On Implementing the Forlan Formal Language Theory Toolset in Standard ML

Alley Stoughton

Forlan Project

- Forlan is a toolset for experimenting with the objects of formal language theory: regular expressions, finite automata, grammars, a universal programming language, etc.
- Forlan is implemented in Standard ML, as a library on top of Standard ML of New Jersey.
- Forlan is used interactively, but users can extend it via SML functions.
- Forlan currently consists of about 12,000 lines of SML code, distributed over about 40 modules. Also a Java application for editing finite automata.
- The Forlan Project also includes a draft textbook, and I've tried to minimize the notational distance between toolset and book.

Rationale for Choosing SML

- I chose SML as the implementation language for Forlan because:
  - SML's concepts and notation are similar to those of mathematics;
  - and yet it is easy to write efficient code in SML.
- My hope was that it would prove possible to:
  - naturally and simply express the definitions of formal language theory in SML; and
  - implement the algorithms of formal language theory in ways that were simultaneously natural and reasonably efficient.
- My talk will assess how well SML and SML/NJ supported achieving this goal.

Conclusions

My conclusions are largely positive:

- It was typically easy to naturally and efficiently implement the book's mathematics in SML, with both the core and module languages serving well.
- For example, most of the objects of formal language theory are naturally implemented as abstract types, and this was neatly done using structures, signatures and opaque ascription.
- SML/NJ proved robust and provided sufficient speed.

Minor Gaps Between Book and Toolset

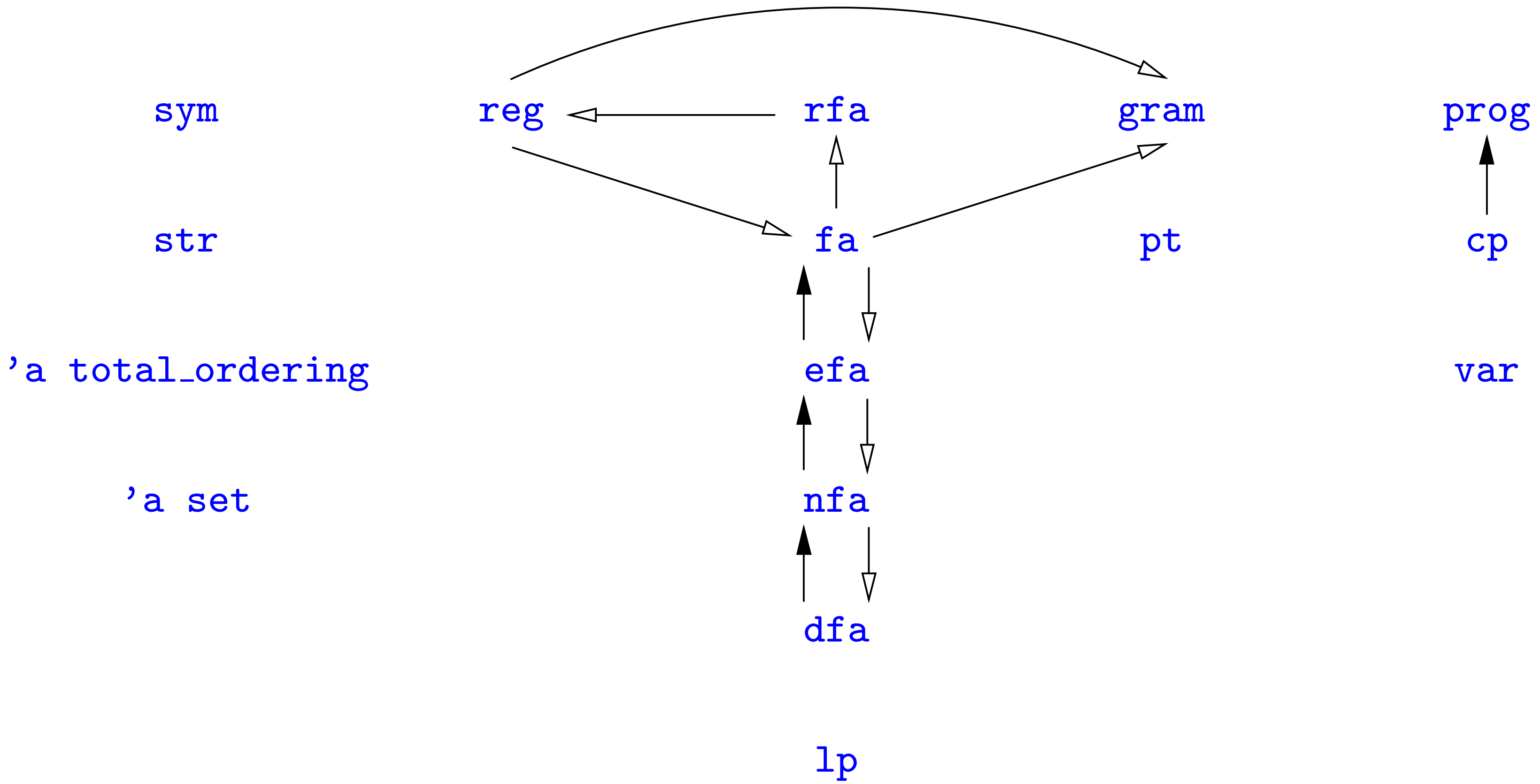
- Lack of `nat` type.
- `int` can't be made arbitrary precision in SML/NJ.
- Can't use successor in patterns:

$$
\begin{aligned}
f\,0 &= \cdots\,, \\
f(n+1) &= \cdots\, f\,n\, \cdots\,, \text{ for all } n \in \mathbb{N}
\end{aligned}
$$

