A Type System for Weighted Automata and Rational Expressions*

Akim Demaille[1], Alexandre Duret-Lutz[1], Sylvain Lombardy[2], Luca Saiu[3,1] and Jacques Sakarovitch[3]

[1] LRDE, EPITA, {akim,adl}@lrde.epita.fr
[2] LaBRI, Institut Polytechnique de Bordeaux, Sylvain.Lombardy@labri.fr
[3] LTCI, CNRS / Télécom-ParisTech, {saiu,sakarovitch}@telecom-paristech.fr

Abstract. We present a type system for automata and rational expressions, expressive enough to encompass weighted automata and transducers in a single coherent formalism. The system allows to express useful properties about the applicability of operations including binary heterogeneous functions over automata.
We apply the type system to the design of the Vaucanson 2 platform, a library dedicated to the computation with finite weighted automata, in which genericity and high efficiency are obtained at the lowest level through the use of template metaprogramming, by letting the C++ template system play the role of a static type system for automata. Between such a low-level layer and the interactive high-level interface, the type system plays the crucial role of a mediator and allows for a cleanly-structured use of dynamic compilation.

1 Introduction

Vaucanson[4] is a free software[5] *platform* dedicated to the computation of and with *finite automata*. It is designed with several use cases in mind. First and foremost it must support experiments by automata theory researchers. As a consequence, *genericity* and *flexibility* have been goals since day one: automata and transducers must support any kind of semiring of weights, and labels must not be restricted to just letters. In order to demonstrate the computational qualities of algorithms, *performance* must also be a main concern. To enforce this we aim, eventually, at applying Vaucanson to linguistics, whose problems are known for their size; on this standpoint we share goals with systems such as OpenFST [2]. Finally our platform should be *easy to use* by teachers and students in language theory courses (a common goal with FAdo [3]), which also justifies our focus on rational expressions.

* This is not the officially submitted "final" version of the paper, despite the content being the same but for an important correction in Fig. 4, added after publication and hence absent in the "final" version. The final publication is available at http://link.springer.com/chapter/10.1007%2F978-3-319-08846-4_12.

[4] Work supported by ANR Project 10-INTB-0203 VAUCANSON 2.

[5] http://vaucanson.lrde.epita.fr

Among our goals flexibility and efficiency are potentially in conflict. The main objective of this work is demonstrating how to reconcile them, and how to use a type system to manage such complexity.

Aiming at both efficiency and flexibility essentially dictates the architecture: the software needs to be rigidly divided into *layers*, varying in comfort and speed.

The bottom layer (named *static*) is a C++ library. For the sake of efficiency the classical object-oriented run-time method dispatch (associated to the C++ `virtual` keyword) is systematically avoided, instead achieving compile-time code generation by using `template` metaprogramming [1]. This results in a *closed world*: new types of automata require the compilation of dedicated code.

At the opposite end of the spectrum, the topmost layer is based on IPython [6]. It is visual (automata are displayed on-screen) and, most importantly, interactive: the user no longer needs to write a C++ or even a Python program, and instead just interacts with the system using Python as a command language. In such a high-level environment the closed-world restriction would be unacceptable, resulting as it would in error messages such as "this type of automaton is not supported; please recompile and rerun". To address this issue VAUCANSON uses on-the-fly generation and compilation of code, relying on our type system in a fundamental way.

This paper builds on top of ideas introduced last year [4][6]. However, in that work contexts were partitioned and entities of different types could not be mixed together. In particular algorithms such as the union of automata were "homogenous": operands had all the same type, which was that of the result. The contribution of this paper is to introduce support for heteregeneous types: the definition of a type calculus, its implementation and, to gain full benefit from it, dynamic code generation.

This paper is structured as follows. In Sec. 2 we describe the types of weighted automata, rational expressions and their components. Then, in Sec. 3, we study how types relate to one another and how to type operations over automata. We introduce the implementation counterpart of types in Sec. 4, which also explains how run-time compilation reconciles performances and flexibility. Sec. 5 discusses the pros and cons of the current implementation.

2 Typing Automata and Rational Expressions

Computing with weighted automata or rational expressions entails reasoning about *types*. We should have a system strong enough to detect some unmet preconditions (for instance applying subset construction on an automaton weighted in $\mathbb{Z}$), and at the same time expressive enough to encompass many different kinds of automata, including transducers.

[6] Names and notations have slightly changed. We now name "Value/ValueSet" the core design principle in Vaucanson, rather than "Element/ElementSet". For consistency with POSIX regular expression syntax, curly braces now denote power: '`a{2}`' means aa instead of $a \cdot 2$, which is now written '`a<2>`'. Similarly, '`a(*min,max)`' is now written '`a{min,max}`'.

