-checks against \( \Delta \) and \( \Xi \) if and only if \( A \) accepts word \( w = \sigma_1 \cdots \sigma_n \).

**Theorem 5.3.** \texttt{Fluent} \( \subseteq \texttt{DCFL} \)

**Proof.** Yamazaki et al. [2019] showed that \texttt{DCFL} \( \subseteq \texttt{Fluent} \), i.e., that any LR language, alternatively, any \texttt{DCFL}, can be encoded in a \texttt{Fluent} program. It remains to show the converse, \texttt{Fluent} \( \subseteq \texttt{DCFL} \). We prove \texttt{Fluent} \( \subseteq \texttt{deep}-\texttt{DPDA} \), noting the folk-lore equality

\[
\texttt{deep}-\texttt{DPDA} = \texttt{DPDA}.
\] (5.3)

The encoding of a \texttt{Fluent} program in a \texttt{deep}-\texttt{DPDA} is reminiscent of the encoding of a program in \texttt{rudimentary}-\texttt{P}_k type system in a vanilla tree automaton in the proof of Thm. 5.2 just above. The full proof of the current theorem is in Sect. C.3. □
Having seen that Fluent is not more expressive than it was intended to be, it is interesting to check whether its expressive power would increase if it allowed unrestricted typeof clauses.

**Theorem 5.4.** full-typeof-Fluent $\supseteq$ DCFL

The proof is by showing that type system full-typeof-Fluent is expressive enough to encode the language w#w, known to be context sensitive. The full proof is in Sect. C.4.

6 OVERLOADING RESOLUTION AND DETERMINISTIC COMPUTATION

Most previous work concentrated in recognition of deterministic languages [Gil and Levy 2016; Gil and Roth 2019; Grigore 2017; Nakamaru et al. 2017]. We show here that type system with Ada-like overloading can encode non-deterministic context free languages as well. Its proof relies on creating a direct correspondence of the type system and context free grammars (CFGs).

**Theorem 6.1.** UFCL $\subseteq\langle$ monadic, eventually-one-type$\rangle$

Proof. Given an unambiguous context free grammar G, we encode it as $\Lambda$, a set of function definitions in $\langle$ monadic, eventually-one-type$\rangle$ such that $G$ derives word $\sigma_1 \cdots \sigma_n$ if, and only if, expression $\epsilon.\sigma_1.\cdots.\sigma_n.\$ ($\$ being a dedicated function symbol) type checks against $\Lambda$.

We redefine CFGs using a notation more consistent with this manuscript: Context free grammar $G$ is a specification of a formal language over alphabet $\Sigma$ in the form of a quadruple $\langle \Sigma, \Gamma, \epsilon, R \rangle$ where $\Sigma$ is the set of $G$'s terminals, $\Gamma$ is the set of grammar variables, $\epsilon \notin \Gamma$ is the start symbol, and $R$ is a set of derivation rules. Each derivation rule $\rho \in R$ is either in the form $\epsilon \rightarrow \omega$, or in the form $\gamma \rightarrow \omega$, where $\gamma \in \Gamma$ and where $\omega$ is a possibly empty sequence of terminals and grammar variables, i.e., $\omega \in \langle \Sigma \cup \Gamma \rangle^*$.

Recall that a grammar is in Greibach Normal Form (GNF) if every rule $\rho \in R$ is in one of three forms (i) the usual form, $\rho = \gamma \rightarrow \sigma\gamma$, where $\sigma \in \Sigma$ is a terminal and $\gamma \in \Gamma^*$ is a sequence of variables, (ii) the initialization form, $\rho = \epsilon \rightarrow \sigma\gamma$, or, (iii) the $\epsilon$-form, $\rho = \epsilon \rightarrow \epsilon$, present only if the grammar derives the empty word $\epsilon \in \Sigma^*$.

For the encoding, first convert unambiguous grammar $G$ into an equivalent unambiguous grammar in GNF. This is done using the algorithm of Nijholt [1979] (also presented in more accessible form by Salomaa and Soittola [1978]).

The type encoding of GNF grammar $G$ uses a monadic generic type $\gamma$ for every symbol $\gamma \in \Gamma$, an additional monadic generic type $\$, and, one non-generic type $\epsilon$, also known as the unit type.

For each derivation rule $\rho \in R$ introduces a function $\delta \in \Lambda$ that uses these types:

- Suppose $R$ includes the $\epsilon$-form rule $\epsilon \rightarrow \epsilon\$, introduce (one overloaded) definition of function $\$ : $\epsilon \rightarrow \epsilon$. Then, $\epsilon\$, the expression corresponding to the empty word, type-checks to type $\epsilon$. (Recall that $\epsilon$ is the single type of the unit type $\epsilon$.)
- If $\rho$ is in the initialization form $\epsilon \rightarrow \sigma\gamma$ then $\delta = \sigma : \epsilon \rightarrow \gamma\$. For such a rule introduce also function $\$ : $\epsilon \rightarrow \epsilon$.
- If $\rho$ is in the usual form $\gamma \rightarrow \sigma\gamma$, then $\delta = \sigma : \gamma\epsilon \rightarrow \gamma\$.

We show by induction on $i = 1, \ldots, n$ the following claim on the partial expression $e_i = \epsilon.\sigma_1.\cdots.\sigma_i$: The set of types assigned by the type checker to $e_i$ includes a type $\gamma\$, $\gamma \in \Gamma^*$, if and only if, there exists a left most derivation (LMD) that yields the sentential form $\sigma_1.\cdots.\sigma_i\gamma$. For the inductive base observe that $e_0 = \epsilon$ and that the set of types of $\epsilon$ includes only the unit type $\epsilon$; indeed there is a (trivial) LMD of the degenerate sentential form $\epsilon\epsilon = \epsilon$.

Consider an LMD of $\sigma_1.\cdots.\sigma_i\gamma\$, where $i < n$, $\gamma' \in \Gamma^*$ and $\sigma_{i+1}$ is the terminal $\zeta \in \Sigma$, $\zeta \neq \$.

We show that $\gamma'$ is a type of $e_{i+1} = \zeta(e_i)$. The said LMD can only be obtained by applying a rule $\rho = \gamma \rightarrow \zeta\gamma$ to the sentential form $\sigma_1.\cdots.\sigma_i\gamma\$, where $\gamma$ is the first symbol of $\gamma$.

By examining the kind of functions in $\Lambda$, one can similarly show that every type $\gamma'$ of $e_{i+1}$ is an evidence of an LMD of a sentential form $\sigma_1.\cdots.\sigma_i\sigma_{i+1}\gamma'$. 

20
The proof is completed by manually checking that a full expression, ending with the $.\$ invocation can only type check to a single type, $\varepsilon$, and this can happen only if the type of $\varepsilon.\sigma_1.\cdots.\sigma_n$ is $\gamma$, where $\gamma$ occurs in an initialization rule $\varepsilon \rightarrow \sigma_n\gamma$. \hfill $\Box$

Sect. D.2 demonstrates the proof by presenting a fluent API of the non-deterministic context free language of even length palindromes.

If final expressions are also allowed to be multi-typed, then we can construct fluent API for all context free languages.

