

CERTIFIED CONTEXT-FREE PARSING: A FORMALISATION OF VALIANT’S ALGORITHM IN AGDA

JEAN-PHILIPPE BERNARDY AND PATRIK JANSSON

Chalmers University of Technology & University of Gothenburg, Sweden
e-mail address: bernardy@chalmers.se

Chalmers University of Technology & University of Gothenburg, Sweden
e-mail address: patrikj@chalmers.se

ABSTRACT. Valiant (1975) has developed an algorithm for recognition of context free languages. As of today, it remains the algorithm with the best asymptotic complexity for this purpose. In this paper, we present an algebraic specification, implementation, and proof of correctness of a generalisation of Valiant’s algorithm. The generalisation can be used for recognition, parsing or generic calculation of the transitive closure of upper triangular matrices. The proof is certified by the Agda proof assistant. The certification is representative of state-of-the-art methods for specification and proofs in proof assistants based on type-theory. As such, this paper can be read as a tutorial for the Agda system.

INTRODUCTION

Context-free grammars [Chomsky, 1957] are the standard formalism to express and study the syntactic structure of programming languages. While numerous algorithms are used for parsing context-free inputs, the subject of this paper is a generalisation of the recognition algorithm discovered by Valiant [1975]. (For simplicity we call the generalisation simply “Valiant’s algorithm” below.)

Valiant’s algorithm has many qualities. First, it is efficient: it is the parsing algorithm with the best asymptotic complexity [Lee, 2002]. Further, it has recently been identified that its performance in common average cases is also excellent [Bernardy and Claessen, 2014], given some care in the representation of grammars and data structures. Second, Valiant’s algorithm is abstract and general. While its main application is parsing, it solves the problem of finding the transitive closure of a generalised relation W given as an upper-triangular matrix, where the underlying element operation is not necessarily associative. Third, Valiant’s algorithm is relatively simple: we will see that it can be expressed as two inductions on the matrix structure.

The combined qualities of Valiant’s algorithm (importance of the problem solved, efficiency, generality and simplicity) make it, in our opinion, one of the first 10 algorithms that every computer scientist should learn. As such, it is a good candidate for being given a fully certified-correct implementation. Such an implementation is the main contribution of this paper.

1998 ACM Subject Classification: F.4.1. Mathematical Logic (Mechanical theorem proving); F.4.2. Grammars and Other Rewriting Systems (Parsing) .

Key words and phrases: Context-Free Parsing, Valiant’s algorithm, Proof, Agda.

The medium chosen for our proof is the Agda proof assistant [Norell, 2007]. One motivation for this choice is that Agda is based on type-theory, and using type-theory as a core means that the tool which verifies our proof is a relatively simple type-checker, which can itself be subject to formal verification. This chain of certification, relying ultimately on a small trusted base shared with many other proofs makes us very confident that our development is correct. (Even though the core of Agda is not currently verified, there is ongoing effort in this direction.)

A secondary goal of this work is to provide an exemplary proof: we have taken particular care to make the proof approachable. In particular: 1. the algorithm is derived from its specification (instead of being first implemented and proved after the fact). 2. the core of formal proof is close to a semi-formal proof developed earlier [Bernardy and Claessen, 2013]. Further, because we assume little knowledge of the Agda proof assistant, our development can be used as a tutorial of algorithm specification and derivation in Agda.

The rest of the paper is organised as follows. In section 1, we review how Valiant’s algorithm reduces parsing to the computation of transitive closure. Section 2 introduces Agda’s syntax and some basic formalisation concepts. Sections 3 to 8 contain the formal development. We review related work in section 9 and conclude with possible extensions in section 10.

1. PARSING AS TRANSITIVE CLOSURE

In this section we review the basic idea underlying Valiant’s algorithm, namely, that context-free parsing can be specified as computing a transitive closure.

1.1. Chomsky Normal Form. The simplest implementation of Valiant’s algorithm takes as input a grammar in Chomsky Normal Form [Chomsky, 1959]. In Chomsky Normal Form, hereafter abbreviated CNF, the production rules are restricted to one the following forms

$$\begin{array}{ll} A ::= BC & \text{(binary)} \\ A ::= t & \text{(unary)} \\ S ::= \epsilon & \text{(nullary)} \end{array}$$

Any context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$ generating a given language can be converted to a grammar \mathcal{G}' in CNF generating the same language. Hence we will assume from now on a grammar provided in CNF. Moreover, because it is easy to handle the empty string specially, we conventionally exclude it from the input language and thus exclude the nullary rule $S ::= \epsilon$ from the set of production rules. The reader avid of details is directed to Lange and Leiß [2009] for a pedagogical account of the process of reduction to CNF.

Given a grammar specified as above, the problem of parsing is reduced to finding a binary tree such that each leaf corresponds to a symbol of the input (and a suitable unary rule) and such that each branch corresponds to a suitable binary rule. Essentially, parsing is equivalent to considering all possible bracketings of the input, and finding one (or more) that form a valid parse.

1.2. Charts. Let w be a vector of input symbols. We define the operations 0 , $+$, \cdot and σ as follows.

Definition 1.1 (0 , $+$, \cdot on $\mathcal{P}(N)$).

$$\begin{aligned} 0 &= \emptyset & x + y &= x \cup y & x \cdot y &= \{A \mid B \in x, C \in y, (A ::= BC) \in P\} \\ & & & & \sigma_i &= \{A \mid (A ::= w[i]) \in P\} \end{aligned}$$

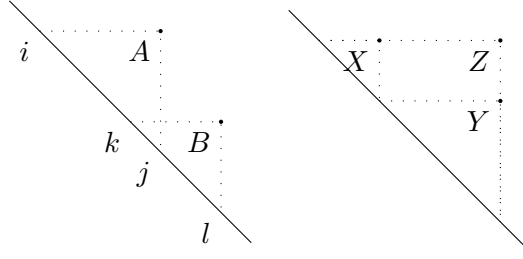


Figure 1: Example charts. In each chart a point at position (x, y) corresponds to a substring starting at x and ending at y . The first parameter (x) grows downwards and the second one (y) rightwards. The input string w is represented by the diagonal line. Dots in the upper-right part represent nonterminals. The first chart witnesses $A \xrightarrow{*} w[i..j]$ and $B \xrightarrow{*} w[k..l]$. An instance of the rule $Z ::= XY$ is exemplified on the second chart.

The (\cdot) operation fully characterises the binary production rules of the grammar, while σ captures the unary ones. We will then use a matrix of sets of non-terminals to record which non-terminals can generate a given substring. Such a matrix is called a chart. If C is a (complete) chart, $A \in C_{ij}$ iff. $A \xrightarrow{*} w[i..j]$. See figure 1 for an illustration. The operation σ is used to construct an initial chart $I(w)$ such that

$$\begin{aligned} I(w)_{i,i+1} &= \sigma_i \\ I(w)_{i,j} &= 0 && \text{if } j \neq i+1 \end{aligned}$$

The matrix $W = I(w)$ is a partial chart: it contains the correct non-terminals for positions $(i, i+1)$, but is zero otherwise. Computing the transitive closure of W will complete the chart; so that at position $(0, n)$ one will find the non-terminals generating the whole input.

Definition 1.2 (Transitive closure). If it exists, the transitive closure of a matrix W , written W^+ , is the smallest matrix C such that

$$C = W + C \cdot C$$

The equation means that C contains all the possible associations of W multiplied by itself. Because C is the least matrix satisfying the equation, it will not contain any junk.

The above procedure specifies a recogniser: by finding the closure of $I(w)$ one finds if w is parsable, but not the corresponding parse tree. However, one can obtain a proper parser by using sets of parse trees (instead of non-terminals) and extending (\cdot) to combine parse trees.

2. AGDA PRELIMINARIES

This section introduces the elements of Agda necessary to understand the upcoming sections. We present both the language itself, and some definitions which are part of the standard library. Throughout the paper, we will use a literate-programming style. The body of this section is a single module Preliminaries which contains “specification building blocks” — a number of definition for later use.

module Preliminaries where

In Agda code, scoping is indicated by indentation. Indentation is rather hard to visualise in a paper which spans over several pages, so we will instead give scoping hints in the text when necessary. The scope of the current module extends to the end of the section.

