A Functional Framework for Result Checking*

Gilles Barthe[1], Pablo Buiras[1,2], and César Kunz[1]

[1] IMDEA Software, Spain
[2] FCEIA, Universidad Nacional de Rosario, Argentina

Abstract. Result checking is a general methodology for ensuring that untrusted computations are valid. Its essence lies in defining efficient checking procedures to verify that a result satisfies some expected property. Result checking often relies on certificates to make the verification process efficient, and thus involves two strongly connected tasks: the generation of certificates and the implementation of a checking procedure. Several ad-hoc solutions exist, but they differ significantly on the kind of properties involved and thus on the validation procedure. The lack of common methodologies has been an obstacle to the applicability of result checking to a more comprehensive set of algorithms. We propose the first framework for building result checking infrastructures for a large class of properties, and illustrate its generality through several examples. The framework has been implemented in Haskell.

1 Introduction

Computer programs are error-prone, making it a challenge to assure the validity of computations. Errors arise from many sources: programming mistakes, rounding-off errors in floating-point computations, defects in the underlying hardware, or simply because part of a computation has been delegated to some untrusted party. A general methodology for ascertaining the correctness of the computations performed by a program F is to rely on an independent result checker V, which guarantees the correctness of the computation performed by F. A simple example of result checker is a boolean-valued predicate between inputs and outputs that only returns true on pairs (x, y) such that $F(x) = y$, where F is a program with a single input x and single output y. For example, a result checker for the program F computing the square root y of x is the program V that returns the boolean expression $(y^2 \leqslant x)\&\&(x < y^2 + 2y + 1)$. However, result checkers may in general rely on additional inputs, called certificates, that guarantee efficient execution. A typical example of result checker which relies on certificates is the checker for greatest common divisor (gcd), which takes as arguments, in addition to a and b for which the gcd must be computed, two additional values u and v (which constitute the certificate), and the candidate gcd d, and returns the boolean value $d = ua + vb \wedge d \mid a \wedge d \mid b$.

* Partially funded by the EU project HATS and Spanish project Desafios-10 and Community of Madrid project Comprometidos.

M. Blume, N. Kobayashi, and G. Vidal (Eds.): FLOPS 2010, LNCS 6009, pp. 72–86, 2010.

While result checking offers a general methodology to guarantee the correctness of computations, and thus is potentially applicable to many domains, its applications have been circumscribed to a few and rather specific settings; see Section 2. The main challenge in broadening the scope of result checking is finding a systematic means of building, for a large class of properties, a result checking framework that provides (a) for every property P in the class, a type wit_P of certificates; (b) a means of generating certificates[1]; and (c) a checker $\mathrm{check}_P : A \to \mathrm{wit}_P \to \mathbf{bool}$ (where A is the carrier of P), such that for all $a : A$ and $w : \mathrm{wit}_P$ we have

$$(\mathrm{check}_P\ a\ w = \mathtt{true}) \Rightarrow P\ a.$$

The purpose of this article is to provide a framework for building and verifying certified results for a large class of algorithms. The framework is implemented on top of the Haskell [9,10] programming language, and provides:

- *certifying combinators*, which extend the usual combinators of functional programming with facilities for turning certificates of the combinators' inputs into certificates of the combinators' outputs. Certifying combinators can be combined to produce certified results;
- a *generic* checker function, that takes a *representation* of a property and behaves like a checker for it.

The combination of certifying combinators and the generic checker allows us to obtain a result checking framework for a large class of properties, including sorting and searching algorithms, or primality testing.

Although our results are developed in the setting of a sequential language, our primary motivation is to provide a certificate-based infrastructure for guaranteeing the correctness of large distributed computations among untrusted hosts [1]. The results of this paper can be embedded in the framework of [16] for this purpose.

Outline. In Section 3, we define a generic checking function for a large class of predicates that includes inductively defined ones. In Section 4, we propose two methods for the generation of certificates. Both are defined as an extension of the original producer algorithm. The first one is based on the recursion pattern defining the producer algorithm. The second one is a general approach for the certification of nondeterministic computations.

2 Related Work

Blum and Kannan [2] were among the first to recognise the importance of result checking as a general method for discovering bugs in programs, and to advocate its superiority over testing, which can be unreliable, or program verification,

[1] Note that no property of the certificate generation mechanism is needed for checked results to be correct, i.e., it is not necessary to trust the certificate generators.

which can be costly. Many of the checkers considered in [2] are probabilistic programs, whose soundness is expressed in probabilistic terms. Blum subsequently reflects on the usefulness of result checking for managing hardware errors, as in the Pentium bug, or round-off errors arising in floating-point computations.