**Theorem 6.2.** \(<\text{monadic, multiple-type}> = \text{CFL}\)

**Proof.** The construction in the proof of Thm. 6.1 works here as well. Note that here the transition from a plain CFG to GNF does not have to preserve unambiguity. \hfill $\Box$

# 7 CONCLUSIONS

**Perspective.** Revisiting Table 3.1, we see that in total it has $|C_1| \cdot |C_2| \cdot |C_3| \cdot |C_4| \cdot |C_5| \cdot |C_6| = 4 \cdot 3 \cdot 2 \cdot 2 \cdot 3 \cdot 3 = 432$ lattice points. Accounting for the fact that in a \textit{nyadic} type system, the values of $C_2$ \textit{(type pattern depth)}, and $C_3$ \textit{(type pattern multiplicity)} are meaningless, we see that lattice $\Xi$ spans $|C_1| \cdot |C_2| \cdot |C_3| = 2 \cdot 3 \cdot 3 = 18$ monomorphic type systems ($\Xi_{\perp}$ among them), and $(|C_1| - 1) \cdot |C_2| \cdot |C_3| \cdot |C_4| \cdot |C_5| \cdot |C_6| = 3 \cdot 3 \cdot 2 \cdot 3 \cdot 3 \cdot 3 = 324$ potential polymorphic type systems ($\Pi$ and \textit{Fluent} among them). To make the count more exact, account for $C_3$ being irrelevant in a \textit{monadic} type system, obtaining $|C_2| \cdot |C_4| \cdot |C_5| \cdot |C_6| = 3 \cdot 2 \cdot 3 \cdot 3 = 36$ \textit{monadic}, yet polymorphic type systems, and $(|C_1| - 2) \cdot |C_2| \cdot |C_3| \cdot |C_4| \cdot |C_5| \cdot |C_6| = 2 \cdot 3 \cdot 2 \cdot 3 \cdot 3 \cdot 3 = 216$ non-\textit{monadic} polymorphic type systems.

Beyond the implicit mention that the type-automata correspondence relates to \textit{monomorphic type systems}, these were not considered here. Our study also invariably assumed \textit{unary-function}, ignoring in characteristic $C_4$ \textit{n-ary-functions type systems} which comprise half of the type systems of $\Xi$.

Even though most of this work was in characterizing the complexity classes of type systems, it could not have covered even the $(36 + 216)/2 = 126$ type systems remaining in scope. The study rather focused on these systems which we thought are more interesting: We gave an exact characterization of the complexity classes of two central type systems, $\Pi_1$ (Thm. 4.1) and \textit{Fluent} (Thm. 5.3), and investigated how this complexity changes if the type systems are made more or less potent along $\Xi$’s characteristics (with the exception of $C_4$, the function arity characteristic). Comparing (3.1) with Table 3.1 we see that \textit{Fluent} can be made more potent along $C_1$, $C_5$, or $C_6$, and, as follows from our results, its complexity class increases in all three cases:

1. In $C_1$, \textit{Fluent} \(\subseteq\) \textit{dyadic-Fluent} = RE, by combining Thm. 4.4 and Thm. 5.1.
2. In $C_5$, \textit{Fluent} \(\subseteq\) \textit{eventually-one-type-Fluent} (Thm. 6.1).
3. In $C_6$, \textit{Fluent} \(\subseteq\) full-typeof-\textit{Fluent} (Thm. 5.4).

Conversely, \textit{Fluent} can be made less potent along characteristics $C_1$, $C_2$ and $C_5$:

1. In $C_1$ complexity decreases, \textit{Fluent} \(\Rightarrow\) \textit{monadic} = FSA \(\subseteq\) \textit{Fluent} (Obs. 1).
2. In $C_2$, (5.3) makes us believe that complexity does not change, \textit{Fluent} \(\Rightarrow\) \textit{deep+shallow} = \textit{Fluent}.
3. In $C_5$, then, by (Obs. 1 and (5.3)), \textit{Fluent} \(\Rightarrow\) \textit{rudimentary} = \textit{deep-RDPDA}. We believe complexity decreases but are unsure.

Type system $\Pi_1$ can be made more potent along characteristics $C_2$, $C_3$, $C_5$ and $C_6$:

1. In $C_2$ complexity increases, $\Pi_1 \not\subseteq$ deep-$\Pi_1$ (Thm. 4.2).

\[\text{footnote}{\text{the ignored} n\text{-ary-functions} \text{correspond to the forest-recognizer brand of automata; however forest-recognizer automata were used in the construction, e.g., in Lem. 5.1.}}\]
(2) In $C_3$ complexity increases, $P_b \subseteq \text{non-linear-}P_b$ (Thm. 4.3).
(3) In $C_3$ complexity does not change, $P_b = \text{rudimentary-typeof-}P_b$ (Thm. 5.2).
(4) In $C_6$ complexity increases, $P_b \subseteq \text{eventually-one-type-}P_b$ (Thm. 6.1).

Type system $P_b$ can be made less potent only along characteristic $C_1$. From Obs. 1 and Thm. 4.1,

$$
FSA = \langle \text{n-ary functional} \rangle \subseteq SRDPDA = \langle \text{monadic} \rangle \subseteq \langle \text{dyadic} \rangle \subseteq \langle \text{polyadic} \rangle \\
\subseteq \langle \text{polyadic} \rangle = \text{DCFL}.
$$

i.e., it is not known whether decreasing $P_b$ along $C_1$ to dyadic reduces its complexity, but decreasing it further to monadic certainly does.

This work should also be viewed as a study of the type-automata correspondence: (i) The results in Sect. 4 revolve around the correspondence between tree-store automata employing tree rewrites, and type system in which the signature of functions employs type pattern to match its argument. (ii) Sect. 5 explored the correspondence between typeof clause in the signature of functions, and $\varepsilon$-transitions of automata. (iii) The correspondence between non-deterministic runs and allowing multiple types of expressions, or at least as a partial step during resolution of overloading was the subject of Sect. 6. Overall, our study confirmed that the type-automata correspondence is a significant aid in the characterization of complexity classes, either by a direct bisimulation between the two, or by employing and adapting (sometimes ancient) contributions in the decades old research of automata.

**Open Problems.** Technically, we leave open the problem of characterizing the complexity class of each of the 126 type systems that were not considered at all, or, considered, but not fully characterized. However, many of these can be trivially solved, e.g., since $T_1 = \langle \text{deep, rudimentary, polyadic} \rangle = \text{RE}$, (Thm. 5.1), $T_2 = \text{RE}$ for all $T_2 \in \mathcal{T}$, $T_2 > T_1$. We draw attention to four type systems for which we are able to set a lower and an upper bound, but still miss precise characterization, e.g., in terms of familiar computational complexity classes.

1. deep-$P_b$, for which we have DCFL $\subseteq$ deep-$P_b \subseteq$ CSL by Thm. 4.2.
2. non-linear-$P_b$, for which we also have DCFL $\subseteq$ non-linear-$P_b \subseteq$ CSL by Thm. 4.3.
3. $\langle \text{deep,non-linear} \rangle \vee P_b$, for which we have again DCFL $\subseteq$ $\langle \text{deep, non-linear} \rangle \vee P_b \subseteq$ CSL by Thms. 4.2 and 4.3.
4. full-typeof-Fluent, for which we have DCFL $\subseteq$ full-typeof-Fluent $\subseteq$ RE by Thm. 5.4.

Also, we do not know yet how these relate to each other in terms of computational complexity, beyond what can be trivially inferred by $\mathcal{T}$’s partial order. Sect. D.3 may offer some insights.

**Expression Trees vs. Expression Words.** Language recognizers, i.e., automata which take trees as inputs were defined and used in the proofs. Still, this study does not offer much on the study of n-ary-functions—the type counterpart of language recognizers. There is potential in exploring the theory of polymorphic types of tree shaped expressions. In particular, it is interesting to study type systems $S_1 = \langle \text{n-ary, deep} \rangle$ and $S_2 = \langle \text{n-ary, deep, non-linear} \rangle$, both modeling static generic multi-argument functions of C# and Java, except that $S_2$ adds the power, and predicament (see List. 3.1), of non-linear type patterns. In the type-automata perspective $S_1$ and $S_2$ correspond to forest-recognizer real-time tree-store brand of automata, which received little attention in the literature. We see two number of potential applications of type theory, for which (say) $P_b$ is insufficient, and could serve as motivation for resolving the open problems above and for the study of $S_1$ and $S_2$.

