Fluent APIs in Functional Languages

ORI ROTH and YOSSI GIL, Technion, Israel

Fluent API is an object-oriented pattern for elegant APIs and embedded DSLs. A smart fluent API can enforce the API protocol or DSL syntax at compile time. Since fluent API implementations typically rely on overloading function names, they are hard to realize in functional programming languages. This work shows how functional fluent APIs can be implemented in the absence of name overloading, by relying on parametric polymorphism and Hindley–Milner type inference. The implementation supports fluent API protocols in the regular- and deterministic context-free language classes, and even beyond.

CCS Concepts: • Software and its engineering → API languages; Functional languages.

Additional Key Words and Phrases: fluent API, API protocols, embedded DSLs

ACM Reference Format:
Ori Roth and Yossi Gil. 2023. Fluent APIs in Functional Languages. Proc. ACM Program. Lang. 7, OOPSLA1, Article 105 (April 2023), 26 pages. https://doi.org/10.1145/3586057

1 INTRODUCTION

An API protocol dictates a set of rules for correct API usage. Breaking these rules may result in runtime errors and undefined or unexpected behavior. The Java FileReader API protocol, for example, states that

“once the stream has been closed, further [API] invocations will throw an IOException.”

Although protocols usually come in written form as part of the API documentation, many protocols can be formalized as grammars, or more generally, a specification of a formal language. Such a specification defines the language (set) of legal sequences of API invocations. For example, the following regular expression describes the FileReader protocol:

\((\text{initialize}) (\text{read} | \text{skip} | \text{reset})^\ast \text{close}^\ast\)

Specifying API protocols using grammars or automata reduces the problem of protocol verification to formal language recognition.

APIs also use grammars and automata to specify domain-specific languages (DSLs). A DSL is a specialized language tailored for the API’s problem domain. Many applications use DSLs: A Python machine learning model uses SQL to load its dataset, a JavaScript frontend verifies textual forms with regular expressions, and a Standard ML (SML) HTTP server serves HTML webpages. To integrate a DSL into an application, we can implement the DSL as an API in the host programming language. The protocol of this embedded DSL (EDSL) is the syntax of the original DSL, i.e., EDSL invocations must describe well-formed DSL programs.

This paper presents methods to create smart and elegant APIs in functional programming languages. These APIs enforce their protocols or DSL syntax at compile time by performing

See a full version of this article [Roth and Gil 2022] and the accompanying artifact [Roth 2023].

1 https://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html#close--

Authors’ address: Ori Roth, soriroth@cs.technion.ac.il; Yossi Gil, yogi@cs.technion.ac.il, Technion, Haifa, Israel.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2023 Copyright held by the owner/author(s).

2475-1421/2023/4-ART105

https://doi.org/10.1145/3586057
language recognition at the type level. For example, Listing 1 demonstrates how we embed an HTML document as an SML expression using our methodology.

```
val webpage = ^^
<html>
  <body>
    <h1> "National Parks" </h1>
    "California:"
    <table>
      <tr>
        <th> "Park Description" </th>
        <th> "Park Picture" </th>
      </tr>
      <tr>
        <td> <p> <b> "Yosemite" </b> national park </p> </td>
        <td> <img src "https://tinyurl.com/yosemite5"/> </td>
      </tr>
    </table>
  </body>
</html>
```

Listing 1. An HTML webpage embedded in SML

Our HTML EDSL is *elegant* in the sense that embedded webpages closely resemble HTML, with minimal syntactic overhead imposed by the host programming language. The EDSL is *smart* as it coerces the SML compiler into enforcing HTML’s syntax, so invalid webpages do not compile. The tags must appear in the correct order (<html> before <body>, <tr> inside <table>, and so on), every opening tag must have a matching closing tag, and all table rows must have the same number of columns. Note that the additional requirement on table rows elevates HTML into a context-sensitive language.

Our API designs are based on *fluent API*, an object-oriented pattern for embedding DSLs. Fluent API techniques typically rely on function overloading, a common object-oriented feature not supported by many functional type systems. We adapt fluent APIs to functional settings by replacing overloading with parametric polymorphism and type inference. Our fluent APIs do not rely on a specific language feature, so they are compatible with most statically-typed functional languages.

### 1.1 Related Work

*Typestates* [Strom and Yemini 1986] serve a similar purpose to fluent APIs: A typestate is nothing but the set of operations permitted on a type in a given program context. Thus, a typestate dictates a protocol by forbidding some type correct operations, depending on the execution context. For example, the FileReader typestate forbids invoking read after close is called. Typestates are realized sometimes as a first-class language abstraction [Aldrich et al. 2009; Degen et al. 2007; DeLine and Fähndrich 2001; Garcia et al. 2014; Kuncak et al. 2002; Sunshine et al. 2011]. Another approach is to infer and verify typestates and API protocols statically [Bodden and Hendren 2012; Field et al. 2003; Fink et al. 2008; Pradel et al. 2012]. Ferles et al. [2021] developed a static analysis method for enforcing API protocols specified by context-free grammars. Our work is similar to that on typestates, except that we tacitly assume that the set of permitted operations in a given context is specified by a formal language. It is different in its modest requirements: Fluent APIs do not rely on abstractions added to the programming language, nor on external tools, but rather on correct
employment of host language’s type system and accompanying compiler for the enforcement of protocols.

Language workbenches [Erdweg et al. 2013; Fowler 2005b] such as Xtext [Eysholdt and Behrens 2010], MPS [Voeller and Pech 2012], and MetaEdit+ [Kelly et al. 1996] are development environments for the making of DSLs. The Spoofax language workbench, for instance, compiles DSL specifications into a full-fledged DSL IDE [Kats and Visser 2010]. SugarJ [Erdweg et al. 2011], Polyglot [Nystrom et al. 2003], and Racket [Felleisen et al. 2015] are examples of extensible programming languages in which the syntax of the language can be extended to incorporate DSLs. The fluent API approach is superior in that it requires no additional tools: Embedding DSLs as a fluent API makes it possible to extend the host programming language. It is inferior in that the embedded DSLs are not free of the host language built-in restrictions, e.g., it is impossible for an EDSL to depart from the tokenization and lexical analysis of the host language.

Programming languages with extensive compile-time metaprogramming capabilities offer a direct solution to the problem of DSL embedding [Czarnecki et al. 2004]. MetaML [Sheard 1999], MetaOCaml [Calcagno et al. 2003; Kiselyov 2014], and Template Haskell [Sheard and Jones 2002] extend existing programming languages with macro systems that enable DSL code generation at compile time. Evidently, metaprogramming in Haskell is expressive enough to embed DSLs even without templates. Haskell supports ad hoc polymorphism through type classes [Wadler and Blott 1989], shown to be useful for DSL embedding [Augustsson et al. 2008; Hudak 1998]. EDSLs often employ Haskell’s do-statement and monadic comprehensions [Brackner and Gill 2014; Gill 2014], especially in parser combinator libraries [Florijn 1995; Hutton and Meijer 1998; Leijen and Meijer 2000, 2001]. In contrast to these approaches, functional fluent APIs do not rely on any macro system, language extension, or special control structure. Some of the Haskell EDSL techniques are ad hoc, whereas our fluent API solutions support well-defined classes of DSLs, namely all regular and deterministic context-free languages.

It is essential to note that existing fluent API designs do not apply to functional languages, and in particular, those that rely on the Hindley-Milner type system. In this system, function name overloading is typically forbidden, lest type inference becomes intractable. Traditional fluent API implementations may be realized in functional languages with ad hoc polymorphism, e.g., as proposed by Bourdoncle and Merz [1997], Shields and Peyton Jones [2001], and Neubauer et al. [2002]: For example, Yamazaki et al. [2019] implemented fluent APIs in Haskell by relying on typeclasses and several other language extensions. Their work and other modern fluent API designs are discussed in Section 2. The present work is the first to discuss fluent APIs in a plain Hindley-Milner type system, and is the first to meet the grave challenge posed by the lack of overloading.

SML was chosen the main language used throughout the paper. We specifically ignore the language’s SML’s rich module system [MacQueen 1984], and limit use of the language to its core constructs [Harper et al. 1986]. To our knowledge, very few attempts were made in fluent API in SML. An exception is the work of Kamin and Hyatt [1997], who presented a specific fluent API-like EDSL for describing graphics. Kamin and Hyatt’s work is notable since it appeared eight years before the term was even coined [Fowler 2005a], but it did not spark followup. The present work is the first to meet the general challenge of fluent APIs.

1.2 Contributions

We present different algorithms for encoding fluent APIs in functional languages. Our fluent API designs support all regular and deterministic context-free API protocols and DSLs and can be implemented in most statically-typed functional languages. This work is accompanied by an
SML implementation of key algorithms. The full paper [Roth and Gil 2022] describes a series of experiments conducted using the implementation.

Outline. The rest of the paper is organized as follows. Section 2 covers preliminary knowledge in automata theory and object-oriented fluent APIs. The next three sections present three methods for encoding regular protocols as functional fluent APIs: bit shuffling in Section 3, Church Booleans in Section 4, and tabulation in Section 5. In Section 6 we show how to extend these techniques to non-regular protocols. Section 7 concludes with a discussion on functional fluent APIs and a comparison of the various encoding methods.

2 PRELIMINARIES

2.1 Automata and Formal Languages

We assume that the reader is familiar with the structure and operation of deterministic finite state machines (FSMs) and pushdown automata (DPDA). Here we briefly review some definitions; for the full theoretical background, we recommend Hopcroft et al. [2007].

FSMs recognize the class of regular languages. An FSM \( M = \langle Q, \Sigma, q_0, \delta, F \rangle \), where \( Q \) is a finite set of states, \( \Sigma \) is a finite set of input letters, \( q_0 \in Q \) is the initial state, \( \delta : Q \times \Sigma \to Q \) is the (partial) transition function, and \( F \subseteq Q \) is the set of accepting states. Let \( \delta_\sigma : Q \to Q \) be \( \delta \)'s projection on \( \sigma \), i.e., \( \delta(q, \sigma) = p \iff \delta_\sigma(q) = p \).

