A Trustworthy Framework for Resource-Aware Embedded Programming

Adam D. Barwell
adb23@st-andrews.ac.uk
University of St Andrews
Scotland, UK

Christopher Brown
cmb21@st-andrews.ac.uk
University of St Andrews
Scotland, UK

ABSTRACT

Systems with *non-functional* requirements, such as Energy, Time and Security (ETS), are of increasing importance due to the proliferation of embedded devices with limited resources such as drones, wireless sensors, and tablet computers. Currently, however, there are little to no programmer supported methodologies or frameworks to allow them to reason about ETS properties in their source code. *Drive* is one such existing framework supporting the developer by lifting non-functional properties to the source-level through the Contract Specification Language (CSL), allowing non-functional properties to be first-class citizens, and supporting programmer-written code-level contracts to guarantee the non-functional specifications of the program are met. In this paper, we extend the *Drive* system by providing rigorous implementations of the underlying proof-engine, modeling the specification of the annotations and assertions from CSL for a representative subset of C, called Imp. We define both an improved abstract interpretation that automatically derives proofs of assertions, and define inference algorithms for the derivation of both abstract interpretations and the context over which the interpretation is indexed. We use the dependently-typed programming language, *Idris*, to give a formal definition, and implementation, of our abstract interpretation. Finally, we show our well-formed abstract interpretation over some representative exemplars demonstrating provable assertions of ETS.

KEYWORDS

Dependent Types, Idris, Lightweight Verification, Non-Functional Properties, Abstract Interpretation, Proof-Carrying Code, Embedded Systems

ACM Reference format:
Adam D. Barwell and Christopher Brown. 2020. A Trustworthy Framework for Resource-Aware Embedded Programming. In *Proceedings of International Symposium on Implementation and Application of Functional Languages, Singapore, September 2019 (IFL'19),* 12 pages.
https://doi.org/10.1145/1122445.1122456

IFL'19, September 2019, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Programs that consider non-functional properties, such as energy consumption or maximum execution time, are of increasing importance due to the proliferation of devices with limited resources; e.g. embedded medical devices, camera pills, drones, wireless sensors, mobile phones and tablets. While conventional understanding of software correctness pertains to the *functional* properties of a program, such as the absence of errors and bugs, resource-limited embedded devices prompt additional conformance to *non-functional* requirements [23]. A system that does not conform to its non-functional specification may ultimately render the system useless, or worse, a potential danger to others; e.g. a drone depleting its battery before it can land safely will crash to the ground. It is therefore necessary to develop such systems with an awareness of, and a demonstration of conformity to, their (non-functional) specification.

Drive [6], is a framework for capturing, and reasoning about, non-functional properties such as Energy, Time and Security (ETS) in C programs. It includes the Contract Specification Language (CSL), an Embedded Domain Specific Language (EDSL) that defines C-statement annotations in order to capture non-functional properties of the statements they annotate, including energy usage, worst-case execution time (WCET), and the degree of vulnerability to side-channel attacks. *Drive* also facilitates reasoning about non-functional properties via assertion annotations; e.g. whether a statement can be executed within an energy budget (Listing 1). Previously, these contracts were automatically verified using a light-weight abstract interpretation implemented in Idris [5]. This lightweight approach to verification is a form of *proof-carrying code* [25] since the abstract interpretation automatically derives a proof of whether each assertion holds true for a given context.

In this paper we extend the *Drive* framework, creating a trustworthy and meaningful proof system for the CSL assertions and ETS properties. Specifically, we provide a rigorous implementation of the abstract interpretation. We model a larger subset of the C language, Imp, facilitating the inference of necessary contextual information by which our generated proofs are now indexed. We define a big-step operational semantics of assertion annotations in conjunction with the semantics of Imp. We additionally parameterise our abstract interpretation over the type of numeric values. Consequently, proofs are no longer limited to natural numbers, but can be generated, e.g., for integers and real numbers, given a suitable representation. Finally, we demonstrate our executable formalisation on a representative example, capturing a range of programmer-provided assertions and non-functional properties, and demonstrable proofs of these assertions. In line with the Curry-Howard correspondence [28, 30], we formulate our definitions of

language, properties, rewrites, and logical and arithmetic formulæ as types, and transformations over types that enact rewrites or determine proofs of properties as total functions. Type-checking ensures the soundness of these functions relative to the definitions given as types, thus ensuring soundness of abstract interpretation and context inference.