Propositions-as-types. The philosophy behind Agda is that each proposition is expressed as a type. That is, proving that a proposition P holds means finding an inhabitant (an element) of P (read as a type). With this in mind, we define the type of relations over an underlying type A as functions mapping two elements of type A to a another type (a *Set* in Agda parlance).

```
Rel : Set → Set1
Rel A = A → A → Set
```

Hence, an element of type $\text{Rel } A$ is a binary relation on A . For example $\text{Rel } A$ will be inhabited by equivalence relations and orderings. A consequence of the above definition is that our relations are constructive: to show that a pair (x, y) is in a relation R we must provide a witness of type $R \times y$. (In passing, the type of relations is a so-called big set (Set_1), because it contains *Sets* itself. This distinction is necessary for the consistency of the logical system.)

As another example, we can define the existential quantifier connective as follows:

```
record ∃ {A : Set} (P : A → Set) : Set1 where
  constructor _,_
  field
    proj1 : A
    proj2 : P proj1
```

That is, the existence of an element satisfying P can be written $\exists (\lambda x \rightarrow P x)$ (or, equivalently, $\exists P$), and proving this proposition means to find a witness $x : A$ and an inhabitant $p : P x$. The constructor keyword here introduces the name $_,_$ as the two-argument constructor of the record type. Infix operators are declared using underscores on both sides of the name so an infix comma $(,)$ can now be used as in the following example: $(x, p) : \exists P$. Note that only the last argument (P) of the \exists symbol is written, the first (A) is left for Agda to infer: we say that it is *implicit*. Implicit parameters are marked by placing them in braces at the declaration site. An implicit argument can be supplied explicitly at a call site, if the programmer encloses it with braces. This syntax can be useful if Agda fails to infer the argument in a certain context.

Entire relations. A binary relation from a set A to a set B is called *entire* if every element of A is related to at least one element of B , and we can encode this definition as follows.

```
Entire : {A B : Set} → (_R_ : A → B → Set) → Set
Entire _R_ = ∀ a → ∃ λ b → a R b
```

Here, again, the use of underscores around R makes it an infix operator (in its scope). Fixity is just a presentation issue, so an equivalent, but shorter, definition is $\text{Entire } R = \forall a \rightarrow \exists (R a)$ where R is prefix and $R a$ is a partial application. A consequence of proving that a relation R is entire in our constructive setting is that we get a function contained in R . We can extract the function using the first field of the \exists record.

```
fun : {A B : Set} → {_R_ : A → B → Set} → Entire _R_ → A → B
fun ent a = proj1 (ent a)
```

The proof that the function is contained in R can be obtained from the second field of \exists :

```
correct : {A B : Set} → {_R_ : A → B → Set} → (e : Entire _R_) →
  let f = fun e in ∀ {a : A} → a R (f a)
correct e {a} = proj2 (e a)
```

The above pattern generalises to relations of any number of arguments. In this paper we need the following version:

Entire3 : {A B C D : Set} → (R : A → B → C → D → Set) → Set
 Entire3 R = $\forall x y z \rightarrow \exists (R x y z)$

with corresponding definitions of fun3 and correct3.

Uniqueness. An element of UniqueSolution \simeq P is a proof that the predicate P has (at most) a *unique* solution relative to some underlying relation \simeq .

UniqueSolution : {A : Set} → Rel A → (A → Set) → Set
 UniqueSolution \simeq P = $\forall \{x y\} \rightarrow P x \rightarrow P y \rightarrow x \simeq y$

A proof usP : UniqueSolution \simeq P is thus a function which given two hidden arguments of type A and two proofs that they satisfy P returns a proof that they are related by \simeq .

Least solutions. In optimisation problems, one often wants to find the least solution with respect to some order \leq . We use LowerBound \leq P for the predicate that holds for an a : A iff. a is smaller than all elements satisfying P. If a lower bound is in the set (satisfies the predicate P) it is called *least*.

LowerBound : {A : Set} → Rel A → (A → Set) → (A → Set)
 LowerBound \leq P a = $\forall z \rightarrow (P z \rightarrow a \leq z)$
 Least : {A : Set} → Rel A → (A → Set) → (A → Set)
 Least \leq P a = P a × LowerBound \leq P a

Note that a proof alP : Least \leq P a is a pair of a proof that a is in P and a function albP : LowerBound \leq P a. And, in turn, albP is a function from some z : A and a proof that z is in P to a proof that a \leq z.

Records and modules. The upcoming proof makes extensive use of *records*, which we review now in detail. Agda record types contain fields and helper definitions. Fields refer to data which is stored in the record, while helper definitions provide values which can be computed from such data. Because Agda treats proofs (of propositions) as data, one can require the fields to satisfy some laws, just by adding (proofs of) those laws as fields. Our first record type example, $\exists P$, has two fields; one element proj_1 and a proof that it satisfies the property P. As a more complex record type example we use the following (simplified) version of IsCommutativeMonoid from Algebra.Structures in the standard library. The record type is parametrised over a carrier set, a relation, a binary operation and its identity element:

```
record IsCommutativeMonoid {A : Set} ( _≈_ : Rel A)
  ( _•_ : A → A → A) (ε : A) : Set₁ where
  field
    isSemigroup : IsSemigroup _≈_ _•_
    identityl   : LeftIdentity _≈_ ε _•_
    comm        : Commutative _≈_ _•_
```

The fields capture the requirements of being a commutative monoid, in terms of three other properties. Here, the first field (isSemigroup) is also of record type; it is in fact common in Agda to define deeply nested record structures.

In Agda every record type also doubles as a module parametrised over a value of that type. For example, within the scope of the above record, given a value $\text{isNP} : \text{IsSemigroup } \mathbb{N} \text{ } _+_$, the phrase `IsSemigroup isNP` designates a module containing a declaration for each field and helper definition of the `IsSemigroup` record type. Consequently, within the scope of the above record, one can access the (nested) fields of `isSemiGroup` in the module `IsSemigroup isSemiGroup`. In fact, it is very common for a record type to re-export all the definitions of inner records. This can be done with the following declaration (still inside the `record`):

open IsSemigroup isSemiGroup **public**

open means that the new names are brought into scope for later definitions inside the record type (module) and **public** means the new names are also exported (publicly visible). This means that the user can ignore the nesting when fetching nested fields (but not when constructing them).

Equality proofs. We finish this section with an example of a helper definition, which also serves as an introduction to *equality-proof notation*. A helper definition of `IsCommutativeMonoid` is the right identity proof `identityr`, which is derivable from commutativity and left identity. We can define it as follows (still inside the `record IsCommutativeMonoid`):

```
identityr : ∀ x → (x • ε) ≈ x
identityr x =
  begin
    x • ε
  ≈⟨ comm x ε ⟩
    ε • x
  ≈⟨ identityl x ⟩
    x
  ■
```

The above is merely the composition by transitivity of $(\text{comm } x \ \varepsilon)$ and $(\text{identity}^l \ x)$, but by using special purpose operators the user can keep track of the intermediate steps in the proof in a style close to pen-and-paper proofs.

3. FORMAL DEVELOPMENT OVERVIEW

In the following sections we expose our formalisation of the specification of the transitive closure algorithm, its implementation (Valiant’s algorithm) and the proof of correctness. The algorithm falls out from a calculational refinement of the specification rather than being exposed *ex-nihilo* and proved separately. The development is presented in a number of stages:

- We define the ring-like algebraic structure where we set our development (section 4),
- We give the specification of the transitive closure (section 5),
- We define the concrete data structures that the algorithm manipulates (section 6),
- We derive Valiant’s algorithm from part of that specification (section 7),
- We conclude by showing that the algorithm satisfies the rest of the specification (section 8).

4. ALGEBRA

We begin by defining a record type called `SemiNearRing`, whose fields and helper definitions capture the algebraic structure that we need for the algorithm development. First we introduce a carrier set `s` with an equivalence relation, a zero, addition and multiplication.

record `SemiNearRing` : `Set1` **where**

field

`s` : `Set`

`_≃s_` : `s` → `s` → `Set`

`0s` : `s`

`_+s_` : `s` → `s` → `s`

`_·s_` : `s` → `s` → `s`

The charts example from section 1 can be seen as a record value of this record type. This value has finite sets of nonterminals for `s` and its operations for the other fields (the usual equality, the empty set as zero, set union as addition and “cross product filtered by the grammar” as multiplication).