The run of word \( w = \sigma_1\sigma_2\ldots\sigma_n \in \Sigma^* \) on FSM \( M \) is a sequence of states \( \rho = \rho_0\rho_1\cdots\rho_n \) such that \( \rho_0 = q_0 \) and for every \( i = 1, \ldots, n \), \( \delta(p_{i-1}, \sigma_i) = p_i \). We say that \( M \) accepts \( w \) if and only if \( \rho_n \in F \); otherwise \( M \) rejects \( w \). The language of \( M \), \( L(M) \) is the set of words \( M \) accepts. Consider, for example, the FSM depicted in Fig. 2.1 in graph form. This FSM recognizes the language \((a^4)^*\) containing all sequences of the letter \( a \) whose length is a multiple of four.

![Fig. 2.1. An FSM accepting the language \((a^4)^*\)](image)

A DPDA is an FSM that uses an auxiliary pushdown store of stack symbols drawn from a finite set \( \Gamma \). The DPDA transition function is

\[
\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*
\]

In each computation step, the automaton reads symbol \( y \in \Gamma \) on the stack and replaces it with a (possibly empty) sequence of symbols \( y' \in \Gamma^* \). If the the automaton tries to read from an empty stack, it gets stuck and rejects the input word. A DPDA transition canconsume an input letter \( \sigma \in \Sigma \) or \( \varepsilon \), the empty word. These \( \varepsilon \)-transitions are used to modify the stack between input reads. A DPDA without \( \varepsilon \)-transitions is called a real-time DPDA, as it makes exactly one operation for each input letter. DPDAs recognize the class of deterministic context-free languages (DCFLs), a proper super-set of regular languages. For instance, a DPDA can recognize the Dyck language of balanced brackets, but an FSM cannot.
2.2 Encoding Automata as Fluent APIs in Object-Oriented Programming Languages

The traditional study of compiler design [Aho and Ullman 1977] provides many parser generation techniques that take formal language specifications and convert them into parsers. Similarly, the academic study of fluent APIs seeks DSL embedding techniques that encode DSL specifications (usually in automaton form) as fluent APIs. We now review some of the existing fluent API encoding methods, designed primarily for object-oriented host languages.

Encoding an FSM as a fluent API is easy. First, create class \( q \) for each state \( q \in Q \). Then, for each transition \( \delta(q, \sigma) = p \), add the method \( \sigma: q \rightarrow p \) to class \( q \). This method receives an (implicit) parameter of type \( q \) and returns an instance of class \( p \). Finally, add a special method \( $ \) only to classes of accepting states \( q_F \in F \) (its return type does not matter). A fluent chain of API calls (call for short) starts with an initial variable \( \_\_ \) of type \( q_0 \), followed by a sequence of API calls \( \sigma_1, \sigma_2, \ldots, \sigma_n \in \Sigma \), and ends with a call to \( $ \):

\[
\_\_.(\_\_.(\_\_.(\_\_.().().().().().$.())().$.())().$.())$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$.())().$) (1)
\]

For example, the fluent API in Listing 2 encodes the language \((a^4)^*\) as defined by the FSM in Fig. 2.1. The fluent API is followed by several chain examples in the form of (1). A given chain type checks if and only if the number of times it calls \( a() \) is a multiple of four. We say that the fluent API enforces (encodes, specifies, recognizes) the language \((a^4)^*\).

```java
interface q0 { q1 a(); // \( \delta(q_0, a) = q_1 \) /\ void $(); // \( q_0 \in F \) /\)
interface q1 { q2 a(); // \( \delta(q_1, a) = q_2 \) /\ }
interface q2 { q3 a(); // \( \delta(q_2, a) = q_3 \) /\ }
interface q3 { q0 a(); // \( \delta(q_3, a) = q_0 \) /\ }
q0 __ = null; // /\ q_0 is the initial state
// /\ example chains /\ {
  __.$(); // compiles, \( e \in L((a^4)^*)\)
  __.a().a().a().a().$.(); // compiles, \( a^4 \in L((a^4)^*)\)
  __.a().a().a().$.(); // does not compile, \( a^5 \notin L((a^4)^*)\)
}
```

Listing 2. A Java fluent API encoding the language \((a^4)^*\)

So, how does it work? The chain (1) essentially simulates the FSM run \( \rho = p_0p_1 \cdots p_n \) of the word \( w = \sigma_1\sigma_2 \cdots \sigma_n \) encoded in the chain. That is, the return type of \( \sigma_i() \) in (1), \( i = 1, 2, \ldots, n \) is \( p_i \). Starting with the initial variable \( \_\_ \) of type \( q_0 \), the initial state, adding a call to \( \sigma_i() \) yields the type \( \delta(p_{i-1}, \sigma_i) = p_i \). The call \( \sigma_n() \) returns an instance of class \( p_n \), so the next call, to the terminal function \( $() \), compiles if and only if \( p_n \in F \), which is exactly the FSM membership condition.

Notice that method \( a \) in Listing 2 is overloaded, since it has four different signatures: \( a: q_0 \rightarrow q_1 \), \( a: q_1 \rightarrow q_2 \), \( a: q_2 \rightarrow q_3 \), and \( a: q_3 \rightarrow q_0 \). In general, this construction induces overloading whenever letter \( \sigma \in \Sigma \) appears in more than one FSM transition.

Although challenging, it is possible to encode non-regular languages as fluent APIs [Gil and Levy 2016; Nakamaru et al. 2017; Xu 2010; Yamazaki et al. 2019]. We focus on the “tree encoding” of DPDAs introduced by Gil and Roth [2019]. Gil and Roth presented an algorithm for converting a DPDA A into a tree encoding that can be written as a fluent API. The resulting fluent APIs look very similar to what we saw in Listing 2, except that they use parametric polymorphism (generics, type parameters). Listing 3 demonstrates how generics are used in the construction of a fluent API from tree encoding. The listing contains a generic fluent API that recognizes the Dyck language—a well-known DCFL. The chain examples that follow the fluent API compile if and only if they encode balanced left (\( L \)) and right (\( R \)) brackets. The fluent API works by encoding the number of open brackets as a Peano number \( \$<\$<\ldots<\$<\ldots<\$>\ldots>\ldots> \) in the chain’s return types.
interface z { s<z> L(); void $(); }
interface s<x> { s<s<x>> L(); x R(); }

z __ = null; // we only care about type checking at the moment
/class example chains /
{
  __.L().L().L().R().R().R().$(); // compiles, the sequence ((())) is balanced
  __.L().L().R().L().R().R().$(); // compiles, the sequence (()()) is balanced
  __.L().R().R().$(); // does not compile, the sequence () is not balanced
  __.L().L().R().$(); // does not compile, the sequence () is not balanced
}

Listing 3. A Java fluent API encoding the Dyck language of balanced L/R brackets

We see Gil and Roth’s tree encoding algorithm as a black box. We only care about the structure of the fluent APIs it yields, as demonstrated in Listing 3. In general, the tree encoding algorithm results in classes $c_0, c_1, \ldots, c_m$ with zero or more type parameters, $c_i<x_0, x_1, \ldots, x_n>$. Functions’ return types are general terms over the available classes $c_0 – c_m$ and parameters $x_0 – x_n$, e.g., $c_1<x_2<x_3, c_3>, x_1>$. Note that the tree encoding algorithm, as well as the encoding techniques proposed in the papers cited above, rely on overloading similarly to the FSM case.

2.3 A Note on Typeclasses and Fluent APIs

The object-oriented fluent API techniques mentioned above, including the more advanced designs of Gil and Roth [2019] and Yamazaki et al. [2019], can be adapted to functional programming languages by substituting typeclasses for function overloading.

Typeclasses were introduced by Wadler and Blott [1989] as a formalism for ad hoc polymorphism in Haskell and other functional programming languages. The authors defined the semantics of typeclasses by translating them to the standard Hindley-Milner type system, in a technique later becoming known as dictionary passing. Listing 4 demonstrates how to use dictionary passing to implement an “overloaded” function in SML.

| line | code |
|------|------|
| 1    | fun square (numD: { mul: 'a → 'a → 'a }) x = #mul numD x x (* an "overloaded" square function *) |
| 2    | fun mulInt (a: int) (b: int) = a * b (* multiplication of integers *) |
| 3    | val intD = { mul = mulInt } (* int dictionary *) |
| 4    | fun mulReal (a: real) (b: real) = a * b (* multiplication of real numbers *) |
| 5    | val realD = { mul = mulReal }; (* real dictionary *) |
| 6    | square intD 2; (*⇒ 4: int*) |
| 7    | square realD 3.0; (*⇒ 9.0: real*) |

Listing 4. Dictionary passing in SML

Function square, the first in Listing 4, can be thought of as being overloaded since it accepts either a pair integers or a pair real numbers. We tell square how to multiply numbers by passing it a dictionary numD that points to the appropriate multiplication method: intD for integers and realD for real numbers.

Although dictionary passing is semantically equivalent to typeclasses, it is not a viable alternative for function overloading. Matching each argument type with a dictionary (intD, realD) takes the same effort from the programmer as having multiple functions (squareInt, squareReal). In Wadler and Blott’s original design, the compiler infers the dictionary from the argument type, so the programmer doesn’t have to specify it explicitly.

Proc. ACM Program. Lang., Vol. 7, No. OOPSLA1, Article 105. Publication date: April 2023.
3 BIT SHUFFLING

The fluent API encoding techniques described in Section 2.2 use function overloading and, therefore, cannot be used in functional languages such as SML. As explained above, dictionary passing does not offer a viable alternative. This section presents bit shuffling, the first of our three alternative methods for encoding FSMs as functional fluent APIs. This method is unique in that it does not rely on unification, resulting in relatively fast compilation even when enforcing highly complex API protocols.

The design regards FSM states as binary vectors and the FSM transitions as a collection of Boolean functions to manipulate these vectors. The current FSM state is encoded as a tuple of types \( f \) and \( t \) that represent the bits 0 and 1. Each API function applies a transformation of this tuple by shuffling the bits comprising it:

\[(f, t, f, t) \implies (t, f, f, t)\]

The function’s body must of course be such that the resulting tuple encodes the subsequent FSM state.