**Types for linear algebra** The matrix product $A \times B$ is defined if matrix $A$ is $m_1 \times m_2$ and matrix $B$ is $m_2 \times m_3$, in which case the result is an $m_1 \times m_3$ matrix. The matrix addition $A + B$ is defined only if both $A$ and $B$ are $m_1 \times m_2$, in which case the result is also $m_1 \times m_2$. The unary encoding of integers and their comparison in one step in the proof of Thm. 4.3 seem to be sufficient for developing a decidable type system that enforces such constraints.
However, unlike type systems for checking fluent API, types for linear algebra implemented this way are impractical: matrices whose dimensions are in the range of thousands are common, e.g., in image processing. But, programmers cannot be expected to encode integers this large in unary, not mentioning the fact that such types tend to challenge compilers’ stability. The problem is ameliorated in $S_2$ in which a decimal (say) representation of integers is feasible. A more precise design is left for future research.

A more difficult challenge is the type system support and checking of operations which involve integer arithmetic. A prime example is numpy\(^{11}\)’s reshape operation which converts, e.g., an $m_1 \times m_2$ matrix to an $m_3 \times m_4$ matrix, where correctness is contingent on the equality if $m_1 \cdot m_2 = m_3 \cdot m_4$. Indeed, we are not aware of any decidable type system that can do integer multiplication.

**Dimensional types** A similar challenge is supporting of physical dimensions, i.e., a design of a type system allowing, e.g., the division of distance quantity by time quantity obtaining speed quantity, and addition and comparison distance quantities, but forbidding, e.g., addition and comparison of time and distance quantities. To do so, the type system should probably encode $\prod_i x_i^{m_i}$, $m_i \in \mathbb{Z}$, the general form of a physical dimension (in say MKS), as a tuple of $r$ of signed integers.

To enforce the rules of addition and comparison of physical dimensions, the type system should be able compare (typically very small) integers, as done in Thm. 4.3, although the implementation should be tweaked to support negative integers. For multiplying and dividing physical quantities, the type system should be able to add (small) integers. We do not know whether this is possible in $S_1$ or $S_2$.

**Modeling type erasure.** Finally, we draw attention to the fact that Java’s type erasure is not accurately modeled by our system. In particular Java forbids function overloading if the type of the overloaded functions becomes identical after type erasure. We propose this type inference rule for type erasure

$$
\frac{\sigma : y(\tau) \rightarrow \tau \quad \sigma : y(\tau') \rightarrow \tau'}{\sigma : \bot}
$$

and leave the problem of studying type systems with type erasure to future research.

REFERENCES

Nada Amin and Ross Tate. 2016. Java and Scala’s type systems are unsound: The existential crisis of null pointers. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016). Association for Computing Machinery, New York, NY, USA, 838–848. https://doi.org/10.1145/2983990.2984004

Jean-Michel Autebert, Jean Berstel, and Luc Boasson. 1997. Context-free languages and pushdown automata. Springer, Berlin, Heidelberg. 111–174 pages. https://doi.org/10.1007/978-3-642-59136-5\_3

Henk Barendregt. 1991. Introduction to generalized type systems. J. Functional Programming 1 (1991), 125–154. Issue 2. https://doi.org/10.1017/s0956796800002025.

Stefan D. Bruda and Selim G. Akl. 1999. On the power of real-time Turing machines: $k$ tapes are more powerful than $k – 1$ tapes. Technical Report. Queen’s University.

H. Comon, Ma Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2007. Tree Automata Techniques and App. Available on: http://grappa.univ-lille3.fr/tata. (2007). release Oct, 12th 2007.

Jean-Luc Coquidé, Max Dauchet, Rémi Gilleron, and Sándor Vágvolgyi. 1994. Bottom-up tree pushdown automata: classification and connection with rewrite systems. Theoretical Comp. Science 127, 1 (1994), 69–98. https://doi.org/10.1016/0304-3975(94)00101-5

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’82). Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

\(^{11}\)https://numpy.org/
Joseph (Yossi) Gil and Ori Roth

Alan A.A. Donovan and Brian W. Kernighan. 2015. The Go Programming Language. Addison-Wesley Professional, Boston, MA, USA.

Yossi Gil and Tomer Levy. 2016. Formal language recognition with the Java type checker. In 30th European Conf. on OO Prog. (ECOOP 2016) (Leibniz International Proceedings in Inf. (LIPIcs)), Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:27. https://doi.org/10.4230/LIPIcs.ECOOP.2016.10

Yossi Gil and Ori Roth. 2019. Fling—a fluent API generator. In 33rd European Conf. on OO Prog. (ECOOP 2019) (Leibniz International Proceedings in Inf. (LIPIcs)), Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 13:1–13:25. https://doi.org/10.4230/LIPIcs.ECOOP.2019.13

Jean-Yves Girard. 1971. Une Extension De ľInterpretation De Gödel a ľAnalyse, Et Son Application a ľElimination Des Coupures Dans L'Analyse Et La Theorie Des Types. In Proceedings of the Second Scandinavian Logic Symposium, J.E. Fenstad (Ed.), Studies in Logic and the Foundations of Mathematics, Vol. 63. Elsevier, 63–92. https://doi.org/10.1016/S0049-237X(08)70843-7

Jean-Yves Girard. 1972. Interpretation fonctionnelle et elimination des coupures de l’arithmetique d’ordre superieur. Ph.D. Dissertation. Universite Paris.

Radu Grigore. 2017. Java generics are Turing complete. SIGPLAN Not. 52, 1 (Jan. 2017), 73–85. https://doi.org/10.1145/3093333.3099871

Irène Guessarian. 1983. Pushdown tree automata. Math. Syst. Theory 16, 1 (1983), 237–263.

R. Hindley. 1969. The principal type-scheme of an object in combinatory logic. Trans. Amer. Math. Soc. 146 (Dec. 1969), 29–60. http://www.jstor.org/stable/1995158

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. Introduction to automata theory, languages, and computation (3rd ed.). Pearson Addison Wesley, Boston, MA.

Richard M. Karp. 1972. Reducibility among combinatorial problems. In Proc. Symp. Complex. Comp., Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger (Eds.). Springer, Yorktown Heights, NY, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_proceedings-98

A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. 1990. ML typability is DEXPTIME-complete. In CAAP ’90, A. Arnold (Ed.). Springer, New York, 206–220.

Donald E. Knuth. 1965. On the translation of languages from left to right. Info & Comp. 8, 3 (1965), 607–639. https://doi.org/10.1016/0019-9958(65)90426-2

Robin Milner. 1978. A theory of type polymorphism in programming. J. Comput. System Sci. 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Tomoki Nakamaru and Shigeru Chiba. 2020. Generating a generic fluent API in Java. The Art, Science, and Eng. of Prog. 4, 3 (Feb. 2020). https://doi.org/10.22152/programming-journal.org/2020/4/9

Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. 2017. Silverchain: a fluent API generator. In Proc. 16th ACM SIGPLANInt. Conf. Generative Prog. (GPCE’17). ACM, Vancouver, BC, Canada, 199–211.

Anton Nijholt. 1979. Grammar functors and covers: From non-left-recursive to Greibach normal form grammars. BIT Numerical Mathematics 19, 1 (01 March 1979), 73–78. https://doi.org/10.1007/BF01931223

Guido Persch, Georg Winterstein, Manfred Daussmann, and Sophia Drossopoulou. 1980. Overloading in Preliminary Ada. SIGPLAN Not. 15, 11 (Nov. 1980), 47–56. https://doi.org/10.1145/947783.948640

Michael O. Rabin. 1963. Real time computation. Israel J. Math. 1 (1963), 203–211.

