

Finger Trees in RUSSELL

Anonymous

Abstract

Finger Trees (Hinze and Paterson 2006) are a general purpose persistent data structure with good performance. Their genericity permits to develop a wealth of structures like ordered sequences or interval trees on top of a single implementation. On the other hand, the type systems used by current functional languages do not permit to guarantee the coherent parameterization and specialization of Finger Trees. This problem of modularity can be overcome in a system with dependent types and higher-rank polymorphism like COQ. We present a certified implementation of Finger Trees in COQ relying on the RUSSELL language which permits writing “non-total”, richly specified functional programs and elaborating them into COQ terms. We not only implement the structure but also prove its invariants along the way, which permit building certified structures on top of Finger Trees in an elegant way.

Keywords Coq, Dependent Types, Finger Trees, Certification

1. Introduction

Finger Trees are based on an optimization of 2-3 trees which gives constant or in the worst case logarithmic amortized time to the usual sequence operations, from adding an element at either end to splitting at an arbitrary location. Suffice to say that it has been integrated as the `Data.Sequence` implementation in HASKELL 6.6 to witness the fact that it is a practical, useful data structure. Our first contribution will be to show that the implementation given by Hinze & Paterson is correct, which means that all functions are terminating and have the right properties and that the datatype invariants are respected. In fact, as the invariants are part of our data structure we will also prove properties on the client side, which lead us to our second contribution, a certified implementation of random-access sequences built on top of Finger Trees. Along the way, we will present the RUSSELL language which permits writing functional programs with rich specifications in the COQ environment. The accompanying COQ scripts are available on the author’s website (Omitted for submission) and will be available as a contribution of the next COQ version.

The remaining of this paper is organized as follows: in sections 2 and 3 we give a short introduction to COQ and RUSSELL respectively, walking through simple examples. We then move on to the implementation of Finger Trees, in section 4, their specialization in section 5 and their extraction in section 6. We finally discuss our method and results in section 7. No prior knowledge of the COQ

tool is assumed, but practice of a typed functional language is recommended. Familiarity with Finger Trees is not required either.

Notations The bulk of this paper consists of a set of literate COQ files processed by the `coqdoc` tool. COQ code is typeset with the usual mathematical style; variable and definition names are typeset in *italic*, inductive types and constructor names are typeset in *sans-serif* and language keywords are typeset in *typewriter* font. An index of all the definitions used in the paper with their type or a reference to their definition is given in the appendix.

2. Preliminaries

In this section we present COQ’s underlying calculus and some features of the tool we use in the rest of the paper. It can be safely skipped by COQ users.

2.1 CIC power

COQ is a proof assistant based on the Calculus of Inductive Constructions (CIC), a very powerful type system and logical language. It relies on the Curry-Howard isomorphism between proofs and programs, types and logical statements to provide an environment in which programming and proving are seemingly intermingled. CIC is an extension of the Calculus of Constructions (CC), the richest type system of Barendregt’s λ -cube, which integrates polymorphic, dependent types with highly expressive (co-)inductive types (Coq). The environment built around this kernel permits to do interactive theorem proving of mathematical developments and certified programming using the underlying λ -calculus. It provides facilities to define datatypes and functions like a regular functional programming language but it also permits to write rich specifications and construct proofs in the same language viewed as a higher-order logic.

We present the typing rules of CC in figure 1 as a reminder. We will extend them when defining RUSSELL’s type system in section 3. The basic calculus is church-style λ -calculus plus a set of sorts $S = \{\text{Prop}\} \cup \{\text{Type}(i) \mid i \in \mathbb{N}\}$ and the dependent product $\Pi x : A, B$ which types abstractions (rules PROD, ABS, APP). We do not detail the relation \mathcal{R} which allows an impredicative Prop sort and a predicative Type hierarchy. The sort $\text{Type}(0)$ represents computational types (e.g. lists, naturals) and Prop logical propositions. Both sorts are typed by $\text{Type}(1)$ which is itself typed using the rule TYPE. An essential rule is CONV which internalizes the fact that β -convertible types can be considered equal during type-checking (\equiv is the β -equivalence relation). Here lies the true power of the system: computation can be interleaved with reasoning anywhere. We do not treat the full kernel of COQ, which also contain sort subtyping and inductive datatypes as they are not essential to describe the core features RUSSELL will add; inductives will be described next by example.

Important properties of this system are Subject Reduction and Strong Normalization, i.e.: all well-typed terms reduce to a value in a finite number of reductions, which ensures that typing is decidable in the CONV rule. With such expressive power and strong

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

$$\begin{array}{c}
\text{WF-EMPTY} \frac{}{\vdash \square \text{ wf}} \quad \text{WF-VAR} \frac{\Gamma \vdash A : s}{\vdash \Gamma, x : A \text{ wf}} \quad s \in \mathcal{S}, x \notin \Gamma \\
\\
\text{PROP} \frac{\vdash \Gamma \text{ wf}}{\Gamma \vdash \text{Prop} : \text{Type}(1)} \\
\\
\text{TYPE} \frac{\vdash \Gamma \text{ wf}}{\Gamma \vdash \text{Type}(i) : \text{Type}(i+1)} \quad i \in \mathbb{N} \\
\\
\text{VAR} \frac{\vdash \Gamma \text{ wf} \quad x : A \in \Gamma}{\Gamma \vdash x : A} \\
\\
\text{PROD} \frac{\Gamma \vdash T : s_1 \quad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash \Pi x : T. U : s_3} \quad (s_1, s_2, s_3) \in \mathcal{R} \\
\\
\text{ABS} \frac{\Gamma, x : T \vdash M : U \quad \Gamma \vdash \Pi x : T. U : s}{\Gamma \vdash \lambda x : T. M : \Pi x : T. U} \\
\\
\text{APP} \frac{\Gamma \vdash f : \Pi x : V. W \quad \Gamma \vdash u : V}{\Gamma \vdash (fu) : W[u/x]} \\
\\
\text{CONV} \frac{\Gamma \vdash t : U \quad \Gamma \vdash U \equiv T : s}{\Gamma \vdash t : T}
\end{array}$$

Figure 1. Calculus of Constructions

metatheoretical properties comes verbosity and inflexibility. We will present in the following sections what mechanisms help overcome these difficulties in the COQ tool. But first we will present the salient features of inductives in COQ.

Inductive types Inductive types are the algebraic datatypes of Type Theories, only more powerful. As usual algebraic datatypes, they are equivalent to their ML counterpart. For example we may define the standard polymorphic lists using the declaration:

```

Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A.

```

Here A is an *inductive parameter* representing the type of elements of the list and each constructor is declared with a product type from its arguments types to the inductive. This presentation as a product comes from the fact that constructor arguments may depend on each other and parameters may change. For example we may declare vectors (lists of fixed length) as:

```

Inductive vector (A : Type) : nat → Type :=
| vnil : vector A 0
| vcons : A → ∀ n, vector A n → vector A (S n).

```

The inductives of COQ (Paulin-Mohring 1993) are in fact parameterized, possibly nested, mutually recursive inductive families. An inductive family is a family of types indexed by a type or a value. Here for example, vector is a family parameterized by a type A of elements and indexed by the naturals nat . The vnil constructor is the empty vector of length 0 and the vcons constructor builds a vector of length $S n$ from an object of type A and a vector of length n .

Finally, we can define nested inductives using non-uniform parameters; this feature is needed to be able to define Finger Trees. A classical example of a nested datatype is the power list of A where the tail of a list is a list of pairs of A . The important point is that we can reinstantiate the parameter A in the constructors.

```

Inductive pow_list (A : Type) : Type :=
| pow_nil : pow_list A
| pow_cons : A → pow_list (A × A) → pow_list A.