The pioneering ideas of Blum and Kannan were further developed in the context of certification trails for common data structures, and certifying algorithms for mathematical software:

- a *certifying algorithm* is one which computes, along with the intended result, a certificate that allows a checker program to verify independently the correctness of the result (independent verification means that it should not be necessary for the checker program to trust the certifying algorithm in any way: results and certificates should speak for themselves). Certifying algorithms are implemented notably in the LEDA and CGAL platforms for computational geometry; the role of the checkers is to increase the reliability of the platform, through a checking phase introduced at the end of each geometric algorithm.
- a *certification trail* [3,14] is a record of selected intermediate results during a program computation. A second algorithm executes more efficiently by using the execution trail of the first program to compute the same result. Then, the original result and the result of the second algorithm are compared. The technique is applicable to algorithms that manipulate data structures, e.g. priority queues.

Result checking is also commonly considered—although not always explicitly so—in formal verification. For example, result checking is a natural approach to connecting proof assistants with external tools, e.g. computer algebra systems or mathematical packages whose results are untrusted. One prominent application of result checking in proof assistants is Grégoire, Théry and Werner's work on primality checking [7], using Pocklington's criterion, and optimizing the checker in order to check large prime numbers. Another example is the work of Harrison [8], based on a sums of squares representation to certify positive semidefinite polynomials. The experiments have been developed by combining the HOL Light machinery with a semidefinite programming package.

Result checking is also used as a proof technique for simplifying the task of program verification; the basic idea is to cut-off program verification tasks by isolating subroutines for which appropriate checkers exist, and then verifying these checkers instead of the aforementioned subroutines. This process is used e.g. in the CompCert project [12], where a formal proof of compiler correctness uses result certification for graph colorings.

Applications of result checking to guarantee trust in distributed environments are relatively recent. One of the most prominent ones is Proof Carrying Code (PCC) [13], which offers trust in a mobile application (the value to be checked) through a formal proof (the certificate) that the application respects a given security or safety policy. However, PCC cannot be used to establish the integrity of distributed computations among untrusted hosts, because one cannot make assumptions on the code executed by remote hosts.

3 Construction of Result Checkers

This section presents a general framework for result checking. Given a predicate P, and assuming that the certificate types and checkers for the atomic predicates that compose P are known in advance, we infer the type of *certificates* of P, and we can check P using a *generic checker*. The generic checker, which we present in Section 3.2, is generator-agnostic, and works regardless of the way in which certificates are generated.

The main difficulty in this section is handling inductively presented predicates, i.e. predicates that are defined by inference rules of the form:[2]

$$\frac{Q\ a}{P\ a}\text{[Base case]} \qquad \frac{P\ r_1 \ldots P\ r_n \qquad R\ \vec{r}\ x}{P\ x}\text{[Inductive case]}$$

where Q is a predicate describing a base case and R is a predicate for the induction step, or equivalently by the equation:

$$P\ x = Q\ x \vee (\exists \vec{r}.\ (\forall i \in \mathsf{bounds}(\vec{r}).\ P\ r_i) \wedge R\ \vec{r}\ x)$$

where $\mathsf{bounds}(v) = \{i \mid 1 \leqslant i \leqslant |v|\}$. Pocklington's criterion is an instance of an inductively presented predicate that has been used for checking primality efficiently.

Example 1. A number N is prime if it satisfies $\mathsf{Prime}\ N$ where

$$\begin{aligned}\mathsf{Prime}\ N &\doteq (N = 2) \vee (\exists \vec{p}.\ (\forall i \in \mathsf{bounds}(\vec{p}).\ \mathsf{Prime}\ p_i) \wedge \mathsf{Pock}\ \vec{p}\ N)\\ \mathsf{Pock}\ \vec{p}\ N &\doteq \exists \vec{\alpha} a.\ |\vec{\alpha}| = |\vec{p}| \wedge p_1^{\alpha_1} \cdots p_k^{\alpha_k} \mid (N-1) \wedge \sqrt{N} < p_1^{\alpha_1} \cdots p_k^{\alpha_k} \wedge\\ &\quad a^{N-1} \bmod N = 1 \wedge \forall i \in \mathsf{bounds}(\vec{p}).\ \mathsf{coprime}\ (a^{\frac{N-1}{p_i}} - 1)\ N\\ &\quad \text{where } k = |\vec{p}|\end{aligned}$$

Inductively presented predicates arise in many contexts, and it is thus desirable for a result checking framework to support them.

3.1 Properties and Certificates

The starting point of the framework is the definition of formulae and predicates in Figure 1. Instead of relying on the usual formalisation of these as vanilla data types, we take advantage of Haskell's support for Generalised Algebraic Data Types (GADTs) [11] and index the type of formulae and predicates with a type of certificates: thus, *Form w* is the type of formulae whose correctness is certificated by terms of type w, and *Pred w a* is the type of predicates over a with certificate type w.