John C. Reynolds. 1974. Towards a theory of type structure. In Programming Symposium, B. Robinet (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 408–425.

A. Salomaa and M. Soittola. 1978. Automata-Theoretic Aspects of Formal Power Series. Springer-Verlag, NY.

J.B. Wells. 1999. Typability and type checking in System F are equivalent and undecidable. Annals of Pure and Applied Logic 98, 1 (1999), 111 – 156. https://doi.org/10.1016/S0168-0072(98)00047-5

Hao Xu. 2010. EriLex: an embedded domain specific language generator. In Objects, Models, Components, Patterns, Jan Vitek (Ed.). Springer, Berlin, Heidelberg, 192–212.

Tetsuro Yamazaki, Tomoki Nakamaru, Kazuhiro Ichikawa, and Shigeru Chiba. 2019. Generating a fluent API with syntax checking from an LR grammar. Proc. ACM Program. Lang. 3, Article Article 134 (Oct. 2019), 24 pages. https://doi.org/10.1145/3360560
A ABBREVIATIONS, ACRONYMS, AND NOTATION

Acronyms

G&R Gil and Roth [2019], page 3
P plain parametric polymorphism, or, polyadic parametric polymorphism, (3.2) and Fig. 3.2, page 3
API application programming interface, page 1
CFG context free grammar, page 20
CFL context free language, page 5
CSL context sensitive language, page 5
DCFL deterministic context free language, page 5
DEXP deterministic exponential, page 1
DFSA deterministic finite state automaton, page 5
FSA finite state automaton, page 1
GHC Glasgow Haskell Compiler, page 2
GNF Greibach normal form (of CFG), page 20
HM Hindley-Milner (type system), page 1
IDE interactive development environment, page 3
LBA linear bounded automaton, page 5
LMD left-most derivation, page 20
LR left-to-right, right-most derivation, page 2
MKS meter-kilogram-second (system of physical units, page 23
ML the ML (“meta-language”) programming language, page 3
PDA pushdown automaton, page 1
PDTA pushdown tree automaton, page 16
RDPDA real-time deterministic pushdown automaton, page 5
REG the set of regular languages, page 5
RTM real-time Turing machine, page 5
SRDPDA stateless real-time deterministic pushdown automaton, page 5
STLC simply typed lambda calculus, page 1
TA tree automaton, i.e., an automaton employing a tree store, page 5
UCFL unambiguous context free language, page 20

List of Symbols

1. Latin Letters Like (upper case)

\( \ell \) forest, or language of trees, \( \ell \subseteq \Sigma^\Delta \), page 9
\( A \) a finite control automaton, page 6
\( A \) a two-dimensional matrix, page 22
\( B \) a two-dimensional matrix, page 22
\( C_i \) a characteristic of lattice \( \mathfrak{X} \), see Table 3.1, page 10
\( C_1 \) number of type arguments (characteristic of lattice \( \mathfrak{X} \)), see Table 3.1, page 10
\( C_2 \) type pattern depth (characteristic of lattice \( \mathfrak{X} \)), see Table 3.1, page 10
\( C_3 \) type pattern multiplicity (characteristic of lattice \( \mathfrak{X} \)), see Table 3.1, page 10
\( C_4 \) arity of functions (characteristic of lattice \( \mathfrak{X} \)), see Table 3.1, page 10
\( C_5 \) type capturing (characteristic of lattice \( \mathfrak{X} \)), see Table 3.1, page 10
\( C_6 \) overloading (characteristic of lattice \( \mathfrak{X} \)), see Table 3.1, page 10
\( F \) the set of accepting states in a finite control automaton, page 6
2. **Latin Letters Like (lower case)**

- *a* an example letter in alphabet, page 2
- *b* an example letter in alphabet, page 2
- *c* an example letter in alphabet, page 16
- *e* expression (abstract syntax category), see Fig. 3.1, page 11
- *h* position of the read/write head on tape auxiliary storage, page 7
- *m* a dimension of a matrix, page 22
- *n* exponent of certain physical unit in a physical dimension such as kilogram/meter-squared, page 23
- *p* length of word input to finite control automaton, page 8
- *p* value of the lattice property *i*, page 5
- *q* a state of a finite control automaton, page 6
- *q* the initial internal state of a finite control automaton, page 6
- *r* number of physical units in a system of physical units such as MKS, page 23
- *r* rank/number of children in a node of a tree in $\Gamma^\Delta$, page 8
- *r* rank of symbol $\gamma$ drawn from a signature, page 8
- *s* tree substitution $\{x_1 \rightarrow t_1, \ldots, x_r \rightarrow t_r\}$, page 8
- *u* a tree in $\Gamma^\Delta$, page 8
- *t* grounded type (abstract syntax category), see Fig. 3.1, page 11
- *u* the word denoting the remainder of input to a language recognizer, page 6
- *w* the input word to language recognizer, page 6
- *x* type variable (abstract syntax category), see Fig. 3.2, page 12
- *x* variable used in a term, page 8
- *x* a physical unit such as centimeter, second, gram, and coulomb, page 23
- *t* formal language of strings, $t \subseteq \Sigma^*$, page 6
- *b* designated blank symbol occupying uninitialized cells of tape auxiliary storage, page 17
- *e* multi-expression (abstract syntax category), see Fig. 3.3, page 13
- *q* multi-state $q_1, q_2, \ldots, q_r$, r determined by context, page 9
- *t* multi-tree $t_1, \ldots, t_r$, r determined by context, page 10
3. **Greek Letters Like (upper case)**

- \( \Gamma \): alphabet of symbols used in auxiliary storage, page 6
- \( \Gamma^\Delta \): set of variables of CFG, page 20
- \( \Gamma^\triangle \): set of all trees over signature \( \Gamma \), page 8
- \( \Gamma^\Xi \): set of all terms over signature \( \Gamma \), page 8
- \( \Delta \): set of input-output items of consuming transition function \( \delta \), page 7
- \( \Delta \): set of primary function definitions (abstract syntax category), see Fig. 3.1, page 11
- \( \Xi \): set of auxiliary function definitions (abstract syntax category), see Fig. 3.4, page 14
- \( \Xi^\varepsilon \): set of input-output items of \( \varepsilon \)-transition function \( \xi \), page 7
- \( \Sigma \): finite alphabet of symbols, page 6
- \( \Sigma^* \): set of all strings (words) over \( \Sigma \), including the empty string, page 6
- \( \Phi \): set of auxiliary function names, disjoint to \( \Sigma \), see Fig. 3.4, page 14
- \( \Gamma \): set of possible contents of auxiliary storage, page 6