1.1 Contributions

(1) We present an abstract interpretation of C, extended with CSL assertions, fully implemented in Idris, by deriving and implementing a small general imperative language, called Imp.
(2) We define and implement a big-step operational semantics for well-formed Imp programs and CSL assertions, thereby facilitating robust inference of necessary contextual information in order to generate proofs for CSL assertions.
(3) We present an implementation, in Idris, of an inference system that automatically (dis)proves programmer-provided assertions in Imp, potentially making use of captured non-functional information provided by the capture annotations in CSL.

2 BACKGROUND

2.1 The Contract Specification Language

The Contract Specification Language (CSL) [6] is an embedded domain-specific language developed collaboratively between the University of St Andrews and Inria, Rennes. CSL extends C with special annotations for both capturing non-functional information about source code, and the ability to make assertions (or contracts) using the captured information.

Listing 1 shows an extract from the Levenshtein Distance algorithm, as defined in the BEEBs benchmarks [27] Two CSL capture annotations and an assertion have been added to the code. The capture annotations at Lines 19 and 23 direct the compiler to invoke a worst-case execution time (WCET) analysis for the assignment statements on Lines 20 and 24–26, respectively. The results of these analyses are assigned to the declared variables that are passed to the capture annotations; i.e. `true_time` and `false_time`. These measurements could be used by the programmer as a simple or coarse-grained check for vulnerability to side-channel attacks; i.e. using differences in execution time to infer information about secret data [20]. The assertion at Line 31 expresses this check: the implementation is vulnerable if the assertion does not hold true.

Brown *et al.* define assertion expressions as being standard Boolean expression evaluating to *true* or *false*. The proof of whether an assertion holds true within a given context (i.e. mapping of variables to values) is inferred by a simple decision procedure implemented in Idris.

2.2 Dependent Types

Dependently typed languages take advantage of the Curry-Howard correspondence, which states that, given a suitably rich type system, (certain kinds of) proofs can be represented as programs [30]. For languages with insufficiently rich type systems, such as C, dependently-typed languages can be used to produce an *abstract interpretation* [11] of a given program in those languages. Such

Listing 1: An extract of the Levenshtein algorithm from the BEEBs benchmarks suite for C

```
1  int levenshtein_distance(const char *s, const char *t) {
2    ...
3    for (j = 1; j <= tl; j++) {
4      for (i = 1; i <= sl; i++) {
5        if (s[i - 1] == t[j - 1]) {
6          __csl_time_worst(&true_time); // WCET of following stmt
7          d[i][j] = d[i - 1][j - 1];
8        }
9        else {
10         __csl_time_worst(&false_time); // WCET of following stmt
11         d[i][j] = min(d[i - 1][j] + 1, // deletion
12                       min(d[i][j - 1] + 1, // insertion
13                           d[i - 1][j - 1] + 1)); // substitution
14       }
15     }
16   }
17   __csl_assert(true_time = worst_time);
18   return d[sl][tl];
19 }
```

abstract interpretations can be used to derive proofs of desired properties [2].

In the case of dependently-typed languages, under the propositions as types view, dependent types are used to represent predicates [31]. For example, `(Even : (n : Nat) -> Type)` defines the type of evidence (or proofs) that a natural number, `n`, is even. In cases where the property does *not* hold true, e.g. `Even 1`, and assuming a suitably restricted definition of that property, the type is uninhabited. An uninhabited type represents falsity. Evidence that a predicate does not hold true can be represented by the type function, `(Not a = a -> Void)`, where `a` is a type variable and `Void` is the empty type; i.e. it has no constructors. Using dependent types in this way, properties that represent a (non-)functional specification can be encoded as predicates (i.e. types). Accordingly, total functions, `f : A -> B`, allow for the derivation of evidence that the predicate `B` can be constructed given evidence of `A`. Type-checking ensures the soundness of these functions [28].