The definition of formulae includes constructors for the true and false values (TT and FF), logical connectives ($\wedge$ and $\vee$), existential quantification (E),

[2] We stress that inductively presented predicates do not require the r_is to be smaller than x with respect to some well-founded order; in particular every predicate P is equivalent to an inductively presented one by taking Q to be false and $n = 1$ and $R\ r\ x$ to be $x = r$.

```
data Form w where
  TT    :: Form ()
  FF    :: Form ()
  (∧)   :: Form w1 → Form w2 → Form (w1, w2)
  (∨)   :: Form w1 → Form w2 → Form (Either w1 w2)
  E     :: Pred w a → Form (a, w)
  (@)   :: Pred w a → a → Form w
  Forall :: Pred w a → [a] → Form [w]
data Pred w a where
  Atom    :: (w → a → Bool) → Pred w a
  Abs     :: (a → Form w) → Pred w a
  RecPred :: Pred w1 a → Pred w2 ([a], a) → Pred (RecWit w1 a w2) a
```

Fig. 1. Definition of formulae (*Form*) and predicates (*Pred*)

universal quantification over lists (*Forall*), and @ for predicate application to a value. For TT and FF, the certificate type is the unit type. For conjunction and disjunction, the certificate types are respectively the product and sum of the certificate types of the two conjuncts. For existential quantification over a predicate P over the type a, a certificate is a pair consisting of an element of a and a certificate for P. Finally, universal quantification requires a list of certificates, and @ expects a certificate for the predicate. We omit treating negation in our framework, since it would be impossible to derive checkers for the negation of an existential.

The definition of predicates includes the constructor *Abs*, which uses higher-order abstract syntax to turn a formulae into a predicate. Atomic predicates, for which we assume that a checker is given, are modeled by encapsulating the corresponding checker with the constructor *Atom*.

The constructor *RecPred* models inductively presented predicates. It takes the predicate Q as first parameter, and R as the second parameter. The data type constructor *RecWit* is introduced to define certificate types for recursive predicates:

```
data RecWit w1 a w2 = Base w1
                    | Rec [a] [RecWit w1 a w2] w2
```

The parameters for this type are:

w_1	the type of certificates for the base case, Q
a	the input type for the predicate
w_2	the type of certificates for predicate R

A certificate of the form *Base w* corresponds to the certificate of the validity of the base case Q, as certificated by w. A certificate of the form *Rec as ws w* proves the existential part, where *as* is the list of values for which the predicate

recursively holds, ws is the list of certificates for the values in as, and w is the certificate for R.

Example 2. Consider the certification of the primality testing algorithm introduced in Example 1. The predicate Prime derived previously from Pocklington's criterion matches the form of inductively presented predicates shown above. Therefore, we can encode the predicate Prime as an expression of type *Pred*, by using the data type constructor *RecPred*: predicate Q corresponds simply to the condition $n \equiv 2$; predicate R corresponds to Pock. We also include the definitions of some auxiliary predicates that we need to express the criterion.

$$
\begin{array}{ll}
\mathit{divides} & = \mathit{Atom}\ (\lambda()\ (n, m) \to m\ `\mathit{mod}`\ n \equiv 0) \\
\mathit{mult\ ps\ as} & = \mathit{product}\ (\mathit{zipWith}\ (\hat{})\ \mathit{ps}\ \mathit{as}) \\
\mathit{coprimes} & = \mathit{Abs}\ (\lambda(a, n, ps) \to \\
 & \quad \mathit{Forall}\ (\mathit{Abs}\ (\lambda p \to \mathit{coprime}@(a\hat{}((n-1)\ `\mathit{div}`\ p) - 1, n)))\ \mathit{ps}) \\
\mathit{coprime} & = \mathit{Atom}\ (\lambda()\ (n, m) \to \mathit{gcd}\ n\ m \equiv 1) \\
\mathit{pock\ ps\ n} & = E\ (\mathit{Abs}\ \$\ \lambda(a, \alpha s) \to \\
 & \quad \mathit{Atom}\ (\lambda()\ (xs, ys) \to \mathit{length}\ xs \equiv \mathit{length}\ ys)@(ps, \alpha s) \wedge \\
 & \quad \mathit{divides}@(\mathit{mult\ ps}\ \alpha s, n-1) \wedge \\
 & \quad \mathit{Atom}\ (\lambda()\ (x, y) \to x\hat{}(y-1)\ `\mathit{mod}`\ y \equiv 1)@(a, n) \wedge \\
 & \quad \mathit{ordP}@(\sqrt{n}, \mathit{mult\ ps}\ \alpha s) \wedge \\
 & \quad \mathit{coprimes}@(a, n, ps)) \\
\mathit{prime} & = \mathit{RecPred}\ (\mathit{Atom}\ (\lambda()\ n \to n \equiv 2)) \\
 & \quad (\mathit{Abs}\ \$\ \lambda(ns, n) \to \mathit{pock}\ ns\ n)
\end{array}
$$

The type of *prime* is

$$\mathit{prime} :: \mathit{Pred}\ (\mathit{RecWit}\ ()\ \mathit{Int}\ ((\mathit{Int}, [\mathit{Int}]), T))\ \mathit{Int}$$

where $T = ((((), ()), ()), [()])$ is the (trivial) certificate type for Pock.