We explain the construction in the example of writing a fluent API for the toy language \((a^4)^*\), whose FSM is depicted in Fig. 2.1. Based on this example, we generalize to Algorithm 1.

3.1 Notation

A Boolean bit \( b \) is a member of \( \{0, 1\} \). We also denote Boolean bits as \( \text{false}, \text{true}, \) or \( \text{true}, \text{false} \). A Boolean vector \( v \) of length \( n \) is a sequence of \( n \) Boolean bits, \( v \in \{0, 1\}^n \). We write \( v \) as \( \langle b_n, \ldots, b_2, b_1 \rangle \) (indexed from the right to the left) or as \( b_n \cdots b_2 b_1 \) for short. An \( n \)-ary Boolean function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \) receives a Boolean vector of length \( n \) and returns one Boolean bit. An \( n \)-ary Boolean vector function \( f : \{0, 1\}^n \rightarrow \{0, 1\}^n \) receives a Boolean vector of length \( n \) and returns a vector of the same length. A Boolean vector function can be decomposed into \( n \) Boolean functions \( f = \langle f_n, \ldots, f_2, f_1 \rangle \) such that \( f_i \) computes the \( i \)th bit of \( f \):

\[f(v) = \langle f_n(v), \ldots, f_2(v), f_1(v) \rangle\]

The composition of Boolean function \( g \) and vector function \( f \) is defined by

\[(g \circ f)(v) = g(f(v))\]

Also, the composition of Boolean vector functions \( g \) and \( f \) is defined as

\[g \circ f = \langle g_n \circ f, \ldots, g_2 \circ f, g_1 \circ f \rangle\]

Note that \((g \circ f)(v) = g(f(v))\).

3.2 Step I: Binary Encoding of States and Transitions

Encode the automaton states \( Q = \{q_0, q_1, q_2, q_3\} \) as Boolean vectors of length \( \log(|Q|) = 2 \) using an arbitrary injective mapping \( \beta : Q \rightarrow \{0, 1\}^2 \):

\[
\begin{align*}
\beta(q_0) &= 00 \\
\beta(q_1) &= 01 \\
\beta(q_2) &= 10 \\
\beta(q_3) &= 11
\end{align*}
\]
Recall that $\delta_a$ is the projection of the transition function $\delta$ on letter $\sigma = a$. Now, construct a Boolean vector function $f^a$ such that $f^a(\beta(q)) = \beta(p) \iff \delta_a(q) = p$:

$$
\begin{align*}
    f^a(00) &= 01 \\
    f^a(01) &= 10 \\
    f^a(10) &= 11 \\
    f^a(11) &= 00
\end{align*}
$$

Let $f^a = \langle f^a_2, f^a_1 \rangle$. Notice that $f^a_2$ is exclusive or (XOR), $f^a_2 = \oplus$ and $f^a_1$ is negation (NOT) applied to the LSB, $f^a_1 = \neg_1$, so we can write

$$
    f^a = \langle \oplus, \neg_1 \rangle
$$

Observe that we can describe the FSM run on word $w = a^n$ using a sequence of $n$ applications of $f^a$:

$$
    (f^a \circ f^a \circ \ldots \circ f^a)(\beta(q_0)) = \beta(p_n)
$$

where $p_n$ is the last state in $w$’s run (this observation is proved by a simple induction on $n$). We can say that $f^a$ consolidates the four FSM transitions comprising $\delta_a$ into a single Boolean vector function.

Intuitively, the consolidation of FSM transitions into a single function helps us overcome the overloading problem, which arose in the first place because of the multiplicity of transitions. It is not enough, however, to implement $f^a$ as a function on Boolean vectors. Let

$$
    \\text{true} \quad \text{false} \quad \text{true} \quad \text{false} \quad \ldots \quad \text{true} \quad \text{false}
$$

be a functional fluent chain, with $\text{true}$ replacing $\_\_\_$ as the initial variable (now a function). We want the chain (4) to compile if and only if the number of times $\_\_$ is called is a multiple of four, i.e., if the chain encodes a word $w \in L((a^4)^*)$. Yet, if we implement the fluent API as a product of Boolean vector functions, say,

```plaintext
val ^\^ = (false, false) (* \beta(q_0) = 00 *)
fun $ (b2, b1) = (b2 \lor b1, not b1) (* f^a = (\oplus, \neg_1 ) *)
fun $ (b2, b1) = if not b2 andalso not b1 then () else raise Fail "!" (* q \in F \iff \beta(q) = 00 *)
```

the bits $b2$ and $b1$ would be evaluated at run time, but the fluent API must inspect them at compile time in order to fail compilation (if needed). To make the Boolean values available at compile time, we implement them as constants of different types:

```plaintext
datatype t = T (* 1, true Boolean *)
  and F = F (* 0, false Boolean *)
```

The return type of $\text{fun} a \ (b2, b1)$, now however, depends on the specific types of its arguments, which brings us back to the overloading problem. We address this issue in Step II.

### 3.3 Step II: Direct Encoding of Boolean Function Evaluations

Consider the following Boolean functions:

$$
\begin{align*}
    g^1(b_2b_1) &= b_2 \land b_1 \\
    g^2(b_2b_1) &= b_2 \land \neg b_1 \\
    g^4(b_2b_1) &= \neg b_2 \land b_1 \\
    g^8(b_2b_1) &= \neg b_2 \land \neg b_1
\end{align*}
$$

Function $g^8$ can be used to check our FSM’s acceptance condition, since

$$
    q \in F \iff q = q_0 \iff g^8(\beta(q)) = 1
$$
In the bit shuffling method, the terminal function $\text{\$}$ does not evaluate $g^8$ by itself, but instead receives its evaluation as an argument $g8$ (to replace $b2$ and $b1$). Then, it is a simple matter to check that the type of $g8$ is $t$, i.e., that $g^8(\beta(p_n)) = 1$.

Let $w = a^n$ be the word encoded by the fluent chain (4), and let $p = p_0p_1 \cdots p_n$ be its run. The initial variable $\text{\$}$ assigns $g8=\text{T}$, since

$$
g^8(\beta(p_0)) = g^8(\beta(q_0)) = 1
$$

The API function $\text{\textbackslash a}$ should receive in parameter $g8$ the evaluation of $g^8(\beta(p_1))$ and return the evaluation of $g^8(\beta(p_{i+1}))$. Since $f^a(\beta(q)) = \beta(p) \iff \delta_a(q) = p$ (see (3)), $g^8(\beta(p_{i+1}))$ can be computed as follows:

$$
g^8(\beta(p_{i+1})) = g^8(\beta(\delta_a(p_i))) = g^8(f^a(\beta(p_i))) = \ldots
$$

Let $\beta(p_i) = b2b1$. We continue:

$$
\ldots = g^8(f^a(b2b1)) = g^8((b2 \oplus b1, \neg b1)) = \neg(b2 \oplus b1) \land b1 = b2 \land b1 = g^1(b2b1) = g^1(\beta(p_i))
$$

(Function $g^1$ is defined in (5).) Overall, we get

$$
g^8(\beta(p_{i+1})) = (g^8 \circ f^a)(\beta(p_i)) = g^1(p_i)
$$

(6)
i.e., the evaluation of $g^8$ on the next state $p_{i+1}$ is equal to applying the composition $g^8 \circ f^a$ to the current state $p_i$, which in turn equals the evaluation of another function $g^1$ on $p_i$.

We let $\text{\textbackslash a}$ receive the value of $g^1(\beta(p_i))$ in yet another parameter $g1$. We then continue recursively to compute $g^1(\beta(p_{i+1}))$, in a similar way to how $g^8(\beta(p_{i+1}))$ was computed:

$$
g^1(\beta(p_{i+1})) = (g^1 \circ f^a)(\beta(p_i)) = g^2(\beta(p_i))
$$

$$
g^2(\beta(p_{i+1})) = (g^2 \circ f^a)(\beta(p_i)) = g^3(\beta(p_i))
$$

$$
g^3(\beta(p_{i+1})) = (g^3 \circ f^a)(\beta(p_i)) = g^4(\beta(p_i))
$$

(7)
The recursion ends on $g^8(\beta(p_i))$, since this value is already given in parameter $g8$. Let $g = \langle g^1, g^2, g^4, g^8 \rangle$ (the ordering of functions in $g$ does not matter). Equations (6) and (7) can be summarized as follows:

$$
g \circ f^a = \langle g^2, g^4, g^8, g^1 \rangle
$$

(8)

Thus, function $\text{\textbackslash a}$ receives a quadruple $(g1, g2, g4, g8)$ representing the four evaluations of $g^j(\beta(p_i))$, $j = 1, 2, 4, 8$, and returns the evaluations of $g^j(\beta(p_{i+1}))$ by shuffling the quadruple into $(g2, g4, g8, g1)$ following (8).

The resulting fluent API is shown in Fig. 2.1. The addition of parameter $f^a$ to functions $\text{\textbackslash a}$ and $\text{\textbackslash a}$ allows us to call them in fluent form, i.e., from the left to the right as in (4).

Listing 5. A shuffling SML fluent API for the FSM of Fig. 2.1
The fluent API in Fig. 2.1 is followed by a few fluent chain examples. Notice that a chain compiles if and only if the number of $\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_$ calls is a multiple of four, i.e., if the chain encodes $w \in L((a^4)^*)$.

### 3.4 The Bit Shuffling Encoding Algorithm

We now generalize the encoding steps described in Sections 3.2 and 3.3 and present an algorithm for any FSM $M$. FSMS are, in general, more complicated than our example FSM, so we need to add some minor steps to the encoding process. First off, notice that our example FSM, depicted in Fig. 2.1, has a total transition function $\delta$, i.e., every state has an outgoing edge for each input letter, and the number of states $|Q| = 4$ is a power of two. These properties make the Boolean vector function $f^a$ well-defined, i.e., $f^a(u)$ is defined for any Boolean vector of length $n = \log(|Q|)$. Given an FSM $M$, we make its transition function total by adding a non-accepting sink state, to which the missing transitions are directed. The sink state loops for any letter $\sigma \in \Sigma$. We then complete the number of states to a power of two by, again, adding sink states. These modifications result in automaton $M' = \langle Q', \Sigma, q_0, F, \delta' \rangle$ that accepts the same language as $M$, $L(M') = L(M)$.

