SNÅRKL: Somewhat Practical, Pretty Much Declarative Verifiable Computing in Haskell

Gordon Stewart(✉) 0000-0003-0244-2980, Samuel Merten 0000-0001-8763-0053, and Logan Leland 0000-0002-1434-386X

Ohio University, Athens, Ohio, USA
{gstewart, sm137907, ll734713}@ohio.edu

Abstract. Verifiable computing (VC) uses cryptography to delegate computation to untrusted workers. But in most VC schemes, the delegated program must first be *arithmetized* – expressed as a circuit with multiplication and addition over a finite field. Previous work has compiled subsets of languages like C, LLVM, and bespoke assembly to arithmetic circuits. In this paper, we report on a new DSL for VC, called SNÅRKL ("Snorkel"), that supports encodings of language features familiar from functional programming such as products, case analysis, and inductive datatypes. We demonstrate that simple constraint-minimization techniques are an effective means of optimizing the resulting encodings, and therefore of generating small circuits.

1 Introduction

It is now possible, using today's cryptographic techniques and systems, to execute a computation remotely – on an untrusted computer such as an AWS virtual machine – while verifying locally without re-execution that the computation was done correctly. Due to recent advances in the systems and theory behind this kind of *verifiable computing* (VC), it is occasionally even practical to delegate a computation in this way: depending on the system and computation, the total latency to arithmetize a program (as an arithmetic circuit or set of arithmetic constraints), set up shared parameters like cryptographic keys, remotely execute the computation, and locally verify the result is now just a few orders of magnitude higher than the time it would have taken to execute the computation locally (cf. [16, §5]).

These performance results have not been easily won, however. Since about 2007,[1] cryptographers have worked to refine the underlying cryptographic and complexity-theoretic techniques – probabilistically checkable proofs [1,2], interactive proofs [13], efficient arguments systems [6]. Most systems now use variants of the protocol and representation published by Gennaro, Gentry, Parno, and Raykova (GGPR) in 2013 [11]. At the same time, researchers in practical cryptography have applied tools from the systems and compilers literatures to build verifiable computing platforms that are approaching practicality [4,8,21].

[1] See Walfish et al.'s ACM survey [22] for a summary of the recent history.

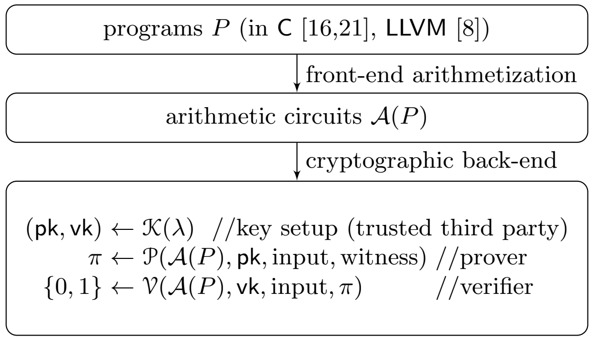


Fig. 1: Architecture of Verifiable Computing (VC) Systems

The architectures of the most recent systems follow a common pattern (Figure 1). At the top of the VC pipeline, a compiler translates the high-level representation of a program – in a language like C or LLVM – to an equivalent representation either as an arithmetic circuit or as a set of constraints that encodes the behavior of an arithmetic circuit. Only terminating programs can be arithmetized in this way.[2] For example, Figure 2 gives an arithmetic circuit respresentation of the expression $\mathsf{out} = \textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{x}\ \textbf{else}\ \mathsf{y}$. The variables b, x, and y are input "wires" to the circuit. The italicized variables x_1 and x_2 are internal wires that must be instantiated by the proving party. The gates perform field operations such as multiplication and addition.

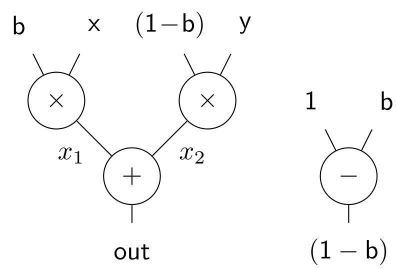


Fig. 2: An arithmetic circuit implementing ($\mathsf{out} = \mathsf{if\ b\ then\ x\ else\ y}$). The variable b ranges over $\{0, 1\}$, a constraint that must be encoded separately.

In the second phase, a cryptographic backend computes from the circuit representation three subroutines:

- a key generator $\mathcal{K}$ which establishes proving and verification keys to be used by the prover (remote) and verifier (local);
- a prover $\mathcal{P}$ which solves for witness values and constructs a succinct cryptographic proof π that the computation was executed correctly; and
- a verifier $\mathcal{V}$ which checks that the proof is valid.

The system is complete when $\mathcal{V}$ never rejects proofs generated by $\mathcal{P}$. The system is secure when, for adversarial but computationally bounded provers $\mathcal{P}'$, the

[2] BCGTV [3] approximates potentially nonterminating programs by first translating to assembly (for the bespoke TinyRAM architecture), then "executing" a bounded number of steps of the program by arithmetizing the transition relation of the underlying instruction set architecture (ISA).

probability that $\mathcal{P}'$ convinces $\mathcal{V}$ to accept a false proof π' is bounded by $\mathsf{negl}(\lambda)$, for some negligible function negl and security parameter λ.

In Figure 1, the input to $\mathcal{P}$ and $\mathcal{V}$ is the assignment of values to the computation's input variables provided by the verifying party. The witness is generated by the prover, and can be understood as a satisfying assignment – given the inputs – of the internal wires (e.g., x_1, x_2) of the circuit that results from arithmetizing P. Some VC systems, such as `libsnark` [5], support zero-knowledge computation in the sense that the verifier learns nothing about witness when verifying π. Verification time after the initial setup phase is usually small – on the order of milliseconds to seconds, depending on the size of the program input. Key-generation and proving can be more expensive, depending on the number of circuit variables and constraints; `libsnark` reports times in the tens of minutes for large circuits.