```

```

Fixpoint pow_map (A B : Type) (f : A → B)
  (l : pow_list A) { struct l } : pow_list B :=
match l with
| pow_nil ⇒ pow_nil B
| pow_cons hd tl ⇒ pow_cons B (f hd)
  (pow_map (A × A) (B × B)
    (fun x : A × A ⇒ (f (fst x), f (snd x)))) tl)
end.

```

Figure 2. pow_map definition

We can of course recurse on elements of these inductive types and in the last case we will even use polymorphic recursion, a feature the type system of ML's lacks. For example the map combinator on power lists can be defined as in figure 2.

To ensure the aforementioned properties are preserved when adding inductives, the kernel implements an incomplete and relatively inflexible syntactic criterion to discard non-structurally recursive definitions where the structurally decreasing argument is given by the $\{\text{struct } id\}$ annotation. In the pow_map example, it checks that tl is indeed a subterm of l . In this definition we also face the foremost difficulty in using CIC as a programming language: verbosity. Clearly, writing all these types is not viable from a software development point of view, so we must find a way to overcome this problem while still retaining decidability of inference and type-checking. COQ's solution to this problem is to use a higher-level language GALLINA which elaborates/compiles into CIC. This approach does not change the kernel's language *at all*, so we keep all the nice properties once our definitions have compiled into it.

2.2 COQ surface: GALLINA

We will now present a few of COQ's surface language features which permit to manage the verbosity and abstraction power of CIC: implicit arguments to omit some type information, notations to abbreviate and beautify the otherwise obfuscated definitions and sections to easily parameterize whole developments.

2.2.1 Recovering inference

Implicit arguments systems permit writing terms omitting as much inferable information as possible. Its effect is best described by an example application. Suppose we want to apply the pow_map function to a function $f : A \rightarrow B$ and a power list $l : \text{pow_list } A$. Without implicit arguments, we must repeat the type of f as the first arguments of the function, like this: $\text{pow_map } A B f l$. In implicit arguments mode, the first two arguments become implicit as the system knows that they will be inferable at application time from the *type* of the function, hence we can write $\text{pow_map } f l$ to achieve the same effect. This permits to write definitions with a syntax very similar to languages based on Hindley-Milner type inference, e.g. the pow_map definition comes down to the code in figure 3.

```

Fixpoint pow_map (A B : Type) (f : A → B)
  (l : pow_list A) { struct l } : pow_list B :=
match l with
| pow_nil ⇒ pow_nil
| pow_cons hd tl ⇒ pow_cons (f hd)
  (pow_map (fun x ⇒ (f (fst x), f (snd x))) tl)
end.

```

Figure 3. pow_map definition with implicit arguments

2.2.2 Notations: pretty parsing/printing

Notations are kind of hygienic macros permitting to abbreviate or beautify specifications and generally give a more mathematical feel to writing COQ. For example we can define the type representing the subset of elements of a type A having a particular property P as in figure 4.

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
| exist : ∀ a : A, P a → sig A P.
```

Figure 4. Subset type in Coq

An element of the type $\text{sig } A \ P$ is a tuple $\text{exist } A \ P \ x \ p$ where x is a witness of type A and p is of type $P \ a$, i.e. p is a proof that the predicate P holds for a . However it is much clearer to write this type as $\{x : A \mid P\}$ rather than $\text{sig } P$ and this is what the notation system permits.

An alternative encoding of the vector inductive structure may be the lists of fixed lengths which we can define using this notation (the types of A and n can be inferred automatically in this case):

```
Definition vector A n := { x : list A | length x = n }.
```

One can define arbitrary notations, possibly for binding constructs, and assign them priorities (levels) and associativities. For example we may define composition using the symbol \circ as a notation:

```
Notation "g 'o' f" := (fun x => g (f x))
(at level 20, left associativity).
```

Armed with these tools we can start our first development.

2.2.3 Sections: implementing the monoid typeclass

As a gentle introduction to COQ, we will build a monoid datatype corresponding to the Monoid typeclass of HASKELL. We recall its signature:

```
class Monoid m where
  () :: m
  (⊙) :: m → m → m
```

Instances should make sure the following properties hold:

```
left identity   () ⊙ x = x
right identity  x ⊙ () = x
associativity    x ⊙ (y ⊙ z) = (x ⊙ y) ⊙ z
```

We first introduce the monoid laws we just described. We will use the section mechanism of COQ extensively in the following. Sections permit to write a bunch of definitions which are parameterized by some variables, e.g. for types and operations. When closing a section, every definition inside it is quantified over the variables it used.

Section *Monoid_Laws*.

The carrier m type may be any type.

```
Variable m : Type.
```

The identity element *empty* and the operation *mappend*.

```
Variables (empty : m) (mappend : m → m → m).
```

We define fancy notations for the element and operation.

```
Notation ε := empty.
Infix "⊙" := mappend (right associativity, at level 20).
```

We now state the properties.

```
Definition monoid_id_l_t : Prop := ∀ x, ε · x = x.
```

```
Definition monoid_id_r_t := ∀ x, x · ε = x.
```

```
Definition monoid_assoc_t := ∀ x y z, (x · y) · z = x · y · z.
```

End *Monoid_Laws*.

Every variable in *Monoid_Laws* has been discharged by now, so we must apply each definition to particular *empty* and *mappend* objects. We can finally define the dependent record which represents monoids on m :