At this stage they are “raw” operations without laws. Here is a summary of the laws needed. We require that $(0, +)$ forms a commutative monoid. We also require that 0 is absorbing for (\cdot) , that (\cdot) distributes over $(+)$ and that (\cdot) is idempotent. Note that the product is not necessarily associative; in fact, if it were, computing the transitive closure would be much easier and parsing would not be an application.

$$x + 0 = x = 0 + x \quad x + y = y + x \quad x + (y + z) = (x + y) + z \quad (\text{now})$$

$$x \cdot 0 = 0 = 0 \cdot x \quad x + x = x \quad x \cdot (y + z) = x \cdot y + x \cdot z \quad (\text{later})$$

(It is easy to check that the definitions given in section 1 have these properties.) We could specify these properties individually, but instead we take advantage of “specification building blocks” from the Algebra modules in the Agda standard library [Danielsson and The Agda Team, 2013]. More specifically we use the record `IsCommutativeMonoid` from the library module `Structures` and we use the left and right zero laws from the parametrised module `FunctionProperties` specialised to the underlying equivalence $(_ \simeq_)$ on our carrier set `s`.

open `Algebra.Structures` **using** `(IsCommutativeMonoid)`

open `Algebra.FunctionProperties` `_≃s_` **using** `(LeftZero; RightZero)`

Armed with these properties we now continue the `SemiNearRing` record type by specifying (as new fields in the record) the laws we require of our operations: `_+s_` is a commutative monoid with `0s` as the unit, `0s` is also a multiplicative zero of `_·s_` and multiplication preserves equivalence.

field

`isCommMon` : `IsCommutativeMonoid` `_≃s_` `_+s_` `0s`

`zerol` : `LeftZero` `0s` `_·s_` -- expands to $\forall x \rightarrow (0_s \cdot_s x) \simeq_s 0_s$

`zeror` : `RightZero` `0s` `_·s_` -- expands to $\forall x \rightarrow (x \cdot_s 0_s) \simeq_s 0_s$

`_◇s_` : $\forall \{x y u v\} \rightarrow (x \simeq_s y) \rightarrow (u \simeq_s v) \rightarrow (x \cdot_s u \simeq_s y \cdot_s v)$

The rest of the record type consists of helper declarations which are useful to have in scope when working with the specification. They will be put in scope whenever we access an instance of the record in the definition of the algorithm or in its proof. Also inside the record type we specify the precedence of operators using the following declarations:

infix 4 `_≃s_`; **infixl** 6 `_+s_`; **infixl** 7 `_·s_`

Exporting inner names. As we mentioned in section 2 the module `Algebra.Structures` includes a parametrised module `IsCommutativeMonoid` which contains the commutative monoid laws. In the `SemiNearRing` record type that we are defining there is already a field `isCommMon` which, in turn, contains the proofs of the monoid laws so that a user with a value `snr : SemiNearRing` can access all these proofs by indexing through the two record layers. But two (and later more) levels of records is inconvenient to use so we include short hand names for the inner record fields as follows.

open `Algebra.Structures.IsCommutativeMonoid` `isCommMon` **public**

```

hiding (refl)
renaming
  (isEquivalence to isEquivs
  ; assoc to assocs
  ; comm to comms
  ; •-cong to _<+>_
  ; identityl to identityls
  )

```

`identityrs = proj2 identity`

Recall that **open** means that the new names are brought into scope for later definitions inside the record type and **public** means the new names are also exported (publicly visible). We rename (with a subscript *s*-suffix) to avoid clashes when we add most of the same for another set *u* later. The infix notation `_<+>_` for congruence is useful in equality proofs: if we have proofs `ap : a ≈s a'` and `bp : b ≈s b'` we get a proof of `(a +s b) ≈s (a' +s b')` which can be layed out nicely:

```

begin
  a +s b
  ≈( ap <+> bp )
  a' +s b'

```

■

Finally `sSetoid` packages up the carrier set *s*, the relation `≈s` and the proof that it is an equivalence. This is useful not only for documentation purposes (*s* is a setoid), but certain parts of the standard library require properties to be packaged in such a manner, for example the module giving convenient syntax for equality proofs, which we use later.

```

sSetoid : Setoid _ _
sSetoid = record { Carrier      = s;
                  ≈_           = ≈s;
                  isEquivalence = isEquivs }

```

open `IsEquivalence` `isEquivs` **public**

```

hiding (reflexive) renaming (refl to refls; sym to syms; trans to transs)

```

The **open public** of `isEquivs` is (again) to avoid the need for multiple layers of record projections.

This marks the end of the definition of the record type `SemiNearRing`. To summarise, a record value `snr : SemiNearRing` contains as fields a carrier set *s*, the operations `(≈s, 0s, +s, ·s)` and the proofs of the properties (`isCommMon`, `zerol`, `zeror`, `_<◇>_`). We will later return to this record type and add a few more fields (in section 5) and helpers (in section 7.2 and section 8) to capture the full specification of the closure algorithm, but for now this will do.

4.1. Matrix Algebra(s). Valiant’s algorithm works on square matrices. We carry on and define the algebraic structures required on matrices. Luckily, most of the structure of matrices is the same as that required on elements. In fact, the name of the carrier set (s) defined earlier was in reference to square matrices.

However, Valiant’s algorithm works only on upper triangular matrices— that is, those whose elements at and below the diagonal are zero. We could have defined upper triangular matrices by quotienting, namely, as the type of square matrices satisfying the triangularity predicate. However, such a definition yields a large amount of tedious work to manipulate the extra predicate.

Instead, we enforce the property of upper-triangularity axiomatically, by defining a separate type u (for upper triangular matrix) and require an embedding from u to s (square matrix). We package the types u and s , together with all their properties, in a single record `SemiNearRing2`: we later build structures by simultaneous recursion on u and s . (Yet we keep `SemiNearRing` as a separate entity, as it is an adequate specification of the elements of the matrices.)

```
record SemiNearRing2 : Set1 where
  field
    snr : SemiNearRing
  open SemiNearRing snr -- later also made public = exported
  field
    u : Set
    _+_u : u → u → u
    _·_u : u → u → u
    u2s : u → s
```

Beside the type u and an embedding into s , we require addition and multiplication over u : this is a simple specification of the property that those operations preserve upper-triangularity. Note that we require no relation between $_+_u$ and $_+_s$, no relation between $_·_u$ and $_·_s$ and no algebraic properties of the u -operations. Most of the link between u and s will become manifest when we see the recursive structure tying them together. The rest of the record contains helper definitions to lift the structure of s onto u using $u2s$. The lifting of binary relations is provided for us by the standard library via the infix operator `Function.on`.

```
_≈u : u → u → Set
_≈u = _≈s Function.on u2s

_⌣s : u → s → s
_⌣s u s = u2s u ·s s

_·s : s → u → s
_·s u s = s ·s u2s u
```

The operator precedences and a `Setoid` instance follow, for completeness.

```
infix 4 _≈u; infixl 6 _+_u; infixl 7 _·u _⌣s _·s
uSetoid : Setoid _ _
uSetoid = record {isEquivalence = Relation.Binary.On.isEquivalence u2s isEquivs}
```

```
open SemiNearRing snr public
```

To summarise, a record value `snr2 : SemiNearRing2` contains the s -operations from a subrecord `snr`, the corresponding operations for u (but no laws) and the embedding $u2s : u \rightarrow s$.

Recall that Valiant’s algorithm aims at finding the smallest matrix C such that $C = W + C \cdot C$. Our goal is to find a function computing C , and to prove it correct. The first step is to formally transcribe the definition of transitive closure. Hence we revisit and extend the `SemiNearRing` structure with a few more components needed to define the closure relation. First we need an order to define “smallest”. Remember that, for parsing, a value of type `s` is a set of non-terminals. Hence, parsing uses set inclusion for the preorder and set union for addition and in this context $a \subseteq b$ iff $a \cup b = b$. A natural generalisation is to define a partial order where $x \leq_s y$ iff $x +_s y \simeq_s y$. For the relation \leq_s to be a preorder we need idempotence of addition (which implies reflexivity and transitivity). With this order addition is automatically monotonous, but for multiplication to be monotonous we need distribution laws: that \cdot_s distributes over $+_s$.