versus

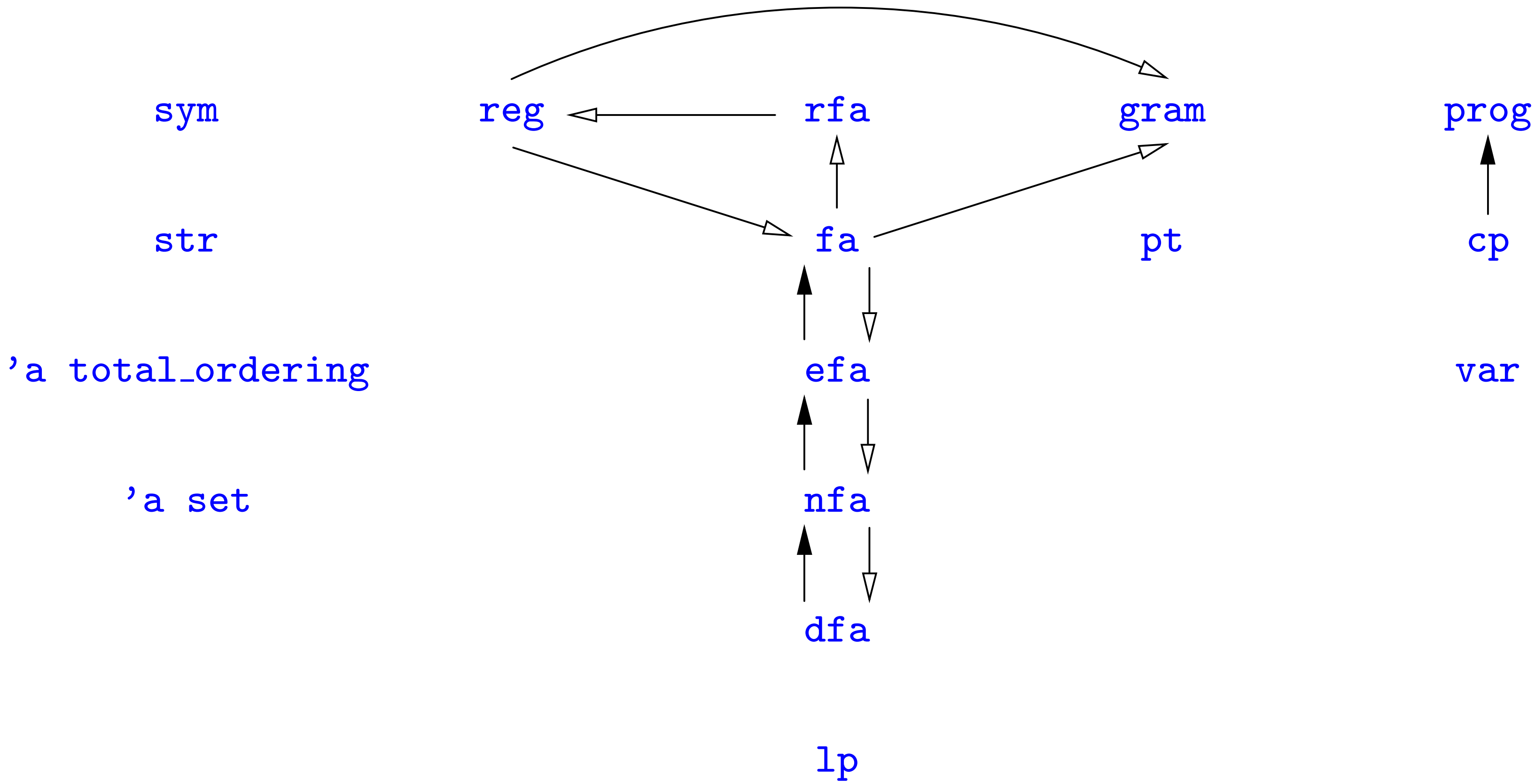
```
fun f 0 = ...
  | f n = ... f(n - 1) ...
```