2.1 Weighted Automata

Usually a weighted automaton $\mathcal{A}$ is defined as a sextuple $(A, \mathbb{K}, Q, I, F, E)$, A being an alphabet (a finite set of symbols), $\mathbb{K}$ a semiring, Q a finite set of states, I/F initial/final (partial) functions $Q \to \mathbb{K}$, and E a (partial) function in $Q \times A \times Q \to \mathbb{K}$. With such a definition, the generalization to transducers involves turning the sextuple into a septuple by adding a second *output* alphabet, changing the transition function domain to also take output labels into account, among the rest. Independently from transducers, definitions also need variants for many alternative cases, such as admitting the empty word as an input or output label. In VAUCANSON this variability is captured by *contexts*, each composed of one *LabelSet* and one *WeightSet*.

Different *LabelSets* model multiple variations on *labels*, members of a monoid:

letterset Fully defined by an alphabet A, its labels being just letters. It is simply denoted by A. It corresponds to the usual definition of an NFA.

nullableset Denoted by $A^?$, also defined by an alphabet A, its labels being either letters or the empty word. This corresponds to what is often called ε-NFAs.

wordset Denoted by A^*, also defined by an alphabet A, its labels being (possibly empty) words on this alphabet.

oneset Denoted by $\{1\}$, containing a single label: 1, the empty word.

tupleset Cartesian product of LabelSets, $L_1 \times \cdots \times L_n$. This type implements the concept of transducers with an arbitrary number of "tapes".

In the implementation LabelSets define the underlying monoid operations, and a few operators such as comparison.

A *WeightSet* is a semiring whose operations determine how to combine weights when evaluating words. Examples of WeightSets include $\langle \mathbb{B}, \vee, \wedge \rangle$, the family $\langle \mathbb{N}, +, \times \rangle$, $\langle \mathbb{Z}, +, \times \rangle$, $\langle \mathbb{Q}, +, \times \rangle$, $\langle \mathbb{R}, +, \times \rangle$ and tropical semirings such as $\langle \mathbb{Z} \cup \{\infty\}, \min, + \rangle$; moreover tuplesets also allow to combine WeightSets, making weight tuples into weights.
In the implementation a WeightSet defines the semiring operations and comparison operators, plus some feature tests such as "star-ability" [5].

We may finally introduce contexts, and the definition of automata used in VAUCANSON — a triple corresponding to its type (context), its set of states and its set of transitions.

Definition 1 (Context). *A* context *C is a pair (L, W), denoted by $L \to W$, where:*
- *L is a* LabelSet*, a subset of a monoid,*
- *W is a* WeightSet*, a semiring.*

Definition 2 ((Typed, Weighted) Automaton). *An automaton $\mathcal{A}$ is a triple (C, Q, E) where:*
- *$C = L \to W$ is a context;*
- *Q is a finite set of states;*

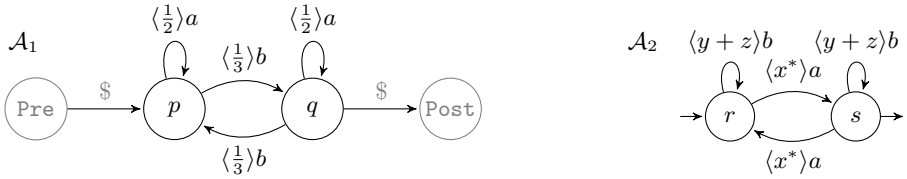

Fig. 1: Two (typed) automata: $\mathcal{A}_1$, whose context is $C_1 = \{a, b, c\} \to \mathbb{Q}$, and $\mathcal{A}_2$, whose context is $C_2 = \{a, b, d\} \to \mathsf{RatE}[\{x, y, z\} \to \mathbb{B}]$, i.e., with rational expressions as weights. In $\mathcal{A}_1$ we reveal the `Pre` and `Post` hidden states.

– *E is a (partial) function whose domain represents the set of transitions, in:* $(Q \times L \times Q) \cup (\{\texttt{Pre}\} \times \{\$\} \times Q) \cup (Q \times \{\$\} \times \{\texttt{Post}\}) \to (W \setminus \{0\})$.

Notice that the initial and final functions are embedded in the definition of E through two special states —the *pre-initial* and *post-final* states `Pre` and `Post`— and a special label not part on L and only occurring on *pre-transitions* (transitions from `Pre`) and *post-transitions* (transitions from `Post`). This somewhat contrived definition actually results in much simpler data structures and algorithms: with a unique `Pre` and a unique `Post` there is no need to deal with initial and final weights in any special way. On Fig. 1, automaton $\mathcal{A}_1$ is drawn with explicit `Pre` and `Post` states, while $\mathcal{A}_2$ is drawn without them.