4. **Greek Letters Like (lower case)**

- \( \gamma \): variable (non-terminal) of CFG, page 20
- \( \delta \): a definition of primary function (abstract syntax category), see Fig. 3.1, page 11
- \( \delta \): the consuming transition function of a finite control automaton, page 7
- \( \iota \): an instantaneous description of a finite control automaton, see Def. 2.3, page 6
- \( \iota_0 \): initial instantaneous description of a finite control automaton, page 6
- \( \xi \): auxiliary function definition (abstract syntax category), see Fig. 3.4, page 14
- \( \xi \): the \( \varepsilon \)-transition function of a finite control automaton, page 7
- \( \rho \): derivation rule of CFG, page 20
- \( \rho \): tree rewrite rule, page 8
- \( \varsigma \): A terminal of a CFG, or the special symbol $, page 20
- \( \sigma \): a letter in alphabet \( \Sigma \), page 7
- \( \sigma \): class name (abstract syntax category), see Fig. 3.1, page 11
- \( \sigma \): name of primary function (abstract syntax category), see Fig. 3.1, page 11
- \( \sigma \): terminal of CFG, page 20
- \( \tau \): term in set \( \Gamma^\Delta \), page 8
- \( \tau \): type pattern, i.e., ungrounded type (abstract syntax category), see Fig. 3.2, page 12
- \( \omega \): sentential form, i.e., a sequence of terminals and variables of a CFG, \( \omega \in (\Sigma \cup \Gamma)^* \), page 20
- \( \varrho \): pseudo expression, an expression whose type is ungrounded (abstract syntax category), see Fig. 3.4, page 14
- \( \phi \): auxiliary function name, drawn from set \( \Phi \) (abstract syntax category), see Fig. 3.4, page 14
- \( \gamma \): a string of symbols drawn from alphabet \( \Gamma \), page 32
- \( \gamma \): entire contents of auxiliary storage, page 6
- \( \gamma \): sequence of CFG variables, \( \gamma \in \Gamma^\ast \), page 20
- \( \gamma_0 \): initial contents of auxiliary storage, page 6
- \( \tau \): multi-term, \( \tau_1, \ldots, \tau_r \), \( r \) determined by context, page 10
- \( \tau \): multi-type pattern, i.e., multi ungrounded type (abstract syntax category), see Fig. 3.2, page 12
- \( \varepsilon \): degenerate tree, also denoting a leaf in any tree in \( \Gamma^\Delta \), page 8
- \( \varepsilon \): designated stack symbol denoting the bottom of the stack, page 32
- \( \varepsilon \): start symbol of CFG, page 20

---

x multi-variable (abstract syntax category), see Fig. 3.2, page 12
5. Other

Depth($t$) depth of tree $t \in \Gamma^\Delta$, $\text{Depth}(\varepsilon) = 0$, page 8

Depth($\rho$) depth of pattern $\rho \in \Gamma^\Delta$, page 9

Depth($\tau$) depth of term $t \in \Gamma^\Delta$, $\text{Depth}(x) = 0$, page 9

Vars($\rho$) set of variables in rewrite $\rho$, page 8

Vars($\tau$) set of variables in term $\tau$, page 8

⊥ the error type (terminal of abstract syntax), see Fig. 3.1, page 11

Fling a fluent API generator contributed by G&R, page 3

Fluent intermediate language used in the implementation of TypelevelLR, page 2

TypelevelLR a fluent API generator due to Yamazaki, Nakamaru, Ichikawa and Chiba [2019], page 2
B  FLUENT API: FROM PRACTICE TO THEORY

An application programming interface (API) provides the means to interact with an application via a computer program. For example, using a file system API we can open, read, and close files from within C code:

```c
open(); // Open file
read(); // Read line
read(); // Read another line
close(); // Close file
```

Accompanied to an API is a protocol of use, defining rules for good API practice. A protocol is usually brought in internal and external documentation, delegating its imposition to the programmer. For instance, a typical file system API protocol disallows `read()` to be called before `open()`, and `close()` to be called twice in a row. Although breaking the protocol may result in malicious run time behaviors, it nonetheless yields coherent, runnable programs.

With object oriented programming (OOP)\textsuperscript{12}, functions (methods) are defined within classes. To invoke a method, it must be sent as a message to an object of the defining class. Methods of an OO fluent API yield objects that accept other API methods:

Listing B.1 Fluent file system API implemented in Java

```java
class ClosedFile {
  OpenedFile open() {...}
}

class OpenedFile {
  OpenedFile read() {...}
  ClosedFile close() {...}
}
```

In this OO file system API there are two classes, `ClosedFile` and `OpenedFile`. Every API call returns either an object of class `ClosedFile` or an object of class `OpenedFile`, and thus may immediately be followed by a successive API call:

Listing B.2 Chain of fluent API method calls

```java
closedFile.open().read().read().close();
```

This expression conducts multiple API calls: Invoking `open` on a `ClosedFile` object yields an `OpenedFile` object. Calling `read` on the `OpenedFile` yields itself, but a `close` invocation returns a `ClosedFile`.

The main advantage of fluent APIs is their ability to enforce a protocol at compile time: The object returned from API call \( \sigma_i() \) is missing method \( f() \), if calling \( f \) at that location (\( \sigma_{i+1} \leftarrow f \)) breaks the protocol. Consider, for instance, finishing the methods chain of List. B.2 with a second `close` call, therefore breaking the file system protocol which forbids double closing: This call fails at compile time, raising a compilation error, as the first `close` call returns a `ClosedFile` object, defined in List. B.1, which lacks a `close` method.

Fluent APIs grew in fame due to their application for domain specific languages (DSLs). In contrast to general purpose programming languages, as Java and C++, DSLs employ syntax and semantics designed for a specific component. Standard query language (SQL), for example, is a DSL for writing database queries. To make use of an application in a general software library, its DSL has to be substituted for an API. Making the API fluent is then ideal: it makes it possible to embed DSL programs in code as chains of method calls, that preserve and enforce the original syntax of the DSL. Additional details on DSLs and fluent APIs may be found in [Gil and Roth 2019].

\textsuperscript{12}Strictly speaking, we need only “object based” programming, which admits classes and objects, but no class inheritance.
A protocol or a DSL may be described by a formal language $\ell$: Then, the fluent API problem is to compile $\ell$ into a fluent API that enforces the protocol. The fluent API problem is parameterized by the complexity of the input language, and the capabilities of the host type system. The file system protocol, for instance, is described by a regular expression,

$$(\text{open} \cdot \text{read}^* \cdot \text{close})^*,$$

and therefore defines a regular language. Given a class of formal languages $L$, we seek a minimal set of type system features required to embed $L$ languages.

As many programming languages and DSLs are not regular, practical interest lies with stronger language classes. A popular approach is to use parametric polymorphism, yet another common OOP feature. A fixed number of polymorphic classes define an infinite number of types ($A, A<A>, A<A<A>>, \ldots$): Intuitively, these types can be used to simulate an unbounded storage, required to accept non-regular languages.

Consider, for example, the following Java definitions: With these definitions, an expression of the form

$$e = \text{new Empty()}.\sigma_1().\sigma_2() \ldots .\sigma_n().\text{empty()}, \quad (B.1)$$

where $\sigma_i \in \{\text{push, pop}\}$ type checks if and only if, $\sigma_1\sigma_2 \ldots \sigma_n$ belongs in the Dyck language of balanced parentheses with the homomorphism

$$h(\sigma) = \begin{cases} \text{push} & \sigma = '(' \\ \text{pop} & \sigma = ')' \end{cases}$$

A pop from empty stack (conversely, unbalanced parenthesis) is signaled by a type error generated at compile time, e.g., in

```java
new Empty().push().pop().pop().empty();
```

the second call to pop() triggers a compile time error, to say that type Empty does not feature this method.

With the fluent API problem trivial for regular languages, recent studies [Gil and Levy 2016; Gil and Roth 2019; Nakamaru and Chiba 2020; Nakamaru et al. 2017; Xu 2010; Yamazaki et al. 2019] introduced various methods for composing fluent APIs of more complex languages. Two promising results are those of G&R and Yamazaki et al. [2019]: Released roughly at the same time, both papers showed any deterministic context free languages (including the Dyck language) can be composed into a fluent API.