```
Record monoid (m : Type) : Type := mkMonoid
{ empty : m ; mappend : m → m → m
; monoid_id_l : monoid_id_l_t empty mappend
; monoid_id_r : monoid_id_r_t empty mappend
; monoid_assoc : monoid_assoc_t mappend }.
```

We will come back to monoids when implementing Finger Trees. We will now present the RUSSELL language which was the essential tool we needed to develop them.

3. RUSSELL in COQ

The lack of distinction between proofs and programs in Type Theory helps having a clear, unified semantics for the complete system but hinges users and implementers alike, because sometimes you do not want to have proofs polluting computations (e.g. when evaluating a term, the way proofs are constructed should not matter, only the algorithm) and you certainly do not want to write complex programs using a procedural language with automation like the language used in proof scripts of COQ but directly inputting terms and have them interpreted in COQ's kernel language. A solution to the first problem is given by the Prop/Type distinction in COQ. If Prop is used as the type of propositions (e.g. conjunction, negation) and Type as the type of computational types (e.g. naturals, lists) then we can immediately see whether something is a proof or a program just by looking at its sort. This is used by the extraction mechanism (Letouzey 2002) to extract *only* the algorithmical code from a COQ term. However, this does not solve the problem of writing complex programs in COQ and it is not used directly inside the proof-checker either, so computations in COQ may freeze because of irreducible proofs in the term, even when they are irrelevant; we will discuss this later issue in section 7. For now we will concentrate on the problem of writing richly specified programs in COQ.

Usually, as soon as you have propositions appearing in the type of a function, you will write it in COQ using tactics instead of using the functional language because otherwise you would have to directly manipulate proof terms. This is a showstopper because it becomes intractable as soon as you have a complex proof to do, especially since there is not much support for incremental refinement in COQ, contrary to AGDA or EPIGRAM (McBride and McKinna 2004), so you have to get the complete proof term right in one step.

RUSSELL is a programming language built on top of COQ which permits to write only the algorithmical code of strongly specified functions and forgetting about the required proofs which COQ needs to establish correctness (i.e. typecheck the term). Its main purpose is to act as a convenient, permissive source language which elaborates into CIC, which can be type-checked by the safe kernel of COQ. Here, permissive means the user is not restricted to using structural recursion, complete pattern-matching and total functions only, as we have seen up to now. Obviously, you have to give evidence that your source term can be seen as a legitimate COQ term, which generally means that what you left out can be proved. This is done by solving obligations which are generated *after* the type-checking procedure of RUSSELL terms by an interpretation into CIC. For a detailed presentation of this procedure, the reader is directed to (Sozeau 2006). The generation of obligations is done in

a similar way as in PVS (Owre and Shankar 1997), using subsets to carry propositions. We will describe this essential feature of the RUSSELL type system next, then move on gradually to our motivating example: Finger Trees.

3.1 Subset equivalence

Subset equivalence is a simple idea: two subsets over the same carrier may be considered equal, given a proof that the subsets are equivalent. We can leave the properties out and just consider the carrier of subsets. That is, if we have an object t of type T where we expect an object of type $\{x : T \mid P\}$, we may say “ok, you will have to give me a proof that $P[t/x]$ holds but we are good for now”. Conversely, it is always permitted to consider an object of type $\{x : T \mid P\}$ as an object to type T . This is the essential idea behind the type system of RUSSELL: it checks type equivalence on the carriers only, ignoring if the properties really hold. We need only enrich the type conversion relation of CIC with the rules given figure 5 to formalize this idea.

$$\frac{\Gamma \vdash T \equiv U : \text{Type} \quad \Gamma, x : U \vdash P : \text{Prop}}{\Gamma \vdash T \equiv \{x : U \mid P\} : \text{Type}} \quad \frac{\Gamma \vdash T \equiv U : \text{Type} \quad \Gamma, x : T \vdash P : \text{Prop}}{\Gamma \vdash \{x : T \mid P\} \equiv U : \text{Type}}$$

Figure 5. Subset equivalence rules

As an example, we can derive:

$$\frac{\Gamma \vdash \mathbb{N} \equiv \mathbb{N} : \text{Type} \quad \Gamma, x : \mathbb{N} \vdash x \neq 0 : \text{Prop}}{\Gamma \vdash 0 : \mathbb{N} \quad \Gamma \vdash \mathbb{N} \equiv \{x : \mathbb{N} \mid x \neq 0\} : \text{Type}} \quad \frac{}{\Gamma \vdash 0 : \{x : \mathbb{N} \mid x \neq 0\}}$$

After a term has been typed in this system, we can automatically generate the obligations and produce a full-fledged, well-typed COQ term once these are solved. Following the example, a single obligation demanding that $0 \neq 0$ be proved will be generated which will certainly not be solved except if Γ is inconsistent. This phase distinction is essential: as soon as we want to do any proof we jump into an undecidable world because we have the full higher-order logic of COQ at hand, so we must not have any proving to do if we want our type-checking to be decidable. This is to contrast with the approaches of other dependently-typed programming languages, from DML (Xi and Pfenning 1999) to EPIGRAM. In these languages either the expressive power is limited so that an automatic prover can solve obligations during type-checking (e.g. Pressburger arithmetic in DML) or there is no limit but proofs and code are intermingled in the same language (for EPIGRAM, COQ and AGDA) or a mix of the two styles in ATS (Xi 2004) or Ω MEGA (Sheard) for example.

3.2 RUSSELL’s incarnation: PROGRAM

Using RUSSELL we can write arbitrarily complex code with arbitrarily complex specifications. It will typecheck because we do not care about proofs at the type-checking step but later, when recombining obligations with the algorithmical skeleton we wrote. Let us define the strongly specified euclidean division of a and b where b must be different from 0. This function is defined by well-founded recursion using the less-than order on the argument a :

```
Program Fixpoint div (a : nat) (b : nat | b ≠ 0) { wf lt } :
  { (q, r) : nat × nat | a = b × q + r ∧ r < b } :=
  if less_than a b then (0, a)
  else dest div (a - b) b as (q', r) in (S q', r).
```

The Program vernacular indicates that we use the RUSSELL type-checker followed by the interpretation into COQ instead of COQ’s

original type-checker. The construct `dest t as p in b` is equivalent to a destructive `let` construct where p is an arbitrary pattern (permitting to pull values from nested tuples for example) and we use sugar for the binder $(x : \text{nat} \mid P) \triangleq (x : \{x : \text{nat} \mid P\})$.

This code type-checks and PROGRAM generates obligations which are discharged automatically by COQ’s tactics except the one for the inductive case. We can discharge it interactively using COQ’s proof language. When all obligations are solved the PROGRAM system adds the completed definition of *div* in the environment. We can then use *div* like any other COQ object, reasoning and computing with it as desired.

We were able to discharge the obligations because hypotheses $a < b$ or $a \geq b$ were present in their contexts. However, these hypotheses are not automatically generated by COQ, they come directly from the term which uses a dependent function *less_than* of type: $\forall x y, \{x < y\} + \{x \geq y\}$. Had we simply used a boolean function instead, we would have no information about the comparison in the obligations. Hence we must always take care of *reflecting* the logic in our code if we want to reason on it later. In case of boolean conditionals, we can always use the combinator *dec* : $\forall b, \{b = \text{true}\} + \{b = \text{false}\}$ to achieve this reflection. The *if* construct works on any inductive type with two constructors.

3.3 Extension to inductive types

It is natural to extend the support for subset types in RUSSELL to inductive families. For example, we have seen we can have two encodings of vectors, one using subsets and another using an inductive definition. The facility we added for handling subset types should then transpose easily to inductive types. Indeed we can obtain the RUSSELL derivation:

$$\frac{\Gamma \vdash x : \text{vector } A \ m \quad \Gamma \vdash \text{vector } A \ m \equiv \text{vector } A \ n : \text{Type}}{\Gamma \vdash x : \text{vector } A \ n}$$

The type conversion results from an instantiation of the rule given figure 6 which states that an inhabitant of an inductive family I with indexes \vec{x}_i also inhabits the type $I \vec{y}_i$.

$$\frac{I \text{ inductive} \quad \Gamma \vdash I \vec{x}_i, I \vec{y}_i : \text{Type}}{\Gamma \vdash I \vec{x}_i \equiv I \vec{y}_i : \text{Type}}$$

Figure 6. Dependent inductive equivalence rule

When interpreting into COQ, proof obligations will be generated to show that indices are equal. For the example above, a proof that m equals n will be required. This extension was crucial in the development of Dependent Finger Trees where a good part of the obligations are generated due to applications of this rule.

Dually to this construction permitting to construct an inductive object with arbitrary index, we have enriched the pattern-matching construct to get information about destructured values and their indexes (i.e. automatically reflecting pattern-matching at the logical level). Informally, our construction permits to type each branch of a pattern-matching construct in a context extended with equalities between the patterns and the matched values up to their indices. For example, when matching a term v of type $\text{vector } A \ (S \ n)$ with two branches having patterns vnil and $\text{vcons } x \ n' \ v'$, the first branch will be typed in a context containing hypotheses $S \ n = 0$ and $v \simeq \text{vnil}$ (hence an absurd case). The symbol \simeq denotes heterogeneous equality which will be presented in § 4.4.1. We will also be able to deduce that $n = n'$ in the second branch because we will have an equality $S \ n = S \ n'$ in the context (and constructors are injective). As a further refinement of pattern-matching, we add inequality hypotheses in the context of branches for which the pattern was intersecting with a previous one. Consider the following code:

```

Program Fixpoint div2 (n : nat) :
  { n' : nat | n = 2 * n' ∨ n = 2 * n' + 1 } :=
  match n with
  | S (S p) ⇒ S (div2 p)
  | x ⇒ 0
  end.

```

The second branch is typed in a context extended with the hypotheses $x = n$ and $\forall p, x \neq S (S p)$, which is crucial to solve the obligation $n = 2 * 0 \vee n = 2 * 0 + 1$.

A conclusion on RUSSELL We think having an unrestricted power of expression in the logic is essential to having useful software development and certification tools and our method permits to achieve it without losing the facilities we are accustomed to in a practical programming language. So we get decidability of type-checking and our programs look exactly like their simply-typed, functional counterparts with no proof handling code appearing in the term but we also have a highly expressive logic and a complete system to prove and check statements in this logic. This combination gives us a certified programming language with unmatched support for proving.

4. Dependent Finger Trees

We will now illustrate RUSSELL's expressiveness by presenting a certified implementation of Finger Trees in COQ. Finger Trees (Hinze and Paterson 2006) are a functional, general purpose, efficient data structure based on an optimization of 2-3 trees. It is implemented in HASKELL as a *nested* datatype which is parameterized by a type of *measures*, which can be instantiated by various types to give specializations of the Finger Trees. The data structure builds upon simpler structures like digits and 2-3 nodes which we will present first.

4.1 Digits

A digit is simply a buffer of one to four values.

Section *Digit*.