2.2 Rational Expressions

Definition 3 ((Typed, Weighted) Rational Expression). *A rational expression $\mathcal{E}$ is a pair (C, E) where:*

– *$C = L \to W$, is a context,*
– *E is a term built from the following abstract grammar*

$$\mathsf{E} \coloneqq 0 \mid 1 \mid \ell \mid \mathsf{E} + \mathsf{E} \mid \mathsf{E} \cdot \mathsf{E} \mid \mathsf{E}^* \mid \langle w \rangle \mathsf{E} \mid \mathsf{E} \langle w \rangle$$

where $\ell \in L$ is any label, and $w \in W$ is any weight.

The set of rational expressions of type $L \to W$ is denoted by $\mathsf{RatE}[L \to W]$, and called a *ratexpset*. With a bit of caution rational expressions can be used as weights, as exemplified by automaton $\mathcal{A}_2$ in Fig. 1: equipped with the sum of rational expressions as sum, their concatenation as product, 0 as zero, and 1 as unit, it is very close to being a semiring[7].

Rational expressions may also serve as labels, yielding what is sometimes named *Extended Finite Automata* [3], a convenient internal representation to perform, for example, state elimination, a technique useful to extract a rational expression from an automaton. So, just like tuplesets, ratexpsets can be used as either a WeightSet or a LabelSet.

[7] Ratexpset do *not* constitute a semiring for lack of, for instance, equality between two rational expressions; however rational expressions provide an acceptable approximation of *rational series* [7, Chap. III], the genuine corresponding semiring.

```
⟨context⟩ ::= ⟨labelset⟩ "→" ⟨weightset⟩
⟨labelset⟩ ::= "{1}" | ⟨alphabet⟩ | ⟨alphabet⟩ "?" | ⟨alphabet⟩ "*"
             | ⟨ratexpset⟩ | ⟨labelset⟩ ×···× ⟨labelset⟩
⟨weightset⟩ ::= "𝔹" | "ℕ" | "ℤ" | "ℚ" | "ℝ" | "ℤ_min"
              | ⟨ratexpset⟩ | ⟨weightset⟩ ×···× ⟨weightset⟩
⟨ratexpset⟩ ::= "RatE" ⟨context⟩
```

Fig. 2: A Grammar of Types

Fig. 2 shows the precise relation among the different entities introduced up to this point: LabelSets, WeightSets, contexts, ratexpsets.

3 The Type System

3.1 Operations on Automata

Several binary operations on automata exist: union, concatenation, product, shuffle and infiltration products, to name a few. To demonstrate our purpose we consider the simplest one, i.e., the union of two automata, whose behavior is the sum of the behavior of each operand.

Definition 4 ((Homogeneous) Union of Automata). *Let* $\mathcal{A}_1 = (C, Q_1, E_1)$ *and* $\mathcal{A}_2 = (C, Q_2, E_2)$ *be two automata of the same type* C*.* $\mathcal{A}_1 \cup \mathcal{A}_2$ *is the automaton* $(C, Q_1 \cup Q_2, E_1 \cup E_2)$*.*

Def. 4 is simple, but has the defect of requiring the two argument automata to have exactly the same type. Overcoming this restriction and making operations such as automata union more widely applicable is a particularly stringent requirement in an interactive system (Sec. 4.3).

Automata union can serve as a good example to convey the intuition of heterogeneous operation typing: if its two operands have LabelSets with different alphabets, the result LabelSet should have their *union* as alphabet; if one operand is an NFA and the other a ε-NFA, their union should also be a ε-NFA. It is also reasonable to define the union between an automaton with spontaneous transitions only (oneset) and an NFA (letterset) as a ε-NFA (nullableset) — a type different from *both* operands', and intuitively "more general" than either.

Much in the same way, some WeightSets are straightforward to embed into others: $\mathbb{Z}$ into $\mathbb{Q}$, and even $\mathbb{Q}$ into $\mathsf{RatE}[L \to \mathbb{Q}]$. Then, let two automata have weights in $\mathbb{Q}$ and $\mathsf{RatE}[L \to \mathbb{Z}]$; their union should have weights in *the least WeightSet that contains both* $\mathbb{Q}$ *and* $\mathsf{RatE}[L \to \mathbb{Z}]$, which is to say $\mathsf{RatE}[L \to \mathbb{Q}]$. Once more the resulting type is new: it does not match the type of either operand.

3.2 The hierarchy of types

The observations above can be captured by introducing a *subtype* relation as a partial order on LabelSets, WeightSets and contexts, henceforth collectively

denoted as *ValueSets*. We write $V_1 <: V_2$ to mean that V_1 is a subtype of V_2; in this case each element of V_1 may be used wherever an element of V_2 would be expected, and we have in particular that $V_1 \subseteq V_2$. Notice that this makes our relation reflexive, so for every ValueSet V we have that $V <: V$.