Main Types



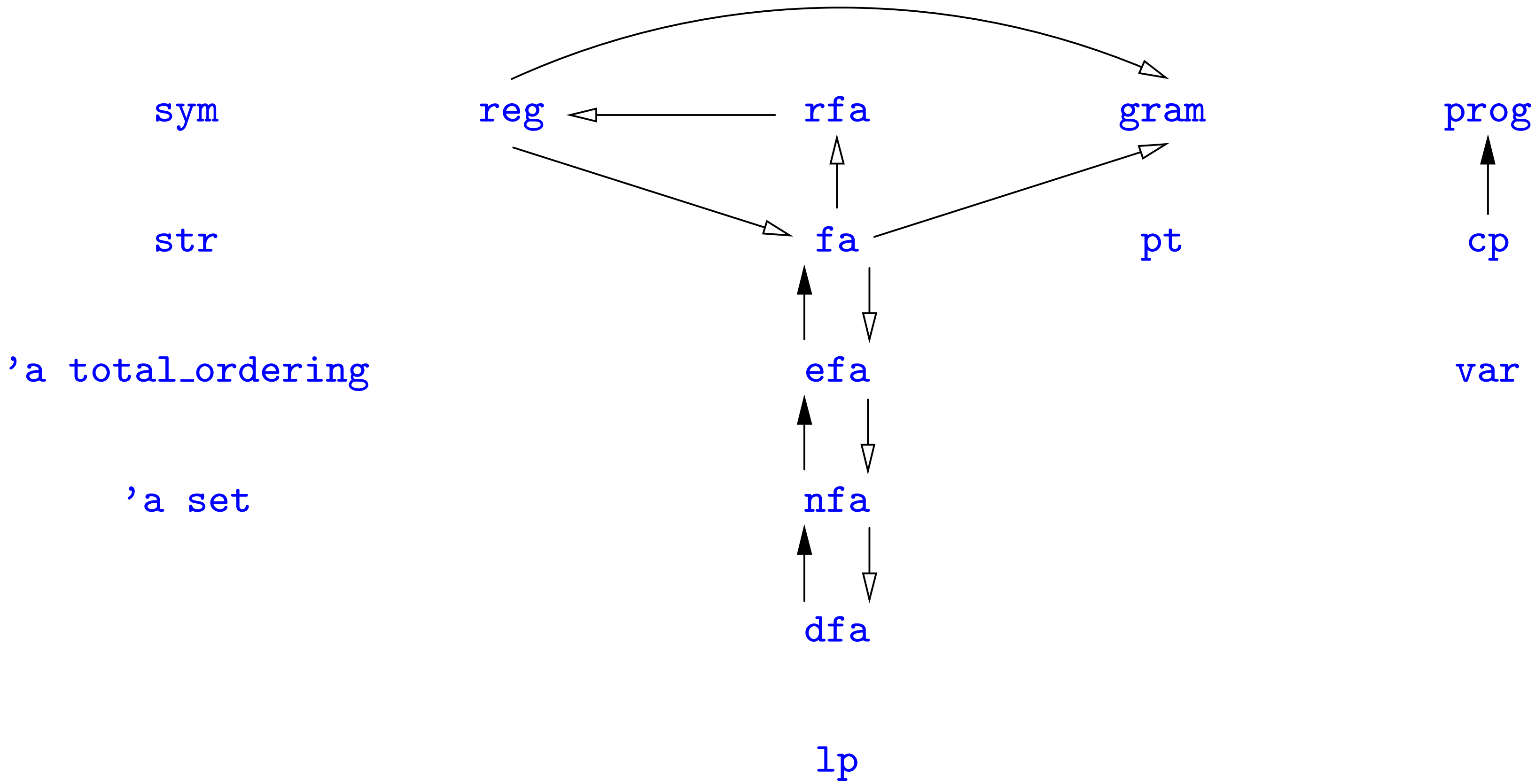
- Open arrows are conversions; Closed arrows are injection/projection pairs.

Main Types



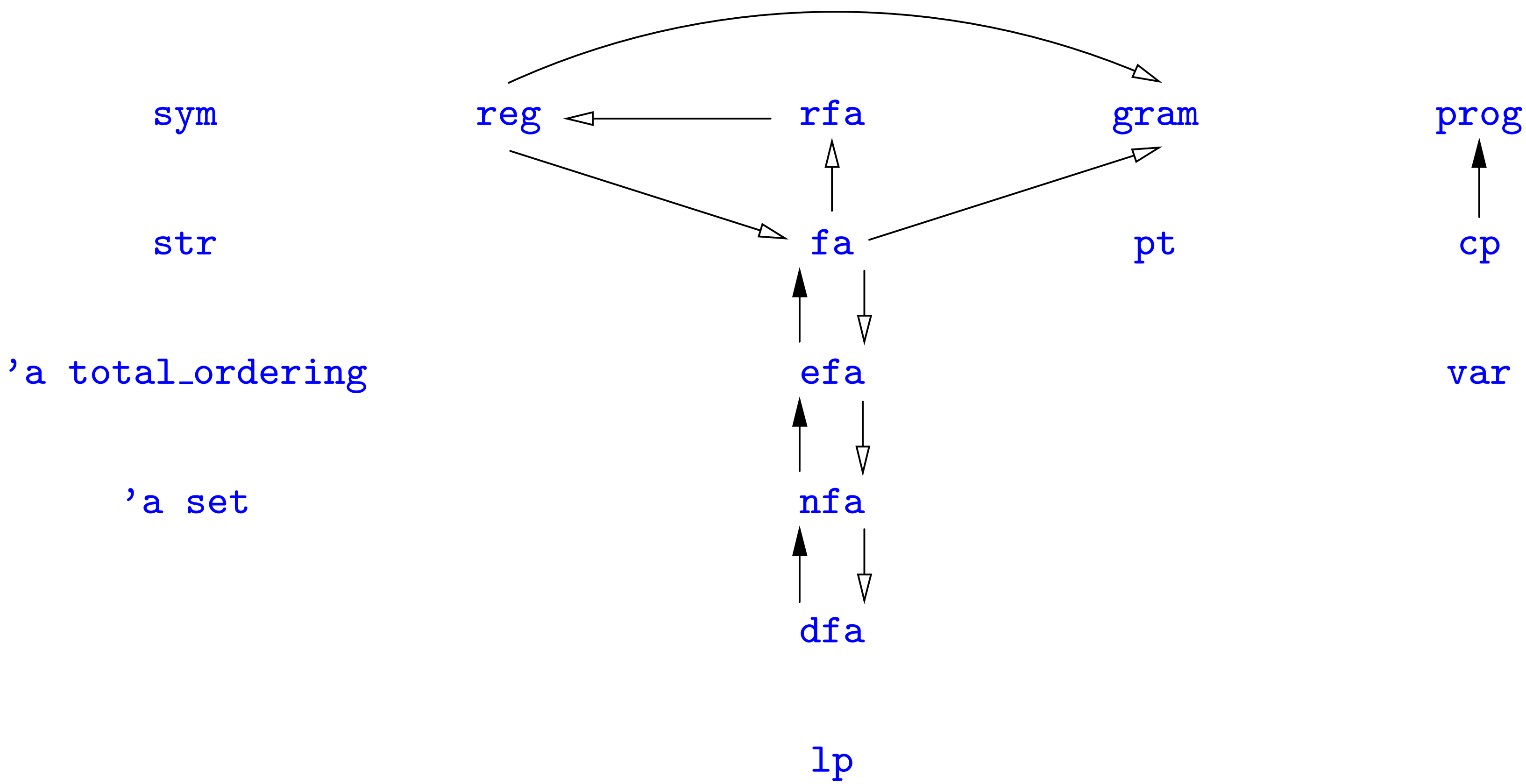
- Symbols: including 0–9, a–z and A–Z.
- Strings: `type str = sym list`