---

13Java generics, C++templates, etc.
14A finite state machine can be encoded using simple OO classes. A Java fluent API generator for regular languages is available at https://github.com/verhas/fluflu.
C PROOFS

C.1 Proof of Thm. 4.2

Recall that $a^n b^n c^n \in \text{CSL}$, and that DCFL $\subset \text{CSL}$. We show that $a^n b^n c^n \in \text{deep-$\mathcal{P}_k$}$. The details are in List. C.1, that employs Java syntax to show a set of definitions that recognizes the language $a^n b^n c^n$.

Listing C.1 Definitions in type system \textit{deep-$\mathcal{P}_k$} (using Java syntax) for the language $a^n b^n c^n$

```java
interface y1<x1, x2> {} // Type after reading $a$ is $y1<\text{Zero}, \text{Zero}>$
interface y2<x1, x2> {} // Type after reading $a^2$ is $y2<\text{Succ}<\text{Zero}>, \text{Zero}>$
interface y3<x2> {} // Type after reading $a^3$ is $y3<\text{Succ}<\text{Succ}<\text{Succ}<\text{Zero}>>, \text{Zero}>$
static y1<Zero, Zero> begin() { return null; } // chain start
static <x1, x2> y2<x1, x2> y1<Succ<x1>, Succ<x2>> a(y1<x1, x2> e) { return null; } // Increment both arguments
static <x1, x2> y2<x1, x2> b(y1<Succ<x1>>, x2> e) { return null; } // b after $a^2$; decrement first argument
static <x1, x2> y2<x1, x2> b(y1<Succ<x1>>, x2> e) { return null; } // b after $a^2$, k > 0; decrement first argument
static <x> y3<x> c(y3<Zero, Succ<x>>, x> e) { return null; } // c after $a^3$; decrement second argument
static <x> y3<x> c(y3<Succ<x>>, x> e) { return null; } // c after $a^3$, $k > 0$; decrement argument
static void end(y3<Zero> e) {} // Accept after $a^3 e^2$, $k = n$
static {} // Test definitions in static initializer
end(c(c(b(b(b(a(a(a(begin())))))))))); // Expression $\gamma_1(a(a(begin())))$ type-checks
end(c(c(b(b(b(a(a(begin())))))))); // Expression $\gamma_1(a(a(begin())))$ does not type-check
```

The three generic types $y_1$, $y_2$ and $y_3$ rely on the unary encoding and increment and decrement operations for maintaining counts of letters $a$, $b$, and $c$ (calls to functions $a()$, $b()$ and $c()$) in the input string:

1. The type of expression
   \[ a(\cdots a(begin()) \cdots) \]
   ($k$ occurrences of $a$) is $y_1<\text{Zero}, \text{Zero}>$, where $u_k$ is the type encoding of $k$;

2. The type of
   \[ b(\cdots b(a(\cdots a(begin()) \cdots)) \cdots) \]
   ($n$ occurrences of $a$, $k$ of $b$) is $y_2<\text{Succ}<\text{Zero}, \text{Zero}>, \text{Zero}>$; and

3. The type of expression
   \[ c(\cdots c(b(\cdots b(a(\cdots a(begin()) \cdots)) \cdots)) \cdots) \]
   ($n$ occurrences of $a$ and $b$; $k$ occurrences of of $c$) is $y_3<\text{Succ}<\text{Succ}<\text{Succ}<\text{Succ}<\text{Succ}<\text{Zero}>>, \text{Zero}>, \text{Zero}>$.

For example, observe the (overloaded) definition of function $b(\cdot)$ in the listing,

```java
static <x1, x2> y2<x1, x2> b(y1<Succ<x1>>, x2> e) { return null; }
```

This version of $b(\cdot)$, intended for expressions of the form

\[ b(a(\cdots a(begin()) \cdots)) \]

converts $y_1<\text{Zero}, \text{Zero}>$, the type of its argument to $y_2<\text{Zero}, \text{Zero}>, \text{Zero}>$.

Consider the general case expression

```java
end(c(\cdots c(b(\cdots b(a(\cdots a(begin()) \cdots)) \cdots)) \cdots))
```

and, starting at the inner most invocation, $\text{begin()}$, whose type is $y_1<\text{Zero}, \text{Zero}>$, and tracing, bottom up, types of the successive nested expressions, we see that:

- First, a count of the $a$’s is recorded in both arguments of generic $y_1$. This count is incremented with each call to $a()$.
- Once the first $b$ is seen, these arguments are passed to generic $y_2$. The first argument of $y_2$ is decremented with each $b$ encountered. The second argument remains however unchanged during these encounters.
- This second argument is then passed to generic $y_3$ when the first $c$ is encountered. It is then decremented for each $c$ encountered.
- Function $\text{end}$ type-checks only if this argument is $\text{Zero}$. 


C.2 Proof of Thm. 4.3

The Java definitions in List. C.2 realize the language \( a^n b^n c^n \in \text{CSL} \).

Listing C.2 Definitions in type system non-linear-\( \text{P} \) (using Java syntax) for the language \( a^n b^n c^n \)

```java
interface y1<x1, x2, x3> { // Type after reading \( a^1 \) is \( y1<\text{Zero,Zero,Zero}> \)
    y1<Succ<x1>, x2, x3> a(); // No phase change: increment the first type argument
    y1<x1, Succ<x2>, x3> b(); // First \( \gamma \) seen: change phase, and increment second argument
}
static {
    interface y2<x1, x2, x3> { // Type after reading \( a^2 \) is \( y2<\text{Zero,Zero,Zero}> \)
        y2<Succ<x1>, x2, x3> b(); // No phase change: increment the second type argument
        y2<x1, Succ<x2>, x3> c(); // First \( \gamma \) seen: change phase, and increment third argument
    }
    static <x> void end(x1, x2, x3> c(); // No phase change: decrement the third type argument
    y2<x1, x2, Succ<x3>> c(); // No phase change: decrement the first type argument
}
interface y3<x1, x2, x3> { // Type after reading \( a^3 \) is \( y3<\text{Zero,Zero,Zero}> \)
    y3<x1, x2, Succ<x3>> c(); // No phase change: decrement the third type argument
}
static {
    interface y1<Zero, Zero, Zero> begin() { return null; } // Start with type \( y1<\text{Zero,Zero,Zero}> \)
    static <> void end(y3<x1, x2, x3> e) { } // Accept only on type \( y3<\text{Zero,Zero,Zero}> \) for some \( n \geq 0 \)
    static { // Test definitions in static initializer
        end(begin().a().a().b().b().b().c().c().c()); // Expression \( e \cdot e \cdot e \cdot c \cdot c \cdot c \) type-checks
        end(begin().a().a().a().b().b().b().c().c().c()); // Expression \( e \cdot e \cdot e \cdot c \cdot c \cdot c \) does not type-check
    }
}
```

The fluent API records the number of \( a \)'s, \( b \)'s and \( c \)'s in three unary integer encodings. The recording is in generic types \( y_1, y_2 \) and \( y_3 \) (each taking three type parameters). As before, type \( y_1 \) is dedicated to the first phase in which the \( a \)'s are encountered, type \( y_2 \) is to the second phase in which the \( b \)'s occur, and type \( y_3 \) to the final phase in which the \( c \)'s show.

When the entire input is read, the three counters are compared by function \( \text{end()} \). This function relies on non-linearity, to check that they are indeed equal.

C.3 Proof of Thm. 5.3

Given is a fluent program \( P = \Delta \Xi e \). We construct from the definitions \( \Delta \) and \( \Xi \) deep-DPDA automaton \( A \). Let \( e = e.\sigma_1.\cdots.\sigma_n \). Then, \( A \) accepts \( w = \sigma_1 \cdots \sigma_n \) if and only if \( P \) is type-correct.