```

Variable A : Type.
Inductive digit : Type :=
| One : A → digit
| Two : A → A → digit
| Three : A → A → A → digit
| Four : A → A → A → A → digit.

```

We build simple functional predicates on digits for use in specifications.

```

Definition full (x : digit) :=
  match x with Four _ _ _ _ ⇒ True | _ ⇒ False end.

```

We now define addition of an element to the left of a digit.

```

Program Definition add_digit_left (a : A)
  (d : digit | ¬ full d) : digit :=
  match d with
  | One x ⇒ Two a x
  | Two x y ⇒ Three a x y
  | Three x y z ⇒ Four a x y z
  | Four _ _ _ _ ⇒ !
  end.

```

It has become interesting here, as we define a *partial* function. We can add to a digit if and only if it is not already full. So, we require the argument digit to be accompanied by a proof that it is not full. We will use it to prove that the last branch is inaccessible; we use the notation ! ('bang') to denote inaccessible program points, which

```

a      : A
d      : digit
Hd     : ¬ full d
x, y, z, w : A
Heq_d  : d = Four x y z w
False

```

Figure 7. Obligation of *add_digit_left*

are points where False can be proved. Note that we can pattern-match on d as if it was just a digit: properties have no influence on code, only on proofs.

The generated obligation (figure 7) is easily solved, as we have a contradiction in the context: both $\neg \text{full } d$ and $d = \text{Four } _ _ _ _$ are present. We can define in a similar fashion addition on the right and the various accessors on digits (*head*, *tail*, *last* and *liat*). From now on we will omit the obligations and the proof scripts used to solve them; they can be found in the COQ contribution.

As an other example of the usefulness of PROGRAM, consider defining the monoid $([], ++)$ on lists. To construct a monoid implementation (c.f. §2.2.3), we have to apply the `mkMonoid` constructor to `[], ++` and proofs of identity and associativity. Fortunately, those can be filled automatically by COQ using lemmas proved in the standard library. When PROGRAM processes the following definition, it turns the holes $(_)$ in the term into obligations, which are thus automatically discharged. We explicitly give the element and operation which are considered implicit by the system as they are derivable from the types of the proofs.

```

Program Definition listMonoid (A : Type) : monoid (list A) :=
  mkMonoid (empty:=nil) (mappend:=app) _ _ _ .

```

We shall now define the Finger Trees over some monoid and measure. We will see in section 5 how particular instantiations of these permit to obtain different structures from this implementation.

Section *DependentFingerTree*.

Variables $(v : \text{Type})$ (*mono* : monoid v).

The variable v is the monoid carrier type, *mono* is the monoid implementation. As before, we use ε and \cdot for the monoid empty element and operation respectively.

4.2 Nodes

Finger Trees are implemented using 2-3 nodes as in 2-3 trees, with the addition of a cached value that stores the combined measure of the subobjects.

Section *Nodes*.

Variables $(A : \text{Type})$ (*measure* : $A \rightarrow v$).
 Notation "||' x ||'" := (*measure* x).

Intuitively, the measure represents an abstraction of the sequence of objects in the node (later, in the tree) considered from left to right. Taking the list monoid previously defined and the measure `[-]` gives the most concrete abstraction of this sequence of elements: the sequence itself. The *measure* function is denoted by `||_||` in the following, when possible (notations local to a function are not supported). We now define 2-3 nodes with cached measures:

```

Inductive node : Type :=
| Node2 : ∀ x y, { s : v | s = || x || · || y || } → node
| Node3 : ∀ x y z, { s : v | s = || x || · || y || · || z || } → node.

```

We use a subset type here to specify the invariant on the cached value. This invariant could not be checked using simple types,

because it is dependent on the values carried by the node. Moreover, it uses the *measure function* as a *datatype parameter* so it would require an additional abstraction mechanism like type classes or functors to achieve it in a regular ML-based functional language. A regular dependent product is used here instead.

We have smart constructors *node2* and *node3* which compute the measure, e.g:

```
Program Definition node2 (x y : A) : node :=
  Node2 x y (|| x || · || y ||).
```

The generated obligations are trivially true as they are of the form $x = x$. Correspondingly, *node_measure* pulls the cached measure.

```
Program Definition node_measure (n : node) : v :=
  match n with Node2 _ _ s => s | Node3 _ _ _ s => s end.
End Nodes.
```

Although it may seem that the *node_measure* function is independent of the measure, it is not. Witness its type after we closed the section:

$$\forall (A : \text{Type})(\text{measure} : A \rightarrow v), \text{node_measure} \rightarrow v$$

The node datatype itself is parameterized by the measure function, hence all operations on nodes take it as a parameter. Had we not added the invariant, we would not need this parameter, but we could not prove much about node measures either, because they might be any element of type v . We could define an inductive invariant on nodes asserting that their measures were built using the measure coherently, but we would need to show that this invariant is preserved in each of the functions, while in our case we get it for free by typing.

4.3 The fingertree datatype

Before presenting the definition of *fingertree* in COQ, we introduce the original HASKELL datatype and justify why the direct translation would be unsatisfactory. In HASKELL, the *FingerTree* algebraic datatype is defined as:

```
data FingerTree v a = Empty | Single a
  | Deep v (Digit a) (FingerTree v (Node v a)) (Digit a)
```

The *Empty* constructor represents the empty tree, the *Single* constructor builds a singleton tree from an element of a and the *Deep* constructor is the branching node. It builds a *FingerTree v a* from a cached measure of type v , two digits of a and a middle *FingerTree v (Node a)* of 2-3 *nodes* of a with measures of type v . This is where the nesting occurs; it is best described by a picture (figure 8).

We could directly define the *Finger Tree* data structure in COQ by translating the HASKELL definition. However doing so would cause a rather subtle problem: the cached measures in two *Finger Trees* could be computed by two *different* measure functions but these trees would still have the same type as the *FingerTree* datatype is parameterized only by the type of the computed measures and not by the measure function. Again, we could live with it and have a predicate formalizing the fact that the measures appearing in a *FingerTree* were built coherently with some measure function, but preservation of this invariant would have to be proved for each new definition on *Finger Trees*. We prefer to view the measure function as an additional parameter of the *Finger Tree* datatype. In this case coherence is ensured by the type system because two *Finger Trees* will have different types if they are built from different measure functions. Hence the following definition:

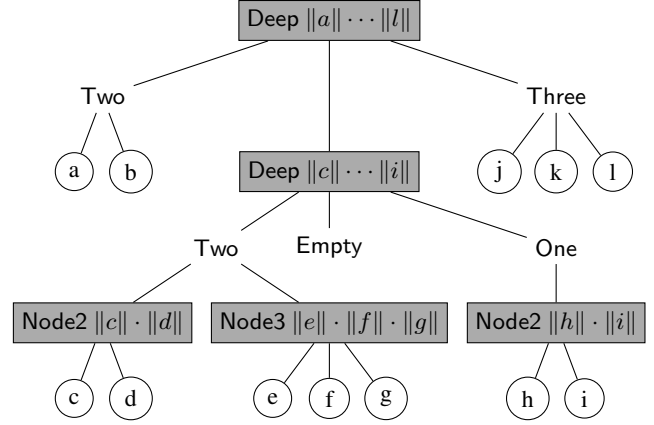


Figure 8. A sample *FingerTree*. At the first level we have a *Deep* node with a 2-digit of a at the left and a 3-digit at the right. The middle tree is a *FingerTree v (Node a)* consisting of a *Deep* node with an *Empty* middle tree. It has a 2-digit of *Node a* at its left and a 1-digit at its right. The blank nodes represents elements of type a and the filled ones indicates loci where cached measures are stored

A *fingertree* inductive is parameterized by a type A and a measure function $\text{measure} : A \rightarrow v$ on this type. Each *fingertree* object is also indexed by its corresponding measure:

- *Empty* builds the empty tree of measure ε ;
- *Single x* builds the tree with sole element x of measure $\|x\|$;
- *Deep pr ms m sf* builds the tree of prefix pr , subtree m of measure ms and suffix sf . Its measure is given by combining the measures of its subtrees. The argument ms will be implicit when constructing *Deep* nodes, as it can be inferred from the type of m . We place this argument ms just before the middle *FingerTree* contrary to the original datatype in which the cached measure is the first field and stores the measure of the *whole* tree whereas for us the latter is given by the index.

We present the definition using inference rules for enhanced readability, omitting A and measure in the premises:

$$\frac{}{\text{Empty} : \text{fingertree } A \text{ measure } \varepsilon}$$

$$\frac{x : A}{\text{Single } x : \text{fingertree } A \text{ measure } (\text{measure } x)}$$

$$\frac{pr, sf : \text{digit } A, ms : v}{\text{Deep } pr \ ms \ m \ sf : \text{fingertree } A \text{ measure } (\text{digit_measure } measure \ pr \cdot ms \cdot \text{digit_measure } measure \ sf)}$$

As we have already seen, this inductive type is not regular and requires *polymorphic* recursion to go into the subtree of a *Deep* node. It is also a dependent type indexed by values of type v . Why we need a dependent type follows from a simple observation. If we want the cached measure on the *Deep* node and the invariant that the measure is really the one of the middle tree, we must have a way to refer to the measure of the middle tree, but we are actually defining trees, so we cannot recurse on them at this time.

Adding elements We can add an element a to the left of a tree t of measure s to get a tree with measure $\|a\| \cdot s$. Due to polymorphic recursion, our functions will now always have A and measure

arguments as they are *real* arguments which will change during recursive calls.

```

Program Fixpoint add_left A (measure : A → v) (a : A)
  (s : v) (t : fingertree measure s) {struct t} :
  fingertree measure (measure a · s) :=
  match t in fingertree _ s
  return fingertree measure (measure a · s) with
  | Empty ⇒ Single a
  | Single b ⇒ Deep (One a) Empty (One b)
  | Deep pr st' t' sf ⇒
    match pr with
    | Four b c d e ⇒
      let sub := add_left (node3 measure c d e) t' in
      Deep (Two a b) sub sf
    | x ⇒ Deep (add_digit_left a pr) t' sf
  end
end.

```

The first match expression uses dependent elimination. Its meaning is that from a particular t of measure s each branch will build a t' of measure $measure\ a \cdot s$ where s has been substituted by the measure of the matched pattern. For example in the first branch we must build an object of type $fingertree\ measure\ (measure\ a \cdot \varepsilon)$. However, the right hand side $Single\ a$ has type $fingertree\ measure\ (measure\ a)$, hence we must use the inductive equivalence rule (figure 6), which generates an obligation $\vdash\ measure\ a \cdot \varepsilon = measure\ a$, easily solved using the right identity of the monoid.

The nested match expression uses an alias for capturing prefixes which are not *Four* and applies the partial function *add_digit_left* defined earlier. There is an application of the subset equivalence rule here, which generates an obligation to show that pr is not full. It can be solved because we have $\forall b\ c\ d\ e, pr \neq Four\ b\ c\ d\ e$ in the context.

It is an essential property of this function that the measure is preserved. To see that, let us instantiate the Finger Tree by the list monoid $([], ++)$ and measure $[-]$ defined earlier. You can check here that adding to the left is prepending the measure of the element to the measure of the Finger Tree, hence consing to the sequence of elements of the tree with the list monoid interpretation. For each of the following operations, this adequacy with lists will hold.

4.4 The view from the left... of a fingertree

We will now construct views on the Finger Trees which permit to decompose a tree into its leftmost element and a remaining Finger Tree (*View_L*) (or dually, the rightmost one and the "preceding" Finger Tree). It serves to abstract from the implementation and give a list-like interface to *fingertree*. The *View_L* inductive is a little less polymorphic, as it does not need carry the measure function which views ignore. However *View_L* still stores the measure s of the rest of the tree in the *cons_L* constructor (s will be implicit).

```

Inductive View_L (A : Type) (seq : v → Type) : Type :=
| nil_L : View_L A seq
| cons_L : A → ∀ s, seq s → View_L A seq.

```

Such a view will be constructed by the *view_L* function, by structural (polymorphic) recursion on the *fingertree*. We can seamlessly use the *digit_tail* function which is partial (it accepts only *digit* which are not *One*) and need not add any type annotations when calling *view_L* recursively on t' . Note that we use a partial type application (*fingertree measure*) in the return type, which is perfectly legal.

```

Program Fixpoint view_L (A : Type) (measure : A → v)
  (s : v) (t : fingertree measure s) {struct t} :
  View_L A (fingertree measure) :=
  match t with
  | Empty ⇒ nil_L
  | Single x ⇒ cons_L x Empty
  | Deep pr st' t' sf ⇒
    match pr with
    | One x ⇒
      match view_L t' with
      | nil_L ⇒ cons_L x (digit_to_tree _ sf)
      | cons_L a st' t' ⇒
        cons_L x (Deep (node_to_digit a) t' sf)
      end
    | y ⇒ cons_L (digit_head y) (Deep (digit_tail y) t' sf)
    end
  end.

```

We can show that *view_L* is measure-preserving; had we used a dependent *View_L* inductive these generation lemmas would have appeared as obligations.

```

Lemma view_L_nil : ∀ A (measure : A → v) s
  (t : fingertree measure s), view_L t = nil_L → s = ε.

```

```

Lemma view_L_cons : ∀ A (measure : A → v) s
  (t : fingertree measure s) x st' t',
  view_L t = cons_L x (s := st') t' → s = measure x · st'.

```

4.4.1 Dependence hell

Our views are useful to give an high-level interfaces to Finger Trees, but in their current state they are very limited as we can write only non-recursive definitions on views. That is, we will not be able to convince COQ that functions defined by recursion on the view of a tree is just as valid as recursion on the tree itself. To do that, we must have a measure on our Finger Trees, e.g. their number of elements (c.f. *tree_size*), from which we can trivially build a measure on the views *View_L_size*. Then it is only a matter of showing that for all t , *view_L t* returns a view of size *tree_size t* to prove that a recursive call on the tail of a view is correct (c.f. §5.1).

However, doing such a thing is not as easy as it looks because *view_L* manipulates objects of dependent types and reasoning about them is somewhat tricky. An essential difficulty is that the usual Leibniz equality is not large enough to compare objects in dependent types as they may be comparable but in different types. A simple example is that the statement $vnil = vcons\ x\ n\ v$ is not well-typed because $vnil$ is of type $vector\ 0$ and $vcons\ x\ n\ v$ of type $vector\ (S\ n)$ which are not convertible.

In our case, we want to say that an arbitrary tree t of measure s with *view nil_L* must be the *Empty* tree, but those two trees do not have the same type. We apply the usual trick of heterogeneous equality: prove they must be in the same type. The inductive *JMeq* defines an heterogeneous equality (previously denoted by \simeq), which permits to compare objects which are not in the same type. Its sole constructor is *JMeq_refl* of type $\forall A\ a, JMeq\ A\ a\ A\ a$. The point of this admittedly strange notion of equality is to postpone the moment at which we need to prove that the types of the compared objects are equal. When we arrive at this point, we can apply the *JMeq_eq* axiom of type $\forall A\ x\ y, JMeq\ A\ x\ A\ y \rightarrow x = y$ to get back a regular equality between the objects.

In the first lemma, we compare t of measure s with *Empty* of measure ε ; clearly, replacing *JMeq* by regular equality would yield a type error here.

This inductive combines both subsets and dependent inductive types, but we can still write our code as usual. In the definition below, we use a *pr* argument of unit type to carry a proposition on *t* to work around the structural recursion criterion of COQ which does not work “up-to subsets” whereas a type-based criterion would accept the more natural definition. Approximately 100 lines of proof are necessary to discharge the generated obligations.