Main Types



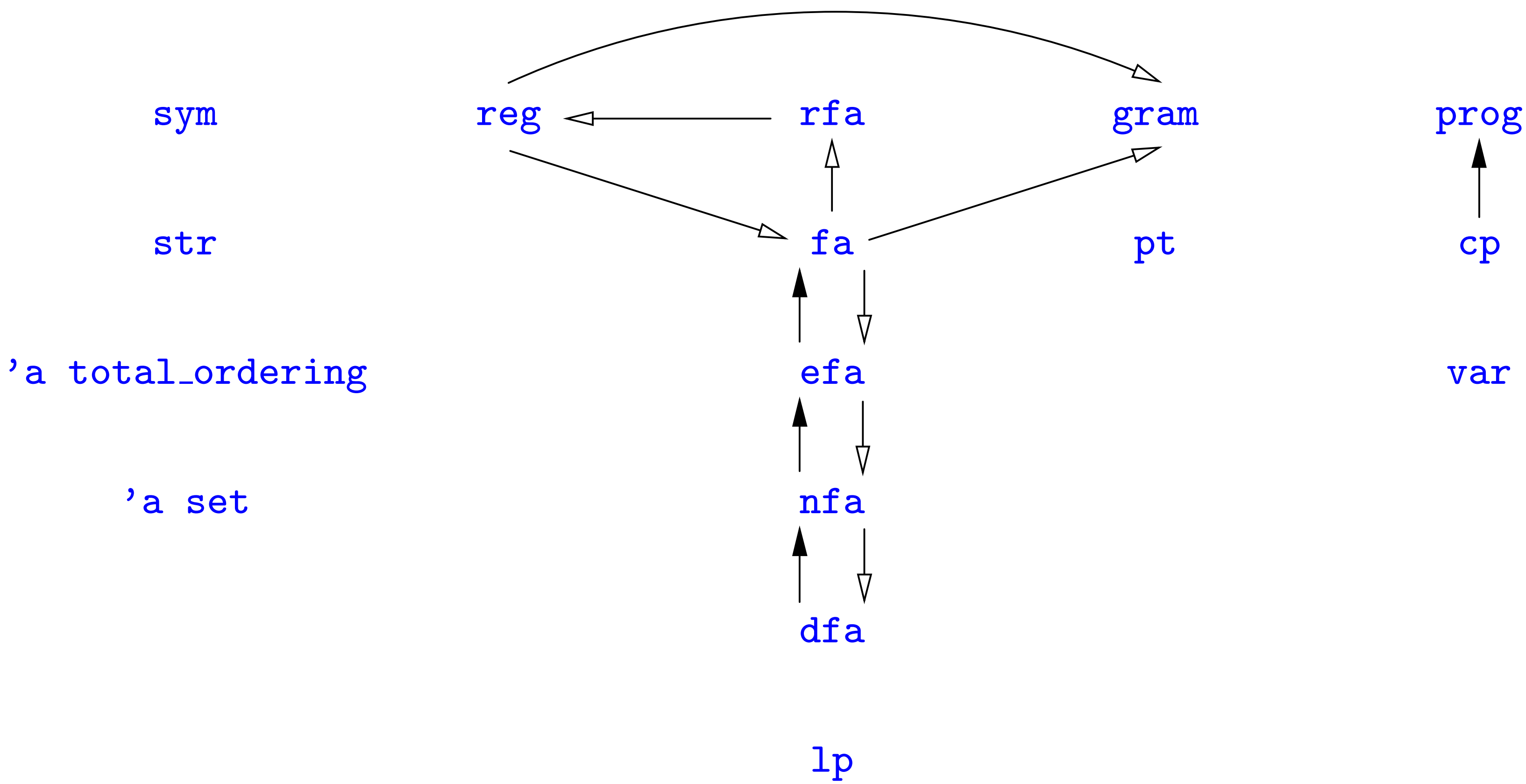
- `type 'a total_ordering = 'a * 'a -> order` plus axioms.
- Ordered sets.
- Regular expressions.