3.2 Generic Checker

This section describes a generic checker for formulae and predicates. The checker is defined by mutual recursion, and consists of functions *check* and *checkPred*, defined in Figure 2. Most equations are straightforward. Atomic predicates are represented by their own checkers, so in *checkPred* we just call the checker function with the appropriate parameters. Predicate abstractions are also easy to deal with: we just compute the formula with the encapsulated function, and then delegate the work of checking this formula to the function *check*, taking care to deliver it the right certificate.

Finally, we must deal with the recursive predicate case, where terms are of the form *RecPred q r*. Recursive certificates either certify base cases (predicate q here), or recursive cases (the existential part of the disjunction). If we have a certificate for the base case, we just check predicate q with the supplied certificate. The recursive case is the trickiest one. On the one hand, we must recursively

```
check                        :: Form w → w → Bool
check TT _                   = True
check FF _                   = False
check (x ∧ y) (w1, w2)       = check x w1 ∧ check y w2
check (x ∨ y) (Left w1)      = check x w1
check (x ∨ y) (Right w2)     = check y w2
check (E p) (x, w)           = checkPred p x w
check (Forall p xs) ws       = all id (zipWith (checkPred p) xs ws)
check (p@x) w                = checkPred p x w

checkPred :: Pred w t → t → w → Bool
checkPred (Atom f) x w                    = f w x
checkPred (Abs f) x w                     = check (f x) w
checkPred (RecPred q r) x (Base w)        = checkPred q x w
checkPred (RecPred q r) x (Rec as rs w2) =
  check (Forall (RecPred q r) as) rs ∧
  checkPred r (as, x) w2
```

Fig. 2. The definition of *check* and *checkPred*

check the predicate for the list of existentially quantified values. On the other hand, we must check that predicate r holds, which we do by calling *checkPred* with the appropriate parameters.

Example 3. Consider again the example of Pocklington's Criterion to verify the primality of an integer n. An implementation of a checker for this criterion, requires a partial decomposition of $n-1$ into prime factors, so we must recursively invoke the checker to verify the primality of these numbers as well.

An instance of a Pocklington certificate is the term

$$Rec\ [2,3]\ [Base\ (), Rec\ [2]\ [Base\ ()]\ ((2,[1]),t)]\ ((11,[4,2]),t)$$

which proves that the number 1009 is prime, where $t = ((((),()),()),[()])$ is the trivial certificate for the side conditions in Pock.

Note that there is unnecessary redundancy in this certificate. There are two instances of the certificate for 2 (i.e. $Base\ ()$), which have to be checked separately by the checker. This is inefficient not only in terms of space, but also in terms of execution time.

The solution to this problem consists in the introduction of *sharing* in the certificate data structure. The idea is to avoid repeated subcertificates by having only one instance of each, and allowing it to be shared in several parts of the structure. Using sharing, the certificate for 1009 is written as

$$\textbf{let}\ w_2 = Base\ ()\ \textbf{in}\ Rec\ [2,3]\ [w_2, Rec\ [2]\ [w_2]\ ((2,[1]),t)]\ ((11,[4,2]),t)$$

Although requiring a more sophisticated certificate generation mechanism, sharing forces memoisation of checker results, thus allowing the proof of the primality

of 2 to be performed only once for this certificate. We would like to stress that no changes in the checker are needed for this to work; it is just a consequence of lazy graph reduction.

Our implementation of Pocklington's criterion is relatively easy to improve by combining the use of clever algebraic properties of the modulo and exponentiation operators, and having certificates for gcd in the *coprime* predicate. Since these optimisations can be thought of as more sophisticated checkers for atomic predicates, it is straightforward to incorporate them into our framework. Other implementations of Pocklington's criterion [7,5] using these techniques are able to cope with huge primes, with approximately 10000 digits.

3.3 Generic Checker Properties

Assuming a standard interpretation function $[\![\cdot]\!]$ that maps formulae and predicates to values in some semantic domain suitable for first-order logic, we say that a checker *checkPred* ϕ is *sound* if *checkPred* $\phi\ x\ w \equiv True \Rightarrow [\![\phi]\!]\ (x)$. It is possible to state a *soundness* property for generic checkers: all checkers of the form *checkPred* ϕ are sound, provided that ϕ only has atomic predicates with sound checkers.