For simplicity we will focus on free monoids only. Let A, B be any alphabets such that $A \subseteq B$. Then we define:

$$\{1\} <: A^? \qquad A <: A^? \qquad A^? <: A^*$$
$$A <: B \qquad A^? <: B^? \qquad A^* <: B^*$$

For WeightSets, if the WeightSet W_1 is a sub-semiring of W_2, it trivially holds that $W_1 <: W_2$; therefore $\mathbb{N} <: \mathbb{Z} <: \mathbb{Q} <: \mathbb{R}$. The WeightSet $\mathbb{B}$, as the WeightSet of language recognizers, is worthy of special treatment; in particular it is convenient to allow heterogeneous operations between automata over $\mathbb{B}$ and automata over other WeightSets, which yields:

$$\mathbb{B} <: \mathbb{N} <: \mathbb{Z} <: \mathbb{Q} <: \mathbb{R} \qquad \mathbb{B} <: \mathbb{Z}_{\min} \tag{1}$$

This allows for instance to restrict the domain of a series realized by a weighted automaton to the rational language described by a Boolean automaton. For this reason it is desirable to have $\mathbb{B}$ at the bottom of the WeightSet hierarchy, so that it can be promoted to any other WeightSet simply by mapping false to the WeightSet zero, and true to its unit. However such conversion requires care and should not be used blindly; in particular converting an ambiguous Boolean automaton to another WeightSet leads in general to an automaton which does *not* realize the characteristic series of the language recognized by the original.

A context C_1 is a subtype of a context C_2 if C_1 has a LabelSet and a WeightSet which are respectively subtypes of the LabelSet and WeightSet of C_2.

$$(L_1 \to W_1) <: (L_2 \to W_2) \quad \text{iff} \quad L_1 <: L_2 \text{ and } W_1 <: W_2 \tag{2}$$

As of today tuples of ValueSets do not mix with other values:

$$(V_1 \times \cdots \times V_n) <: (V'_1 \times \cdots \times V'_n) \quad \text{iff} \quad (V_i <: V'_i) \text{ for all } 1 \leq i \leq n \tag{3}$$

Interestingly, rational expressions can play the role of both labels and weights:

$$\begin{aligned} \mathsf{RatE}[C_1] <: \mathsf{RatE}[C_2] \quad &\text{iff} \quad C_1 <: C_2 \\ L_1 <: \mathsf{RatE}[L_2 \to W_2] \quad &\text{iff} \quad L_1 <: L_2 \\ W_1 <: \mathsf{RatE}[L_2 \to W_2] \quad &\text{iff} \quad W_1 <: W_2 \end{aligned} \tag{4}$$

The subtype relations between LabelSets are summarized in Fig. 3. If two LabelSets L_1 and L_2 admit a least upper bound (*resp.* a greatest lower bound), we call it the join (*resp.* the meet) of these two LabelSets and we denote it by $L_1 \vee L_2$ (*resp.* the $L_1 \wedge L_2$). The cases where no join or meet exists correspond in practice to compilation errors about undefined cases. The join and meet operations extend naturally to other ValueSets such as WeightSets, tuples,

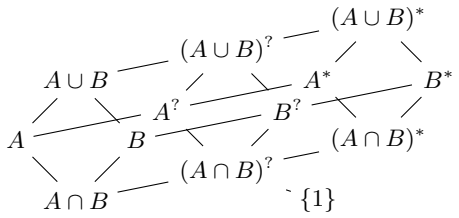

Fig. 3: The Hasse diagram of the LabelSets generated by the two alphabets A and B showing, for instance, that $A^? \vee B = (A \cup B)^?$.

contexts and rational expressions, as per Equations (1) to (4)). For instance, for any LabelSet L_1, L_2 and any WeightSet W_1, W_2:

$$\begin{aligned}
\mathsf{RatE}[L_1 \to W_1] \vee L_2 &\coloneqq \mathsf{RatE}[(L_1 \vee L_2) \to W_1] \\
\mathsf{RatE}[L_1 \to W_1] \vee W_2 &\coloneqq \mathsf{RatE}[L_1 \to (W_1 \vee W_2)] \\
\mathsf{RatE}[L_1 \to W_1] \vee \mathsf{RatE}[L_2 \to W_2] &\coloneqq \mathsf{RatE}[(L_1 \to W_1) \vee (L_2 \to W_2)]
\end{aligned}$$

At this point we are ready to describe typing for binary operations on heterogeneous automata more formally. An operation on two automata with contexts $L_1 \to W_1$ and $L_2 \to W_2$ will yield a result with context $(L_1 \vee L_2) \to (W_1 \vee W_2)$. As an example we can extend Def. 4 into:

Definition 5 (Heterogeneous Union of Automata). *Let $\mathcal{A}_1 = (C_1, Q_1, E_1)$ and $\mathcal{A}_2 = (C_2, Q_2, E_2)$ be two automata. $\mathcal{A}_1 \cup \mathcal{A}_2 \coloneqq (C_1 \vee C_2, Q_1 \cup Q_2, E_1 \cup E_2)$.*

3.3 Type restriction

The specific semantics of some binary operations let us characterize the result type more precisely. For instance spontaneous-transition-removal applied to an automaton with LabelSet $A^?$ returns a *proper* automaton, i.e., an automaton with LabelSet A. Another interesting example is the product of automata labeled by letters[8], whose behavior is the Hadamard product of series of the behavior of each operand, if the WeightSet is commutative.

Definition 6 (Product of Automata). *Let $\mathcal{A}_1 = ((L_1 \to W_1), Q_1, E_1)$ and $\mathcal{A}_2 = ((L_2 \to W_2), Q_2, E_2)$ be two automata, where L_1 and L_2 are* lettersets. *$\mathcal{A}_1 \,\&\, \mathcal{A}_2$ is the accessible part of the automaton $(C_\&, Q_\&, E_\&)$ where $C_\& = (L_1 \wedge L_2) \to (W_1 \vee W_2)$, $Q_\& = Q_1 \times Q_2$, and*

$$((q_1, q_2), \ell, (q_1', q_2')) \in \mathsf{Dom}(E_\&) \quad \text{iff} \begin{cases} (q_1, \ell, q_1') \in \mathsf{Dom}(E_1), \\ (q_2, \ell, q_2') \in \mathsf{Dom}(E_2); \end{cases}$$

$$E_\&((q_1, q_2), \ell, (q_1', q_2')) = E_1(q_1, \ell, q_1') \cdot E_2(q_2, \ell, q_2').$$

[8] The product operation can actually be extended to nullablesets, using a more complex algorithm related to weighted transducer composition.

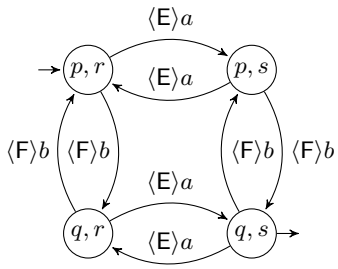

Fig. 4: $\mathcal{A}_3 = \mathcal{A}_1 \,\&\, \mathcal{A}_2$ (see Fig. 1), with $\mathsf{E} = \langle\frac{1}{2}\rangle x^*$ and $\mathsf{F} = \langle\frac{1}{3}\rangle(y+z)$. Its type is $C_3 = \{a,b\} \to \mathsf{RatE}[\{x,y,z\} \to \mathbb{Q}]$.

Like for other binary operations it would be correct to describe the type of the result of a product as the join of its operand types; however in this case the specific operation semantics permits us to be more precise: a product result transition is created if and only if labels match in the two argument automata, and therefore the result LabelSet happens to lie in the *meet* of the argument LabelSets. By contrast, each weight is computed as the product of argument weights, in general belonging to two different WeightSets: the WeightSet of the product hence lies in the *join* of the argument WeightSets.

Fig. 4 shows the heterogeneous product of $\mathcal{A}_1$ and $\mathcal{A}_2$ from Fig. 1.

4 Implementation Facet

4.1 The Value/ValueSet Design Principle

The implementation of Vaucanson closely follows its algebraic design illustrated in Sec. 2 in terms of labels, weights, automata and rational expressions. Other entities not shown here also exist, such as polynomials.

In a typical object-oriented implementation each of these concepts would be implemented as a class, possibly templated. For instance a Boolean weight would be an instance of some class `boolean_weight` having a `bool` attribute. However some of these concepts require run-time meta-data; for instance a letterset needs a set of letters, so a `letter_label` would aggregate not only a `char` for the label, but also *the whole alphabet*, as a `char` vector. As a context aggregates a LabelSet and a WeightSet it requires run-time meta-data as well, and since rational expressions can also be used as weights, they, too, depend on run-time meta-data. Therefore weights and LabelSets both need to be associated to meta-data at run time.

However it would result in an unacceptable penalty to have every instance carry even a mere pointer to meta-data such as an alphabet (a simple `char` label, because of alignment, would then require at least eight bytes, a 8× space penalty on a 32-bit architecture!). To cut this Gordian knot, as a design principle, we split traditional values into *Value/ValueSet* pairs. The value part is but the implementation of a datum; the *ValueSet*, on the other hand, stores *only one copy* of the meta-data related to the type (such as the alphabet) and performs

the operations on values (such as `+` for $\mathbb{Z}$ and `min` for $\mathbb{Z}_{\min}$) without relying on dynamic dispatch.