Main Types



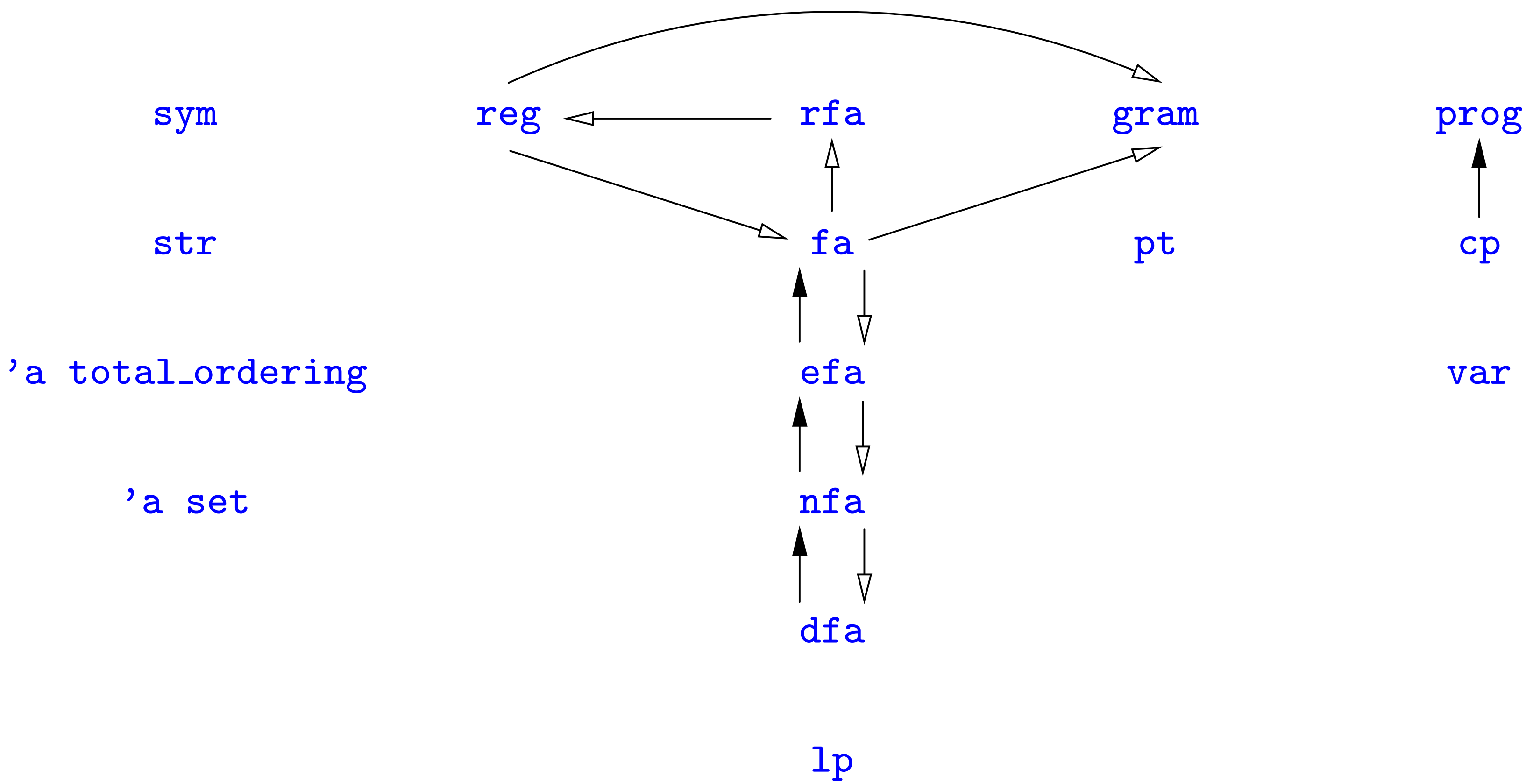
- Finite automata meaning given using labeled paths.
- `rfa`: transitions labeled by regular expressions.
- `fa`: transitions labeled by strings.

Main Types



- `efa`: transitions labeled by strings of length at most 1.
- `nfa`: transitions labeled by strings of length 1.
- `dfa`: `nfa` plus determinism.

Main Types



- Context-free grammars given meaning using parse trees.
- Functional programs for computability theory.
- Closed programs—no free variables—can be evaluated.

Example

Define

$$\mathbf{AllFollow} \in \{0,1\}^* \times \{0,1\}^* \to \mathcal{P}(\{0,1\}^*)$$

by: for all $x, y \in \{0,1\}^*$,

$$\begin{aligned}\mathbf{AllFollow}(x, y) \\ &= \{w \in \{0,1\}^* \mid \text{for all } u, v \in \{0,1\}^*, \text{ if } w = uxv, \\ &\qquad \text{then } y \text{ is a substring of } v\}.\end{aligned}$$

Let's see how we can find, as a function of $x, y \in \{0,1\}^*$, a minimized DFA accepting $\mathbf{AllFollow}(x, y)$.

Example

In the file `examp.sml` we put:

```
val regToEFA  = faToEFA o regToFA;
val efaToDFA  = nfaToDFA o efaToNFA;
val regToDFA  = efaToDFA o regToEFA;
val minAndRen = DFA.renameStatesCanonically o DFA.minimize;
val allStrDFA =
      minAndRen(regToDFA(Reg.fromString "(0 + 1)*"));
val allStrEFA = injDFAToEFA allStrDFA;
val strToEFA  = faToEFA o FA.fromStr;
```

- The lack of subtyping necessitates use of explicit injections.

Example

```
fun hasSubEFA x =
      EFA.concat
      (allStrEFA,
       EFA.concat(strToEFA x, allStrEFA));

val hasSubDFA = minAndRen o efaToDFA o hasSubEFA;

fun hasNotSubDFA x =
      minAndRen(DFA.minus(allStrDFA, hasSubDFA x));
```

- The lack of dependent types means functions that may only be called with strings in $\{0,1\}^*$, and which return automata whose alphabets are $\{0,1\}$, will have imprecise types.

Example

```
fun someNotFollowEFA(x, y) =
      EFA.concat
      (allStrEFA,
       EFA.concat
       (strToEFA x, injDFAToEFA(hasNotSubDFA y)));

val someNotFollowDFA =
      minAndRen o efaToDFA o someNotFollowEFA;

fun allFollowDFA(x, y) =
      minAndRen
      (DFA.minus(allStrDFA, someNotFollowDFA(x, y)));
```