Contributions. Existing VC systems support imperative source languages like C [16,21] and LLVM IR [8] but not features found in functional languages like sums, products, user-defined inductive datatypes and case analysis. This paper reports on the first DSL supporting such features that compiles to a verifiable computing back-end, `libsnark`, using tools that apply systematic and general constraint-minimization techniques to the arithmetic encodings of such programs in order to generate small circuits. Our primary contributions are threefold:

– We show encodings into arithmetic constraints of language features familiar from type theory and functional programming: sums and products, inductive datatypes, and case-analysis (§3). As far as the authors are aware, no other VC tool has direct encodings for these features.
– We demonstrate that straightforward constraint minimization, when applied systematically to the arithmetic encodings of such programs, is a viable method of generating and of solving small circuits. Small circuits lead to concomitant low key-generation and proving times (§5).
– We implement everything described in the paper as a prototype Haskell DSL, called SNÅRKL ("Snorkel"), that is open source and freely available.[3]

Organization. In Section 2 we introduce the fundamentals of SNÅRKL by example. Section 3 presents the compilation toolchain, and gives arithmetic encodings of language features like sums, products, inductive datatypes, and case-analysis. Section 4 is devoted to SNÅRKL's constraint-minimization algorithm. We report in Section 5 on preliminary measurements of SHA3 KECCAK-f and other microbenchmarks and, in Section 6, put SNÅRKL in its broader research context.

Zero-Knowledge Proof. SNÅRKL's verifiable computing backend, `libsnark`, supports the construction of zero-knowledge proofs (the π's of Figure 1), in which the verifier $\mathcal{V}$ learns only the validity of the witness, not the witness itself. While we do not stress zero-knowledge proof in the remainder of the paper, we point out here that SNÅRKL is entirely compatible with zero knowledge as implemented in `libsnark`: whether π can be made zero knowledge depends on the cryptographic backend (`libsnark`), not the compiler that arithmetizes programs (SNÅRKL).

[3] `https://github.com/gstew5/snarkl`

Listing 2.1: Syntax of SNÅRKL's typed expression language TExp

```
data TExp :: Ty → ∗ → ∗ where
  TEVar :: TVar ty → TExp ty a
  TEVal :: Val ty a → TExp ty a
  TEUnop :: Typeable ty1 ⇒ TUnop ty1 ty → TExp ty1 a → TExp ty a
  TEBinop :: (Typeable ty1, Typeable ty2) ⇒
    TOp ty1 ty2 ty → TExp ty1 a → TExp ty2 a → TExp ty a
  TEIf :: TExp 'TBool a → TExp ty a → TExp ty a → TExp ty a
  TEAssert :: Typeable ty ⇒ TExp ty a → TExp ty a → TExp 'TUnit a
  TESeq :: TExp 'TUnit a → TExp ty2 a → TExp ty2 a
  TEBot :: Typeable ty ⇒ TExp ty a
data Ty where
  TField :: Ty
  TBool :: Ty
  TArr :: Ty → Ty
  TProd :: Ty → Ty → Ty
  TSum :: Ty → Ty → Ty
  TMu :: TFunct → Ty
  TUnit :: Ty deriving Typeable
data TFunct where
  TFConst :: Ty → TFunct
  TFId :: TFunct
  TFProd :: TFunct → TFunct → TFunct
  TFSum :: TFunct → TFunct → TFunct
  TFComp :: TFunct → TFunct → TFunct deriving Typeable
```

2 SNÅRKL by Example

SNÅRKL programs are embedded in Haskell through the use of GHC's [12] `RebindableSyntax` and `DataKinds` language extensions. `RebindableSyntax` co-opts Haskell's do-notation for sequencing SNÅRKL commands. `DataKinds` is used to embed SNÅRKL's type system into Haskell. As an example, consider the following snippet of SNÅRKL code.

```
1 arr_ex :: TExp 'TField Rational → Comp 'TField
2 arr_ex x = do
3  a ← arr 2
4  forall [0..1] (λ i → set (a,i) x)
5  y ← get (a,0)
6  z ← get (a,1)
7  return $ y + z
```

Line 3 uses the arr keyword to allocate an array of size 2, bound in the remainder of the function body to variable a. In line 4, SNÅRKL's forall combinator, of type

[b] → (b → Comp 'TUnit) → Comp 'TUnit

initializes a. The function set in the body of the lambda is the standard array update, with complement get satisfying the usual McCarthy laws. Lines 5 and 6 read twice from a, at indices 0 and 1.

In the type of arr_ex, TExp t r is the type of expressions in SNÅRKL's typed intermediate language, with t ranging over SNÅRKL types and the metavariable r a Haskell type. Comp is SNÅRKL's compilation monad (about which we say more in Section 3). Higher-level SNÅRKL code is built using combinators that operate over and return TExps, in the style of an embedded DSL. The full syntax of the TExp expression language is given in Listing 2.1. In what follows, we discuss the relevant points.

SNÅRKL*'s type system* is embedded into Haskell using the GADT [23] TExp. TExp is parameterized by a SNÅRKL type t, of (data-)kind Ty, and a Haskell type r (of kind $*$). The type system is mostly standard. TField is the type of field elements in the underlying field, typically **Rational**. In expression types TExp t r, we often omit the r to save space in listings. In each such case, r is specialized to **Rational**. The constructor TEBot provides an escape hatch (used to compile sums and bounded recursion, Section 3). There are no constructors for the complex types in Ty (TProd, TSum, etc.). Values of these types are built using higher-level Haskell combinators.