We define a *certifying* variant of a function of type $A \to B$ as a function of type $A \to (B, W)$, where W is the type of the witness. A *specification* for such a certifying function is a pair (ϕ,ψ) where ϕ is a precondition with trivial witness, i.e. of type $Pred\ ()\ A$; and ψ is a postcondition of type $Pred\ W\ (A, B)$.

The generic checker can be used at runtime to ensure the partial correctness of certifying functions. One can define a function *wrap* that lifts a certifying computation into the *Maybe* monad:

$$
\begin{array}{l}
wrap :: (Pred\ ()\ a, Pred\ w\ (a, b)) \to (a \to (b, w)) \to a \to Maybe\ b \\
wrap\ (pre, post)\ f\ x \\
\quad \mid checkPred\ pre\ x\ () \equiv False = Nothing \\
\quad \mid otherwise = \mathbf{let}\ (y, w) = f\ x \\
\qquad\qquad\quad \mathbf{in\ if}\ checkPred\ post\ (x, y)\ w \\
\qquad\qquad\qquad \mathbf{then}\ Just\ y \\
\qquad\qquad\qquad \mathbf{else}\ Nothing
\end{array}
$$

The function *wrap* $(\phi, \psi)\ f$ checks that ϕ holds on the input, then computes f, and finally checks that ψ holds on the output, returning *Nothing* if any check fails. It has the useful property of turning a certifying (not necessarily correct) function into a correct one:

Theorem. *Let f be a certifying function with specification (ϕ, ψ). Then, wrap $(\phi, \psi)\ f$ is* partially correct *with respect to the specification (ϕ, ψ'), where $\psi'\ (x, y) = y \equiv Nothing \ \vee\ (y \equiv Just\ z \wedge \psi\ (x, z))$.*

4 Certificate Generation

Certificate generation is a much harder problem than certificate checking. One cannot program a generic certificate generator that automatically builds

certificates of complex properties, even when having certificate generators for atomic predicates. The problem arises from existentials: recall that a certificate for $\exists x : T.\ \phi(x)$ is a pair (t, p) where t is an element of T and p is a certificate of $\phi(t)$. In general, it is unfeasible to find t. The problem also appears in the case of inductively presented predicates. In the general case, one cannot find the witnesses that are used to infer that an element verifies the predicates. It should be pointed however, that if we require that the witnesses $\vec{r}$ are smaller than x w.r.t. a well-founded order, then one can write a generic certificate generator. Of course, such a generic certificate may be hopelessly inefficient.

In this section, we present two different approaches to certificate generation for particular classes of problems. They share the underlying notion of *certifying higher-order recursion operators*, which abstract away common patterns of certificate generation. In particular, we focus on structural recursion and generation patterns, otherwise known as *folds* and *unfolds* in the functional programming literature, because it is possible to express all general recursive functions as compositions of these combinators. In both cases, the point is to generate results and their certificates in tandem, instead of doing it in a separate post-processing step.

4.1 Certificates as Monoids

It is possible to define certifying versions of *fold* and *unfold*, namely *cfold* and *cunfold*, which build certificates incrementally: at each node of the structure, a *local* certificate is generated; then, all local certificates merge to form a certificate for the whole computation. For this merging to be well-defined, we require that certificates form a *monoid*.

Let us consider the certification of the output of sorting algorithms. In a pure formulation of the problem, a sorting algorithm is fed with an input list L, and produces an output list L' such that

i. The list L' is in *nondecreasing* order (according to some total order on the elements of L);
ii. The list L' is a *permutation* of L.

A certifying sorting algorithm would then have to *prove* that these conditions are met by its output. A checker would need no hints to prove the first condition, since it is possible to efficiently check the sortedness of a list. Therefore, certification in this case boils down to checking the second condition, i.e. that L' has the same elements as L. The certificate we require is just a mapping that describes the sorting performed by the algorithm.

Consider the Quicksort algorithm, which can be visualised as building a tree of elements such that, when flattened, yields a sorted permutation of the input list. We can make this intermediate structure explicit, and write the tree-generation step as an *unfold*, and the flattening step as a *fold*.

$$\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a = E \mid N\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)\\
&\mathit{unfoldT} :: (s \to \mathit{Maybe}\ (s, a, s)) \to s \to \mathit{Tree}\ a\\
&\mathit{unfoldT}\ f\ s = \textbf{case}\ f\ s\ \textbf{of}
\end{aligned}$$

```
  Nothing          → E
  Just (s1, x, s2) → N (unfoldT f s1) x (unfoldT f s2)
foldT :: (b → a → b → b) → b → Tree a → b
foldT f z E              = z
foldT f z (N lt x rt) = f (foldT f z lt) x (foldT f z rt)
qsort = foldT (λls x rs → ls ++ x : rs) [] ∘ unfoldT build
  where build []       = Nothing
        build (x : xs) = let (l, r) = partition (<x) xs
                         in Just (l, x, r)
```

By expressing the algorithm in terms of higher-order recursion operators, we can produce a certificate for the result in a compositional way which is directed by the structure of the recursion. For Quicksort, since we are building a tree of elements, we would only be required to show a certificate for every node of the tree, and the higher-order operator could combine them into a certificate for the whole tree.