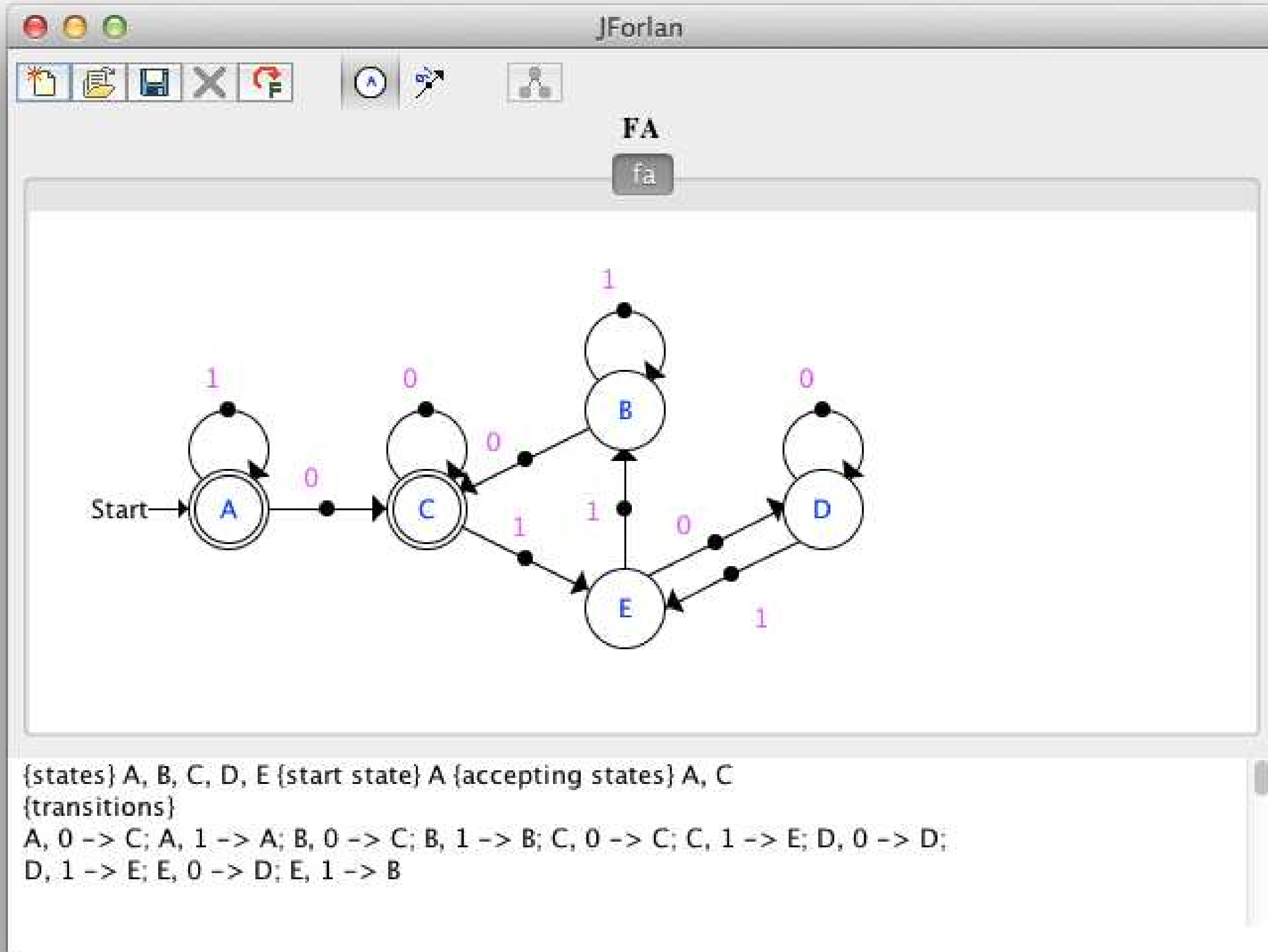
Example

```
- use "examp.sml";
[opening examp.sml]
val regToEFA = fn : reg -> efa
val efaToDFA = fn : efa -> dfa
val regToDFA = fn : reg -> dfa
val minAndRen = fn : dfa -> dfa
val allStrDFA = - : dfa
val allStrEFA = - : efa
val strToEFA = fn : str -> efa
val hasSubEFA = fn : str -> efa
val hasSubDFA = fn : str -> dfa
val hasNotSubDFA = fn : str -> dfa
val someNotFollowEFA = fn : str * str -> efa
val someNotFollowDFA = fn : str * str -> dfa
val allFollowDFA = fn : str * str -> dfa
val it = () : unit
```

Example

```
- val dfa =
=       allFollowDFA
=       (Str.fromString "01", Str.fromString "10");
val dfa = - : dfa
- FA.jforlanEdit(injDFAToFA dfa);
val it = - : fa
```