The construction maintains the invariant that after \( A \) consumes \( \sigma_i \) and conducting all (any) subsequent \( \varepsilon \)-transitions, its stack contents encodes \( t_i \), the type of the partial expression \( e = e.\sigma_1.\cdots.\sigma_i \). Concretely, since \( \text{Fluent} \) is a monadic type system, \( t_i \) must be in the (full) form \( y_{i-1} \cdot y_i(e) \cdot \cdots \). The stack encoding of \( t_i \) is \( y_1 y_2 \cdots y_k e \), i.e., the monadic abbreviation of the full form augmented with a designated symbol \( e \) for denoting the stack’s bottom. For this reason, the set of stack symbols of \( A \) includes a symbol \( y \) for every type name used in \( \Delta \cup \Xi \), and the extra symbol \( e \).

The set of internal states of \( A \) includes an initial and accepting state \( q_0 \). The automaton will be in state \( q_0 \) initially, and then whenever it exhausted all possible \( \varepsilon \)-transitions after consuming a letter, and is ready to consume the next input symbol. Also, \( A \) has an internal (not-accepting) state \( q_{\phi} \) for every auxiliary function name \( \phi \) used in \( \Xi \). These states are used while executing \( \varepsilon \)-transitions, which emulate the resolution of the rudimentary typeof clauses allowed in \( \text{Fluent} \).

As in the proof of Thm. 5.2, the rudimentary-typeof property of the type systems makes it possible to classify any function definition in \( \Delta \cup \Xi \) as either direct, if its type signature is \( \tau \rightarrow \tau' \), or as forwarding, in case it is \( \tau \rightarrow \text{typeof} \ \tau' . \phi \).

Every \( \text{Fluent} \) function is encoded in one (consuming- or \( \varepsilon \)-) transition item of \( A \). In this encoding, the function type signature uniquely determines the stack rewrite rule \( \rho \), but unlike in the proof of Thm. 5.2, \( \rho \) is not identical to the type signature.

To see why, recall first that since \( \text{Fluent} \) is monadic, we can write any term \( \tau \) as \( yx \) where \( y \in \Gamma^* \) (in the case \( \tau \) is a proper term) or as \( y \) (in the case it is a grounded). If a function’s type is \( yx \rightarrow y' \), then to maintain the invariant, \( A \) needs to push the string \( y'x' \) to stack after emptying it, by popping first the \( y \) fixed portion, and then the \( x \) variable portion which may of unbounded length. Alas, this \( x \) portion cannot be cleared with the single stack rewrite allowed in the single transition encoding a \( \text{Fluent} \) function.
For this reason, we use instead a stack rewrite $\rho = \gamma x \rightarrow \gamma' ex$ in this case, i.e., emulating stack emptying by pushing another copy of $e$, the bottom of the stack symbol. Automaton $A$ is oblivious to the trick, since none of the rewrites in its transitions of removes a $e$ symbol off the stack.

With the definition of $\rho(\tau \rightarrow \tau')$ by

$$\rho(\tau \rightarrow \tau') = \begin{cases} \gamma x \rightarrow \gamma' x & \text{if } \tau = \gamma x \text{ and } \tau' = \gamma' x \\ \gamma e \rightarrow \gamma' e & \text{if } \tau = \gamma y \text{ and } \tau' = \gamma' y \\ \gamma x \rightarrow \gamma' ex & \text{if } \tau = \gamma x \text{ and } \tau' = \gamma' y \\ \end{cases} \quad (C.1)$$

we can describe the transition encoding of each of the four kinds of functions that may occur in $P$.

1. **Primary function definitions**, found in $\Delta$, are encoded as consuming transitions of $A$:
   a. **Direct definition** $\sigma : \tau \rightarrow \tau'$ as $(\sigma, q_0, \rho(\tau \rightarrow \tau'), q_0)$,
   b. **Forwarding definition** $\sigma : \tau \rightarrow \text{typeof } \tau'.\phi$ as $(\sigma, q_0, \rho(\tau \rightarrow \tau'), q_\phi)$.

2. **Auxiliary function definitions**, found in $\Xi$, are encoded as $e$ transitions of $A$:
   a. **Direct definition** $\phi : \tau \rightarrow \tau'$ as $(q_\phi, \rho(\tau \rightarrow \tau'), q_0)$.
   b. **Forwarding definition** $\phi : \tau \rightarrow \text{typeof } \tau'.\phi'$ as $(q_\phi, \rho(\tau \rightarrow \tau'), q_\phi')$.

We can now verify that automaton $A$ iteratively computes the type of the word-encoded input expression: Consuming transitions correspond to type checking of primary function invocation, while $e$-transitions make the detour required to compute the type of functions defined by a `typeof` clause. If the input expression fails type checking, then automaton $A$ hangs (whereby rejecting the input), due to failure to find an appropriate transition for the current stack contents, internal state (and the current input symbol, when appropriate).

### C.4 Proof of Thm. 5.4

We present a set of *full-typeof-Fluent* definitions that encodes the language $w#w \in \text{CSL}$.

---

Listing C.3 C++, *full-typeof-Fluent* program recognizing the CSL $w#w$

```cpp
struct E {} // Bottom type
template<type A> struct A {} // Generic type, stands for a
template<type B> struct B {} // Generic type, stands for b
template<type S> struct S {} // Generic type, stands for #
at E* a() {} // Begin expression with a
B* E* b() {} // Begin expression with b

template<typename T> struct S<T> {} // Generic type, stands for T

B* reverse(E* E) { return reverse(b(reverse(T()))); } // Expression has ended, eventually match a
B* reverse(B* B) { return reverse(b(reverse(T()))); } // Eventually match b
B* S(0) { return reverse(S(0)); } // # encountered, reverse the second w
B* A(T) { return match_a(S(T())); } // Expression encoding $a$ # $a$
B* S(T) { return match_b(S(T())); } // Expression encoding $b$ # $b$
B* A(0) {} // Correct $a$ # $b$

E reverse(E* E) { return reverse2start_a(reverse2end_a(A(T()))); } // Reattach a
B* reverse2end_a(A(T)) { return append2start_a(reverse2end_a(A(T()))); } // Reattach b
B* reverse2end_a(A(0)) { return append2start_a(reverse2end_a(A(0))); } // Reattach b
B* append2end_a(A(0)) {} // Append a to the end
B* append2end_a(A(T)) { return append2start_a((append2end_a(A(T()))); } // Reattach a
B* append2end_a(A(0)) { return append2start_a((append2end_a(A(0))); } // Reattach b
B* append2end_a(A(T)) { return append2start_a((append2end_a(A(T()))); } // Reattach a
B* append2end_a(A(0)) { return append2start_a((append2end_a(A(0))); } // Reattach b
B* append2end_a(A(T)) { return append2start_a((append2end_a(A(T()))); } // Reattach a
B* append2end_a(A(0)) { return append2start_a((append2end_a(A(0))); } // Reattach b

int main() {
    E w1=S(a(b(a(s(a(b(a(a(a())))))))))); // Expression encoding $w = aba#aba$ type-checks
    E w2=S(a(b(a(s(a(b(a(b(a(b(a(a())))))))))))); // Expression encoding $w = aba#aba$ does not type-check
    E w3=S(b(a(s(a(b(a(b(a(b(a(a())))))))))))); // Expression encoding $w = baa#baa$ does not type-check
```
The definitions first accumulate the input to a monadic type, e.g., where expression \(a(b(s(a(b()))))\) is typed as \(A<B<S<A<B<E>>\), type \(E\) is the bottom type (C++ identifiers \(s\) and \(S\) stand for \#). Actual computation is done only when reaching method \$, which terminate all expressions.