Certificates for Quicksort must be permutations. We shall represent mappings as lists of pairs, where the mapping (x, y) maps the element in position x to position y in the list.

```
newtype Map = Map [(Int, Int)]
```

We can make our *Map* data type an instance of *Monoid* in this way:

```
instance Monoid Map where
  ϵ = Map []
  (Map xs) ⊕ (Map ys) = Map (norm (xs ++ ys))
```

The identity is defined as the empty mapping (represented by []), and the merging operator is the concatenation ++ (taking care to normalise the list resulting from the append operation with the auxiliary function *norm*). This function *norm* removes pairs coinciding on the first element to make sure that the list still represents a valid function.

We can now write *certifying* versions of fold and unfold for binary trees.

```
cunfoldT :: (Monoid w) ⇒ (s → Maybe (s, (a, w), s)) → s → Tree (a, w)
cunfoldT f s = case f s of
  Nothing          → E
  Just (s1, (x, w), s2) → N (cunfoldT f s1) (x, w) (cunfoldT f s2)
cfoldT :: (Monoid w) ⇒ (b → a → b → (b, w)) → b → Tree (a, w) → (b, w)
cfoldT f z E                = (z, ϵ)
cfoldT f z (N lt (x, w) rt) =
  let (v1, w1) = cfoldT f z lt
      (v2, w2) = cfoldT f z rt
      (v, fw)  = f v1 x v2
  in (v, fw ⊕ w1 ⊕ w ⊕ w2)
```

As can be seen, the modified intermediate tree contains nodes of pairs (τ, w), where certificates of type w are generated by the coalgebra of the function $cunfoldT$. All certificates in the intermediate tree are combined together by the $\oplus$ operator, when the tree is flattened by the function $cfoldT$.

We use these combinators to write a certifying version of Quicksort:

$$
\begin{array}{l}
qs :: (Ord\ a) \Rightarrow [(a, Int)] \rightarrow ([a], Map) \\
qs = cfoldT\ (\lambda xs\ (x, \_)\ ys \rightarrow (xs +\!\!+ x : ys, \epsilon))\ [\,] \circ cunfoldT\ build \\
build\ [\,] \qquad\qquad\qquad\qquad = Nothing \\
build\ (x@(\_, from) : xs) = \\
\quad \textbf{let}\ (l, r) = partition\ (<x)\ xs \\
\quad \textbf{in}\ Just\ (l, (x, Map\ (zip\ (map\ snd\ l)\ [1\,..] +\!\!+ \\
\qquad\qquad\qquad\qquad [(from, length\ l + 1)] +\!\!+ \\
\qquad\qquad\qquad\qquad zip\ (map\ snd\ r)\ [(length\ l + 2)\,..])), r)
\end{array}
$$

We assume that the input for qs has been preprocessed into a form where every element is paired with its inital position in the list. We omit this transformation since it is trivial and can be done in linear time, so it will not dominate the time complexity of the algorithm.

Note that, essentially, all we have to do is to specify the permutation induced by each call to *partition* (in the function *build*). The certifying combinators serve to abstract away the combination of small, locally generated certificates into large, global certificates for the whole computation.

Example 4 (Insertion sort). Another sorting algorithm where certifying fold can be used is Insertion sort. This algorithm is traditionally written as follows:

$$
\begin{array}{l}
isort :: Ord\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha] \\
isort = foldr\ insert\ [\,] \\
\quad \textbf{where}\ insert\ x\ [\,] \qquad\qquad\qquad\quad = [x] \\
\qquad\qquad insert\ x\ (y : ys) \mid x \leqslant y \qquad = x : y : ys \\
\qquad\qquad\qquad\qquad\qquad\quad \mid otherwise = y : insert\ x\ ys
\end{array}
$$

The certifying version is a little bit more involved. The basic idea is the same as with Quicksort, i.e. to produce an appropriate certificate for each step of the computation, and have the certifying combinator merge them together. We shall present the final version of the algorithm, and invite the reader to work out the details.