To support user-defined inductive types, the recursive-type constructor TMu quantifies over a user-defined type functor TFunct. In the signatures of SNÅRKL's (iso-recursive) roll and unroll combinators, we use a Haskell type family Rep

```
type family Rep (f :: TFunct) (x :: Ty) :: Ty
type instance Rep ('TFConst ty) x = ty
type instance Rep 'TFId x = x
type instance Rep ('TFProd f g) x = 'TProd (Rep f x) (Rep g x) ...
```

to encode the semantics of these functors. The signatures of roll and unroll are:

```
unroll :: ... ⇒ TExp ('TMu f) → Comp (Rep f ('TMu f))
roll :: ... ⇒ TExp (Rep f ('TMu f)) → Comp ('TMu f)
```

Elided in ... are Typeable-instance constraints for type Rep f ('TMu f) and the promoted[4] type f. These constraints, which appear elsewhere in Listing 2.1, facilitate reflective programming on TExps. For example, it is possible to write a function var_is_bool with type Typeable ty ⇒ TVar ty → **Bool** that determines statically whether a given program variable x is boolean.

More interesting programs are also encodable. Consider the following code, which implements the type of untyped lambda-calculus terms.

[4] The effect of GHC's `DataKinds` extension is to implicitly promote datatypes like TFunct to kinds, and constructors of user-defined datatypes (TFConst, TFId, etc.) to type constructors. Type constructors that have been promoted in this way are marked by an initial apostrophe, as in 'TFId.

```
type TTerm = 'TMu TF
type TF = 'TFSum ('TFConst 'TField) ('TFSum 'TFId ('TFProd 'TFId 'TFId))
```

In math, the functor TF is $F(\tau) = \mathsf{TField} + \tau + \tau \times \tau$. A lambda term (in DeBruijn-style) is either a field element (type TField) encoding a DeBruijn index, an abstraction with body of type μF, or an application (a pair of lambda terms $\mu F \times \mu F$). The constructor for application is:

```
1 app :: TExp TTerm → TExp TTerm → Comp TTerm
2 app t1 t2 = do
3  t ← pair t1 t2
4  t' ← inr t
5  v ← inr t'
6  roll v
```

Assuming t1 and t2 are lambda terms (Snårkl expressions of type TTerm), pair t1 t2 constructs an expression t of type 'TProd TTerm TTerm (line 3). Lines 4 and 5 inject t to an expression v of type 'TField+(TTerm+(TTerm×TTerm)). In line 6, we roll v as an expression of type 'TMuTF=TTerm.

3 Compiling to R1CS

Encoding a small functional language into Haskell is all well and good. But how do we go about compiling to arithmetic circuits? Figure 3 provides an overview of the general strategy. The target language, Rank-1 Constraint Systems (R1CS), is `libsnark`'s input specification. At the top of the compiler stack, we elaborate

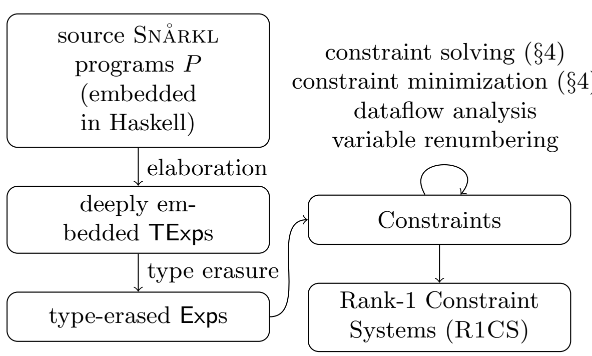


Fig. 3: The Snårkl compiler

Snårkl programs P to the deeply embedded TExp language of Section 2. Then we erase types, which facilitates later phases, by compiling TExps to a similar but untyped language Exp. Exps are compiled to a language of Constraints designed for easy optimization. It is at this Constraints level that we run most optimizations, including constraint minimization (Section 4) and dataflow

SNÅRKL Code (from §2)	Step-by-Step Elaboration to TExp

```
1   arr_ex2 :: Comp 'TField       // let elaboration environment ρ_0 = ∅ in
2   arr_ex2 = do
3    x ← fresh_input              // freshvar x_0; mark x_0 as input; let x = TEVar x_0 in
4    a ← arr 2                    // freshloc l_0; freshvars a_0, a_1; let a = l_0 in
5                                 // let ρ_1 = ρ_0[(a, 0) ↦ a_0][(a, 1) ↦ a_1] in
6    set (a,0) x                  // let ρ_2 = ρ_1[(a, 0) ↦ x] in
7    set (a,1) x                  // let ρ_3 = ρ_2[(a, 1) ↦ x] in
8    y ← get (a,0)                // let y = ρ_3[(a, 0)] in
9    z ← get (a,1)                // let z = ρ_3[(a, 1)] in
10   return $ y + z               // TEBinop (TOp Add) y z
```

Fig. 4: SNÅRKL to TExp

analysis. The minimizer doubles as a constraint solver for generating witness values (given inputs) to assign to the internal "wires" in the circuit representation of a computation (the witness of Figure 1).

3.1 Elaboration

Elaboration uses a code-generation state monad Comp that incorporates `gensym` for fresh names and a compile-time symbol table that maps "objects" in the source language (values of nonscalar types such as arrays, products, sums) to associated constraint variables. As an example, consider the array code we presented in Section 2, re-listed and slightly modified in the first column of Figure 4.