Function \$ first traverses the first \(w\) of \(w\#w\), while replacing types \(A\) and \(B\) with calls to \(\text{match}_A\) and \(\text{match}_B\) respectively. Upon reaching type \(S\), encoding \#, function \$ encodes the second \(w\) as a type, and reverses it; then functions \(\text{match}_A\) and \(\text{match}_B\) proceed to match the words in the correct order. For example, expression \$\(a(b(s(a(b()\cdots))\) changes first into \(\text{match}_A(\text{match}_B(\$\(s(a(b()\cdots)), and then into \(\text{match}_A(\text{match}_B(B<A<E>>));\) next the \(\text{match}\) functions match \(b\) and then \(a\), and return the bottom type \(E\), successfully terminating the typing process. If the word before the \# differs from the word after it, this matching is ought to fail (if typing has not yet failed).

Rest of the code deals with implementing function \(\text{reverse}\). Function \(\text{reverse}\) appends the current type \(A\) (resp. \(B\)) to the end of the type, recursively, using function \(\text{append2end}_A\) (\(\text{append2end}_B\)). Function \(\text{append2end}_A\) examines its argument \(A<T>\) (\(B<T>\)), replaces it with a call to \(\text{append2start}_A\) (\(\text{append2start}_B\)) and continue recursively into \(T\); type \(A\) (\(B\)) is reattached after the process has ended. Function \(\text{append2end}_B\) is implemented in a similar way.
D SUPPLEMENTARY MATERIAL

D.1 Full Encoding of a Turing Machine in \(\langle\text{deep, rudimentary}\rangle \cap \mathcal{P}_0\)

Thm. 5.1 above showed that any Turing machine can be encoded by a program in type system

\[ T = \langle\text{deep, rudimentary}\rangle \cap \mathcal{P}_0 \]

The proof of theorem used Fig. 5.1 depicting an example of such a machine. For the sake of completeness, List. D.1 here presents the full encoding in \(T\) of the Turing machine of Fig. 5.1.

Listing D.1 C++ program encoding the Turing Machine of Fig. 5.1

D.2 Fluent API for the Language of Palindromes

Here we demonstrate Thm. 6.1 and its proof, by constructing a fluent API library for palindromes in an Ada like type system, i.e., a type system with eventually-one-type style of overloading resolutions.
Consider the formal language of even length palindromes over alphabet \{a,b\}, as defined by the following context free grammar

\[
\begin{align*}
\varepsilon & \rightarrow a\varepsilon a \\
& \quad \rightarrow b\varepsilon b \\
& \quad \rightarrow \varepsilon.
\end{align*}
\]  

(D.1)

It is well known that the language (D.1) is not-deterministic yet unambiguous. Rewriting its grammar in Greibach normal form gives

\[
\begin{align*}
\varepsilon & \rightarrow ay_1 \\
& \quad \rightarrow by_2 \\
y_1 & \rightarrow ay_1y_3 \\
& \quad \rightarrow by_2y_3 \\
& \quad \rightarrow a \\
y_2 & \rightarrow ay_1y_4 \\
& \quad \rightarrow by_2y_4 \\
& \quad \rightarrow b \\
y_3 & \rightarrow a \\
y_4 & \rightarrow b.
\end{align*}
\]  

(D.2)

Applying the construction in the proof of Thm. 6.1 to the grammar (D.2) gives the program in List. D.2, that realizes a fluent API for (D.1).

**Listing D.2** Definitions in type system \langle monadic, eventually-one-type \rangle (using Java-like syntax) encoding the language of even length palindromes

```java
interface \varepsilon {
    \gamma_1<> a();
    \gamma_2<> b();
}

interface \gamma_1<T> {
    \gamma_1<>\gamma_3<T> a();
    \gamma_2<>\gamma_3<T> b();
    T a(); // Java error, overloaded functions cannot differ only by return type
}

interface \gamma_2<T> {
    \gamma_1<>\gamma_4<T> a();
    \gamma_2<>\gamma_4<T> b();
    T b(); // Java error, overloaded functions cannot differ only by return type
}

interface \gamma_3<T> {
    T a();
}

interface \gamma_4<T> {
    T b();
}

interface \$ {
    void $();
}

new \varepsilon().a().a().b().b().a().a().$.();
```

Note that even though the program in the listing uses Java syntax, it would not provide the desired result if compiled by a Java compiler. The reason is that Java does not permit multiple types for sub-expressions.

Expression `new \varepsilon().a().a().b().b().a().a().$.();` in List. D.2 is phrased as `aabbaa`—with this prefix, the center of the word (denoted by `·`), separating \(w\) from \(w^R\), can be in three places: `aab-baa`, in case \(w = aab\), `aabba-a`, in case \(w = aabba\), or `aabbaa-r`, in case \(w = aabbaar\). These three
Ties between Type Systems and Automata

possibilities correspond to three types deduced for the expression. Yet, when reaching method $()$, the type checker settles the ambiguity to the favor of the first option, as only after reading $ww^R$ type $\$ with method $()$ is returned. As there is exactly one way to type the entire expression, type checking is successful.

D.3 On the Complexity of Deep Polyadic Parametric Polymorphism

We take particular interest in type system deep-$\Pi$, since it models generic non-method functions. Also, this type system might be applicable for the software engineering applications mentioned in Sect. 7.

We don’t know the exact complexity class of deep-$\Pi$, but here are few comments and observations that might be useful towards characterizing it.

1. A tree automaton with $\epsilon$-transitions is even more potent than a two-pushdown automaton, which is equivalent to a Turing machine. This equivalence does not hold for the tree automaton in point, which is real-time.

2. A direct comparison of our real-time (and hence linear time) tree automata to real-time (or linear time) Turing machines is not possible, since an elementary operation of tree automata may involve transformations of trees whose size may be exponential.

3. We can still describe an emulation of the computation of real-time Turing machine (RTM, see Table 2.1 above) by a deep tree automaton, by breaking the machine’s tape into two stacks, and store these stacks as branches of the same tree, RTM $\subseteq$ deep-TA. Let RTM$_n$ be an RTM equipped with $n \geq 0$ linear bounded tapes. A classical result of Rabin [1963] separates the class RTM = RTM$_1$ from RTM$_2$, showing $|RTM|_1 \subseteq RTM_2$. Subsequently, Bruda and Akl [1999] generalized Rabin’s result for any number of tapes, showing that RTM$_n \subseteq RTM_{n+1}$, for all $n \geq 1$. Extending the tree automaton emulation of RTMs, to run concurrently on any (fixed) number of tapes, we obtain that the entire non-collapsing hierarchy of RTMs is contained in deep-TA, i.e, that RTM$_n \subseteq deep$-TA for all $n \geq 1$.

4. It does not seem likely that a linear time tree automata can recognize an arbitrary context sensitive language, a problem which is known to be PSPACE complete [Karp 1972]. We conjecture that deep-TA $\subset$ CSL.

5. A hint to the complexity of class deep-$\Pi$ may be found in the fact that it is closed under finite intersection and finite union. (The proof is by merging the respective tree automata by running their rewrites in tandem on two distinct branches of the same tree. The merged automata recognizes intersection if there is an accept both branches; it recognizes the union, if there is an accept in one of the branches.)

6. On the other hand, we claim that deep-TA is not closed under complement (equivalently, set difference): Consider (yet again) the language $a^n b^n c^n \in deep$-TA. If there was an automaton that recognizes the complement of the language, it should accept the word $abca$, but reject its prefix $abc$. Alas, a stateless automaton such as ours, can only reject by reaching a configuration where there are no further legal transitions, and hence cannot recover from the rejection of this prefix.

We are however able to show that stateful-TA is closed under complement.

7. Coquidé et al. [1994] discuss tree automata models similar to ours, and show that some restrictions on pattern depths and signature ranks are interchangeable.