```
Program Fixpoint split_tree' (A : Type) (measure : A → v)
  (i s : v) (t : fingertree measure s) (pr : unit | → isEmpty t)
  {struct t} : tree_split measure i s := ...
```

We can of course define a wrapper to get rid of the unit argument.

```
Program Definition split_tree (A : Type) (measure : A → v)
  (i s : v) (t : fingertree measure s | → isEmpty t) :
  tree_split measure i s := split_tree' i t tt.
```

End *Trees*.

This closes our implementation of (dependent) Finger Trees with PROGRAM. The complete file we walked through comprises 600 lines of specifications (declarations, types, programs, proof statements) and 450 lines of proof. In comparison the source file provided by Hinze & Paterson is 650 LoC. We argue that this is proof of the scalability of RUSSELL as a programming language.

5. Instances

Obviously, we can instantiate the Finger Trees just like in the original paper of Hinze & Paterson using weak, simply typed specifications and extract the code, but that would not be too exciting. Nevertheless, we will first present a way to recover the simply-typed interface and instantiate it to ordered sequences. We will then focus on proving a particular specialization using the dependent interface directly.

5.1 FingerTree: a non-dependent interface

As shown before, we can wrap the measure and the tree in a dependent pair to offer a simpler interface to Finger Trees.

Section *FingerTree*.

```
Variables (v : Type) (mono : monoid v).
Variables (A : Type) (measure : A → v).
```

```
Definition FingerTree :=
  { m : v & fingertree mono measure m }.
```

This gives us a type *FingerTree v mono A measure*, which can be compared with the original *FingerTree v a* datatype in HASKELL. It adds the monoid implementation *measure* as explicit parameters whereas in HASKELL they are passed implicitly using the typeclass mechanism. Morally, the parameterization we did using the section mechanism correspond to the systematic prefixing of functions on *FingerTree* objects by a *Measured v a* constraint in HASKELL (*Measured v a* is a class with a single function *measure* of type $a \rightarrow v$ where *v* must have a Monoid instance). We think this equivalence indicates an interesting field of investigations on how to integrate overloading in a language like RUSSELL, or a section system in HASKELL.

We do not describe the wrapping in detail, it is just destructuring a *FingerTree* object, calling the appropriate function on *fingertrees* and packing back the measure and the tree. However, an important point is that we can wrap views too and obviously recover the associated measure:

```
Inductive left_view : Type :=
  | nil_left : left_view | cons_left : A → FingerTree → left_view.
```

```
Lemma view_left_size : ∀ t x t', view_left t = cons_left x t' →
  tree_size t' < tree_size t.
```

We can finally wrap splitting on *FingerTrees* along a predicate. We offer the same simply-typed interface as HASKELL by defining the function *split_with p x*. It splits any tree *x* by first checking if *x* is empty and if not calls the *split_tree* wrapper on it using the predicate *p*. Its first and second projections give the *takeUntil* and *dropUntil* functions which return respectively the prefix of the tree for which the predicate was false and the rest of the tree.

```
Program Definition split_with (p : v → bool)
  (x : FingerTree) : FingerTree × FingerTree :=
  if is_empty_dec x then (empty, empty)
  else let (l, x, r) := split p ε x in (l, cons x r).
```

End *FingerTree*.

As an example use of the simple interface we develop the specialization of Finger Trees to Ordered Sequences. The main idea is to use as a measure on elements of *A* the element itself and have the monoid implement an operation which returns the rightmost element in the tree. Then if the operations maintain the invariant that the sequence of elements in the tree are ordered we have an ordered sequence where the maximal element can be retrieved in constant time.

```
Module OrdSequence(AO : OrderedType).
```

We use modules instead of the section mechanism as it gives compile-time instantiation which is exactly what we need here. The module system of COQ is a superset of OCAML’s. We suppose given an implementation of an ordered type.

```
Definition A : Type := AO.t.
Definition measure (x : A) := Some x.
```

We measure one element by itself.

```
Module KM := KeyMonoid AO.
Import KM.
```

The *KeyMonoid* module implements the monoid (*None*, *keyop*) on the type option *A*, where *keyop* is defined by:

```
keyop x None      = x
keyop _ (Some _ as y) = y
```

Clearly this will give the rightmost element of the structure as its measure if there is one. It also defines the function *key_gt* which lift the operation of the ordered type to the monoid’s type. We can then declare the ordered sequence type as *FingerTree* applied to the monoid and measure.

```
Definition OrdSeq := FingerTree m measure.
```

We can get the maximal element in constant time by taking the measure of the whole *FingerTree*.