The main difference is at line 3 where the variable x is now a program input (an "input wire" in the resulting arithmetic circuit) as opposed to a parameter of the Haskell function arr_ex. Also, the forall that was previously at line 4 has been unrolled. This function arr_ex2 is elaborated by SNÅRKL to a TExp package (TExpPkg), which records the total number of variables allocated during elaboration, the input variables, and the TExp itself:

$$\mathsf{TExpPkg}\ \{\ \mathsf{allocated\_vars} = 3,\ \mathsf{input\_vars} = \{x_0\},\\ \mathsf{texp} = \mathsf{TEBinop}\ (\mathsf{TOp\ Add})\ (\mathsf{TEVar}\ x_0)\ (\mathsf{TEVar}\ x_0)\ \}$$

The resulting TExp ranges over the single input variable x_0 (the two other variables allocated during elaboration do not appear). The expression returns the result of doubling the input variable x_0, the same behavior as arr_ex2.

The elaboration process is explained in Figure 4. The environment $\rho : \mathsf{loc} \times \mathsf{int} \to \mathsf{var} + \mathsf{loc}$ maps symbolic locations (introduced during elaboration) and integer offsets to SNÅRKL program variables and other symbolic locations. The declaration x ← fresh_input on line 3 allocates a new variable TEVar x_0 bound to x in the remainder of the function. In line 4, we "allocate" an array (of field elements) of size 2. At elaboration, the effect of this command is to:

- generate a fresh symbolic location l_0, the base of the array a;
- generate two fresh variables a_0 and a_1, the array's initial contents;
- update the elaboration environment ρ to map $(\mathsf{a}, 0)$ to a_0 and $(\mathsf{a}, 1)$ to a_1.

The array updates of lines 6 and 7 overwrite ρ to map both $(\mathsf{a}, 0)$ and $(\mathsf{a}, 1)$ to the input variable x. The array gets of lines 8 and 9 look up the bindings associated with a at offsets 0 and 1. The Haskell metavariables a, x, y, and z are used only during elaboration, and are distinct from the object-language variables x_0, a_0, and a_1, which may appear in the generated TExp. The location l_0 is drawn from a distinct namespace and does not appear in the elaborated expression.

3.2 Products, Sums, Recursion

Products can be elaborated as if they were heterogeneous two-dimensional arrays. For example, the code fragment **do** { p ← pair 1.0 2.0; fst_pair p } that builds a pair and projects its first element elaborates to

```
TESeq (TEAssert (TEVar p0) (TEVal 1.0))
  (TESeq (TEAssert (TEVar p1 (TEVal 2.0))) (TEVar p0)).
```

Here p_0 and p_1 are variables that stand for the first and second projections of the pair. Behind the scenes, a location $\mathsf{p} = l_0$ was allocated such that $\rho[(\mathsf{p}, 0)]$ maps to p_0 and $\rho[(\mathsf{p}, 1)]$ maps to p_1. TEAssert (TEVar p_0) (TEVal 1.0) – asserting that the variable p_0 equals 1.0 – ensures that p_0 is resolved, in the eventual rank-1 constraint system, to the value 1.0.

Compiling sums is trickier. Since the target execution model is arithmetic circuits (specifically, their generalization as the arithmetic constraint language R1CS), we cannot – when implementing case-analysis – just "jump" to the code for the left or right of a match on an expression like

```
e : TExp ('TSum 'TBool TField) Rational.
```

Whether e was built with inl or inr may depend on an input variable of the compiled circuit, as in:

```
do { b ← fresh_input; x ← inl false; y ← inr 0.0; z ← if b then x else y;
     case_sum z (λ b0 → ...) (λ n0 → ...) }
```

Snårkl's solution is to elaborate both branches of the case_sum and combine the results, dependent on the value of the input b (not known at compile-time). To avoid large blowups in the size of the generated code, the compiler performs constant propagation to eliminate spurious branches whenever possible. When a conditional cannot be determined statically, the compiler *zips* (Figure 5) the branches to the leaves of the syntax tree to ensure that expressions of compound type (TSum, TProd, etc.) are represented by location expressions at elaboration time – an invariant that facilitates the compilation of eliminators such as fst_pair.

Internally, sums are represented as pairs (b, (e1, e2)) where b is a boolean expression indicating left or right, e1 is the left-hand expression of the sum (if one exists) and e2 the right-hand (if one exists). In the constructors inl and inr, the uninstantiated branch (right for inl, left for inr) is populated by the expression TEBot, which may assume any type. The elaborator implements a simple static analysis to track both TEBots and boolean expressions with known values.

Modulo such optimizations, case_sum is implemented:

$$\boxed{\vdash^{\mathsf{b}}_{\tau} \mathsf{e1} \bowtie \mathsf{e2} = \mathsf{e12}}$$

$$\frac{}{\vdash^{\mathsf{b}}_{\mathsf{TUnit}} \mathsf{e1} \bowtie \mathsf{e2} = \mathsf{TEVal\ VUnit}}\ \mathsf{zipUnit} \qquad \frac{\tau \in \{\mathsf{TField}, \mathsf{TBool}\}}{\vdash^{\mathsf{b}}_{\tau} \mathsf{e1} \bowtie \mathsf{e2} = \mathsf{TEIf\ b\ e1\ e2}}\ \mathsf{zipBase}$$

$$\frac{\vdash^{\mathsf{b}}_{\tau_1} (\mathsf{fst\_pair\ e1}) \bowtie (\mathsf{fst\_pair\ e2}) = \mathsf{p1} \qquad \vdash^{\mathsf{b}}_{\tau_2} (\mathsf{snd\_pair\ e1}) \bowtie (\mathsf{snd\_pair\ e2}) = \mathsf{p2}}{\vdash^{\mathsf{b}}_{\mathsf{TProd}\ \tau_1\ \tau_2} \mathsf{e1} \bowtie \mathsf{e2} = \mathsf{pair\ p1\ p2}}\ \mathsf{zipProd}$$