The algebraic structure we have specified so far in this record type is nearly, but not quite, an idempotent semiring. We just lack (and don't need) associativity (and unit) of our multiplication.

$$\text{LowerBound}_s = \text{LowerBound} \leq_s$$
$$\begin{aligned} \leq_{u-} &: u \rightarrow u \rightarrow \text{Set} \\ \leq_{u-} &= \leq_{s-} \text{ Function.on } u2s \end{aligned}$$
$$\text{record ClosedSemiNearRing : Set}_1 \text{ where}$$

We can now finally give the specification and we do it in three steps: the relation Q capturing the “quadratic” equation, the relation Closure capturing only the “least” solutions of Q and finally $\text{Entire } Q$ which says that there is a total function inside the relation.

```

Q : u → u → Set
Q w c = w +u c ·u c ≈u c
Closure : u → u → Set
Closure w c = Least _≤u_ (Q w) c
field
  entireQ : Entire Q

```

From the `entireQ` field (which we will populate later) we can extract the closure algorithm (`closure`) and part of its correctness proof (`closureHasAll`).

```

closure : u → u
closure = fun entireQ
closureHasAll = correct entireQ

```

Proving `Entire Closure` means to show that `closure` yields the smallest possible solution, and is deferred until section 8.

```

open SemiNearRing2 snr2 public

```

6. MATRICES

In the specification and implementation we make heavy use of square matrices of size $m \times m$ for different values of m . For concision we make the simplifying assumption that m is a power of two ($m = 2^n$) — handling the general case involves indexing matrices with their sizes, which is straightforward but clutters the development.

There are many different ways to represent matrices. As usual when working with dependent types, some definitions yield a concise presentation, while others require large amounts of boilerplate, obscuring the intent. A judicious choice is thus in order. Possibilities include:

- A vector of vectors of elements, where a vector is

```

data Vec a n where
  Nil   : Vec a 0
  Cons : a → Vec a n → Vec a n → Vec a (1 + n)

```

- A function from (two) indices to elements $\text{Bin } n \rightarrow \text{Bin } n \rightarrow a$ (where `Bin` stands for a binary number of n bits)
- A recursive data-type (a quad tree):

```

data Mat a n where
  Unit : a → Mat a 0
  Quad : Mat a n → Mat a n → Mat a n → Mat a n → Mat a (1 + n)

```

- A function from n to `Set`:

```

Mat a 0      = a
Mat a (suc n) = Square (Mat a n)
where Square t = t × t × t × t

```

Here we choose the last approach, which is specifically tailored to the problem at hand. Indeed, while choosing a simple quadruple for `Square` is sufficient to define just the datatype of (block) matrices, we can use other (much) more complicated functors, which carry the ring-like structures and all the axiomatisation we need, up to the full closure specification. In fact, most of the rest of the paper is devoted to the definition of that functor: when it is fully defined we have a proven-correct implementation of Valiant's algorithm. In this section we show the abstraction over `SemiNearRing` and the construction of matrices, up to the full definition of `Mat`.

The main functor: Square. We attack the recursive case right away by defining the function `Square` which lifts our algebraic structure (the types `s` and `u`, the operations on them and all the laws) from elements to 2-by-2 (block) matrices. It can be seen as an implementation of a (first-class, but otherwise) ML-style functor.

`Square : ClosedSemiNearRing → ClosedSemiNearRing`

`Square csnr = CSNR where`

`open ClosedSemiNearRing csnr`

The rest of this section is the body of this `where` clause where all operations and laws from `csnr` are in scope.

Lifting types. We start by defining two types `U` and `S`, instantiating the `u` and `s` types for two-by-two (block) matrices. Throughout the rest of the paper, we use the convention of using capitals for the values of the fields of the record created (the target of the functor).

For type `S` we could use `S = s × s × s × s` but we prefer to use a record type for clarity. The constructor name uses angle brackets to lift any ambiguity.

`record S : Set where`

`constructor ⟨-, -, -, -⟩`

`field`

`s00 : S; s01 : S`

`s10 : S; s11 : S`

`infix 4 ⟨-, -, -, -⟩`

The two-by-two upper triangular matrix is composed of two (smaller) upper-triangular matrices, and a square matrix for the top-right corner. (The bottom-left corner is zero.) Basically, `U = u × s × u` but again, regular nested pairs obscure the intent. Here we add a box inside the name of the record constructor, as a reminder that the bottom left corner is empty.

`record U : Set where`

`constructor ⟨-, -, □, -⟩`

`field`

`uu00 : u; us01 : S;`

`uu11 : u`

`infix 4 ⟨-, -, □, -⟩`

Lifting operations. We lift the operations $0, +, \cdot$ from the underlying semi-near-ring to matrices over that semi-near-ring, in the usual manner.

$$\begin{aligned} \text{-+}_S &: S \rightarrow S \rightarrow S \\ \text{-+}_S \langle a, b, c, d \rangle &= \langle a', b', c', d' \rangle = \\ &\langle a +_s a', b +_s b', c +_s c', d +_s d' \rangle \end{aligned}$$

Matrix multiplication is defined as expected.

$$\begin{aligned} \text{-}\cdot_S &: S \rightarrow S \rightarrow S \\ \text{-}\cdot_S \langle a, b, c, d \rangle \langle a', b', c', d' \rangle &= \langle (a \cdot_s a') +_s (b \cdot_s c'), (a \cdot_s b') +_s (b \cdot_s d') \\ &\quad, (c \cdot_s a') +_s (d \cdot_s c'), (c \cdot_s b') +_s (d \cdot_s d') \rangle \end{aligned}$$

infixl 6 $\text{-}\cdot_S$

infixl 7 $\text{-}\cdot_S$

Zero is defined similarly.

$$\begin{aligned} \text{zer}_S &: S \\ \text{zer}_S &= \langle 0_s, 0_s, 0_s, 0_s \rangle \end{aligned}$$

We can then define the operations on upper-triangular matrices. In particular, we give full definitions of point-wise upper triangular matrix addition and multiplication. Having to define operations both for U and S is a consequence of our choice of separating u and s in the algebraic structure. This may look redundant, but in fact the definitions for U serve the purpose of proving that upper-triangularity is preserved by those operations.

$$\begin{aligned} \text{-+}_U &: U \rightarrow U \rightarrow U \\ \text{-+}_U \langle xl, xm, \square, xr \rangle \langle yl, ym, \square, yr \rangle &= \langle xl +_u yl, xm +_s ym, \square, xr +_u yr \rangle \\ \text{-}\cdot_U &: U \rightarrow U \rightarrow U \\ \text{-}\cdot_U \langle xl, xm, \square, xr \rangle \langle yl, ym, \square, yr \rangle &= \langle xl \cdot_u yl, xl \cdot_{u's} ym +_s xm \cdot_{s'u} yr \\ &\quad, \square, xr \cdot_u yr \rangle \end{aligned}$$

Equivalence is structural: two matrices are equivalent if all the elements are equivalent.

$$\begin{aligned} \text{-}\simeq_S &: S \rightarrow S \rightarrow \text{Set} \\ \text{-}\simeq_S \langle a, b, c, d \rangle \langle a', b', c', d' \rangle &= (a \simeq_s a') \times (b \simeq_s b') \\ &\quad \times (c \simeq_s c') \times (d \simeq_s d') \end{aligned}$$

infix 4 $\text{-}\simeq_S$

Embedding upper triangular matrices as (regular) matrices is straightforward:

$$\begin{aligned} \text{U2S} &: U \rightarrow S \\ \text{U2S} \langle uu_{00}, us_{01}, \square, uu_{11} \rangle &= \langle u2s \ uu_{00}, us_{01}, \\ &\quad 0_s, u2s \ uu_{11} \rangle \end{aligned}$$