Let $n = \log(|Q'|)$, $n \in \mathbb{N}$ be the number of bits in mapping $\beta$. Our example fluent API required the evaluation of four Boolean functions, $g_1, g_2, g_3$, and $g_4$. Instead of calculating which $g^\sigma$ functions are required for $M'$, we just take all the Boolean functions on $n$ bits, $g^n = \langle g^1, g_2, g_3, \ldots \rangle$ (there is a finite number of such functions). In addition, notice that the example language $(a^4)^*$ uses only a single letter $a$, but we want to support alphabets $\Sigma$ of any finite size. Given alphabet $\Sigma$ of arbitrary size, we simply repeat the encoding steps described above for each letter $\sigma \in \Sigma$, i.e., compute $f^\sigma$ and $g^n \circ f^\sigma$ and encode the result in a function $\sigma$.

Finally, the example FSM specifies a single accepting state $F = \{q_0\}$, but FSMS in general may have multiple accepting states. Therefore, we need to find a Boolean function $g^F$ such that $g^F(\beta(q)) = 1 \iff q \in F$. There must be (exactly one) such function in $g^n$. The terminal function simply checks that the type of parameter $g^F$, holding the evaluation of $g^F$, is $T$.

The complete bit shuffling algorithm is presented in Algorithm 1.

**Algorithm 1.** The bit shuffling algorithm for encoding an FSM as an SML fluent API

Some of the code excerpts printed in Algorithm 1 contain mathematical symbols—we now explain how these are rendered as text:

- A Boolean bit $b$ is rendered as $\top$ if $b = 1$ or as $\false$ if $b = 0$.
A Boolean vector \(a = \langle b_n, \ldots, b_2, b_1 \rangle\) is rendered as a tuple \(\langle b_2, \ldots, b_2, b_1 \rangle\).

A Boolean function \(g\) is rendered as \(\overline{g}\) i.e., we ignore its intent and print its name.

A Boolean vector function \(g = \langle g_1, g_2, \ldots, g_n \rangle\) is rendered as a tuple \(\langle g_1, g_2, \ldots, g_n \rangle\).

## 3.5 Encoding Improvements

We now propose two optional improvements of the bit shuffling algorithm as it is described in Algorithm 1. First, notice that Algorithm 1 computes all the binary functions on \(n = \log(|Q'|)\) bits. Overall, there are

\[2^{2^n} = 2^{|Q'|} \leq 2^{2^{|Q|}} = 4^{|Q|}\]

such functions, so the run time of the algorithm, as well as the size of the code it prints, are exponential in the size of the FSM. Nevertheless, our example fluent API (Listing 5) uses only four of these functions (out of 16), indicating that it might be possible to get rid of at least some functions. To determine which functions are necessary and which can be omitted, we run the recursive procedure described in Section 3.3: Start with \(g^a = \langle g^F \rangle\), and recursively add function \(g\) if there exists \(\sigma \in \Sigma\) and \(g' \in g^a\) such that \(g' \circ f^\sigma = g\). From our experience, this construction of \(g^a\) reduces its size significantly for “reasonable” FSMs. Determining the exact degree to which \(g^a\) is reduced and, specifically, whether or not its sizes stays exponential in \(|Q|\), is left as an open problem.

The second improvement relates to the fluent API user experience. The fluent API printed by Algorithm 1 allows any sequence of API calls to be made and checks whether or not it makes sense only in the terminal method \$. Therefore, the programmer using the fluent API is not alerted when they call a wrong API method, and when they do (on calling \$), they are not told where the mistake is located. Let

\[\overset{\wedge}{\sigma_1} \sigma_2 \ldots \sigma_n\]  \(\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (9)\]

be a partial fluent chain encoding \(w = \sigma_1 \sigma_2 \ldots \sigma_n\), and let \(\rho = p_0 p_1 \ldots p_n\) be \(w\)’s run on the FSM. To make our fluent API more user friendly, we ensure that a call to \(\sigma_{n+1} \in \Sigma\) after (9) will not compile if the resulting chain cannot be completed correctly by some sequence of API calls. We call this property of fluent APIs early failure.

Let \(R \subseteq Q\) be the set of states from which an accepting state is reachable (when viewing the FSM in graph form). Then, the call to \(\sigma_{n+1}\) should compile if and only if

\[\delta(p_n, \sigma_{n+1}) \in R\]  \(\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (10)\]

We can check this condition using the Boolean function \(g^R\) (similar to \(g^F\) used by \$) that holds

\[g^R(\beta(q)) = 1 \iff q \in R\]

We require that the evaluation of \(g^R\), stored in parameter \(gR\), is always true throughout the chain to achieve early failure. Let \(\sigma \in \Sigma\) be an API function, and let \(g1\) be the parameter it assigns in the position of \(gR\) in the shuffled tuple on function \(\sigma\)’s right-hand side, i.e., \(g^R(\beta(p_{j+1})) = g_i(\beta(p_j))\). If \(g^R\) evaluates to false after calling \(\sigma\), then our encoding ensures that \(g1\) is also false. Function \(\sigma\) should, accordingly, check that the type of \(g1\) is \(T\) by replacing the parameter \(g1\) with \(T\), as done in function \$ for parameter \(gF\).

When the two improvements described above are applied together, the recursive construction of \(g^a\) should start with \(g^a = \langle g^F, g^R \rangle\) to ensure that the parameter \(gR\) is present in the final vector.

### 4 CHURCH BOOLEANS

This section describes our fluent API encoding method based on the Church encoding of Booleans. In contrast to bit shuffling, presented in Section 3, the current technique relies on type inference and thus incurs longer compilation times. Nevertheless, our technique demonstrates an application.
of the classic Church Booleans to API design which, we believe, is a fascinating concept whose full potential is beyond the scope of the current paper.

Recall that bit shuffling encodes FSM states as Boolean vectors and FSM transitions as functions over these vectors. The same idea is at the base of our Church encoding. But instead of pre-computing function evaluations, as done in bit shuffling, the new technique directly computes bit functions at compile time by replacing the true and false monotypes with the Church Booleans. While Church Boolean circuits are usually applied in an untyped manner, they work mostly the same at the type level, except for a minor digression: duplicating a bit into two identical bits must be expressed explicitly using a fanout gate.

Before presenting the encoding, we review Church Booleans and how they can be implemented in SML:

```sml
fun T x y = x (∗ true, T : 'a→'b→'a *)
fun F x y = y (∗ false, F : 'a→'b→'b *)
```

_TRUE_ is a function that receives two arguments (Currying style) and returns the first; _False_ also receives two parameters but returns the latter. The type of true is _T_: 'a→'b→'a and the type of false is _F_: 'a→'b→'b. These are the Church Boolean functions:

```sml
fun Not b = b F T
fun Or (b2, b1) = b2 T b1
fun And (b2, b1) = b2 b1 F;
Not T; (∗ false, 'a→'b→'b *)
Or (And (T, F), And (T, T)); (∗ true, 'a→'b→'a *)
```

Can we use these simple Boolean functions to describe complex Boolean circuits? The following attempt fails to compile:

```sml
fun Xor (b2, b1) = Or (And (b2, Not b1), And (Not b2, b1));
Xor (T, F); (∗ compilation error: circularity *)
```

Evidently, the SML compiler fails to assign reasonable types to the parameters of function _Xor_. Mairson [2004] explains that function _Xor_ is problematic because it is non-linear, in the sense that it uses the bits _b2_ and _b1_ more than once in the same expression. To restore linearity, Mairson proposes to explicitly implement a fanout gate _Copy_ that takes a single bit and returns two equivalent bits:

```sml
fun Pair x y z = z x y
fun Copy p = p (Pair T T) (Pair F F)
```

Function _Copy_ is used this way:

```
Copy b (fn b1 => fn b2 => e)
```

_Copy_ evaluates the inner expression _e_ by assigning _b_’s type (Boolean value) to _b1_ and _b2_. Therefore, _e_ can use two instances _b1_ and _b2_ of the same Boolean bit _b_. Let us re-implement _Xor_ using _Copy_:

```sml
fun Xor (b2, b1) =
  Copy b2 (fn b2_1 => fn b2_2 =>
    Copy b1 (fn b1_1 => fn b1_2 =>
      Or (And (b2_1, Not b1_1), And (Not b2_2, b1_2))
    )
  )
Xor (T, F); (∗ true, 'a→'b→'a *)
Xor (F, F); (∗ false, 'a→'b→'b *)
```

---

2If you try to run this code using SML/NJ, you will get “dummy types” instead of plain _'a_ and _'b_ type parameters. We ignore dummy types in the current discussion.
After learning how to compute complex Boolean circuits at compile time, we are ready to describe the Church encoding of fluent APIs. Again, we start by encoding a fluent API for our example language \((a^I)^*\) and then generalize the construction for any FSM in Algorithm 2. The first step of the encoding is identical to Step I of bit shuffling (described in Section 3.2), so we can continue straight to the second step.