$$\frac{\vdash^{\mathsf{b}}_{\mathsf{TProd\ TBool\ (TProd}\ \tau_1\ \tau_2)} (\mathsf{rep\_sum\ e1}) \bowtie (\mathsf{rep\_sum\ e2}) = \mathsf{p}}{\vdash^{\mathsf{b}}_{\mathsf{TSum}\ \tau_1\ \tau_2} \mathsf{e1} \bowtie \mathsf{e2} = \mathsf{unrep\_sum\ p}}\ \mathsf{zipSum}$$

$$\frac{\vdash^{\mathsf{b}}_{\mathsf{Rep\ f\ (TMu\ f)}} (\mathsf{unroll\ e1}) \bowtie (\mathsf{unroll\ e2}) = \mathsf{r}}{\vdash^{\mathsf{b}}_{\mathsf{TMu\ f}} \mathsf{e1} \bowtie \mathsf{e2} = \mathsf{roll\ r}}\ \mathsf{zipRec}$$

Fig. 5: Type-directed zipping

```
case_sum :: forall τ1 τ2 τ. ... ⇒
  (TExp τ1→ Comp τ)→ (TExp τ2→ Comp τ)→ TExp ('TSum τ1 τ2)→ Comp τ
case_sum f1 f2 e =
  do { let p = rep_sum e;
       b ← fst_pair p; p_rest ← snd_pair p;
       e1 ← fst_pair p_rest; e2 ← snd_pair p_rest;
       le ← f1 e1; re ← f2 e2;
       zip_vals τ (not b) le re }
```

When e = (b, (e1, e2)), neither e1 nor e2 is known to evaluate to TEBot, and the value of b is not known statically, case_sum generates code for the left branch (f1 e1) and the right branch (f2 e2) and applies the transformation zip_vals – the $\bowtie$ relation of Figure 5 – to the resulting expressions. Indexing the relation are the type τ of e1 and e2 and the boolean conditional **not** b (inl is defined to let b = false, hence the negation). The $\bowtie$ relation maps two TExps e1 and e2 to a result e12 in which the b branch – deciding between e1 and e2 – has been pushed to the leaves of the syntax tree, enforcing the invariant that TExps of nonbase-type such as TSum or TProd are represented as symbolic locations during elaboration. The relation itself is defined by case analysis on the structure of τ. In the definitions of case_sum and $\bowtie$, the coercions

```
  rep_sum :: TExp ('TSum τ1 τ2) → TExp ('TProd 'TBool ('TProd τ1 τ2))
unrep_sum :: TExp ('TProd 'TBool ('TProd τ1 τ2)) → TExp ('TSum τ1 τ2)
```

cast between sums as products (rep_sum), and back again (unrep_sum).

Snårkl supports recursive functions through the use of a (bounded) fixpoint combinator fix whose type is:

```
fix :: ((TExp τ1 → Comp τ2) → (TExp τ1 → Comp τ2)) → TExp τ1 → Comp τ2
```

At a user-configurable depth[5] d the expression fix f e returns TEBot, indicating delayed error; if the output of the resulting circuit, given user inputs, depends on

[5] The recursion bound is necessary to ensure that elaboration terminates.

the TEBot expression (it exceeds the recursion bound – perhaps the user input is the serialization of a list of size $d+1$), the circuit evaluation will go wrong.

3.3 From **TExps** to R1CS

Compiling TExps to Rank-1 Constraint Systems is more straightforward, and in general follows previous work on arithmetizing general-purpose programs. The main difference is that between TExp and R1CS we employ an intermediate constraint representation CONSTRAINTS that is more suitable than R1CS for optimization. We present R1CS first, then CONSTRAINTS and the encoding of select TExps into CONSTRAINTS. Section 4 shows how to optimize CONSTRAINTS.

The input specification language of `libsnark`, Rank-1 Constraint Systems (R1CS), builds on the QAP arithmetic constraint representation of GGPR [11]. A rank-1 constraint system is a system of constraints on degree-1 polynomials over a finite field, e.g.:

$$\begin{array}{ccccc} A & * & B & = & C \\ (2x_0 + 3x_1) & * & (-3x_1) & = & 2x_0 + 4x_1 \end{array}$$

The variables x_0, x_1 range over a finite field $\mathcal{F}_p$ of prime characteristic p. A system of such constraints encodes the behavior of an arithmetic circuit (cf. GGPR [11] for additional details).

Listing 3.1: SNÅRKL's representation of Rank-1 Constraint Systems (R1CS)

```
1 type Assgn a = Map.IntMap a
2 data Poly a where Poly :: Field a ⇒ Assgn a → Poly a
3 data R1C a where R1C :: Field a ⇒ (Poly a, Poly a, Poly a) → R1C a
4 data R1CS a = R1CS {
5   r1cs_clauses :: [R1C a], r1cs_num_vars :: Int,
6   r1cs_in_vars :: [Var], r1cs_out_vars :: [Var],
7   r1cs_gen_witness :: Assgn a → Assgn a }
```

SNÅRKL's representation of R1CS is given in Listing 3.1. An assignment (line 1, Assgn a) maps variables (type Var = **Int**) to values of type a. A rank-1 polynomial (line 2) is just an assignment in which a has the operators of a field and variable -1 is by convention the constant term. A rank-1 constraint (line 3) is a polynomial constraint $A * B = C$ in which A, B, and C are all polynomials. The R1CS type collects a list of rank-1 constraints, the number of variables appearing in the constraints, which variables are inputs and outputs, and a function, r1cs_gen_witness, that maps input assignments to satisfying witnesses.