This design is asymmetric: ValueSets implement the operations on their Values; conversely from a Value there is no means to reach the corresponding ValueSet. Values may in fact ultimately come down to plain data types like `int` or `char`.

Following the Value/ValueSet design principle, VAUCANSON implements LabelSets such as `oneset`, `letterset<`*generatorset*`>`[9], `nullableset<`*generatorset*`>`, `wordset<`*generatorset*`>`, and WeightSets such as `b`, `z`, ..., `ratexpset<`*context*`>`; finally, `tupleset<`*ValueSet*$_1$`, ..., `*ValueSet*$_n$`>` implements Cartesian products.

4.2 Computations on Types

Two different sets of routines are needed to support heterogeneous operations such as the product and sum of automata or rational expressions: first a computation on types based on join and meet, then a conversion of values to these types.

The computation of joins and meets on basic types is straightforward.

```
r join(const r&, const b&) { return r(); }
r join(const r&, const z&) { return r(); }
r join(const r&, const q&) { return r(); }
```

The code snippet above states that $\mathbb{R} \vee W \coloneqq \mathbb{R}$ for $W \in \{\mathbb{B}, \mathbb{Z}, \mathbb{Q}\}$. Composite types such as rational expressions, tuples or even contexts follow the same pattern, but are computed recursively.

Some features new to C++11 let us express the product context computation (as per Def. 6) quite cleanly, as follows:

```
template <typename LhsLabelSet, typename LhsWeightSet,
          typename RhsLabelSet, typename RhsWeightSet>
auto product_ctx(const context<LhsLabelSet, LhsWeightSet>& lhs,
                 const context<RhsLabelSet, RhsWeightSet>& rhs)
  -> context<decltype(meet(lhs.LabelSet(),  rhs.LabelSet())),
             decltype(join(lhs.WeightSet(), rhs.WeightSet()))>
{
  auto ls = meet(lhs.LabelSet(),  rhs.LabelSet());
  auto ws = join(lhs.WeightSet(), rhs.WeightSet());
  return {ls, ws};
}
```

Two WeightSets are involved in the process of value conversions: the source one, which is used below as a key to select the proper `conv` routine, and the

[9] *generatorset* provides type and value information on the monoid generators; in practice this corresponds to the type of characters and the alphabet, as a vector of characters of the appropriate type.

destination one (`r` in the following example). Type conversion may require run-time computation such as the floating-point division below, or even something more substantial like the construction of a rational expression in other cases.

```
class r {
  using value_t = float;
  ...
  value_t conv(b, b::value_t v) { return v; }
  value_t conv(z, z::value_t v) { return v; }
  value_t conv(q, q::value_t v) { return value_t(v.num)/value_t(v.den); }
  ...
};
```

The process generalizes in a natural way to the case of composite types.

The `join`, `meet` and `conv` functions are used in the implementation of binary operations such as the product, shown below as an example[10]; the idea is to first compute the result type `ctx`, and then use it to create the result automaton `res`.

```
template <typename Ctx1, typename Ctx2>
auto product(const automaton<Ctx1>& lhs, const automaton<Ctx2>& rhs)
  -> ...
{
  auto ctx = product_ctx(lhs.context(), rhs.context());
  auto res = make_automaton(ctx);
  auto ws = res.WeightSet();  // a shorthand to the resulting WeightSet.
  ...
  return res;
}
```

The core of the algorithm consists in an iteration over each reachable left-right pair of states (`lhs_src`, `rhs_src`); for each pair of transitions with the same label from `lhs_src` and `rhs_src`, it adds a transition from the source state pair to the destination state pair, with the same label and the product of weights as weight.

```
for (auto lhs_trans : lhs.out(lhs_src))
  for (auto rhs_trans : rhs.out(rhs_src, lhs_trans.label))
  {
    auto weight = ws.mul(ws.conv(lhs.WeightSet(), lhs_trans.weight),
                         ws.conv(rhs.WeightSet(), rhs_trans.weight));
    res.add_transition({lhs_src, rhs_src}, {lhs_trans.dst, rhs_trans.dst},
                       lhs_trans.label, weight);
  }
```

Three WeightSets play a role in the computation of the resulting `weight`: first `ws.conv(lhs.WeightSet(), lhs_trans.weight)` promotes the left-hand side weight from its original WeightSet `lhs.WeightSet()` to the resulting one `ws`, and likewise for the second weight; finally the resulting WeightSet multiplies the weights (`ws.mul(...)`). For instance in Fig. 4 there is a transition from state

[10] In the following code excerpts some details have been omitted for clarity.