### 4.1 Step II: Encoding FSM States as Vectors of Church Booleans

In Step I (Section 3.2) we constructed mapping \(\beta : Q \rightarrow \{0, 1\}^2\) from states to bit vectors. Function \(\beta\) is used to encode the FSM states \(Q\) as Church Boolean vectors:

\[
\begin{align*}
\text{val q0} &= (F, F) \quad \langle q_0 = 00, q_0 : (a \rightarrow b \rightarrow \langle b \rightarrow \langle c \rightarrow d \rightarrow \langle d \rightarrow 1\rangle \rangle) \rangle \\
\text{val q1} &= (F, T) \quad \langle q_1 = 01, q_1 : (a \rightarrow b \rightarrow \langle b \rightarrow \langle c \rightarrow d \rightarrow \rangle \rangle) \rangle \\
\text{val q2} &= (T, F) \quad \langle q_2 = 10, q_2 : (a \rightarrow b \rightarrow \langle b \rightarrow \langle c \rightarrow d \rightarrow \rangle \rangle) \rangle \\
\text{val q3} &= (T, T) \quad \langle q_3 = 11, q_3 : (a \rightarrow b \rightarrow \langle b \rightarrow \langle c \rightarrow d \rightarrow \rangle \rangle) \rangle 
\end{align*}
\]

We also defined the vector function \(f^a : \{0, 1\}^2 \rightarrow \{0, 1\}^2\) where

\[
f^a(\beta(q)) = \beta(p) \iff \delta_a(q) = p
\]

This function can be described as a composition of two Boolean functions \(f^a = (\oplus, \neg_1)\), i.e.,

\[
f^a(b \oplus b_1, \neg b_1) = \langle b_2 \oplus b_1, \neg b_1 \rangle
\]

Function \(f^a\) can be encoded in SML using Church Booleans and Mairson’s `Copy` function as follows:

```sml
fun a (b2, b1) = Copy b1 (fn b1_1 => fn b1_2 =>
  (Xor (b2, b1_1), Not b1_2)
)
```

(for Not and Xor described above). Let us see function \(\bar{a}\) in action:

\[
\begin{align*}
\text{a q0}: \quad &\langle q_0 = 00, q_0 : (a \rightarrow b \rightarrow \langle b \rightarrow \langle c \rightarrow d \rightarrow \rangle \rangle) \rangle \\
\text{a q1}: \quad &\langle q_1 = 01, q_1 : (a \rightarrow b \rightarrow \langle b \rightarrow \langle c \rightarrow d \rightarrow \rangle \rangle) \rangle \\
\text{a q2}: \quad &\langle q_2 = 10, q_2 : (a \rightarrow b \rightarrow \langle b \rightarrow \langle c \rightarrow d \rightarrow \rangle \rangle) \rangle \\
\text{a q3}: \quad &\langle q_3 = 11, q_3 : (a \rightarrow b \rightarrow \langle b \rightarrow \langle c \rightarrow d \rightarrow \rangle \rangle) \rangle 
\end{align*}
\]

The type of \(\bar{a}\) applied on \(q_1\) is the type of \(q_j\) where \(\delta_a(q_j) = q_j\). Therefore, we can simulate an FSM run by recursively applying \(\bar{a}\) to \(q_0\). Let \(w = a^n\) and let \(p_n\) be the last state in \(w\)’s run. Then, applying function \(\bar{a}\) to \(q_0\) \(n\) times results in an expression of type \(p_n\) (the inductive proof is simple).

The terminal function \(\$\) should check that the last state in the run is accepting. Recall that in our example, this condition is verified by

\[
g^8(b_2b_1) = \neg b_2 \land \neg b_1
\]

as \(g^8(\beta(q)) = 1 \iff q \in F\). Since function \(g^8\) is linear, it can be encoded simply as

```sml
fun g8 (b2, b1) = And (Not b2) (Not b1)
```

Function \(\$\) applies \(g^8\) to \(b_2b_1\) and compels the result \(b\) to be true:

```sml
fun $ (b2, b1) = let val b = g8 (b2, b1) in (b \theta "") + 0 end
```

The expression \((b \theta "")\) has type `int` if \(b\) is true or `string` otherwise, so by adding \(+\theta\) we force \(b\) to be true, or else compilation will fail since a `string` cannot be added to an `int` in SML.

The full encoding of \((a^I)^*\) is shown in Listing 6, together with a few example chains.

### 4.2 The Church Boolean Encoding Algorithm

As in bit shuffling, the general Church Boolean encoding algorithm starts by converting the input FSM \(M\) into an equivalent FSM \(M'\) with a total transition function and whose number of states is a
Listing 6. A Church-encoded SML fluent API for the FSM of Fig. 2.1

As before, the code excerpts in Algorithm 2 contain mathematical symbols. These are rendered in text as follows:

- A Boolean bit \( b_i \) is rendered as \( \mathsf{T} \) if \( b_i = 1 \), as \( \mathsf{F} \) if \( b_i = 0 \), or as \( \mathsf{b}i \) if it is a variable (lines 9 and 11).
We view expressions \(e\) which cannot be evaluated at compile time, we write it as an indexing function \(\texttt{(unit)}\) that returns the index of \(e\) as an integer, i.e., once every four calls to \(\texttt{a}\). In SML, however, infinite expressions with infinite types cannot be described. For example, the following attempt to define \(e\) fails due to circularity:

\[
\text{val } e = \text{let fun } e'() = (\texttt{(unit)}, (\texttt{(unit)}), (\texttt{(unit)}), (\texttt{(unit)}), (\texttt{(unit)}), (\texttt{(unit)}))\text{ in } e'() \text{ end}
\]

We can describe expression \(e\) using four expressions as follows:

\[
e_0 = (\texttt{(unit)}, e_1) \\
e_1 = (\texttt{(unit)}), e_2) \\
e_2 = (\texttt{(unit)}), e_3) \\
e_3 = (\texttt{(unit)}), e_0) \\
e = e_0
\]

(12)

We view expressions \(e_0\) through \(e_3\) as the rows of a \(4 \times 2\) table, where the tuples’ left-hand is the first column, and the right-hand side is the second column. This *tabular* layout of \(e\) reveals its tight connection to the underlying \((a^4)^*\) FSM (Fig. 2.1): A table row \(e_i\) encodes the corresponding FSM state \(q_i\), where the first column encodes membership in \(F\) using \(\texttt{(unit)}\) \((e_i \notin F)\) and \(\texttt{(unit)}\) \((e_i \in F)\), and the second column encodes the output of the transition function \(\delta_a(e_i)\). For instance, expression \(e_2 = (\texttt{(unit)}), e_3)\) encodes state \(q_2\), since \(q_2 \notin F\) and \(\delta_a(q_2) = q_3\) (encoded by \(e_3\)). Let’s explicitly place expressions \(e_0\) through \(e_3\) in a table:

\[
e = \langle e_0, e_1, e_2, e_3 \rangle
\]

(13)

To untangle the recursive definition of expressions \(e_0\) through \(e_3\) (12), we replace each use of expression \(e_j\) with its index in table \(e\) (13), which is \(j\). Instead of writing index \(j\) as an integer, which cannot be evaluated at compile time, we write it as an indexing function \(I_j\) that returns the \(j^{th}\) entry of a (general) quadruple:
fun I0 (x0, x1, x2, x3) = x0
fun I1 (x0, x1, x2, x3) = x1
fun I2 (x0, x1, x2, x3) = x2
fun I3 (x0, x1, x2, x3) = x3
val e0 = ($$, I1)
val e1 = ((), I2)
val e2 = ((), I3)
val e3 = ((), I0)
val e = (e0, e1, e2, e3)

Now we can encode the API function \( a \) as follows:

fun a (\_, I) = I e

Applying \( a \) to expression \( e_i = (\cdot, I_j) \) evaluates to the \( j \)th element of quadruple \( e, I_j e \). Since \( j = (i + 1) \mod 4 \) for every expression \( e_i \), this element is \( e_{(i+1) \mod 4} \). Therefore, function \( a \) correctly encodes the FSM transition function:

\[
a e_i = e_j \iff \delta_a(q_i) = q_j
\]

The full fluent API encoding is shown in Listing 7. The API is followed by several chain examples.

```sml
 datatype $$ = $$
 fun I0 (x0, x1, x2, x3) = x0
 fun I1 (x0, x1, x2, x3) = x1
 fun I2 (x0, x1, x2, x3) = x2
 fun I3 (x0, x1, x2, x3) = x3
 val e0 = ($$, I1)
 val e1 = ((), I2)
 val e2 = ((), I3)
 val e3 = ((), I0)
 val e = (e0, e1, e2, e3)

 fun $$ f' = f' e0 (** q0 is the initial state **)
 fun a (\_, I) f' = f' (I e)
 fun $ ($$, _) = ($$);

 (** chain examples:**
 ^ $$; (** OK, \( e \in L((a^4)^*) \)**)
 ^ a a a a $$; (** OK, \( a^4 \in L((a^4)^*) \)**)
 ^ a $$; (** error, \( a^1 \notin L((a^4)^*) \)**)
 ^ a a $$; (** error, \( a^2 \notin L((a^4)^*) \)**)
 ^ a a a a a $$; (** error, \( a^5 \notin L((a^4)^*) \)**)

Listing 7. A tabulation SML fluent API for the FSM of Fig. 2.1
```

A general FSM with \( n = |Q| \) states and \( m = |\Sigma| \) letters is encoded by an \( n \times (m + 1) \) table, i.e., each letter is given its own column. State \( q_i \) (a table row) is encoded by the expression

\[
\text{val } e_i = (a, (I^{(1)}, \ldots, I^{(m)}))
\]

where \( a = $$ \) if \( q_i \in F \) or \( \cdot \) otherwise and \( I^{(j)} = I_k \) where \( \delta(q_i, \sigma_j) = q_k \). If \( \delta(q_i, \sigma_j) \) is not defined, set \( I^{(j)} = \cdot \), making compilation fail whenever the underlying FSM gets stuck. This means that in order to get the early failure property, described in Section 3.5, we simply need to remove from the FSM every state that cannot reach an accepting state.