SNÅRKL's constraint language presents an abstraction layer on top of R1CS, making it easier to optimize R1CS-style encodings. The main datatype is:

```
data Constraint a =
   CAdd a [(Var,a)]
  | CMult (a,Var) (a,Var) (a,Maybe Var)
  | CMagic Var [Var] ([Var] → State (SEnv a) Bool).
```

$$\boxed{[\![\ \mathsf{e}\ ]\!]_{\mathsf{out}} = [\mathsf{CAdd}\ \ldots, \ldots]}$$

$$
\begin{aligned}
[\![\ \mathsf{EVar\ x}\ ]\!]_{\mathsf{out}} &= [\mathsf{CAdd\ 0\ (fromList\ [(out}, 1),\ (\mathsf{x}, -1)]]\\
[\![\ \mathsf{EVal\ c}\ ]\!]_{\mathsf{out}} &= [\mathsf{CAdd\ c\ (fromList\ [(out}, -1)]]\\
[\![\ \mathsf{EAssert\ (EVar\ x1)\ e2}\ ]\!]_{\mathsf{out}} &= [\![\ \mathsf{e2}\ ]\!]_{\mathsf{x1}}\\
[\![\ \mathsf{EBinop\ Or\ e1\ e2}\ ]\!]_{\mathsf{out}} &= [\![\ \mathsf{e1}\ ]\!]_{\mathsf{e1\_out}} +\!\!+ [\![\ \mathsf{e2}\ ]\!]_{\mathsf{e2\_out}}\\
&+\!\!+ [\![\ \mathsf{EBinop\ Mult\ (EVar\ e1\_out)\ (EVar\ e2\_out)}\ ]\!]_{\mathsf{e12\_out}}\\
&+\!\!+ [\![\ \mathsf{EBinop\ Sub\ (EBinop\ Add\ (EVar\ e1\_out)\ (EVar\ e2\_out))\ (EVar\ out)}\ ]\!]_{\mathsf{e12\_out}}
\end{aligned}
$$

Fig. 6: TExps to CONSTRAINTS (excerpts)

The type a is usually specialized to field elements. The additive constraint CAdd a [(Var, a)] asserts that the linear combination of a constant (of type a) with the variable–coefficient terms ([(Var, a)]) equals 0. For example, the constraint CAdd 2 [(x,1), (y,−3)] is $2 + 1x - 3y = 0$. Multiplicative constraints CMult ... encode facts like 2x∗3y = −7z. In general, CMult (c,x) (d,y) (e,**Just** z) means $cx * dy = ez$. When the second element of the third pair is **Nothing**, the interpretation is $cx * dy = e$.

Compiling both additive and multiplicative constraints to R1CS is straightforward. For example, the additive constraint CAdd 3 [(y,−5), (z,23)] yields:

R1C (const_poly one) (Poly (fromList [(x_c,3), (y,−5), (z,23)])) (const_poly **zero**).

The variable $x_c = -1$ is reserved for the polynomial's constant term. The function const_poly c constructs the constant polynomial equal c. Multiplicative constraints are equally straightforward. For example, CMult (3,x) (4,y) (5,**Just** z) results in the rank-1 constraint 3x ∗ 4y = 5z.

So-called CMagic constraints are hints to SNÅRKL's constraint solver that encode nondeterministic "advice" – used to resolve the values of variables introduced by the nondeterministic encodings of expressions such as disequality tests (about which we say more below).

Compiling TExps to constraints follows previous work (e.g., [16,18]), yet some of the encodings are nonobvious. Consider boolean disjunction in TExps of the form TEBinop (TOp Or) e1 e2. The encoding – after types have been erased – is given in Figure 6, along with that of variables, values, and assertions. The compilation relation $[\![\ \cdot\ ]\!]_{\mathsf{out}}$ is indexed by an output variable out that corresponds one-to-one with the output "wire" of the resulting arithmetic circuit, itself encoded as a list of constraints of type Constraint a. For example, compilation of EVar x, with output variable out, constructs the polynomial constraint 0 + 1∗out + −1x = 0 asserting that out = x. The encoding EVal c is similar.

To compile boolean disjunction EBinop Or e1 e2, we first recursively compile e1 and e2 – sending their values through fresh output variables e1_out and e2_out. Then we compile the TExp that encodes the constraint

e1_out+e2_out − out = e1_out∗e2_out.

As long as e1_out, e2_out, and out range over boolean values 0, 1 – a constraint we encode separately as the additional fact x∗x = x for each boolean variable x – the equality above is satisfiable iff out = e1_out ∨ e2_out.

Listing 4.1: Constraint minimization

```
simplify_rec :: Field a ⇒ ConstraintSet a → State (SEnv a) (ConstraintSet a)
simplify_rec S = do
  S′ ← simplify_once S
  if size S′ < size S then simplify_rec S′
  else if S − S′ ⊆ ∅ then return S′ else simplify_rec S′
where simplify_once S =
  do {S′ ← go ∅ S; remove_tauts S′}
      go W U | size U == 0 = return W
      go W U | otherwise =
        let (given, U′) = deleteFindMin U in do
        in do given′ ← subst_constr given
              given_taut ← is_taut given'
              if given_taut then go W U′
              else do {learn given′;
                       go (W ∪ {given′}) U′}
```

Many of the remaining compilation rules are straightforward (we do not show them in Figure 6). One exception is disequality testing. Here SNÅRKL uses a nondeterministic encoding borrowed from PINOCCHIO [16] and Setty et al. [18] that relies on CMagic constraints to resolve the values of a nondeterministic witness variable. Assume the expression is y = x!=0 ? 1 : 0, which we represent in C-style syntax. Both x and y are variables. The encoding is, there exists an m such that both x∗m = y and (1−y)∗x = 0. Since m is not uniquely determined by the above two facts, we use a CMagic constraint to resolve its value when solving for the witnesses of Figure 1: if x = 0 then let m = 0. Otherwise, let m equal the modular multiplicative inverse x^{-1} of x in the underlying field $\mathcal{F}_p$.