Lifting laws. At this point we must verify that the above constructions preserve the laws seen so far. Most of this verification (of sym_S , trans_S , cong_S , etc.) is by straightforward, and omitted, lifting of the proofs from the underlying structure, for example refl_S :

```

reflS : {x : S} → x ≃S x
reflS = refls, refls,
      refls, refls

```

We include the preservation of distributivity as another example of equational reasoning in Agda. We start by proving the following lemma about commutativity. The underscores after assoc_s and comm_s are placeholders for arguments which Agda can infer.

```

swapMid : ∀ {a b c d} → (a +s b) +s (c +s d)
                ≃s (a +s c) +s (b +s d)

```

```

swapMid {a} {b} {c} {d}
= begin
  (a +s b) +s (c +s d)
≈⟨ assocs _ _ _ ⟩
  a +s (b +s (c +s d))
≈⟨ refls <+> sym (assocs _ _ _) ⟩
  a +s ((b +s c) +s d)
≈⟨ refls <+> (comms _ _ <+> refls) ⟩
  a +s ((c +s b) +s d)
≈⟨ refls <+> (assocs _ _ _) ⟩
  a +s (c +s (b +s d))
≈⟨ sym (assocs _ _ _) ⟩
  (a +s c) +s (b +s d)

```

■

where open EqReasoning sSetoid

```

distlS : (x y z : S) → x ·S (y +s z) ≃S x ·S y +s x ·S z
distlS _ _ _ = distrHelp, distrHelp, distrHelp, distrHelp
  where distrHelp : ∀ {a b c d e f} → a ·s (b +s c) +s d ·s (e +s f)
                ≃s (a ·s b +s d ·s e) +s (a ·s c +s d ·s f)
  distrHelp = transs (distl _ _ _ <+> distl _ _ _) swapMid

```

The record return value from `Square` should eventually contain all the fields, including the proof of closure — but we leave the rest to the upcoming sections. Thus we leave the definition of the functor `Square` for now and instead give the the base case of the inductive structure.

Base case: 1-by-1 matrices. A 1-by-1 upper triangular matrix is zero, can thus be represented by the unit type ($\text{tt} : \top$) and is therefore trivially equipped with closure.

```

OneByOne : SemiNearRing → ClosedSemiNearRing
OneByOne snr = record {snr2 = record {snr = snr; u = ⊤; -+u = λ _ _ → tt;
                                     u2s = λ _ → 0s; ·u = λ _ _ → tt}
}
  where open SemiNearRing snr using (0s)

```

Finally we give the recursion schema, following the pattern that we hinted at at the beginning of the section: we define $\text{Mat } n$ as the semi-near-ring structures lifted n times.

$\text{Mat zero } \text{el} = \text{OneByOne el}$
 $\text{Mat (suc } n) \text{ el} = \text{Square (Mat } n \text{ el)}$

The fields of $\text{Mat } n$ are of special interest to us. In particular the u field is the type of matrices of size 2^n , which we will eventually show how to equip with a closure operation.

$\text{Upper} : \mathbb{N} \rightarrow \text{SemiNearRing} \rightarrow \text{Set}$
 $\text{Upper } n \text{ el} = \text{ClosedSemiNearRing}.u \text{ (Mat } n \text{ el)}$

7. TRANSITIVE CLOSURE: DERIVATION

We have now completed the definition of all the structures necessary to develop our proof. In this section we describe (and partially prove) the closure algorithm.

We do so by deriving it from the specification $\text{Closure } w \text{ c} = \text{Least } \leq_u \text{ (Q } w) \text{ c}$, where $\text{Q } w \text{ c} = w +_u \text{ c} \cdot_u \text{ c} \simeq_u \text{ c}$ and we start with the equational part, the Entire Q requirement, and return to Least in section 8. Technically, we give a definition for the $\text{entireQ} : \text{Entire Q}$ field in both cases of the Mat function.

We first proceed semi-formally, to show what a non-certified proof of the algorithm looks like, and to be able to compare it with the fully formal, certified Agda proof that we subsequently present.

7.1. Outer induction: closure of triangular matrices. Recall that our task is to find a function \cdot^+ which maps an upper triangular matrix W to its transitive closure $C = W^+$.

If W is a 1 by 1 matrix, $C = W = 0$. Otherwise, let us divide W and C in blocks as follows:

$$W = \begin{bmatrix} A & Y \\ 0 & B \end{bmatrix} \quad C = \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix}$$

Then the condition that C satisfies Q becomes:

$$\begin{bmatrix} A & Y \\ 0 & B \end{bmatrix} + \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix} \cdot \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix} = \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix}$$

Applying matrix multiplication and addition block-wise:

$$\begin{aligned} A + A'A' &= A' \\ Y + A'X' + X'B' &= X' \\ B + B'B' &= B' \end{aligned}$$

Because A and B are smaller matrices than W (and still upper triangular), we know how to compute A' and B' recursively ($A' = A^+$, $B' = B^+$). Before showing how X' is computed, we show how to formalise the above reasoning in Agda. The main job is to populate the entireQ field in the $\text{ClosedSemiNearRing}$ record. We have to do so both for the base case and the recursive case of the Mat construction. The base case being trivial, we show here the inductive case.

We first define the Q relation for 2-by-2 (block-)matrices:

$$\text{QU} = \lambda W C \rightarrow (W +_U (C \cdot_U C)) \simeq_U C$$

and then we proceed with the proof that it is Entire: for any input matrix W we construct another matrix C and a proof that it is a closure (strictly speaking, so far only that it satisfies $\text{QU } W \text{ C}$). One

remarkable feature is that Agda can infer the solution matrix ($C = _$), from the proof. The proof follows exactly the semi-formal development given at the beginning of the subsection.

```

entireQU :  $\forall W \rightarrow \exists (QU W)$ 
entireQU W = C, proof where
  C : U
  C = _
  open EqReasoning (SemiNearRing2.uSetoid SNR2)
  proof :  $(W +_U (C \cdot_U C)) \simeq_U C$ 
  proof = begin
     $(W +_U (C \cdot_U C))$ 
   $\equiv \langle \text{refl} \rangle$  -- expand matrix components
    let  $\langle A, Y, \square, B \rangle = W$ 
       $\langle A', Y', \square, B' \rangle = C$  in
       $\langle A, Y, \square, B \rangle +_U (\langle A', Y', \square, B' \rangle \cdot_U \langle A', Y', \square, B' \rangle)$ 
   $\equiv \langle \text{refl} \rangle$  -- expand definition of  $\cdot_U$ 
       $\langle A, Y, \square, B \rangle +_U \langle A' \cdot_u A', (A' \cdot_{us} Y' +_s Y' \cdot_{su} B'), \square, B' \cdot_u B' \rangle$ 
   $\equiv \langle \text{refl} \rangle$  -- by def. of  $+_U$ 
       $\langle A +_u A' \cdot_u A', Y +_s (A' \cdot_{us} Y' +_s Y' \cdot_{su} B'), \square, B +_u B' \cdot_u B' \rangle$ 
   $\approx \langle \text{congU closureHasAll completionHasAll closureHasAll} \rangle$ 
       $\langle A', Y', \square, B' \rangle$ 
   $\equiv \langle \text{refl} \rangle$ 
    C

```

■

The use of equational reasoning in proof shows another very useful feature of the proof notation: using a normal **let-in** expression together with the distfix transitivity operator $\equiv \langle _ \rangle$ we can do what is often done in paper-proofs: introduce new names in the middle of the reasoning chain. The new names (here A, Y, B etc.) are in scope in the rest of the proof. Another interesting feature is that this proof is that many of the steps can be justified simply by $\text{refl} : x \equiv x$. Indeed, Agda automatically expands definitions during type-checking, and thus automatically expands the definitions of operators on block matrices. In fact, because refl is “the unit of transitivity”, Agda would be just as happy with only:

```
proof = congU closureHasAll completionHasAll closureHasAll
```

Yet, this one line still holds the full proof of the inner induction (`completionHasAll`) which is the topic of section 7.2. (Remember that `closureHasAll` = correct `entireQ` from section 5 (page 11) plays the role of induction hypothesis for the “outer induction”.)