We take advantage of the above features by implementing our system in the dependently-typed language Idris, a functional language developed at the University of St Andrews [5]. The syntax of Idris is similar to Haskell [19], and like Haskell, Idris supports algebraic data types with pattern matching, type classes, and do-notation. Unlike Haskell, Idris evaluates its terms eagerly. Definitions, e.g. of languages and well-formedness, are defined by giving their definitions as types in Idris. For example, the aforementioned `Even` predicate can be defined:

```
1  data Even : (n : Nat) -> Type where
2    Zero : Even Z
3    Succ : (ek : Even k) -> Even (S (S k))
```

where, `Even` is the name of the type being defined, `Nat` is the type of natural numbers, `(n : Nat)` is a (named) argument to the type, and both `Zero` and `Succ` are constructors. Constructors may have (named) arguments; e.g. `(ek : Even k)`. Constructors may also restrict the values of their arguments; e.g. `Zero` explicitly states that `n = 0`, and `Succ` states that `n = S (S k)` (i.e. `k + 2`), given an inhabitant of `Even k`. As is desired, under this definition, there is no way to construct an element of, e.g., `(Even 1)` using either constructor.

In order to determine whether `(Elem n)` is inhabited for a given `n`, we define the function `isEven`.

```
1  isEven : (n : Nat) -> Dec (Even n)
2  isEven Z = Yes Zero
3  isEven (S k) with (k)
4    isEven (S k) | Z = No absurd
5    isEven (S k) | (S j) with (isEven j)
6      isEven (S k) | (S j) | (Yes prf) = Yes (Succ prf)
7      isEven (S k) | (S j) | (No contra) =
8        No (\(Succ x) => contra x)
```

Here, `Dec` is the type for a decidable property, where `Yes` holds a proof of the property and `No` holds a proof of its contradiction. The `with` rule is used to pattern match on intermediate values, similar to a `case` expression in Haskell. The function `(absurd : Uninhabited t => t -> a)` is a convenience function for contradictions where a type, here `(Even 1)`, is uninhabited. The argument to `No` on Line 8 is a function of type `(Even (S (S j)) -> Void)` and represents a contradiction of `(Even (S (S j)))` when `(Even j)` is uninhabited. Since this is a total definition, as guaranteed by the type-checker, `isEven` is a decision procedure for the type (predicate) `(Elem n)` for all values of `n`.

2.3 Non-Functional Properties

The *Drive* system focuses on three non-functional properties that we consider to be the most commonplace. The first two are *time* and *energy*, and the third property, *security*, which we do not consider in this paper, is typically one that is required as time and energy properties are leaked from a program and used by adversaries to obtain information about the algorithm. In *Drive*, measurements for these non-functional properties are provided by third-party tools and models. In this paper, we omit the details of obtaining such non-functional information, leaving it for future work to extend the *Drive* system with support to link to automated tools to obtain the information automatically.

For *time*, we consider the worst-case execution time (*WCET*) obtained by executing the code with various underlying profiling tools such as the WCC compiler produced by the University of Hamburg [12].

Energy measurements are typically obtained by using a model such as those provided by Eder et al. [24] or by measuring the amount of energy in Joules (J) that is used by a complete processor package; i.e. by measuring the total energy that is drawn by each hardware CPU socket, and energy usage is typically calculated by computing the rate of change in power per unit of time using the formulae shown below:

$$Energy = Power \times Time$$
$$Joules = Watts \times Seconds$$

3 REPRESENTING WELL-FORMED PROGRAMS IN IDRIS

In this section we introduce and define a simple imperative language, Imp, in order to facilitate the definition of our semantics for CSL annotations and assertions. Accordingly, we define an equivalent version of CSL for Imp, denoted CSL_{Imp}. Imp can be considered a modified standard While language [26], adding arrays and restricting iteration. In principle, Imp can be extended with additional language constructs, such as tuples in order to allow the representation of a wider range of C programs in Imp. In the following sections, Idris code snippets have been simplified using mathematical notation where possible to aid clarity[1].