4 Constraint Minimization

Key generation and proving times in VC systems typically depend on the size, e.g., in number of constraints, of the arithmetization of the source program. Previous work (e.g., [3,16,8]) uses clever encodings of individual program constructs to optimize encoding size but no system we know of applies systematic constraint minimization.

Why is systematic optimization problematic? If the original source program is interpreted in order to find satisfying assignments, as in systems such as GEPPETTO [8], then optimizing the constraint system makes it more difficult to map particular variables and constraints back to program points in the source program; minimization may remove variables and constraints entirely. We solve this problem by having the constraint minimizer perform double duty; for a particular problem instance with concrete inputs provided by the verifying party, simply rerun the constraint minimizer with those concrete initial values. The result, using the constraint minimization algorithm we describe in this section, is a satisfying assignment for the entire constraint system.

Both constraint minimization and solving happen at the level of SNÅRKL's CONSTRAINTS intermediate language. The main data structure is an environment SEnv a = SEnv { eqs :: UnionFind a, solve_mode :: SolveMode } that stores a union-find instance, for mapping variables to their equivalence classes (or to constants) as new variable equalities are learned during optimization, and a flag solve_mode = UseMagic | JustSimplify that tells the simplifier whether to ignore CMagic constraints. If solve_mode = UseMagic (the simplifier is in solve mode), magic constraints are used to resolve the values of nondeterministic witness variables. Otherwise (simplifier mode), the simplifier ignores CMagic constraints.[6]

The main minimization routines, operating over a set of constraints S, are given in Listing 4.1. The idea (simplify_rec, line 2) is to repeatedly apply the simplification procedure simplify_once (line 7) as long as each application (line 4) successfully removes at least one constraint from the set S, because it was able to determine that the constraint was tautological. It is also possible (line 5) that some constraint has been simplified, yet the total number of constraints remains the same. In this case, we continue simplifying. If no new constraints are removed or simplified, we halt with S'.

The function go (beginning at line 8) operates over two sets, a working set of constraints W and an unselected set U. Originally, all constraints are in U. At each iteration, the function deletes the smallest constraint from U (under a particular total order, line 10), simplifies the constraint (line 11) under the equalities currently recorded in the simplification environment, SEnv, then checks whether the resulting constraint is tautological (line 13). If it is, the tautological constraint is removed and go continues to the next iteration, throwing the clause away (line 13). Otherwise (line 14), we attempt to learn new equalities from the constraint (between variables and variables, and variables and constants) and continue (line 15) with the new clause in W.

The function learn (called in line 14) implements just a few simplification rules. For example, from constraints CAdd -1 [(x,c)] (expressing $-1 + \mathsf{c}x = 0$) we learn $x = \mathsf{c}^{-1}$ as long as c is invertible. Likewise, from CAdd 0 [(x,c), (y,d)] (expressing $0 + \mathsf{c}x + \mathsf{d}y = 0$) we learn $x = y$ as long as $\mathsf{c} = -\mathsf{d}$ and c is nonzero. The function subst_constr, which substitutes the equalities currently in context into a constraint, is also straightforward. When applied to, e.g., CAdd constraints it replaces all variables by their union-find roots, replaces certain variables by constants, folds constants, and filters out terms with coefficient 0.

5 Measurements

Since SNÅRKL uses a standard VC backend, our analysis in this section forgoes a direct evaluation of the practicality of the underlying cryptography[7] in favor of answering the following questions:

[6] It would be unsound to rely on these constraints to learn new facts.

[7] `libsnark` was evaluated in [3].

Fixed Matrix Multiply a fixed $n \times n$ matrix M (known at compile time) by an n-length input vector A, resulting in the n-length output vector $M \cdot A$. Output the sum of the elements in $M \cdot A$. This microbenchmark reproduces the "Fixed Matrix, Medium" benchmark of PINOCCHIO [16, §4.3], with parameter $n = 600$.

Input Matrices Multiply an $n \times n$ input matrix M_1 by a second $n \times n$ input matrix M_2. Output the sum of the elements in $M_1 \cdot M_2$. This microbenchmark reproduces PINOCCHIO's "Two Matrices, Medium" benchmark [16, §4.3] with $n = 70$.

Keccak-f(800) The main function of SHA3's "sponge" construction. The lane width ($= 32$) is a parameter known at compile time. As input, Keccak-f(800) takes a 3-dimensional array of size $5 \times 5 \times 32$ bits. It outputs the exclusive or of the 800-bit array that results after applying 22 rounds of Keccak-f.

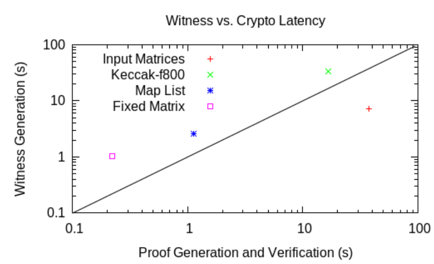
Map List Map the function $(\lambda x.x + 1)$ over a list of field elements of size 50 and return the list's last element. The size and contents of the list are circuit inputs. The generated circuit supports input lists up to size 100 elements.

Fig. 8: Description of the Benchmarks

1. Does SNÅRKL's general-purpose constraint minimizer (§4) produce circuits of comparable size to those encoded by hand in systems like PINOCCHIO?
2. How much overhead is imposed, over proof generation in `libsnark`, by using the constraint minimizer of §4 to generate circuit witnesses?