The general tabulation encoding algorithm is described in Algorithm 3.
### Fluent APIs in Functional Languages

**Input** FSM $M = (Q, \Sigma, q_0, F, \delta)$

1. **Print** `datatype $S = S$`
2. Enumerate $Q = \{q_1, q_2, \ldots, q_n\}$
3. **For all** $j = 1, 2, \ldots, n$ **do**
   
4. **Print** `fun $I_j(x_1, x_2, \ldots, x_n) = x_j$`
5. Enumerate $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$
6. **For all** $i = 1, 2, \ldots, n$ **do**
7. Let $a \leftarrow $ if $q_i \in F$ or else $(\cdot)$
8. **For all** $j = 1, 2, \ldots, m$ **do**
9. Let $I^{(j)} \leftarrow I_k$ where $\delta(q_i, \sigma_j) = q_k$
10. Let $I^{(j)} \leftarrow ()$ where $\delta(q_i, \sigma_j)$ is not defined
11. **Print** `val ei = (a, (I^{(1)}, I^{(2)}, \ldots, I^{(m)}))`
12. **Print** `val e = (e_1, e_2, \ldots, e_n)`
13. **Print** `fun ^^ f' = f' $e$` where $q_i$ is the initial state
14. **For all** $j = 1, 2, \ldots, m$ **do**
15. **Print** `val $\sigma_j(\_\_, (\_\_, \ldots, I, \ldots, \_\_)) = f' (I \ e)` where the inner tuple has $m$ entries and argument $I$ appears in the $j^{th}$ entry
16. **Print** `fun $(S, \_\_)$ = ()`

---

**Algorithm 3.** The tabulation algorithm for encoding an FSM as an SML fluent API

---

**6 BEYOND REGULAR LANGUAGES**

In Sections 3 to 5 we described three methods for encoding FSMs as fluent APIs. These methods can be used to embed regular DSLs in functional programming languages. This leaves us wondering about DSLs that are not regular. It is actually not hard to come up with a fluent API that recognizes a non-regular language. For example, the fluent API shown in Listing 8 recognizes the Dyck language of balanced $L/R$ brackets, known to be non-regular. (This is a functional re-implementation of the object-oriented fluent API in Listing 3.)

---

**Listing 8.** An SML fluent API encoding the Dyck language of balanced $L/R$ brackets

```sml
datatype Z = Z
and 'x S = S of 'x
fun L x f' = f' (S x)
fun R (S x) f' = f' x
fun ^^ f' = f' Z
fun $ Z = ()
val w1 = ^^ L R $ (+ compiles, '()) are balanced +)
val w2 = ^^ L L R R R R $ (+ compiles, '(())) are balanced +)
val w3 = ^^ L L R L R L $ (+ does not compile, '(())) are not balanced -)
```

---

The current section introduces two fluent API encoding techniques for non-regular languages. The first technique integrates several simple APIs (as seen in Listing 8) into one, more complex API. The second technique generalizes our tabulation encoding (presented in Section 5) to support all DCFLs.

**6.1 Product APIs**

Consider the fluent APIs $A_2$ and $A_3$ shown in Listing 9. These APIs encode the languages $(a^2)^*$ and $(a^3)^*$, respectively, using shuffle encoding (cf. Listing 5).
datatype t = T and f = F
structure A2 = struct
  fun ^^ f' = f' (T, F)
  fun a (x1, x2) f' = f' (x2, x1)
  fun $ (T, _) = ()
end
structure A3 = struct
  fun ^^ f' = f' (T, F, F)
  fun a (x1, x2, x3) f' = f' (x3, x1, x2)
  fun $ ((T, _), (T, _, _)) = ()
end

Listing 9. SML APIs for the languages \((a^2)^*\) and \((a^3)^*\)

We create the product of APIs \(A2\) and \(A3\) by pairing the inputs and outputs of each API function, as shown in Listing 10. For instance, the expression \(((T, F), (T, F, F))\) (line 2) is a tuple of the respective expressions \((T, F)\) in \(A2\) and \((T, F, F)\) in \(A3\). The resulting API \(A6\) simulates the two original APIs simultaneously: the simulation of \(A2\) is managed on the left-hand side of each tuple and \(A3\) on the right-hand side.

structure A6 = struct
  fun ^^ f' = f' ((T, F), (T, F, F))
  fun a ((x1, x2), (x1', x2', x3')) f' = f' ((x2, x1), (x3', x1', x2'))
  fun $ ((T, _), (T, _, _)) = ()
end

Listing 10. The product of the APIs in Listing 9

What, accordingly, is the language encoded by \(A6\)? Intuitively, since the termination function \$ enforces the acceptance conditions of both \(A2\) and \(A3\) simultaneously, the language encoded by \(A6\) is the intersection of their languages:

\[
(a^2)^* \cap (a^3)^* = (a^6)^*
\]

In general, we can use this product method to encode a complex DSL \(L\) by (i) describing it as the intersection of two simpler languages, \(L = L_1 \cap L_2\), (ii) encoding these languages as fluent APIs \(L_1\) and \(L_2\), and (iii) writing the product API of \(L_1\) and \(L_2\).

The \(L/R\) API presented at the start of this section (Listing 8) recognizes a non-regular language by simulating a stack machine (DPDA) at the type level. We can use similar stack-based APIs in combination with our shuffle, Church, and tabulation encoding methods to create powerful product APIs. For example, the (enhanced) HTML API discussed in Section 1 is the product of a shuffle API and several stack machines. The shuffle API enforces the order of the HTML tags, e.g., that \(<html>\) can be followed by \(<body>\) but never by \(<p>\), but does not check that each tag has a matching closing tag. Tag matching is verified in a separate stack-based API, that ignores the ordering of tags. Yet another stack-based API is used to check that all the rows of an HTML table have the same number of columns. Although each of these APIs alone encodes a nonsensical language, their product API exactly encodes enhanced HTML, a complex, context-sensitive language.

6.2 Functional Fluent APIs for DCFLs

Gil and Roth [2019] presented an algorithm for encoding DCFLs as fluent APIs in object-oriented languages. Their algorithm relies on a minimal set of common object-oriented type system features,
including parametric polymorphism and overloading. For example, a method created by Gil and Roth’s construction might look like this:

```java
interface A<x> {
    B<C<x>, x> f();
}
```

We denote the signature of method $f$ as follows:

$$f : A(x) \rightarrow B(C(x), x)$$

Note that $f$ explicitly accepts an object of the containing class $A<x>$ as input. There are no additional input parameters in the basic encoding.

In general, object-oriented fluent API methods have the following form:

$$\sigma : \gamma(x) \rightarrow \tau \quad (14)$$

where $\sigma$ is the method name, $\gamma$ is the name of the containing class, $x$ are its type parameters, and $\tau$ is the return type. Type $\tau$ is a (possibly) non-grounded type comprising variables $x$ and other API types. The fluent API also includes an initial variable:

$$^\wedge : t_0 \quad (15)$$

(for ground type $t_0$) and termination methods that return `void` (unit):

$$\$: \gamma(x) \rightarrow \emptyset \quad (16)$$

Observe that API method signatures (14) are essentially tree rewrite rules. Thus, we can see a fluent chain of method calls as a series of rewrites of the initial tree (type) $t_0$ of the initial variable (15). The termination methods (16) simply separate the API types into accepting and rejecting types, i.e., by checking whether or not $\$` accepts type $\gamma$ as input.

For example, the fluent API shown in Listing 3 in Section 2, which encodes an L/R Dyck language, defines the following signatures:

1. $^\wedge : z \quad (a)$
2. $L : z \rightarrow s(z) \quad (b)$
3. $L : s(x) \rightarrow s(s(x)) \quad (c)$
4. $R : s(x) \rightarrow x \quad (d)$
5. $\$: $z \rightarrow \emptyset \quad (e)$

The fluent API in (17) overloads method $L$ and, therefore, cannot be directly encoded in a functional programming language. It is definitely possible, however, to write an equivalent functional API—we saw one in Listing 8. We now describe the construction of another functional fluent API for the L/R Dyck language. This time, we take a more methodological approach by converting the object-oriented fluent API in (17) into a functional fluent API, and then generalize this method for all DCFLs.

The idea is to apply tabulation encoding, presented in Section 5, on top of Gil and Roth’s construction. Recall that in the FSM case, we encoded each machine state as a table row (tuple)

$$q \Rightarrow (a, (I^{(1)}, \ldots, I^{(m)}))$$

where $a \in \{\emptyset, $$\}$ indicates whether state $q$ is accepting and the $I$’s are indexing functions encoding the outgoing $q$ edges. In the non-regular case, we encode generic types instead of states. Instead of outgoing edges, we index the rewrite rules of API methods: (17.b) is $\rho_1$, (17.c) is $\rho_2$, and (17.d) is $\rho_3$. In addition, the encoding of generic type $\gamma(t)$ recursively includes the encodings of its type arguments $t$. Thus, type $\gamma(t_1, \ldots, t_k)$ is encoded as follows:

$$\gamma(t_1, \ldots, t_k) \Rightarrow (a, (I^{(1)}, \ldots, I^{(m)}), (e_1, \ldots, e_k))$$
where (i) $\gamma$ indicates whether $\gamma$ is accepting, (ii) $I_j$ is the index of $\sigma_j : \gamma(x) \rightarrow \tau$ in the table, and (iii) $e_1$, through $e_k$ are the encodings of types $t_1$ through $t_k$.

In our example, types $z$ and $s(x)$ are encoded as follows:

```
val z = ($$, (I1, (), ()))
fun s x = (((), (I2, I3), (x)))
```

Type $z$ is accepting, $a = $$, by (17.e), but $s$ is rejecting since function $\$ does not accept $s$. Functions $I_1$, $I_2$, and $I_3$ are the indices of rules $\rho_1$, $\rho_2$, and $\rho_3$ of the API methods $L$ and $R$. Since $z : R \rightarrow \tau$ is not defined, we put $\{}$ in $R$’s locations in the encoding of $z$. Finally, type $z$ is non-generic and thus its encoding is a value. On the other hand, $s$ has a type parameter $x$, so we encode it as a function with a single parameter that captures $x$’s encoding. Observe that by encoding generic types as functions, we can create the encodings of complex types by writing them directly as expressions, e.g., the encoding of type $s(s(z))$ is given by the expression $s(s(z))$.

Next, we encode the rewrite rules $\rho_1$, $\rho_2$, and $\rho_3$ as functions and collect them in table $r$:

```
fun r1 () = s(z) (*17.b*)
fun r2 (x) = s(s(x)) (*17.c*)
fun r3 (x) = x (*17.d*)
val r = (r1, r2, r3)
```

These functions do not specify the containing class $\gamma$, but are otherwise identical to the signatures from which they originated (17.b–d).