$n : \mathbb{N}$	::=	The natural numbers.
c : Type	::=	A pointed carrier type.
$x : \mathcal{X}$	::=	Numeric variable symbols.
$\alpha^n : \mathcal{Y}$	::=	Array variable symbols, indexed by length.
$i : \mathcal{I}$	::=	Index variable symbols.

Figure 1: Key to terms and notation.

3.1 Variable Representation

Variables are represented as three disjoint sets: $\mathcal{X}$ denotes the set of *numeric* variable symbols, $\mathcal{Y}$ denotes the set of *array* variable symbols; and $\mathcal{I}$ denotes the set of *index* variable symbols for accessing elements in arrays. This separation trivially ensures that array variables are not used where an arithmetic expression is expected, and vice versa. Index variable symbols are used to facilitate checking that array accesses are not out of bounds.

```
1  data VarKind = Numerical | Index | Array
2
3  data Var : (ty : VarKind) -> Type where
4    NumVar : Var Numerical
5    IdxVar : Var Index
6    AryVar : Var Array
7
8  data VarSet : (numvs : Vect n (Var Numerical))
9             -> (idxvs : Vect k (Var Index))
10            -> (aryvs : Vect m (Var Array)) -> Type where
11   MkVarSet : (numvs : Vect n (Var Numerical))
12           -> (idxvs : Vect k (Var Index))
13           -> (aryvs : Vect m (Var Array))
14           -> VarSet numvs idxvs aryvs
15
16 {- 𝒳 = Elem NumVar numvs -}
17 {- 𝒴 = Elem AryVar aryvs -}
18 {- ℐ = Elem IdxVar idxvs -}
```

In our Idris implementation, variables are represented using proofs of existence in a vector; e.g. $\mathcal{X}$ `= Elem NumVar numvs`. Here, `numvs` is a vector of `n` elements with type `Var Numerical`, where `Numeric` defines the set to which the variable belongs. Variables in $\mathcal{I}$ and $\mathcal{Y}$ are defined analogously. Each array variable, $\alpha^n : \mathcal{Y}$, is indexed by the length of the array that it represents, $n : \mathbb{N}$, where $n > 0$. Array sizes occur at the expression and statement level in our implementation. Index variables are used to access elements in arrays; they are mapped to natural numbers and may only be incremented. An array access expression, e.g. $\alpha^n[i]$, is deemed to be out-of-bounds when $i \geq n$.

Data definitions representing syntax are all indexed by the type `VarSet`, which is functionally equivalent to the triple $(\mathcal{X}, \mathcal{I}, \mathcal{Y})$, and allows variables to be used in Imp programs. In order to simplify our presentation, we use $x, x_1, x_2, \cdots : \mathcal{X}$, $i, i_1, i_2, \cdots : \mathcal{I}$, and $\alpha^n, \alpha_1^m, \alpha_2^k, \cdots : \mathcal{Y}$ to represent variables.

[1] Our full implementation can be found at https://github.com/adbarwell/IFL2019-Drive

Example 3.1 (Numeric Variables). The numeric variables, $x_1, x_2 : \mathcal{X}$, can be represented in our implementation via the definitions: x_1 `= Here` and x_2 `= There Here`, where `numvs = [NumVar,NumVar,...]`.