The base case is completely trivial; a 1-by-1 upper triangular matrix contains no non-zero element and is represented by the unit type. Thus `entireQBase tt = tt, refls`.

7.2. Inner induction: completion of square matrices. As in the previous subsection, we first proceed semi-formally. The problem is to find a recursive function V which maps A, B and Y to $X = V(A, Y, B)$, such that $X = Y + A \cdot X + X \cdot B$. In terms of parsing, the function V combines

the chart A of the first part of the input with the chart B of the second part of the input, via a *partial* chart Y concerned only with strings starting in A and ending in B , and produces a full chart X . We proceed as before and divide each matrix into blocks:

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \quad Y = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \quad A = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix}$$

The condition on X then becomes

$$\begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} + \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} \cdot \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} + \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix}$$

By applying matrix multiplication and addition block-wise:

$$\begin{aligned} X_{11} &= Y_{11} + A_{11}X_{11} + A_{12}X_{21} + X_{11}B_{11} + 0 \\ X_{12} &= Y_{12} + A_{11}X_{12} + A_{12}X_{22} + X_{11}B_{12} + X_{12}B_{22} \\ X_{21} &= Y_{21} + 0 + A_{22}X_{21} + X_{21}B_{11} + 0 \\ X_{22} &= Y_{22} + 0 + A_{22}X_{22} + X_{21}B_{12} + X_{22}B_{22} \end{aligned}$$

By commutativity of $(+)$ and 0 being its unit:

$$\begin{aligned} X_{11} &= Y_{11} + A_{12}X_{21} + A_{11}X_{11} + X_{11}B_{11} \\ X_{12} &= Y_{12} + A_{12}X_{22} + X_{11}B_{12} + A_{11}X_{12} + X_{12}B_{22} \\ X_{21} &= Y_{21} + A_{22}X_{21} + X_{21}B_{11} \\ X_{22} &= Y_{22} + X_{21}B_{12} + A_{22}X_{22} + X_{22}B_{22} \end{aligned}$$

Now we have four equations, all of the form that V can compute solutions to. Because each of the sub-matrices is smaller and because of the absence of circular dependencies, X can be computed recursively by V . The internal dependencies dictate the order: start computing X_{21} , use that to compute X_{11} and X_{22} (possibly in parallel) and finally compute X_{12} :

$$\begin{aligned} X_{21} &= V(A_{22}, Y_{21}, B_{11}) \\ X_{11} &= V(A_{11}, Y_{11} + A_{12}X_{21}, B_{11}) \\ X_{22} &= V(A_{22}, Y_{22} + X_{21}B_{12}, B_{22}) \\ X_{12} &= V(A_{11}, Y_{12} + A_{12}X_{22} + X_{11}B_{12}, B_{22}) \end{aligned}$$

A graphical summary is shown in fig. 2.

We proceed to certify this proof step in Agda. The problem is to solve the equation L , defined as follows. (We pick the name L for linear equation as we used Q for quadratic earlier.)

$L : u \rightarrow s \rightarrow u \rightarrow s \rightarrow \text{Set}$

$L \ a \ y \ b \ x = y +_s (a \ u \ s \ x +_s x \ s \ u \ b) \simeq_s x$

We must prove that the relation L is entire, and thus get an algorithm for inhabiting it as well as its correctness proof. To this end, we add the appropriate field to the `ClosedSemiNearRing` record. We now reap the full benefits of using a record structure, which frees us from repeating the recursion pattern.

field

entireL : Entire3 L

completion : $u \rightarrow s \rightarrow u \rightarrow s$

completion = fun3 entireL

completionHasAll : $\forall \{a \ y \ b\} \rightarrow L \ a \ y \ b \text{ (completion } a \ y \ b)$

completionHasAll = correct3 entireL

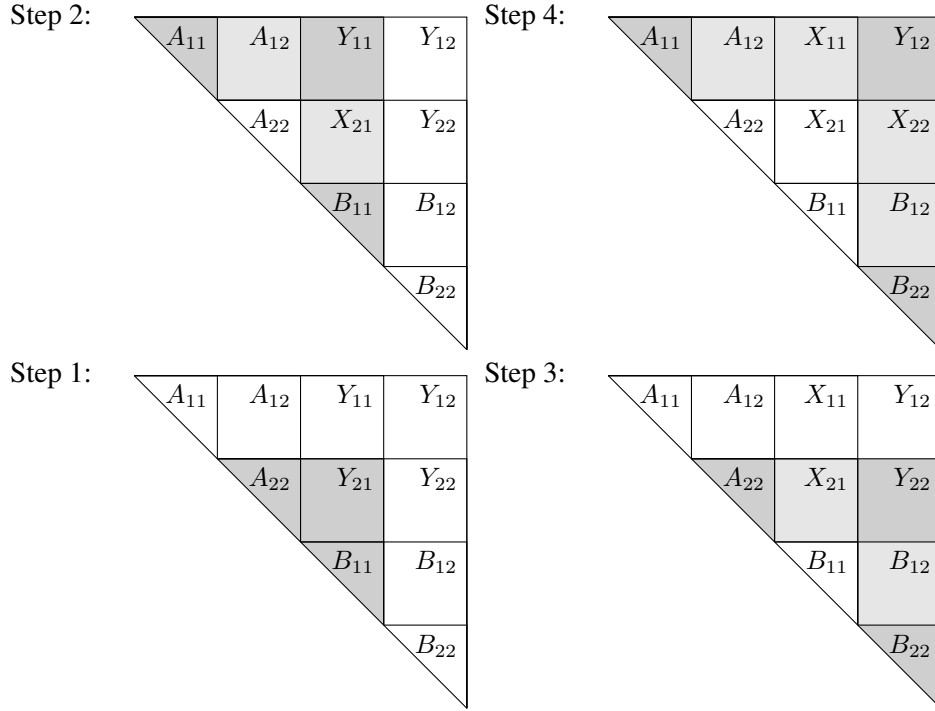


Figure 2: The recursive step of function V . The charts A and B are already complete. To complete the matrix Y , that is, compute $X = V(A, Y, B)$, one splits the matrices and performs 4 recursive calls. Each recursive call is depicted graphically. In each figure, to complete the dark-gray square, multiply the light-gray rectangles and add them to the dark-gray square, then do a recursive call on triangular matrix composed of the completed dark-gray square and the triangles.

Again, the bulk of the proof is the recursive case (part of the definition of `Square`), the lifting of `completionHasAll` to 2-by-2 block matrices. Here as well, the semi-formal proof is faithfully represented in Agda.

`entireLS` : $\forall (A : U) (Y : S) (B : U) \rightarrow \exists (\lambda X \rightarrow Y \text{ }_{+S} (A \text{ }_{U \cdot S} X \text{ }_{+S} X \text{ }_{S \cdot U} B) \simeq_S X)$

`entireLS` $A \ Y \ B = X$, **proof** **where**

$X : S; X = _$

proof : $Y \text{ }_{+S} (U2S \ A \text{ }_S X \text{ }_{+S} X \text{ }_S U2S \ B) \simeq_S X$

open `EqReasoning` (`SemiNearRing2.sSetoid` `SNR2`)

proof = -- continued below to fit the width of the paper (it is still in the **where** clause)

begin

$(Y \text{ }_{+S} (A \text{ }_{U \cdot S} X \text{ }_{+S} X \text{ }_{S \cdot U} B))$