Finally, we can encode the fluent API functions. The initial function simply returns the encoding of the initial type $t_0 = z$ (17.a):

```
fun ^^ f' = f' z
```

Functions $L$ and $R$ are encoded as follows:

```
fun L (_, (I, _), X) f' = f' (I r X)
fun R (_, (_, I), X) f' = f' (I r X)
```

Each function extracts its respective index $I$ from the type encoding, applies it to table $r$ to get the appropriate rewrite function, and then applies this function to the type arguments $X$. For instance, when we apply function $L$ to encoding $s(z)$, the index is $I = I1$, the rewrite function is $I r = r2$, and the resulting expression $r2 \ s(z) = s(s(z))$ correctly encodes the application of $L$ to type $s(z)$ in the original API (17.c). The termination function $\$ simply verifies that $a = $$:

```
fun $ ($$, _, _) = ()
```

The complete functional fluent API is shown in Listing 11.
The general DCFL encoding algorithm is presented in Algorithm 4. This algorithm starts by applying Gil and Roth’s construction to produce an object-oriented fluent API, and then applies the (modified) tabulation encoding to remove overloading.

| Input DPDA A |
|-------------|
| 1. Apply the tree encoding algorithm of Gil and Roth [2019] to convert A into an object-oriented fluent API P as specified in Eqs. (14) to (16). |
| 2. Print **datatype** $$ = $$ |
| 3. Enumerate P’s API functions (14) \( R = \{ \rho_1, \rho_2, \ldots, \rho_n \} \) |
| 4. For all \( j = 1, 2, \ldots, n \) do |
| 5. Print **fun** \( I_j (x_1, x_2, \ldots, x_n) = x_j \) |
| 6. Enumerate P’s API functions \( \sigma \) for all \( \sigma \) where \( \sigma(x) \to \tau \) is not defined in \( P \) |
| 8. For all \( i = 1, 2, \ldots, l \) do |
| 9. Let \( a \leftarrow $$ \) if \( P \) includes the method \( \gamma_i(x) \to () \) or else \( () \) |
| 10. For all \( j = 1, 2, \ldots, m \) do |
| 11. Let \( I^{(j)}_i \leftarrow I_k \) where \( I_k = (\sigma \colon \gamma_i(x) \to \tau) \in P \) |
| 12. Let \( I^{(j)}_i \leftarrow () \) where \( \sigma \colon \gamma_i(x) \to \tau \) is not defined in \( P \) |
| 14. Print **val** \( y_i = (a, (I^{(1)}_i, I^{(2)}_i, \ldots, I^{(m)}_i), ()) \) |
| 15. else |
| 16. Print **fun** \( y_i (x) = (a, (I^{(1)}_i, I^{(2)}_i, \ldots, I^{(m)}_i), (x)) \) |
| 17. For all \( i = 1, 2, \ldots, n \) do |
| 18. Let \( \sigma(x) \to \tau \leftarrow \rho_i \) |
| 19. Print \( r_i (x) = \tau \) |
| 20. Print **val** \( r = (r_1, r_2, \ldots, r_n) \) |
| 21. Print **fun** \( f' = f'' \) where \( f'' : t_0 \) is \( P \)'s initial variable |
| 22. For all \( j = 1, 2, \ldots, m \) do |
| 23. Print **val** \( \sigma_j (\_, \_, \_, \_, I, \_, \_, \_, X) = f'' (I \ r \ X) \) where the inner tuple has \( m \) entries and argument \( I \) appears in the \( j \)th entry |
| 24. Print **fun** $$ ($$, _, _\_ = (\_) |

Algorithm 4. The tabulation algorithm for encoding a DPDA as an SML fluent API

Algorithm 4 accepts as input a DPDA accepting the target DCFL. In line 1, it converts the input into an object-oriented fluent API using tree encoding. In lines 8 through 16 the algorithm encodes the API types as functions and variables. The tree rewrite rules are encoded in lines 17 through 20. The tree pattern \( \tau \) is printed as-is. Finally, the algorithm prints the API functions in lines 21 through 24.

7 DISCUSSION

7.1 Functional vs. Object-Oriented Fluent APIs

We showed that despite the predicament of lack of overloading, fluent APIs and DSL embedding can be carried out in functional programming languages just as in non-functional languages. The implementation methods are largely independent of the syntax of the host functional language. Also, the algorithms presented in Sections 3 to 6 are general, and can be thought of as compiler-compilers for the respective classes of formal languages. The contribution is in line with recent advances in
the study of object-oriented fluent APIs, focusing on fluent APIs for DCFLs [Gil and Levy 2016; Gil and Roth 2019; Yamazaki et al. 2019].

Note that in using functional languages, the syntax of fluent APIs is cleaner: The method chaining invocation style \( x.f().g() \) is three times longer (character wise) than \( x \ f \ g \), the functional equivalent. Although probably technically possible to support, the terse functional form of chaining is not found in any object-oriented language other than Groovy’s specialized support for fluent APIs [Apache 2023].

DSL embedding in functional languages is expected to be syntactically cleaner, providing more freedom to the language designer than in non-functional languages, for four reasons: functional languages tend to use few reserved words, to support alphanumeric identifiers, to realize operators as such identifiers, and to implement these operators as libraries rather than built-in operators. These advantages are demonstrated in the embedding of HTML in SML (Section 1).

Our encoding schemes however are not able to solve the overloading predicament when it comes to literals: Literals of the embedded DSL cannot be function names, and therefore must be realized as function arguments. For example, in the embedded HTML document in Listing 1, attribute \texttt{src} in the \texttt{img} tag is a function that receives a string argument specifying the image source:

\begin{verbatim}
<br src="https://tinyurl.com/yosemite5"/>
\end{verbatim}

A function however can receive a string literal or a URL object, but not both.

### 7.2 Toolbox of Functional Fluent API Encoding Methods

Sections 3 to 5 present three different algorithms for encoding regular languages as functional fluent APIs. The most promising method is probably bit shuffling, described in Section 3. Bit shuffling fluent APIs do not employ unification, so they are expected to induce reasonable compilation times across various languages and compilers. The bit shuffling improvements described in Section 3.5 greatly enhance utility by reducing code size and improving error reporting.

The crux of the bit shuffling construction is to describe the FSM as a collection of circuits or logic gates. In Section 4, we showed how to implement these circuits at the type level using Church Booleans and Mairson’s [2004] fanout gate. Compile-time circuits may have some additional implications for API design beyond fluent API. For instance, they can be used to conduct numeric calculations in a type-safe physical dimensions library [Kennedy 1994].

Tabulation encoding (Section 5) is a flexible embedding method that can be generalized for non-regular DSLs (Section 6). It provides the theoretical basis for a functional fluent API generator in the style of Fling [Gil and Roth 2019; Roth and Gil 2019] and TypeLevelLR [Yamazaki et al. 2019]. Such tools compile LL, LALR, and LR grammars into fluent APIs that validate the DSL syntax at compile time and produce abstract syntax trees (ASTs) at run time.

### 7.3 Functional Subchaining

A fluent API embeds a DSL program as a single chain of method calls. This design can be restrictive when we want to create excerpts of the DSL program. In the HTML document of Listing 1, we might decide, for example, to place the HTML table in a variable and then reference it in the complete webpage. Recently, Yamazaki et al. [2022] addressed the issue of generating object-oriented fluent APIs that support subchains—fluent chains that embed DSL fragments. Since our functional fluent API designs use curried functions, they come with subchaining (almost) out of the box. Listing 12 demonstrates functional subchaining using our HTML fluent API.

Function \texttt{sc}, defined in line 1 of Listing 12, is an auxiliary function for creating subchains. Our fluent API designs require no additions or modifications to support subchaining besides \texttt{sc}. In lines 2 through 6, we define an HTML excerpt containing an HTML table. The boilerplate code
fun sc a b = b a (* subchaining helper function *)
val table = fn f' => sc f' (* HTML table subchain *)
<tr> <th> "Image" </th> </tr>
<tr> <td> <img src "https://tinyurl.com/yosemite5"/> </td> </tr>
val webpage = ^^ (* using the subchain in a complete embedding *)
<html>
<body> table (* subchain usage *)
</body>
</html>

Listing 12. Subchaining with a functional fluent API

fn f' => sc f'

in line 2 must precede every subchain. Variable table containing the HTML table is used as part of a complete HTML embedding in lines 7 through 13.

At the type level, using a subchain is equivalent to inlining its code, so the fluent API correctly reports API misuses in subchains. However, the error messages are displayed where the subchain is referenced and not at the error’s actual location. We speculate that more sophisticated fluent API designs could force the compiler to check the correctness of subchains at the place of definition.

7.4 Performance

This work is accompanied by Flunct [Roth 2023], an SML implementation of the bit shuffling, Church, and tabulation encoding techniques. Flunct compiles a given FSM $M$ into a functional fluent API that validates $M$ at compile time. Figure 7.1 compares the compilation times using Flunct bit shuffling, Church, and tabulation fluent APIs using the MLton SML compiler for the formal language $(a^{16})^*$. 

![Fig. 7.1. Chain length vs. compilation time of bit shuffling, Church, and tabulation fluent APIs using MLton for the formal language $(a^{16})^*$.](image-url)
As can be seen in the figure, bit shuffling, together with the improvements described in Section 3.5, performs the best in terms of compilation time, which is expected considering that bit shuffling is the only technique that doesn’t rely on unification. A complete description of the experiments is found in the full paper.

REFERENCES

Alfred V. Aho and Jeffrey D. Ullman. 1977. Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing). Addison-Wesley Longman Publishing Co., Inc., USA.

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-Oriented Programming. In Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (Oklahoma, Florida, USA) (OOPSLA ’09). Association for Computing Machinery, New York, NY, USA, 1015–1022. https://doi.org/10.1145/1639950.1640073

Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. 2008. Paradise: A Two-Stage DSL Embedded in Haskell. In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (Victoria, BC, Canada) (ICFP ’08). Association for Computing Machinery, New York, NY, USA, 225–228. https://doi.org/10.1145/1411204.1411236

Eric Bodden and Laurie Hendren. 2012. The Clara Framework for Hybrid Typestate Analysis. Int. J. Softw. Tools Technol. Transf. 14, 3 (June 2012), 307–326.