3.2 Numeric Values

Instead of defining Imp with a specific numeric data type, e.g. the natural numbers, as in [6], we aim to build a framework that allows for generic representations of numeric values. Taking inspiration from Slama and Brady [30], we index expressions and statements with a setoid, `(set : Setoid c (≃))` and an algebraic structure, `(Struct c set kind)`, defined on a carrier type, `(c : Type)`. Our implementation differs from Slama and Brady's in that we define both `Setoid` and algebraic structures as *data types* instead of interfaces in order to simplify their use as constraints in other data type declarations. Additionally, we extend our setoid definition with both a zeroth element, `zero`, that is used as a default value for array elements, and a boolean equivalence operator that is required to be equivalent to (≃); i.e. $x_1 \equiv x_2$ iff $x_1 \simeq x_2$ for all $x_1, x_2 : c$. Due to this requirement, we do not also require proofs of symmetry, transitivity, etc. for ≡.

```
1  data PropEq : (c : Type) -> ((≃) : c -> c -> Type) -> Type where
2    MkPropEq : ((≃)  : c -> c -> Type)
3            -> (refl  : (x : c) -> x ≃ x)
4            -> (sym   : {a,b : c} -> a ≃ b -> b ≃ a)
5            -> (trans : {w,y,z : c} -> w ≃ y -> y ≃ z -> w ≃ z)
6            -> (set_cong : (i,j : c) -> Dec (i ≃ j))
7            -> (cong_preserves_cong : (t,s,c1,c2 : c)
8                                   -> (t ≃ c1) -> (s ≃ c2)
9                                   -> (c1 ≃ c2) -> t ≃ s)
10           -> PropEq c (≃)
11
12 data DefnEq : (c : Type) -> Type where
13   MkDefnEq : ((≡)  : c -> c -> Bool) -> DefnEq c
14
15 data Setoid : (c : Type) -> ((≃) : c -> c -> Type) -> Type where
16   MkSetoid : (c : Type)
17            -> (zero : c)
18            -> (propEq : PropEq c (≃))
19            -> (defnEq : DefnEq c)
20            -> (cong_equiv_agree : (x,y : c) -> (x ≃ y)
21                                -> So ((π(≡) defneq) x y))
22            -> (equiv_cong_agree : (x,y : c)
23                                -> So ((π(≡) defneq) x y) -> x ≃ y)
24            -> Setoid c (≃)
```

We define Imp for six fundamental algebraic structures, `Magma`, `Semigroup`, `Monoid`, `Group`, `AlebelianGroup` and `Ring`, shown in the following listings. Algebraic structures are defined in the type `Struct`.

```
1  data StructKind : Type where
2    Magma : StructKind
3    Semigroup : StructKind
4    Monoid : StructKind
5    Group : StructKind
6    AbelianGroup : StructKind
7    Ring : StructKind
8
9  data Struct : (c : Type) -> (set : Setoid c (≃))
10             -> (kind : StructKind) -> Type where
11   Magma : (set : Setoid c (≃)) -> ((+) : BinOp c)
12        -> Struct c set Magma
13   ...
```

We define an ordering over these algebraic structures, where `Magma < Semigroup < Monoid < Group < AbelianGroup < Ring`. This ordering is used to ensure that we can only attempt to project functions that express a structure's requirements from structures that define them; e.g. the (additive) identity element from monoids or greater. Our implementation can be extended with additional structures, e.g. fields, or a total ordering over `c` that enables inequalities in boolean expressions.

```
1  data TotalOrder : (c : Type) -> (set : Setoid c (≃))
2                 -> Type where
3    MkTotalOrder : (lte : (i : c) -> (j : c) -> Type)
4                -> (is_lte : (i : c) -> (j : c) -> Dec (lte i j))
5                -> (def_lte : (i : c) -> (j : c) -> Bool)
6                -> ...
7                -> TotalOrder c set
```

`Magma` takes a proof that there is a `Setoid` on `c` and are equipped with a binary operation, `(+) : c -> c -> c`.

```
1  data Struct : (c : Type) -> (set : Setoid c (≃))
2             -> (kind : StructKind) -> Type where
3    ...
4    Semigroup : (magma : Struct c set Magma)
5              -> (assoc : Associativity c set (π(+) magma))
6              -> Struct c set Semigroup
7    ...
```