```
Definition max (x : OrdSeq) := tree_measure x.
```

We skip the other operations defined in the original paper which permit to partition an ordered sequence or insert or delete in it, they are no more difficult to code. We focus on the merge operation which is the only termination challenge in the whole development. Merging of ordered sequences simply does a merge sort of the list of elements of both sequences. It does so using the left view defined earlier and hence needs to recurse on results of the *view_left* function. We can use *pair_size* measure defined below to define the function by well-founded recursion on the pair of sequences. The generated obligations can be solved using the *view_left_size* lemma defined earlier.

Definition *pair_size* ($x : \text{OrdSeq} \times \text{OrdSeq}$) :=
 let (l, r) := x in $\text{tree_size } l + \text{tree_size } r$.

Program Fixpoint *merge* ($x : \text{OrdSeq} \times \text{OrdSeq}$)
 {measure *pair_size*} : OrdSeq :=
 dest x as (xs, ys) in
 match *view_left* xs with
 | *nil_left* $\Rightarrow ys$
 | *cons_left* $x xs' \Rightarrow$
 dest *split_with* ($\text{fun } y \Rightarrow \text{key_gt } y (\text{measure } x)$) ys
 as (l, r) in *cat* $l (\text{cons } x (\text{merge } (r, xs')))$
 end.

5.2 Dependent Sequences.

We will now define random-access sequences as a specialization of *fingertree*. This structure provides insertion and deletion at both ends in constant amortized time, concatenation and splitting in logarithmic time and access (*get*) and mutation (*set*) operations in logarithmic time too. We will build a certified implementation of this structure by showing that the operations respect a functional specification. This method slightly differs from the usual separation of operation definition and invariant proving. Here, invariant preservation will be proved simultaneously with the definition of operations. We define sequences over a type of elements A .

Section *DependentSequence*.

Variable $A : \text{Type}$.

First, a useful definition: the type *below* i represents the naturals below some i , so *below* 0 is not inhabited.

Definition *below* $i := \{ x : \text{nat} \mid x < i \}$.

Lemma *not_below_0* : *below* 0 \Rightarrow False.

We use naturals to measure sequences by their length. Our measure is a bit special as it also carries a function giving the map realized by the Finger Tree. It is used solely for specification and could be removed in extracted code using a dead code analysis or even rejected at extraction using a finer type system.

Definition $v := \{ i : \text{nat} \ \& \ (\text{below } i \rightarrow A) \}$.

We define a notation for our measure objects: an object $i \succ m$ is a dependent pair composed by an index i and a map m from naturals below i to elements of A . The epsilon will represent empty sequences. As it contains no elements, no value is returned by the function. We have an obligation to show that in a context with an hypothesis of type *below* 0, we can prove False: we just use *not_below_0*.

Program Definition $\varepsilon : v := 0 \succ (\text{fun } _ \Rightarrow !)$.

Appending two maps requires some reindexing: the new map is formed by projecting the new index to either the first or the second map.

Program Definition *append* ($xs \ ys : v$) : v :=
 let (n, fx) := xs in let (m, fy) := ys in
 ($n + m$) $\succ (\text{fun } i \Rightarrow \text{if } \text{less_than } i \ n \ \text{then } fx \ i \ \text{else } fy \ (i - n))$.

We can build a monoid *seqMonoid* from these operations, we skip the proofs which are relatively easy.

The measure of a singleton sequence built from x gives the constant map to x .

Program Definition *single* ($x : A$) : $v := 1 \succ (\text{fun } _ \Rightarrow x)$.

We define an abbreviation *seq* for our sequence type. Sequences are a specialization of *fingertree* with the *seqMonoid* monoid and

single measure defined above. Hence an object of type *seq* ($i \succ m$) is a Finger Tree representing a sequence of objects of length i given by the map m .

Definition *seq* ($x : v$) := *fingertree seqMonoid single* x .

We can get the length of the sequence in constant time as it is part of the cached measure.

Program Definition *length* ($s : v$) ($x : \text{seq } s$) : nat :=
 dest s as $\text{size } \succ _$ in size .

The constructor *make* $n \ x$ creates a sequence of length n with value x in each cell. Note that obligations are generated to prove that e.g., when *make* $n \ x$ is of type *seq* ($n \succ \text{fun } _ \Rightarrow x$) then *add_left* x (*make* $n \ x$) is of type *seq* ($S \ n \succ \text{fun } _ \Rightarrow x$). We prove the preservation of semantics of our sequence's *make* operation directly here.

Program Fixpoint *make* ($i : \text{nat}$) ($x : A$) {struct i } :
seq ($i \succ (\text{fun } _ \Rightarrow x)$) :=
 match i with
 | 0 $\Rightarrow \text{empty}$
 | $S \ n \Rightarrow \text{add_left } x (\text{make } n \ x)$
 end.

We now define the strongly specified *get* function which gets the j th element of a sequence x of length i . We ensure that no out-of-bound access is made because j is of type *below* i and we assert that *get* indeed returns the right value of the map m . Note that m is never used in the code, only in the types. Again, some obligations are generated to prove that this specification is indeed a correct abstraction of the semantics of the code. It is crucial that the *split_tree* function be dependent and return a *tree_split* object carrying proofs to solve these obligations. As before, the code stays simple and compact. However we skip around 50 lines of proof needed to discharge the obligations. The boolean predicate *lt_idx* $i \ x$ tests whether an index i is less than the first component of a measure x .

Program Definition *get* $i \ m$ ($x : \text{seq } (i \succ m)$)
 ($j : \text{below } i$) : {value : A | value = $m \ j$ } :=
 dest *split_tree* (*lt_idx* j) $\varepsilon \ x$
 as *mkTreeSplit* $ls \ lx \ rs \ r$ in x .

The natural companion of *get* is *set* which given a sequence x of length i sets its j th cell to *value*. We have as precondition that j is below i and as postcondition that the new sequence has *value* at its j th index and the same elements as x otherwise. The function denoted by $=n$ decides equality on naturals.

Program Definition *set* $i \ m$ ($v : \text{below } i \rightarrow A$) ($x : \text{seq } (i \succ m)$)
 ($j : \text{below } i$) ($\text{value} : A$) :
seq ($i \succ (\text{fun } \text{idx} \Rightarrow \text{if } \text{idx} =n \ j \ \text{then } \text{value} \ \text{else } m \ \text{idx})$) :=
 dest *split_tree* (*lt_idx* j) $\varepsilon \ x$ as *mkTreeSplit* $ls \ l \ rs \ r$
 in *add_right* $l \ \text{value} \ ++ \ r$.

Now that we have defined our operations with built-in invariants, it is a simple matter of using the type information to relate operations. Here we formalize how *get* and *set* behave together. The proofs use solely the types of *get* and *set*, not their code. We show that our sequences respect the two axioms of functional arrays: if getting at an index j just set to *value* we get *value* (*get_set*), otherwise we get the value from the original map (*get_set_diff*).

Program Lemma *get_set* : $\forall i \ m$ ($x : \text{seq } (i \succ m)$)
 ($j : \text{below } i$) ($\text{value} : A$), $\text{value} = \text{get } (\text{set } x \ j \ \text{value}) \ j$.

We were able to state this lemma only because we are using the RUSSELL type-checker, indeed there is an automatically inserted coercion at the right hand side of the equality to go from a subset

of A to an object of A . This seamless integration of PROGRAM with COQ as a theorem prover is also an improvement over previous solutions to provide a programming environment in COQ, like (Parent 1995). However, we are not yet able to get the most natural statements in RUSSELL. In the next definition for example, we coerce objects j and k of type *below* i to their underlying nat components and the results of *get* to A . This is because the comparison of objects from subset types only makes sense on their first component, the witness, whereas comparing proofs is useless. Our pragmatic solution is to let the user insert a cast to get the expected behavior, but in a proof-irrelevant Type Theory the problem disappears because comparison of proofs become trivial. We discuss this solution in §7.1.

```
Program Lemma get_set_diff : ∀ i m (x : seq (i > m))
  (j : below i) (value : A) (k : below i),
  (j : nat) ≠ k → (get x k : A) = get (set x j value) k.
```

Finally, we define splitting of sequences. It requires creating two projections of the original map, hence the two following auxiliary functions.

```
Program Definition split_l (i : nat) (j : below i)
  (idx : below j) : below i := idx.
Program Definition split_r (i : nat) (j : below i)
  (idx : below (i - j)) : below i := j + idx.
```

We can finally split a sequence x at a particular index j , returning two sequences whose maps are just projections of the splitted x . Approximately a hundred lines of proof are necessary to discharge the three obligations generated by PROGRAM.