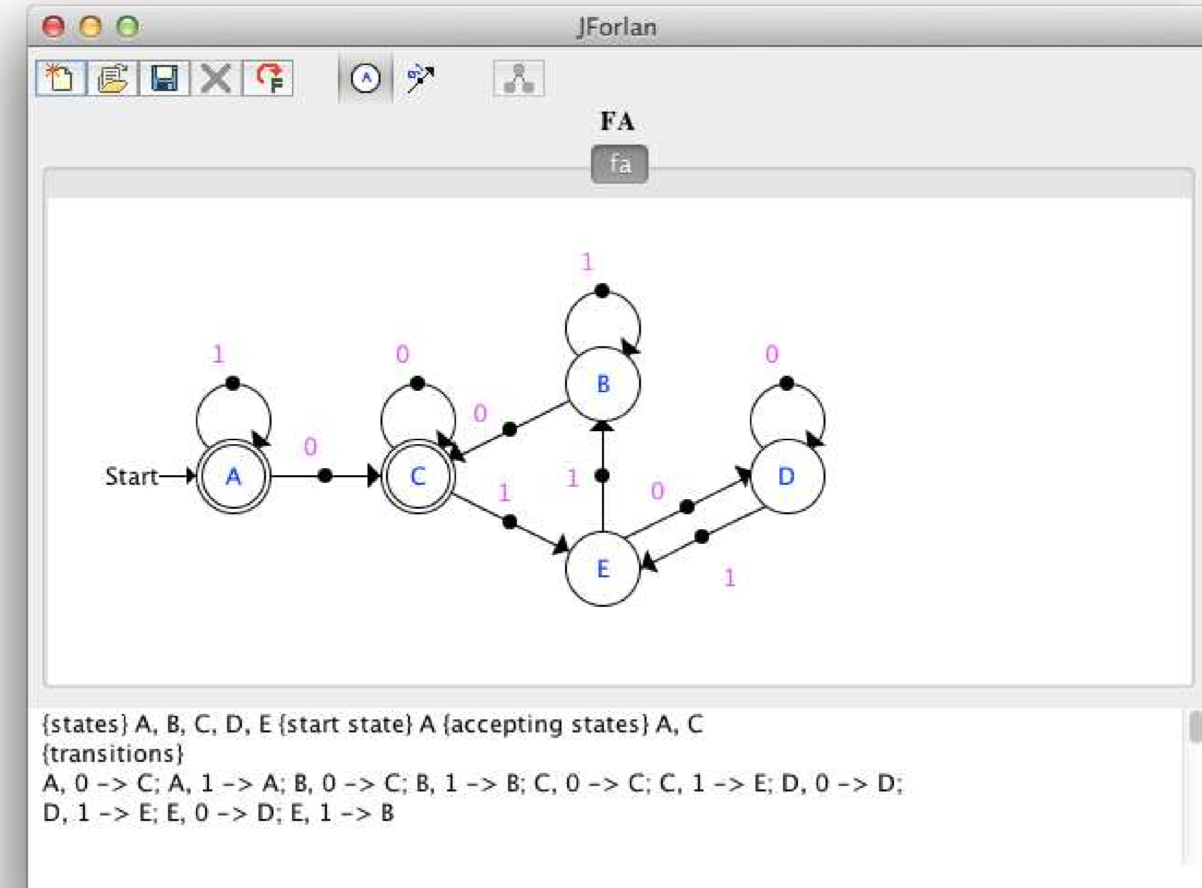
Example


JForlan
FA
fa
Start
A
B
C
D
E
{states} A, B, C, D, E {start state} A {accepting states} A, C
{transitions}
A, 0 -> C; A, 1 -> A; B, 0 -> C; B, 1 -> B; C, 0 -> C; C, 1 -> E; D, 0 -> D;
D, 1 -> E; E, 0 -> D; E, 1 -> B

Example

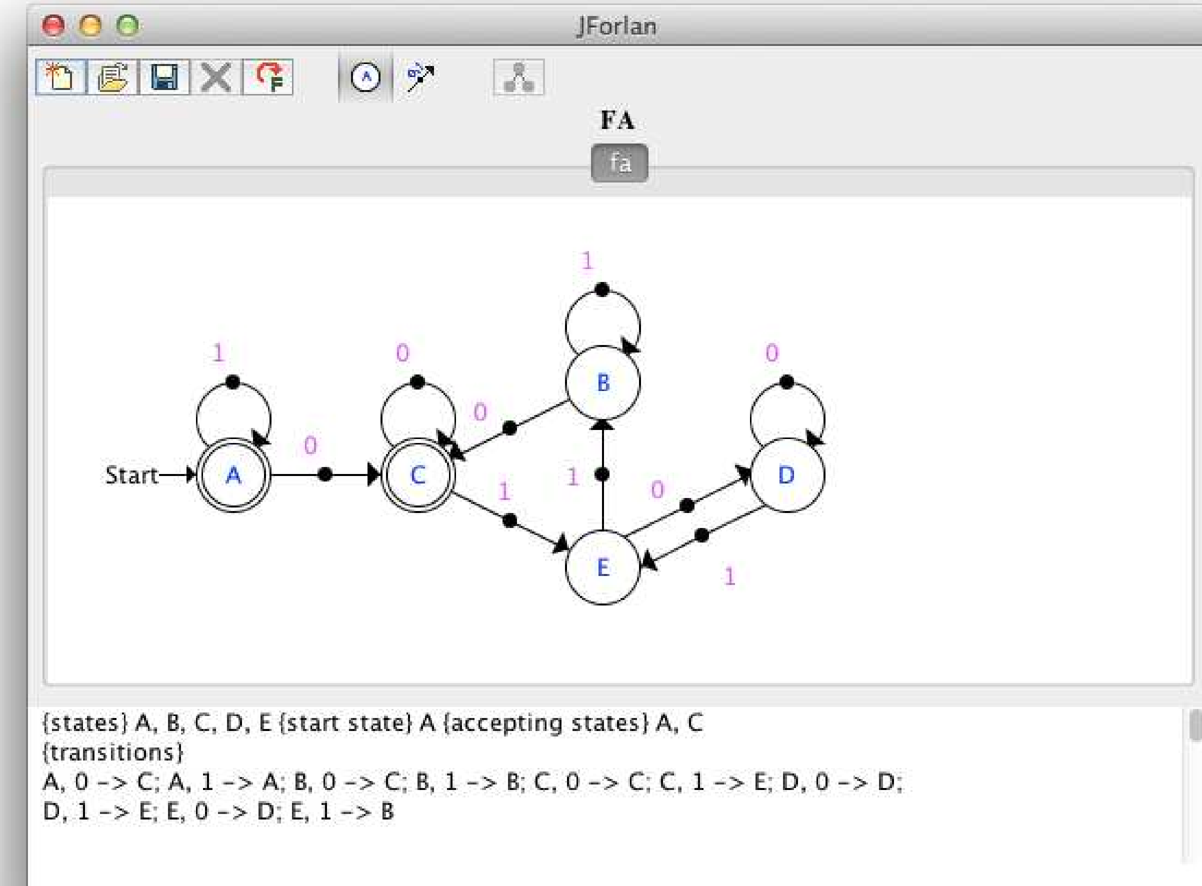
```
- val accepted = DFA.accepted dfa;
val accepted = fn : str -> bool
- val find = DFA.findAcceptingLP dfa;
val find = fn : str -> lp
```