(p, r) to state (p, s) with label a, and whose weight is the product of $\frac{1}{2}$ and x^*. The conversion of the first weight corresponds to '$C_3.W.\texttt{conv}(C_1.W, \frac{1}{2})$', which results in $\langle\frac{1}{2}\rangle 1$; likewise for the second weight: '$C_3.W.\texttt{conv}(C_2.W, \langle\mathbf{1}\rangle x^*)$ `=` $\langle\frac{1}{1}\rangle x^*$'. The resulting WeightSet, C_3 then multiplies them: '$C_3.W.\texttt{mul}(\langle\frac{1}{2}\rangle 1, \langle\frac{1}{1}\rangle x^*))$', i.e., $\langle\frac{1}{2}\rangle x^*$.

4.3 On-the-Fly Compilation

Code snippets shown so far are all part of the *static* layer, the statically-typed, lowest-level Application Program Interface (API) of VAUCANSON, which strictly follows the Value/ValueSet principle. As long as this API is used the compiler will take care of generating the appropriate versions of the routine for the types at hand, with no run-time overhead. Programming at this level however offers little flexibility: the program is written and then compiled, period. Moreover, types have to be explicitly spelled out in the program.

On top of this static layer, the *dyn* API takes care of the template parameter book-keeping, memory allocation and deallocation, and even re-unites split objects: for example a `dyn::ratexp` aggregates both a (static-level) rational expression and its (static-level) ratexpset. By design *dyn* only includes a handful of types such as `dyn::context`, `dyn::automaton`, `dyn::weight` and `dyn::label`: all the wide variety of static-level entities is collapsed into a few categories of objects carrying their own run-time type information (exposed to the user as `dyn::context` objects), so that operations can automatically perform their own conversions without exposing the user to the type system.

The *static/dyn bridge* works with registries, one per algorithm. They play a role similar to *virtual tables* in C++: to select the precise implementation of an algorithm that corresponds to the effective type of the operands. These registries are just dictionaries, mapping each given list of argument types to the corresponding specific (static) implementation. This mechanism and other details on the static/dyn bridge have been described in a previous work [4, Sec. 4.2]; its complete treatment is beyond the scope of this paper.

Several commonly-used basic contexts are precompiled — in other words registries are initially loaded for some specific types. However, not only the number of contexts is too large to permit a "complete" precompilation (24: 4 basic LabelSets times 6 WeightSets), but tupleset and ratexpset also let the user define an *unbounded* number of composite ones. Moreover, as demonstrated in Fig. 4, some operation results belong to contexts that were not even in the operands. For these reasons only some select contexts can be precompiled, which will certainly frustrate some users.

On top of dyn VAUCANSON offers IPython support (see Fig. 5). IPython is an enhanced interactive Python environment [6]. Thanks to specific hooks, entities such as rational expressions feature nice LaTeX-based rendering, and automata are rendered as pictures. This binding of dyn features the familiar Python object-oriented flavor as in "*`automaton`*`.minimize()`", and operator overloading as in "*`automaton & automaton`*". In such an interactive environment (similar to what

```
In [4]: ctx = vcsn.context("lal_char(ab)_ratexpset<lal_char(xyz)_b>"); ctx
Out[4]: {a, b} → RatE[{x, y, z} → 𝔹]

In [5]: r2 = ctx.ratexp('(<x*>b)*<y+z>a(<x*>b+<y+z>a(<x*>b)*<y+z>a)*'); r2
Out[5]: (⟨x*⟩b)* ⟨y + z⟩a (⟨x*⟩b + ⟨y + z⟩a (⟨x*⟩b)* ⟨y + z⟩a)*

In [6]: a2 = r2.derived_term().minimize()
        a1|a2
Out[6]:
```

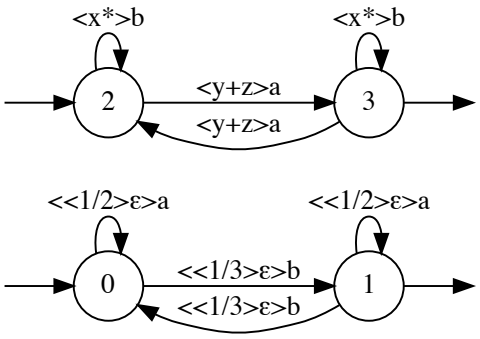

Fig. 5: The computation of $\mathcal{A}_1 \cup \mathcal{A}_2$ in the *IPython notebook* interface of VAUCANSON. The symbol ε denotes the empty word. Weights in $\mathbb{Q}$ such as $\frac{1}{2}$ have been lifted into the WeightSet of C_3: $\langle \frac{1}{2} \rangle \varepsilon \in \mathsf{RatE}[\{x, y, z\} \to \mathbb{Q}]$.

formal mathematics environments offer), working with just a finite, predefined set of types would be unacceptable.