François Bourdoncle and Stephan Merz. 1997. Type Checking Higher-Order Polymorphic Multi-Methods. In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Paris, France) (POPL ’97). Association for Computing Machinery, New York, NY, USA, 302–315. https://doi.org/10.1145/263699.263743

Jan Bracker and Andy Gill. 2014. Sunroof: A Monadic DSL for Generating JavaScript. In Practical Aspects of Declarative Languages, Matthew Flatt and Hai-Feng Guo (Eds.). Springer International Publishing, Cham, 65–80.

Cristiano Calcegno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In Generative Programming and Component Engineering, Frank Pfennig and Yannis Smaragdakis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 57–76.

Krzysztof Czarnecki, John T. O’Donnell, Jörg Striegnitz, and Walid Taha. 2004. DSL Implementation in MetaOCaml, Template Haskell, and C++. Springer Berlin Heidelberg, Berlin, Heidelberg, 51–72. https://doi.org/10.1007/978-3-540-25935-0_4

Markus Degen, Peter Thiemann, and Stefan Wehr. 2007. Tracking Linear and Affine Resources with Java(X). In ECOOP 2007 – Object-Oriented Programming, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 550–574.

Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In Proceedings of the ACM SIGCIC 2001 Conference on Programming Language Design and Implementation (Snowbird, Utah, USA) (PLDI ’01). Association for Computing Machinery, New York, NY, USA, 59–69. https://doi.org/10.1145/378795.378811

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-Based Syntactic Language Extensibility. In Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (Portland, Oregon, USA) (OOPSLA ’11). Association for Computing Machinery, New York, NY, USA, 391–406. https://doi.org/10.1145/2048066.2048099

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martijn Palatnik, Risto Pohjonen, Eugen Schindler, Clemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eleco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In Software Language Engineering, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer International Publishing, Cham, Switzerland, 197–217.

Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster than the Quick and Dirty Way. In Proceedings of the 19th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (Reno, Nevada, USA) (OOPSLA ’10). Association for Computing Machinery, New York, NY, USA, 307–309. https://doi.org/10.1145/1869542.1869625

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In Proceedings of the 24th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications Companion (Victoria, BC, Canada) (ICOOP ’15). Association for Computing Machinery, New York, NY, USA, 550–574.

Kostas Ferles, Jon Stephens, and Isil Dillig. 2021. Verifying Correct Usage of Context-Free API Protocols. Proc. ACM Program. Lang., 5, POPL, Article 17 (Jan. 2021), 30 pages. https://doi.org/10.1145/3434298

John Field, Deepak Goyal, G. Ramalingam, and Eran Yahav. 2003. Typestate Verification: Abstraction Techniques and Complexity Results. In Static Analysis, Radia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 439–462.

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective Typestate Verification in the Presence of Aliasing. ACM Trans. Softw. Eng. Methodol. 17, 2, Article 9 (May 2008), 34 pages. https://doi.org/10.1145/105:24 Ori Roth and Yossi Gil
Gert Florijn. 1995. Object Protocols as Functional Parsers. In *ECOOP ’95 — Object-Oriented Programming*, 9th European Conference, Aarhus, Denmark, August 7–11, 1995, Mario Tokoro and Remo Pareschi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–373.

The Apache Software Foundation. 2023. Domain-Specific Languages. Retrieved March 10, 2023 from https://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html

Martin Fowler. 2005a. FluentInterface. Retrieved January 27, 2023 from https://www.martinfowler.com/bliki/FluentInterface.html

Martin Fowler. 2005b. Language Workbenches: The Killer-App for Domain Specific Languages? Retrieved January 27, 2023 from https://martinfowler.com/articles/languageWorkbench.html

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.*, 36, 4, Article 12 (Oct. 2014), 44 pages. https://doi.org/10.1145/2629609

Yossi Gil and Tomer Levy. 2016. Formal language recognition with the Java type checker. In *30th Europ. Conf OOProg. (ECOOP 2016)* (Leibniz International Proceedings in Inf. (LIPIcs), Vol. 56), Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), 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 Europ. Conf OOProg. (ECOOP 2019)* (Leibniz International Proceedings in Inf. (LIPIcs), Vol. 134), Alastair F. Donaldson (Ed.), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 13:1–13:25. https://doi.org/10.4230/LIPIcs.ECOOP.2019.13

Andy Gill. 2014. Domain-Specific Languages and Code Synthesis Using Haskell. *Commun. ACM* 57, 6 (June 2014), 42–49. https://doi.org/10.1145/2605205

Robert Harper, David MacQueen, and Robin Milner. 1986. *Standard ML*. Department of Computer Science, University of Edinburgh, The King’s Buildings, Edinburgh.

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

Paul Hudak. 1998. Modular domain specific languages and tools. In *Proceedings. Fifth International Conference on Software Reuse* (Cat. No.98TB100203). IEEE Computer Society, USA, 134–142. https://doi.org/10.1109/ICSR.1998.685738

Graham Hutton and Erik Meijer. 1998. Monadic parsing in Haskell. *Journal of Functional Programming* 8, 4 (1998), 437–444. https://doi.org/10.1017/S0956796898003050

Samuel N. Kamin and David Hyatt. 1997. A Special-Purpose Language for Picture-Drawing. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL) ’97*, USENIX Association, USA, 23.

Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA ‘10)*. Association for Computing Machinery, New York, NY, USA, 444–463. https://doi.org/10.1145/1869459.1869497

Steven Kelly, Kalle Lyytinen, and Matti Rossi. 1996. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In *Advanced Information Systems Engineering*, Panos Constantopoulos, John Mylopoulos, and Yannis Vassiliou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–21.

Andrew Kennedy. 1994. Dimension Types. In *Programming Languages and Systems — ESOP ’94*, Donald Sannella (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–362.

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

Viktor Kuncak, Patrick Lam, and Martin Rinard. 2002. Role Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL ’02)*. Association for Computing Machinery, New York, NY, USA, 17–32. https://doi.org/10.1145/503272.503276

Daan Leijen and Erik Meijer. 2000. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages* (Austin, Texas, USA) *(DSL ’99)*. Association for Computing Machinery, New York, NY, USA, 109–122. https://doi.org/10.1145/331960.331977

Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Technical Report UU-CS-2001-27. Dept. of Computer Science, Universiteit Utrecht.

David MacQueen. 1984. Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA) *(LISP ’84)*. Association for Computing Machinery, New York, NY, USA, 198–207. https://doi.org/10.1145/800055.800236

Harry G. Mairson. 2004. Linear lambda calculus and PTIME-completeness. *Journal of Functional Programming* 14, 6 (2004), 623–633. https://doi.org/10.1017/S0956796804005131
Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. 2017. Silverchain: a fluent API generator. In \textit{Proc. 16th ACM SIGPLAN Int. Conf Generative Prog. (GPCE’17)}. ACM, Vancouver, BC, Canada, 199–211.

Matthias Neubauer, Peter Tiemann, Martin Gasbichler, and Michael Sperber. 2002. Functional Logic Overloading. In \textit{Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages} (Portland, Oregon) (POPL ’02). Association for Computing Machinery, New York, NY, USA, 233–244. \url{https://doi.org/10.1145/503272.503294}

Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. 2003. Polyglot: An Extensible Compiler Framework for Java. In \textit{Compiler Construction}, Görel Hedin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 138–152.

M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In \textit{2012 34th International Conference on Software Engineering (ICSE 2012)}. IEEE Computer Society, Los Alamitos, CA, 925–935. \url{https://doi.org/10.1109/ICSE.2012.6227127}

Ori Roth. 2023. Flunct: Functional Fluent API Generator. \url{https://doi.org/10.5281/zenodo.7723110}

Ori Roth and Yossi Gil. 2019. Fling - A Fluent API Generator (Artifact). \textit{Dagstuhl Artifacts Series} 5, 2 (2019), 12:1–12:9. \url{https://doi.org/10.4230/DARTS.5.2.12}

Ori Roth and Yossi Gil. 2022. Fluent APIs in Functional Languages (full version). \url{https://doi.org/10.48550/ARXIV.2211.01473}

Tim Sheard. 1999. Using MetaML: A Staged Programming Language. In \textit{Advanced Functional Programming}, S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–239.

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In \textit{Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell} (Pittsburgh, Pennsylvania) (Haskell ’02). Association for Computing Machinery, New York, NY, USA, 1–16. \url{https://doi.org/10.1145/581690.581691}

Mark Shields and Simon Peyton Jones. 2001. Object-Oriented Style Overloading for Haskell (Extended Abstract). \textit{Electronic Notes in Theoretical Computer Science} 59, 1 (2001), 89–108. \url{https://doi.org/10.1016/S1571-0661(05)80455-4}

Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. \textit{IEEE Transactions on Software Engineering} SE-12, 1 (1986), 157–171. \url{https://doi.org/10.1109/TSE.1986.6312929}

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-Class State Change in Plaid. In \textit{Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications} (Portland, Oregon, USA) (OOPSLA ’11). Association for Computing Machinery, New York, NY, USA, 713–732. \url{https://doi.org/10.1145/2048066.2048122}

Markus Voelter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In \textit{Proceedings of the 34th International Conference on Software Engineering} (Zurich, Switzerland) (ICSE ’12). IEEE Press, Los Alamitos, CA, USA, 1449–1450.

Philip Wadler and Stephen Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In \textit{Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages} (Austin, Texas, USA) (POPL ’89). Association for Computing Machinery, New York, NY, USA, 60–76. \url{https://doi.org/10.1145/75277.752823}

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

Tetsuro Yamazaki, Tomoki Nakamaru, and Shigeru Chiba. 2022. Yet Another Generating Method of Fluent Interfaces Supporting Flat- and Sub-Chaining Styles. In \textit{Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering} (Auckland, New Zealand) (SLE 2022). Association for Computing Machinery, New York, NY, USA, 249–259. \url{https://doi.org/10.1145/3567512.3567533}

Tetsuro Yamazaki, Tomoki Nakamaru, Kazuhiro Ichikawa, and Shigeru Chiba. 2019. Generating a Fluent API with Syntax Checking from an LR Grammar. \textit{Proc. ACM Program. Lang.} 3, OOPSLA, Article 134 (Oct. 2019), 24 pages. \url{https://doi.org/10.1145/3360560}

Received 2022-10-28; accepted 2023-02-25