Example




```
- accepted(Str.fromString "0101010");
val it = false : bool
```

Example




```
- LP.output("", find(Str.fromString "010110"));
A, 0 => C, 1 => E, 0 => D, 1 => E, 1 => B, 0 => C
val it = () : unit
```

Ordered Sets

The signature `SET` of the structure `Set` includes:

```
type 'a set
val memb     : 'a total_ordering -> 'a * 'a set -> bool
val fromList : 'a total_ordering -> 'a list -> 'a set
val toList   : 'a set -> 'a list
val empty    : 'a set
val sing     : 'a -> 'a set
val map      :
      'b total_ordering -> ('a -> 'b) -> 'a set -> 'b set
val union    :
      'a total_ordering -> 'a set * 'a set -> 'a set
```

- Each value of type `'a set` has a corresponding total ordering cmp.
- But the absence of dependent types means that a total ordering may not be compatible with a set or sets.

Ordered Sets

As a workaround, I've used specifications to explain how the total orderings and sets must correspond.

```
val union     :
      'a total_ordering -> 'a set * 'a set -> 'a set
```

If xs and ys are compatible with cmp, then `union`$(xs,\ ys)$ returns the set that is compatible with cmp and is the union of xs and ys, i.e., the set zs such that, for all values z of type `'a`, $\mathbf{memb}\, cmp\, (z, zs)$ iff $\mathbf{memb}\, cmp\, (z, xs)$ or $\mathbf{memb}\, cmp\, (z, ys)$.

But such specifications aren't even typechecked, much less subjected to automatic or semi-automatic theorem proving.

Ordered Sets

For each instance of `'a set` that is used frequently by end users, a specialized structure is provided that hard-wires the right total ordering.

E.g., the signature `SYM_SET` includes:

```
val memb     : Sym.sym * Sym.sym Set.set -> bool
val fromList : Sym.sym list -> Sym.sym Set.set
val map      :
      ('a -> Sym.sym) -> 'a Set.set -> Sym.sym Set.set
val union    :
      Sym.sym Set.set * Sym.sym Set.set -> Sym.sym Set.set
```

And the structure `SymSet` includes:

```
val memb     = Set.memb Sym.compare
val fromList = Set.fromList Sym.compare
val map      = fn f => Set.map Sym.compare f
val union    = Set.union Sym.compare
```

Lack of Support for Specifications

- In documenting Forlan's implementation, I was frustrated by the lack of support in SML for expressing specifications, both of correctness and termination.
- SML even lacks syntax for specifying the types of local functions.
- Making the programmer express correctness and termination specifications in an adhoc syntax within comments means that such specifications can't even be typechecked.
- I believe a modern language should allow specifications to be formally expressed, so that implementations may type-check them, and—ideally—will employ a theorem prover or proof assistant to assess their validity.
- Experience with Leino's Dafny suggests that—in a version of SML with support for specifications—many specifications could be automatically verified.

Lack of Recursive Modules

SML's lack of recursive modules necessitated careful, and sometimes non-optimal, placement of functions involving types from multiple modules.

For example, the `RFA` structure is concerned with converting FAs to regular expressions:

```
val fromFA  : (Reg.reg -> Reg.reg) -> FA.fa -> rfa
val toReg   : (Reg.reg -> Reg.reg) -> rfa   -> Reg.reg
val faToReg : (Reg.reg -> Reg.reg) -> FA.fa -> Reg.reg
```

Because `faToReg`'s type doesn't involve `rfa`, the user might expect either or both of `FA` or `Reg` to mirror it.

But this is impossible:

- it can't be mirrored by `FA`, because `RFA` references `FA`; and
- it can't be mirrored by `Reg`, because `RFA` references `Reg`.

Lack of Subtyping

The signature `FA` of the `FA` structure includes:

```
type concr =
       {stats     : Sym.sym Set.set,
        start     : Sym.sym,
        accepting : Sym.sym Set.set,
        trans     : Tran.tran Set.set}
type fa
val fromConcr : concr -> fa
val toConcr   : fa -> concr
```

The abstract types `EFA.efa`, `NFA.nfa` and `DFA.dfa` are implemented as—increasingly restrictive—subsets of `fa`.

But because SML lacks subtyping:

- explicit injection functions are needed; and
- the `EFA`, `NFA` and `DFA` structures must pack and unpack their values using `FA`.

Lack of Subtyping

For instance, the `DFA` structure includes:

```
type dfa = FA.fa
fun injToFA(dfa : dfa) : FA.fa = dfa
fun injToEFA dfa = EFA.projFromFA(injToFA dfa)
fun injToNFA dfa = NFA.projFromFA(injToFA dfa)
```

And to operate on a DFA, it must do something like:

```
fun foo dfa =
      let val {stats, start, accepting, trans} =
                FA.toConcr dfa
          ...
          val concr =
                {stats     = ...,
                 start     = ...,
                 accepting = ...,
                 trans     = ...}
      in FA.fromConcr concr end
```

Summary

My experience using SML and SML/NJ to implement Forlan was generally positive:

- It was typically easy to naturally and efficiently implement the book's mathematics in SML.
- Both the core and module languages served well.
- SML/NJ proved robust and provided sufficient speed.

But the lack of some language features was frustrating:

- type `nat`, successor as a constructor, `int` not arbitrary precision;
- subtyping;
- dependent types;
- recursive modules; and
- support for specifications.

Forlan Distribution

The Forlan distribution, including

- the draft textbook,
- the Forlan toolset, and
- a link to a paper on Forlan,

can be obtained from

`http://alleystoughton.us/forlan`