$\equiv \langle \text{refl} \rangle$ -- name the components

let $\langle y_{00}, y_{01}, y_{10}, y_{11} \rangle = Y$

$\langle a_{00}, a_{01}, \square, a_{11} \rangle = A$

$\langle b_{00}, b_{01}, \square, b_{11} \rangle = B$

$\langle x_{00}, x_{01}, x_{10}, x_{11} \rangle = X$

in $Y \text{ }_{+S} (A \text{ }_{U \cdot S} X \text{ }_{+S} X \text{ }_{S \cdot U} B)$

```

≡⟨ refl ⟩ -- expand  $U \cdot S$  and  $S \cdot U$  and use components
  let AU*SX = ⟨ a00  $u \cdot s$  x00 +s a01  $\cdot_s$  x10 , a00  $u \cdot s$  x01 +s a01  $\cdot_s$  x11
               , 0s  $\cdot_s$  x00 +s a11  $u \cdot s$  x10 , 0s  $\cdot_s$  x01 +s a11  $u \cdot s$  x11 ⟩
    XS*UB = ⟨ x00  $s \cdot u$  b00 +s x01  $\cdot_s$  0s , x00  $\cdot_s$  b01 +s x01  $s \cdot u$  b11
             , x10  $s \cdot u$  b00 +s x11  $\cdot_s$  0s , x10  $\cdot_s$  b01 +s x11  $s \cdot u$  b11 ⟩
  in Y +S (AU*SX +S XS*UB)

≡⟨ refl ⟩ -- Expand  $\simeq_S$ , +S and collect components
  ⟨ y00 +s ((a00  $u \cdot s$  x00 +s a01  $\cdot_s$  x10) +s (x00  $s \cdot u$  b00 +s x01  $\cdot_s$  0s))
    , y01 +s ((a00  $u \cdot s$  x01 +s a01  $\cdot_s$  x11) +s (x00  $\cdot_s$  b01 +s x01  $s \cdot u$  b11))
    , y10 +s ((0s  $\cdot_s$  x00 +s a11  $u \cdot s$  x10) +s (x10  $s \cdot u$  b00 +s x11  $\cdot_s$  0s))
    , y11 +s ((0s  $\cdot_s$  x01 +s a11  $u \cdot s$  x11) +s (x10  $\cdot_s$  b01 +s x11  $s \cdot u$  b11)) ⟩

≈⟨ congS zeroLemma00 zeroLemma01 zeroLemma10 zeroLemma11 ⟩
  -- assoc. and comm. of +; zero absorption.
  ⟨ y00 +s a01  $\cdot_s$  x10 +s (a00  $u \cdot s$  x00 +s x00  $s \cdot u$  b00)
    , y01 +s a01  $\cdot_s$  x11 +s x00  $\cdot_s$  b01 +s (a00  $u \cdot s$  x01 +s x01  $s \cdot u$  b11)
    , y10 +s (a11  $u \cdot s$  x10 +s x10  $s \cdot u$  b00)
    , y11 +s x10  $\cdot_s$  b01 +s (a11  $u \cdot s$  x11 +s x11  $s \cdot u$  b11) ⟩

≈⟨ congS completionHasAll completionHasAll completionHasAll completionHasAll ⟩
  ⟨ x00, x01, x10, x11 ⟩
≡⟨ refl ⟩
  X

```

■

The series of zeroLemmas use that zeros are absorbing, in addition to commutativity and associativity of addition. One more time, the actual solution of the equation X can be inferred by Agda on the basis of the proof. The base case is a mere application of the properties of zero.

open import ZeroLemmas snr

entireLBase : (a : T) (y : s) (b : T) → ∃ (λ x → y +_s (a $u \cdot s$ x +_s x $s \cdot u$ b) \simeq_s x)

entireLBase tt y tt = y, zero^lrLemma y y y

We have now completely specified Valiant's algorithm, and it can be accessed via the appropriate field of Mat n:

valiantAlgorithm : (el : SemiNearRing) → ∀ n → Upper n el → Upper n el

valiantAlgorithm el n u = ClosedSemiNearRing.closure (Mat n el) u

8. SMALLEST

Only one bit of proof remains to obtain full correctness: namely, we should prove that the solution computed by our algorithm is a lower bound of the solution of the Q equation. This time, we skip the semi-formal outline and jump straight to the fully formal proof. We start by stating two lemmas:

1. the relation L admits a unique solution in its last argument and 2. the relation L is a congruence in its second argument.

$\text{UniqueL} = \forall \{a y b\} \rightarrow \text{UniqueSolution} \simeq_{s-} (L a y b)$
 $\text{CongL} = \forall \{a x b\} \rightarrow \forall \{y y'\} \rightarrow y \simeq_s y' \rightarrow L a y b x \rightarrow L a y' b x$

We then prove those two lemmas, as well as monotonicity of completion and the main theorem of this section (`closurelsLeast`). We do so by induction on the matrix structure. As we have seen before, the induction hypothesis is strengthened by adding fields to our `ClosedSemiNearRing` record.

field

`uniqueL` : `UniqueL`
`congL` : `CongL`
`completionMono` : $\forall \{a a' y y' b b'\} \rightarrow a \leq_u a' \rightarrow y \leq_s y' \rightarrow b \leq_u b' \rightarrow$
 $\text{completion } a y b \leq_s \text{completion } a' y' b'$
`closurelsLeast` : $\{w : u\} \rightarrow \text{LowerBound } \leq_{u-} (Q w) (\text{closure } w)$

The uniqueness of `L` immediately implies that completion gives a least solution:

open OrderLemmas snr public

`completionlsLeast` : $\forall (a : u) (y : s) (b : u) \rightarrow \text{LowerBound}_s (L a y b) (\text{completion } a y b)$
`completionlsLeast` $a y b z p = \simeq_{sTo\leq s} (\text{uniqueL } \text{completionHasAll } p)$

We then proceed with the induction proofs. First, we prove our theorem from the induction hypotheses. As usual, the upper-right corner is the difficult case, and requires the monotonicity lemma.

$\leq_{U-} = \text{SemiNearRing2}.\leq_{u-} \text{SNR2}$
 $\leq_{S-} = \text{SemiNearRing2}.\leq_{s-} \text{SNR2}$
`completionlsLeastS` : $\forall \{W\} \rightarrow \text{LowerBound } \leq_{U-} (QU W) (\text{fun entire } QU W)$
`completionlsLeastS` $\langle z_{00}, z_{01}$
 $\quad , \square, z_{11} \rangle$ -- every matrix Z
 $\quad (p_{00}, p_{01}$ -- which satisfies $(QU W Z)$
 $\quad , - , p_{11}) =$
let `q10` = `identityls 0s`
`q00` = `closurelsLeast` `z00` `p00` -- is bigger than $C = \text{fun } (\text{Ent entire } QU) W$
`q11` = `closurelsLeast` `z11` `p11`
`vs01` = `completionlsLeast` `- - - z01` `p01`
`mono01` = `completionMono` `q00` $(\simeq_{sTo\leq s} \text{refl}_s)$ `q11`
in $(q_{00}, \leq_{-} \text{trans}_s \text{mono01 } vs01$
 $\quad , q_{10}, q_{11})$

Uniqueness of `L` uses the induction hypotheses, replicating the structure derived in the previous section. The only slight difficulty is to use the identity of 0 at the appropriate places.

`uniqueLS` : `SemiNearRing2.UniqueL` `SNR2`

`uniqueLS` $(p_{00}, p_{01}, p_{10}, p_{11}) (q_{00}, q_{01}, q_{10}, q_{11}) = \text{eq00}, \text{eq01}, \text{eq10}, \text{eq11}$

where mutual

`s00` = `congL` $(\text{refl}_s \triangleleft + \triangleright (\text{refl}_s \triangleleft \triangleright \text{sym eq10}))$
`s11` = `congL` $(\text{refl}_s \triangleleft + \triangleright (\text{sym eq10} \triangleleft \triangleright \text{refl}_s))$
`s01` = `congL` $((\text{refl}_s \triangleleft + \triangleright (\text{refl}_s \triangleleft \triangleright \text{sym eq11})) \triangleleft + \triangleright (\text{sym eq00} \triangleleft \triangleright \text{refl}_s))$
`r10` = `trans` (sym zeroLemma10) `q10`
`r00` = `s00` $(\text{trans } (\text{sym zeroLemma00}) \text{q}_{00})$
`r11` = `s11` $(\text{trans } (\text{sym zeroLemma11}) \text{q}_{11})$

```

r01 = s01 (trans (sym zeroLemma01) q01)
eq10 = uniqueL (trans (sym zeroLemma10) p10) r10
eq00 = uniqueL (trans (sym zeroLemma00) p00) r00
eq11 = uniqueL (trans (sym zeroLemma11) p11) r11
eq01 = uniqueL (trans (sym zeroLemma01) p01) r01

```

Completion monotonicity is also straightforward, as is the base case. (The operators $[+]$ and $[*]$ are monotonicity for sum and product.)

```

completionMonoS : ∀ {a a' y y' b b'} → a ≤U a' → y ≤S y' → b ≤U b' →
    proj1 (entireLS a y b) ≤S proj1 (entireLS a' y' b')

```

```

completionMonoS (p00, p01, p10, p11) (q00, q01, q10, q11)
    (r00, r01, r10, r11) = m00, m01, m10, m11

```

```

where m10 = completionMono p11 q10 r00
      m00 = completionMono p00 (q00 [+] p01 [*] m10) r00
      m11 = completionMono p11 (q11 [+] m10 [*] r01) r11
      m01 = completionMono p00 (q01 [+] p01 [*] m11 [+] m00 [*] r01) r11

```

```

congLS : SemiNearRing2.CongL SNR2

```

```

congLS P Q = transS (symS P <+S> reflS) Q

```

```

leastQBase = λ _ _ → identityls 0s

```

This concludes our proof. The complete development is checked by Agda 2.4, and is available as supplementary material for this paper. The set of Agda file is written in literate programming style, and doubles up as the input for the typesetting program.