#Constraints	SNÅRKL	PINOCCHIO
Fixed Matrix	601	600
Input Matrices	347,901	347,900

(a) Constraints per benchmark



(b) Witness generation vs. cryptographic proof generation and verification latency

Fig. 7: Results

We consider the four benchmarks described in Figure 8. For benchmarks that have been implemented in PINOCCHIO (**Fixed Matrix** and **Input Matrices**) we report (Figure 7a) the number of constraints generated by SNÅRKL vs. those in PINOCCHIO's manual encoding, as reported in [16]. In each case, we generate just one additional constraint, resulting from the fact that we return the sum of the resulting matrix in addition to performing the multiplication (thus preventing over-optimization of the resulting circuit by the SNÅRKL compiler).

For each benchmark, we also measured (using Citerion [15]; confidence intervals were small) the relative latency of witness generation as performed by the constraint minimizer of Section 4 versus cryptographic proof generation and verification in `libsnark` (Figure 7b). Both of these procedures must be performed online once per problem instance. The results here are more mixed. Only one benchmark (**Input Matrices**) falls below the line, and therefore has lower witness generation than proof generation and verification latency. In the remaining benchmarks, the cost of witness generation exceeds that of proof generation but the difference is usually small. This is despite the fact that our constraint minimizer has not yet been highly optimized.

6 Related Work

There has been a great deal of work in verifiable computing over the past few years [3,4,7,8,9,10,16,18,21]. With Pinocchio [16] and its most recent incarnation Geppetto [8], researchers at MSR and elsewhere have built VC systems that incorporate novel techniques like MultiQAPs for sharing state between reusable circuit components, and energy-saving circuits for reducing cryptographic costs in programs with conditional branches. These new techniques are complementary to the work we present in this paper. Because Snårkl compiles to the clearly defined R1CS interface (Figure 3), future improvements to `libsnark` resulting from cross-fertilization by tools such as Pinocchio and Geppetto will bring immediate benefit, even without change to the compiler.

In parallel to systems like Pinocchio, Pantry [7] and its successor Buffet [21] (both refinements of previous systems Ginger [18] and Pepper [19]) showed new techniques for efficiently compiling RAM programs. Buffet, for example, adapts the RAM abstraction of TinyRAM to the compilation model of Pantry, resulting in large cryptographic speedups over previous systems. That said, Buffet's imperative input language is still a subset of C; while other tools support other (generally, subsets of) imperative languages like LLVM [8], no tool we know of directly supports functional programs as in Snårkl.

The work on TinyRAM [3,4], which is implemented as an extension of core `libsnark`, represents an interesting third point in the design spectrum: instead of directly compiling C programs to constraints, TinyRAM modifies `gcc` to output assembly programs in a small bespoke assembly language, then "executes" the programs by encoding the semantics of the TinyRAM ISA as arithmetic constraints. This execution strategy is implementable in Snårkl. In fact, one immediate goal of future work is the implementation of other kinds of abstract machines beyond just ISAs – such as interpreters and type-checkers for lambda calculi. With such tools, it may be possible to recast, e.g., dependent type systems in a VC mold: the proof that term e has type τ is a VC proof π that the arithmetization of a type-checking function f applied to e evaluates to $\mathsf{Some}\ \tau$. Finally, the design of Snårkl's frontend has benefited from long lines of work on embedded DSLs (e.g., [14]) and on multi-stage programming (e.g., [20]). Recent work on specialized type rules for DSLs (e.g., [17]) may provide a method for improving the reporting of type errors in Snårkl's embedded type system.

7 Conclusion

Verifiable computing is approaching practicality. But there is still work to do. In this paper, we report on Snårkl ("Snorkel"), a DSL embedded in Haskell for functional programming against a verifiable computing backend. We demonstrate that simple constraint minimization techniques – when applied systematically to a carefully designed intermediate representation – are an effective means of generating small circuits. Our DSL and implementation support familiar features from functional programming such as sums, products, inductive datatypes, and case analysis.

References

1. S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *JACM*, 45(3):501–555, 1998.
2. S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *JACM*, 45(1):70–122, 1998.
3. E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*. Springer, 2013.
4. E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, 2014.
5. E. Ben-Sasson et al. The libsnark library. https://github.com/scipr-lab/libsnark. [Online; accessed 23-9-2015].
6. G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, 1988.
7. B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, pages 341–357. ACM, 2013.
8. C. Costello et al. Geppetto: Versatile verifiable computation. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, volume 15. IEEE, 2014.
9. C. Fournet, M. Kohlweiss, G. Danezis, and Z. Luo. ZQL: A Compiler for Privacy-Preserving Data Processing. In *USENIX Security*, pages 163–178, 2013.
10. M. Fredrikson and B. Livshits. ZØ: An optimizing distributing zero-knowledge compiler. In *USENIX Security*, 2014.
11. R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.
12. GHC Team. The glorious Glasgow Haskell compilation system user's guide, 2005.
13. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 291–304. ACM, 1985.
14. G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *ICFP '08*, 2008.
15. B. O'Sullivan. The Criterion library. http://www.serpentine.com/criterion. [Online; accessed 23-9-2015].
16. B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013.
17. A. Serrano and J. Hage. Type error diagnosis for embedded dsls by two-stage specialized type rules. In *ESOP*, 2016.
18. S. T. Setty et al. Taking Proof-Based Verified Computation a Few Steps Closer to Practicality. In *USENIX Security*, 2012.
19. S. T. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.
20. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *PEPM*, 1997.
21. R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.
22. M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *CACM*, 58(2):74–84, 2015.
23. H. Xi, C. Chen, and G. Chen. Guarded Recursive Datatype Constructors. In *POPL*, pages 224–235. ACM, 2003.