`Semigroup` (as shown in the listing above at Line 4) extends `Magma` with the requirement that (+) is associative. We define these requirements via separate data types; e.g. for associativity,

```
1  data Associativity : (c : Type) -> (set : Setoid c (≃)) -> (
       BinOp c set) -> Type where
2    MkAssociativity : ((x,y,z : c)
3                    -> ((x + y) + z) ≃ (x + (y + z)))
4                    -> Associativity c set (MkBinOp (+) p)
```

Such property types, have single constructors whose arguments express the relevant requirements. Here, associativity requires a function that calculates an explicit witness of type `((x + y) + z) ≃ (x + (y + z))`, for the given binary operation `(+)`, definition of (propositional) equality (≃), and all natural numbers, `x`, `y`, and `z`. We define $\pi_{(+)}$ `: Struct c set k -> BinOp c set`, used in the definition of `Semigroup`, *inter alia*, as a function that projects the binary operation from a `Magma`.

```
1  data Struct : (c : Type) -> (set : Setoid c (≃))
2             -> (kind : StructKind) -> Type where
3    ...
4    Monoid : (semigroup : Struct c set Semigroup)
5           -> (𝟎 : Identity c set (π(+) semigroup))
6           -> Struct c set Monoid
7    Group : (monoid : Struct c set Monoid)
8         -> (inv : Inverse c set (π(+) mnd) (π𝟎 mnd MndSgp))
9         -> Struct c set Group
10   AbelianGroup : (group : Struct c set Group)
11                -> (comm : Commutativity c set (π(+) group))
12                -> Struct c set AbelianGroup
13   ...
```

Similar to `Magma` and `Semigroup` definitions, a `Monoid` extends a given `Semigroup` with the requirement that `c` has an (additive) identity element, a `Group` extends a given `Monoid` with the requirement that there is a function `(-) : c -> c` that produces the (additive) inverse of its argument, and an `AbelianGroup` extends a given `Group` with the requirement that the `(+)` operation is commutative.

```
1  data Struct : (c : Type) -> (set : Setoid c (≃))
2             -> (kind : StructKind) -> Type where
3    ...
4    Ring : (abelgroup : Struct c set AbelianGroup)
5         -> ((×) : BinOp c)
6         -> (assoc× : Associativity c set (×))
7         -> (𝟏× : Identity c set (×))
8         -> (dist : Distributive c set (π(+) abelgroup) (×))
```

```
9         -> Struct c set Ring
10    ...
```

In order to illustrate that we are not limited to a single binary operation, we include `Ring`. A `Ring` extends a given `AbelianGroup` with a secondary binary operation over `c`, denoted $(\times)$, which is associative, has an identity element $(\mathbf{1}_\times)$, and distributes over $(+)$.

```
1   data Struct : (c : Type) -> (set : Setoid c (≃))
2              -> (kind : StructKind) -> Type where
3     ...
4     OrdSemigroup : (sgp : Struct c set Semigroup)
5                 -> (ord : TotalOrder c set)
6                 -> Struct c set OrdSemigroup
```

Similarly, we might extend any of the above structures with the total ordering over `c`. Here, we have extended `Semigroup`.

Example 3.2 (The Natural Numbers as a Semigroup). We can take advantage of the Idris Prelude definitions of propositional equality, addition, and lemmas over the natural numbers in order to define the `Semigroup` structure for natural numbers.

```
1   setoidNat : Setoid Nat (=)
2   setoidNat =
3     MkSetoid Nat (=) Refl symNat transNat set_eq eq_preserves_eq 0
4
5   semigroupNat : Struct Nat setoidNat Semigroup
6   semigroupNat =
7     Semigroup (Magma setoidNat
8                 (MkBinOp (Nat.plus) add_preserves_eq))
9               (MkAssociativity (\x, y, z => sym (Nat.
        plusAssociative x y z)))
```

Example 3.3 (The Natural Numbers as an Ordered Semigroup). We can further extend the definition in Example 3.2 with the Idris Prelude definitions of inequalities (`LTE`).

```
1   totalOrderNat : TotalOrder Nat NatTestSyntax.setoidNat
2   totalOrderNat = MkTotalOrder (LTE) (isLTE) lte ...
3
4   ordsgpNat : Struct Nat NatTestSyntax.setoidNat OrdSemigroup
5   ordsgpNat = OrdSemigroup semigroupNat totalOrderNat
```

3.3 Intrinsically Typed Syntax

Since we are only concerned with well-formed input, it makes sense to restrict the programs that are expressible in Imp as early as possible. Thus, we begin with syntax that is already type-safe; other aspects of well-formedness will be covered in Section 3.4.