$$
\begin{array}{l}
isort :: [((Int, Int), [(Int, Int)])] \rightarrow ([Int], [(Int, Int)]) \\
isort = cfoldL\ insert\ [\,] \\
\quad \textbf{where}\ insert\ (x, from)\ ys = \\
\qquad \textbf{let}\ ins\ x\ [\,]\ n \qquad\quad = ([x], n) \\
\qquad\quad ins\ x\ (y : ys)\ n \\
\qquad\qquad \mid x \leqslant y \qquad = (x : y : ys, n) \\
\qquad\qquad \mid otherwise = \\
\qquad\qquad\quad \textbf{let}\ (zs, n') = ins\ x\ ys\ n
\end{array}
$$

$$\begin{aligned} &\textbf{in } (y : zs, n' + 1) \\ (zs, to) \quad &= ins\ x\ ys\ 1 \\ \textbf{in } &(zs, [(to + from - 1, from)]) \end{aligned}$$

4.2 Certificates for Nondeterministic Computations

In this section we consider the application of certifying higher-order recursion operators to derive certificate generators for nondeterministic algorithms. *Nondeterministic programming* is a design methodology that consists in searching for solutions to a constraint satisfaction problem within a given space of potential solutions, which is generally expressed as a tree. Candidate solutions are evaluated in some fixed order, typically depth-first. Although usually requiring some form of *backtracking* to manage control flow, this type of search is more efficient in terms of program complexity than brute force enumeration of all the candidates, and also easier to express in a functional setting than breadth-first traversal.

In general, there is a notion of *position* or *index* for traversable container types, i.e. they are isomorphic to the type $P \to A$, where P is the type of positions and A the type of contained elements. Lists, for instance, have integers as positions, since we can regard them as functions from integers into values (of the list element type). Nondeterministic algorithms can be extended to record the path in the tree leading to a valid solution as a certificate, allowing the checker to reproduce the search strategy. This effect can be captured using monads [15], with which we assume the reader is familiar.

To support nondeterministic programming, a monad must provide two additional operations: *mzero* and *mplus*. The former denotes a failing computation, returning no results. The latter makes a nondeterministic choice between two computations. The generality and expressiveness of Haskell's type system allows us to write nondeterministic computations that are parameterised by an arbitrary monad, providing it supports these operations. We encode the fact that a monad supports nondeterminism with instances of the following classes:

```
class Monad m ⇒ MonadZero m where
  mzero :: m α
class MonadZero m ⇒ MonadPlus m where
  mplus :: m α → m α → m α
```

Using these operators, let us write a function that searches a tree for an element satisfying a given predicate:

```
find :: MonadPlus m ⇒ (α → Bool) → Tree α → m α
find p = foldT mzero test
  where test flt x frt
    | p x = return x
    | otherwise = flt `mplus` frt
```

We are using the same *Tree* type and *foldT* operator as in the previous section. Now, let us propose a certifying version of *find*, namely *cfind*. The function *cfind* behaves exactly like *find*, but it additionally returns the position of the element in the structure. For this purpose, we introduce yet another certifying version of *foldT*, and only make minor changes in *find*. Our certifying operator does most of the work, basically keeping track of the path that we take for each alternative.

First of all, we need a data type for describing positions.

$$\textbf{data}\ \mathit{Choice} = L \mid R$$
$$\textbf{type}\ \mathit{Position} = [\mathit{Choice}]$$

At every node in the tree, we can either go left or right (denoted by L and R, respectively). Any node in the tree can be identified by the sequence of choices we have to make so as to get there from the root of the tree, so we define the position to be a list of choices.

To keep a record of the current path in the tree, we need the monad to hold a trace of type *Position*. So as to keep the monad type abstract, we will resort to the standard *MonadWriter* class. Monads in this class support the following operation (among others):

$$\mathit{tell} :: \mathit{MonadWriter}\ p\ m \Rightarrow p \to m\ ()$$

Type p is the type of positions, which does not have to be fixed in general. Every time we want to add a position p to the trace, we perform *tell* p. The type of positions should be a *Monoid*, and every *tell* operation appends its parameter to the trace using $(\oplus)$. In the running example, positions are lists, which are already defined as monoids in the Haskell module *Data.Monoid*. We can now give a definition for our certifying *fold*:

$$\begin{aligned}
&\mathit{cfoldT} :: (\mathit{MonadWriter}\ \mathit{Position}\ m) \Rightarrow \\
&\qquad m\ b \to (m\ b \to t \to m\ b \to m\ b) \to \mathit{Tree}\ t \to m\ b \\
&\mathit{cfoldT}\ z\ f\ \mathit{Empty} = z \\
&\mathit{cfoldT}\ z\ f\ (\mathit{Node}\ lt\ x\ rt) = f\ (\mathit{tell}\ [L] \gg \mathit{cfoldT}\ z\ f\ lt) \\
&\qquad\qquad\qquad\qquad\quad x \\
&\qquad\qquad\qquad\qquad\quad (\mathit{tell}\ [R] \gg \mathit{cfoldT}\ z\ f\ rt)
\end{aligned}$$

The function *cfoldT* is a just like its non-certifying counterpart, but it records every choice of branching in the trace monad: if we examine the left branch, we append L to the trace; if we examine the right branch, we append R.