To address these limitations VAUCANSON's *dyn* layer features run-time code generation, compilation, and dynamic object loading. The code generation is a simple translation from the context object into C++ code instantiating the existing algorithms for a given context and then entering into the appropriate registries. Once the context-plugin is successfully compiled and linked, it is loaded into the program via `dlopen`; however, differently from the usual practice, we do not need `dlsym` calls to locate plugin functions one by one; rather, when a plugin is loaded, its initialization code simply adds its functions to the registries. In other words a plugin compiled on the fly and loaded at run-time is treated exactly like precompiled contexts.

Because a call in IPython eventually resolves into a call in the static library, one benefits from both flexibility *and* efficiency — when C++ algorithms take most of the time; of course Python-heavy computations would be a different matter.

5 Future Work and Improvements

The subtype relation among semirings we introduced is natural; however a closer look at these definitions reveals that several mechanisms are involved, which may deserve more justification.

5.1 Syntactic and Semantic Improvements of Contexts

Contexts proved to be a powerful concept; however some early design decisions resulted in limitations, to be lifted in the near future.

First, the concrete syntax the user must use to define the context is cumbersome. For instance $C_3 = \{a, b\} \to \mathsf{RatE}[\{x, y, z\} \to \mathbb{Q}]$ has to be written `lal_char(ab)_ratexpset<lal_char(xyz)_q>` (see Fig. 5); a syntax closer to the mathematical notation would be an improvement.

Second, the *quantifiers* '`?`' and '`*`' should probably apply to an entire LabelSet, and not just to an alphabet like in Fig. 2:

```
⟨labelset⟩ ::= "{1}" | ⟨alphabet⟩ | ⟨labelset⟩ "?" | ⟨labelset⟩ "*"
             | ⟨ratexpset⟩ | ⟨labelset⟩ ×···× ⟨labelset⟩
```

which would allow to define, for instance, two-tape transducers whose labels are either a couple of letters, or the empty (two-tape) word: $(\{a, b\} \times \{x, y\})^?$.

Third, our implementation of automata does not follow the Value/ValueSet pattern, which prevents us from using them like other entities.

5.2 Dynamic Compilation Granularity

The compilation of plugins today is *coarse-grained*, in that we compile "all" the existing algorithms for a given context. This is at the same time too much, and not enough.

It is too much as it may suspend an interactive IPython session for half a minute even on a fast laptop, to compile and load the given context library; caching compiled code however makes this cost a one-time penalty.

It is not enough because algorithms such as union have an open set of possible signatures. The resulting type of the union of two automata might not be precompiled, in which case, for lack of support for the resulting context, the computation would fail. An unpleasant but effective workaround consists in warning the system, at runtime, that a given context will be needed.

To address both shortcomings we plan to support fine-grained plugins able to generate, compile and load code for *one* function with *one* signature.

6 Conclusion

We presented a type system for weighted automata and rational expressions — a novel feature to the best of our knowledge— currently implemented in our Vaucanson 2 system, but not coupled to any particular platform.

Types lie at the very foundation of our design. At the lowest level, where performance concerns are strong, we follow the *Value/ValueSet* principle and types parameterize C++ template structures and functions; there a calculus on types based on a subtype relation allows to define operations on automata of different types and handles value conversions. At a higher level types make up the bridge between the static low-level API and a dynamic one built on top of it.

Finally, run-time translation of types into C++ code allows to compile, generate, and load plugins during interactive sessions, for instance under IPython.

VAUCANSON 2 is free software. Its source code is available at `http://vaucanson.lrde.epita.fr`, along with virtual machine images to let users experiment and play with the system without need for an installation.

Acknowledgements

We wish to thank the anonymous reviewers for their helpful and constructive suggestions.

References

1. A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
2. C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *CIAA'07*, vol. 4783 of *LNCS*, pp. 11–23. Springer, 2007. `http://www.openfst.org`.
3. A. Almeida, M. Almeida, J. Alves, N. Moreira, and R. Reis. FAdo and GUItar. In *CIAA'09*, vol. 5642 of *LNCS*, pp. 65–74. Springer, 2009.
4. A. Demaille, A. Duret-Lutz, S. Lombardy, and J. Sakarovitch. Implementation concepts in Vaucanson 2. In *CIAA'13*, vol. 7982 of *LNCS*, pp. 122–133, July 2013. Springer.
5. S. Lombardy and J. Sakarovitch. The validity of weighted automata. *Int. J. of Algebra and Computation*, 23(4):863–914, 2013.
6. F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. `http://ipython.org`.
7. J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009. Corrected English translation of *Éléments de théorie des automates*, Vuibert, 2003.