```
Program Definition split i m (x : seq (i > m)) (j : below i) :
  seq (j > (m ◦ split_l j)) × seq ((i - j) > (m ◦ split_r j)) :=
  dest split_tree (lt_idx j) ε x as mkTreeSplit ls l x rs r in
  (l, add_left x r).
```

End *DependentSequence*.

This gives us a certified implementation of sequences in less than 500 lines whose dependent interface can be used directly to implement programs and prove properties at the same time. One thing we learned is that once we begin to use dependent types, embracing them completely become a sensible and desirable goal. PROGRAM permits doing this in a tractable and hopefully maintainable way.

6. Extraction

We can extract the code we just certified to both HASKELL and OCAML (bypassing the type checker for polymorphic recursion) to get certified implementations of Finger Trees and sequences. We are guaranteed to get the same algorithmic code than what we wrote using PROGRAM. This is an important aspect of developing with RUSSELL, because usually when developing programs with rich specifications in COQ one tends to use the proof language as much as the programming language and this can have devastating results on the extracted code. Our method enforces a distinction between code and proof that is healthy in this respect.

Unsurprisingly, our extracted Finger Trees have the same performance as the original implementation: they have the same code. The `Data.Sequence` implementation in HASKELL is in fact a specialization of the `FingerTree` implementation to a particular monoid and measure, permitting to avoid dynamic lookups and unnecessary boxing. Had we used the module system of COQ to parameterize our development instead of the section mechanism, we would have had this instantiation at no cost. However the lack of a powerful module system in HASKELL prevented us from doing that, as we would not be able to extract to it then.

Unfortunately, extraction of our dependent sequences won't give excellent results because the measure contains a function giving the map which will be extracted even though it has no algorithmical role. We are in a case where only part of an index need be stored, so a finer distinction than `Prop/Type` has to be found to distinguish between algorithmical and non-algorithmical content. Current work by B. Barras and B. Bernardo on an adaptation of the Calculus of Implicit Constructions (Miquel 2001) as the core calculus of a dependently-typed language ought to give the expressivity we seek.

We conclude that RUSSELL is not overly verbose and that doing the proofs is actually not an insurmountable task, knowing that most of them were solved automatically using a normalization tactic for monoid expressions.

7. Discussion

7.1 Proof carrying code

As displayed before, COQ ultimately melts proof and code in its terms, so as to be able to check the correctness of any part of a term at any time. On the surface, we can manage to separate the two activities of coding and proving using PROGRAM, but there are parts of the system where it still bites. The same notable problem appear during proofs and computations. When proving, we are faced with terms containing proofs we do not want to name or manipulate and when computing (in the call-by-value case only) the system itself is stuck with irreducible opaque proofs. However, in the two cases the proofs appear only in parts of the term that should not be reduced because they have no algorithmical content and need not be manipulated as their structure has no importance. This is formalized by proof-irrelevance (PI), the statement that two proofs of the same proposition are equal. The meta theory of the Calculus of Constructions extended with PI has been studied extensively by Werner and Miquel. Recently in (Werner 2006) the former also gave a method to implement it effectively in COQ, which is currently pursued by the author. It has also been studied in the context of Observational Type Theory, the (would be) core calculus of EPIGRAM 2. Having PI in the system would permit to get rid of these two problems and would probably improve performance of computations inside COQ too.

7.2 A note on the implementation

PROGRAM is composed of two distinct parts: the type-checker and the proof-obligation handling machinery. In fact only a small part of the COQ type-checker needed to be changed, so RUSSELL's type-checker is just a different instantiation of the typing functor of COQ which is parameterized by the conversion algorithm. This is in contrast with the work by Catherine Parent on the previous PROGRAM tactic which was a completely different solution for program-synthesis in COQ. The system was not integrated with COQ and so went unmaintained.

7.3 Related Work

RUSSELL can be seen as one point in the design space of programming languages with dependent types. Its most distinctive feature is the separation of code from proof, à la PVS, and its treatment of subsets and indexed data types. It is also original in the sense that it *elaborates* into a well-studied, safe and mature language while most other solutions coming from the programming languages community have no such safety net or only for part of the system. RUSSELL could in fact be implemented in EPIGRAM or AGDA rather easily as they are based on roughly the same foundations as COQ. The main difference is that COQ already has a comprehensive standard library and a tactic system. Compared to programming languages like ATS, CONCOQTION or HASKELL, RUS-

SELL has no effects and only well-founded recursion. We found that in practice the programs we write usually have simple termination arguments and non-termination can be accommodated by using a coinductive encoding. We think having a layered approach to treating computational effects using functional encodings like monads is a more disciplined approach to development and certification than trying to design a system with such and such effects built-in, if this is at all possible while keeping strong properties.

Other certifications of tree-based data structures include a certification of the AVL trees used in OCAML and also red-black trees in COQ (Filliâtre and Letouzey 2004). The development was done by first building a dependent interface and providing a simpler one on top of it. The second author reported great success adapting the implementation which was sometimes done using tactics to PROGRAM.

8. Conclusion

What we would like to assess is that writing programs in RUSSELL is not more difficult than in HASKELL, with the added benefit of a complete environment for proving properties about your programs in a highly expressive specification language. Clearly, there is room for improvement on the side of programming language technology, for example some kind of overloading mechanism would allow more conciseness, but no programming language matches RUSSELL in terms of expressiveness and actual support for proving properties.

Index

All functions are parameterized by a type A .

| Function name | Reference or Specification |
|--------------------------------|---|
| <i>full</i> | §4.1 |
| <i>single</i> | $\text{digit } A \rightarrow \text{Prop}$ |
| <i>add_digit_left</i> | §4.1 |
| <i>digit_head</i> | $\text{digit } A \rightarrow A$ |
| <i>digit_tail</i> | $(d : \text{digit } A \mid \neg \text{single } d) \rightarrow \text{digit } A$ |
| <i>digit_measure</i> | $(A \rightarrow v) \rightarrow \text{digit } A \rightarrow v$ |
| <i>option_digit_measure</i> | $(A \rightarrow v) \rightarrow \text{option } (\text{digit } A) \rightarrow v$ |
| <i>add_left</i> | §4.3 |
| <i>add_right</i> | $(m : A \rightarrow v) (s : v) (t : \text{fingertree } m \ s)$ $(x : A), \text{fingertree } m (s \cdot m \ x)$ |
| <i>digit_to_tree</i> | $(m : A \rightarrow v) (d : \text{digit } A) \rightarrow$ $\text{fingertree } m (\text{digit_measure } m \ d)$ |
| <i>option_digit_to_tree</i> | $(m : A \rightarrow v) (d : \text{option } (\text{digit } A)) \rightarrow$ $\text{fingertree } m (\text{digit_measure } m \ d)$ |
| <i>tree_size</i> | $(m : A \rightarrow v) (s : v),$ $\text{fingertree } m \ s \rightarrow \text{nat}$ |
| <i>View_L_size</i> | $(m : A \rightarrow v),$ $\text{View_L } A (\text{fingertree } m) \rightarrow \text{nat}$ |
| <i>view_L</i> | §4.4 |
| <i>deep_L</i> | §4.4.1 |
| <i>isEmpty, isEmpty_dec</i> | §4.4.2 |
| <i>head, last, tail, liat</i> | §4.4.2 |
| <i>split_digit, split_node</i> | §4.5 |
| <i>app, split_tree</i> | §4.5 |
| <i>cat</i> | $\text{FingerTree } A \times \text{FingerTree } A \rightarrow$ $\text{FingerTree } A$ |
| <i>cons</i> | $A \rightarrow \text{FingerTree } A \rightarrow \text{FingerTree } A$ |
| <i>split_with</i> | §5.1 |
| <i>empty</i> | $\text{seq } A \ \text{epsilon}$ |
| <i>make, get, set, split</i> | §5.2 |

References

- The Coq proof assistant. URL <http://coq.inria.fr>.
- Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *Proceedings of The European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 370–384, Barcelona, Spain, April 2004. URL <http://www.lri.fr/~filliatr/ftp/publis/fpp.ps.gz>.
- Ralf Hinze and Ross Paterson. Finger Trees: A Simple General-purpose Data Structure. *J. Funct. Program.*, 16(2):197–217, 2006. URL <http://www.soi.city.ac.uk/~ross/papers/FingerTree.html>.
- Pierre Letouzey. A new extraction for coq. In Herman Geuvers and Freek Wiedijk, editors, *TYPES*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002. ISBN 3-540-14031-X.
- Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.
- Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In S. Abramsky, editor, *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Applications*, volume 2044, pages 344–359. Springer-Verlag, Berlin, 2001.
- Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.
- Catherine Parent. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In Bernhard Möller, editor, *MPC*, volume 947 of *Lecture Notes in Computer Science*, pages 351–379. Springer, 1995. ISBN 3-540-60117-1.
- Christine Paulin-Mohring. Inductive definitions in the system COQ. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.
- Tim Sheard. Languages of the future. URL <http://www.cs.pdx.edu/~sheard/papers/LangOfTheFuture.ps>.
- Matthieu Sozeau. Subset coercions in Coq. 2006. Accepted for TYPES’06 post-proceedings.
- Benjamin Werner. On the strength of proof-irrelevant type theories. *3rd International Joint Conference on Automated Reasoning*, 2006.
- Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.
- Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, San Antonio, Texas, pages 214–227, January 1999.