9. RELATED WORK

9.1. Efficient Parsing. One of the main motivations for this work is the discovery that Valiant's algorithm is not only interesting theoretically (as it gives an upper bound on the complexity of context-free recognition), but also practically.

Indeed [Bernardy and Claessen \[2013, 2014\]](#) have recently shown that the divide and conquer structure of Valiant's parsing algorithm yields an efficient parallel algorithm, given commonly occurring conditions on the input. In sum, if the input is organised hierarchically, then the conquer step is $O(\log^2 n)$, instead of being as complex as matrix multiplication.

9.2. Certified Parsing. Certification of parsing algorithms is a field in its infancy. Certification of regular parsing have only been carried out recently [[Firsov and Uustalu, 2013](#), [Coquand and Siles, 2011](#)]. Valiant's algorithm is an important parsing method, as it gives the best known asymptotic bounds.

A previous formalisation of Valiant's algorithm was carried by [Bååth Sjöblom \[2013\]](#) under the supervision of the authors of this paper. The present work is a redevelopment of the proof from scratch. Indeed, the proof produced by [Bååth Sjöblom](#) is opaque: its structure does not match the informal proof. A close match between the formal and the informal proof was enabled by two key design choices: 1. our representation of matrices as an (extensible) record with all the necessary lemmas and 2. the use of two different types for square and upper triangular matrices (instead of using quotient types).

9.3. Certified Parsing Combinators. In functional programming, one often uses parsing combinators as a language formalism. This means that the grammar is represented directly as code in the host language, instead of data structures. On the one hand, combinator parsing is generally less efficient than context-free parsing: the latter technique has access to the full grammar data, including its recursive structure, and thus more intelligent processing of the input is possible. On the other hand, combinators can in principle describe languages which are not context free.

[Danielsson \[2010\]](#) has formalised combinator parsing in Agda. The expressivity of Danielsson’s formalism is maximal: it can express all languages which can be decided by an Agda algorithm.

9.4. Algebra of Programming. The idea of deriving programs from specifications comes from the school of “Algebra of Programming” (AoP) [[Bird and de Moor, 1997](#)]. While AoP uses relational algebra, our setting is Agda. This paper is thus a direct descendent of “Algebra of Programming in Agda” [[Mu et al., 2008, 2009](#)], and as such leverages the power not only of Agda, but of an extensive set of standard libraries, developed over last decade mainly by [Danielsson and The Agda Team \[2013\]](#). These libraries have already been used for several applications.

Yet, during the maturation of the present work we have found the AoP canon too limited for our purposes, and have departed from it significantly. We have not derived the algorithm from the full specification, as AoP prescribes, but only part of it (Equation Q).

Further, while the AoP school prescribes to perform derivations outside the recursive structure (and thus makes extensive usage of catamorphisms and their properties), we have first expanded the recursive structure, and used equational derivations for the inductive case. This primary unrolling of recursion significantly simplified our proof, as every theorem (and lemma) proved using the same induction pattern is simply added as a component of a record (coding up the induction hypothesis).

10. EXTENSIONS

10.1. Boolean Grammars. [Okhotin \[2014\]](#) has shown how to extend Valiant’s algorithm to parse Boolean grammars. Boolean grammars allow to define the generation of non-terminals not only by union of production rules, but also intersection and complement. They can characterise non context-free languages, such as $\{a^n b^n c^n \mid n \in \mathbb{N}\}$. In this case, the ring-like structure that we have used is not sufficient: one must apply a Boolean function to all possible combination of non-terminals before obtaining the parses of a given substring.

We believe that a straightforward extension of our proof to Okhotin’s variant is possible.

10.2. Semirings and relatives. Our formalisation work showed that the specification could be generalised to work for (matrices over) arbitrary closed semirings [Dolan \[2013\]](#) and even further. Comparing to the theory explored in Dolan’s paper they mention already on their page 1: “If we have an affine map $x \rightarrow ax + b$ in some closed semiring, then $x = a * b$ is a fixpoint”. It appears that our linear equation $x = y + a \times x + x \times b$ in section 7.2 is the natural generalisation of the affine map fixed point to the case of non-commutative algebra and that (the corner case V of) Valiant’s algorithm computes this fixed point for upper triangular matrices. Future work includes exploring this relation in more detail and perhaps generalise our development to arbitrary matrices.

10.3. Sparse matrices. The efficiency of Valiant’s algorithm in the average case relies on using sparse matrices [Bernardy and Claessen, 2014]. The above proof does not deal with sparseness. Yet, it is straightforward to support sparseness as outlined by Bernardy and Claessen [2014]: one needs to change the U type to be a disjunction between the 2-by-2 case and the empty matrix case.

REFERENCES

- J.-P. Bernardy and K. Claessen. Efficient divide-and-conquer parsing of practical context-free languages. In *Proc. of ICFP 2013*, pages 111–122, 2013.
- J.-P. Bernardy and K. Claessen. Efficient parallel and incremental of practical context-free languages, 2014. Under consideration for publication in the J. of Funct. Prog.
- R. Bird and O. de Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice-Hall International, 1997.
- T. Bååth Sjöblom. *An Agda proof of the correctness of Valiant’s algorithm for context free parsing*. MSc thesis, Chalmers University of Tech., 2013.
- N. Chomsky. *Syntactic Structures*. Mouton de Gruyter, 1957.
- N. Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959.
- T. Coquand and V. Siles. A decision procedure for regular expression equivalence in type theory. In *Certified Programs and Proofs*, pages 119–134. Springer, 2011.
- N. A. Danielsson. Total parser combinators. In *Proc. of ICFP 2010*, ICFP ’10, pages 285–296. ACM, 2010.
- N. A. Danielsson and The Agda Team. The Agda standard library, version 0.7, 2013.
- S. Dolan. Fun with semirings: A funct. pearl on the abuse of linear algebra. In *Proc. of the 18th ACM SIGPLAN International Conf. on Funct. Prog.*, ICFP ’13, pages 101–110. ACM, 2013.
- D. Firsov and T. Uustalu. Certified parsing of regular languages. In *Certified Programs and Proofs*, pages 98–113. Springer, 2013.
- M. Lange and H. Leiß. To CNF or not to CNF? an efficient yet presentable version of the CYK algorithm. *Informatica Didactica* (8)(2008–2010), 2009.
- L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. of the ACM (JACM)*, 49(1):1–15, 2002.
- S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming using dependent types. In *Mathematics of Program Construction*, volume 5133/2008 of *LNCS*, pages 268–283. Springer, 2008. doi: 10.1007/978-3-540-70594-9_15.
- S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in Agda: dependent types for relational program derivation. *J. Funct. Program.*, 19:545–579, 2009. doi: 10.1017/S0956796809007345.
- U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers Tekniska Högskola, 2007.
- A. Okhotin. Parsing by matrix multiplication generalized to boolean grammars. *Theor. Comp. Sci.*, 516(0):101 – 120, 2014.
- L. Valiant. General context-free recognition in less than cubic time. *J. of computer and system sciences*, 10(2):308–314, 1975.