However, we would like *cfind* to work in an arbitrary nondeterminism monad. By using a standard *monad transformer*, called *WriterT*, we can add tracing capabilities to any monad. Thus, for every monad m, the monad *WriterT Position* m encapsulates the same effects as m, plus a *Position* trace. This monad provides an operation

$$\mathit{runWriterT} :: \mathit{WriterT}\ w\ m\ a \to m\ (a, w)$$

which performs all tracing effects, accumulating the resulting trace in the second component of the result. This operation, along with *cfoldT*, allows us to define *cfind* as follows:

```
cfind :: MonadPlus m ⇒ (a → Bool) → Tree a → m (a, Position)
cfind p = runWriterT ∘ cfoldT mzero test
  where test flt x frt
      | p x = return x
      | otherwise = flt `mplus` frt
```

The function *cfind* captures the essence of a nondeterministic depth-first search in a binary tree. It takes a predicate p and a tree t, and it finds the first element x in t such that $p\ x$. In addition, it outputs the path leading to x in the tree as certificate, so that a checker can efficiently locate the element. It is interesting to note that *cfind* works independently of the way in which we implement nondeterminism: the monad m encapsulates these details.

5 Conclusion

We have provided a general framework for building and checking certificates for a large class of programs and properties. The framework relies on certifying combinators, that carry certificates throughout computations and eventually to results, and checking combinators, that construct checkers for complex properties from checkers for simpler properties. The usefulness of the framework has been demonstrated through a set of examples.

It could be argued that the main challenge with result checking is to find *ad hoc* certifying algorithms, and is thus more central to algorithmics than programming languages. Yet, providing good support for result checking in programming languages is essential to promoting its generalisation. In this sense, we hope that the framework presented in the paper will contribute to increase the number of applications that make beneficial use of result checking.

Improving the efficiency of the generic checker seems necessary. One goal would be to check primality using Pocklington's criterion, achieving performances on a par with [7]. A further goal would be to apply our framework to known certifying algorithms; in particular, considering certifying algorithms for graphs, based on previous work on functional graph algorithms [6,4].

Finally, it would be appealing to implement a result checking infrastructure in the context of a distributed functional language; the implementation should address several interesting issues, including setting up protocols for building and transmitting certificates (certificates, like results, may be built by several cooperating parties), guaranteeing the correctness of result checkers (checkers can be downloaded from untrusted third parties), and providing access to a database of mathematical theorems that can be used to check result checkers (so that the proof of the result checker based on Pocklington's criterion does not need to contain a proof of the criterion itself).

References

1. Barthe, G., Crégut, P., Grégoire, B., Jensen, T.P., Pichardie, D.: The mobius proof carrying code infrastructure. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 1–24. Springer, Heidelberg (2008)
2. Blum, M., Kannan, S.: Designing programs that check their work. J. ACM 42(1), 269–291 (1995)
3. Bright, J.D.: Checking and Certifying Computational Results. PhD thesis (1994)
4. Brunn, T., Moller, B., Russling, M.: Layered graph traversals and hamiltonian path problems-an algebraic approach. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 96–121. Springer, Heidelberg (1998)
5. Caprotti, O., Oostdijk, M.: Formal and efficient primality proofs by use of computer algebra oracles. J. Symb. Comput. 32(1/2), 55–70 (2001)
6. Erwig, M.: Functional programming with graphs. In: Proceedings of the second ACM SIGPLAN international conference on Functional programming, pp. 52–65. ACM, New York (1997)
7. Grégoire, B., Théry, L., Werner, B.: A computational approach to pocklington certificates in type theory. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 97–113. Springer, Heidelberg (2006)
8. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 102–118. Springer, Heidelberg (2007)
9. Hudak, P., Peyton Jones, S.L., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J.H., Guzmán, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R.B., Nikhil, R.S., Partain, W., Peterson, J.: Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. SIGPLAN Notices 27(5), R1–R164 (1992)
10. Hudak, P., Peterson, J., Fasel, J.: A gentle introduction to Haskell 98 (1999), `http://www.haskell.org/tutorial/`
11. Jones, S.P., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming, pp. 50–61. ACM, New York (2006)
12. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Morrisett, J.G., Peyton Jones, S.L. (eds.) POPL, pp. 42–54. ACM, New York (2006)
13. Necula, G.C.: Proof-carrying code. In: POPL, pp. 106–119 (1997)
14. Sullivan, G.F., Masson, G.M.: Using certification trails to achieve software fault tolerance. In: 20th International Symposium on Fault-Tolerant Computing, FTCS-20. Digest of Papers, June 1990, pp. 423–431 (1990)
15. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995)
16. Zipitría, F.: Towards secure distributed computations. Master's thesis, Universidad de la República, Uruguay (2008)