3.3.1 Arithmetic Expressions. Arithmetic expressions comprise literal values, numeric variables, array accesses, and an addition operator for all fundamental algebraic structures that we consider. For `Group` and above, negation is available, representing inverses. Similarly, for `Ring`, a multiplication operator is available. Arithmetic expressions are defined in our formalism by the type `AExp`.

```
1   data AExp : (c : Type) -> (cnst : Struct c set kind)
2            -> (𝒳, ℐ, 𝒴) -> Type where
3     Val : (n : c) -> AExp c cnst vs
4     Var : (var : 𝒳) -> AExp c cnst vs
5     Acc : (var^len : 𝒴) -> (idx : ℐ) -> AExp c cnst vs
6     Neg : (a : AExp c cnst vs)
7        -> (GT kind Group) -> AExp c cnst vs
8     Add : (a1 : AExp c cnst vs) -> (a2 : AExp c cnst vs)
9        -> AExp c cnst vs
10    Mul : (a1 : AExp c cnst vs) -> (a2 : AExp c cnst vs)
11       -> (GT kind Ring) -> AExp c cnst vs
```

Here, `(SKOrd : StructKind -> StructKind -> Type)` is used to restrict the use of constructors `Neg` and `Mul` to when the appropriate algebraic structure is defined over `c`. We omit this restriction on `Add`, since the addition operator is defined as a requirement of `Magma`, the least element in our ordering. Array access expressions, `Acc`, require the array variable, `var`, the length of the array being accessed, `len`, and an index variable, `idx`. In our presentation, `(var : 𝒴) -> (len: Nat)`, $(\texttt{var}^{\texttt{len}} : \mathcal{Y})$, and $\alpha^n : \mathcal{Y}$ are all equivalent.

Example 3.4 (Arithmetic Expressions for Natural Numbers). Given the definitions for natural numbers as a `Semigroup` in Example 3.2, we can define the exemplar arithmetic expression

```
1   a : AExp Nat semigroupNat (𝒳, ℐ, 𝒴)
2   a = Add (Val 42) (Acc α₂⁵ i₁)
```

Here, we add a literal value to the element in the array α_2^5 at index i_1. Since a `Semigroup` is defined over `Nat`, any occurrences of `Neg` or `Mul` in an arithmetic expression will lead to a type error; therefore, such arithmetic expressions cannot be constructed. Any occurrences of a numeric variable or literal in an array access expression, e.g. `(Acc` x_1 `(Val 42))`, or an array variable outside of an array access, e.g. `(Add` α_2^5 `(Var 42))`, are similarly invalid.

3.3.2 Boolean Expressions. Boolean expressions comprise equality and inequality comparisons.

```
1   data BExp : (c : Type) -> (cnst : Struct c set kind)
2            -> (𝒳, ℐ, 𝒴) -> Type where
3     Eq : (a₁ : AExp c cnst vs) -> (a₂ : AExp c cnst vs)
4       -> BExp c cnst vs
5     LTE : (a1 : AExp c cnst vs) -> (a2 : AExp c cnst vs)
6        -> (ok : HasOrdering kind) -> BExp c cnst vs
```

Definitions for both equality and inequalities for a given `c` are provided by `cnst`. As with `Neg` and `Mul` above, `LTE` has an additional argument that requires demonstration that a total ordering is defined for the given algebraic structure.

Example 3.5 (Boolean Expressions for Natural Numbers). Given the definitions for natural numbers as a `Semigroup` in Example 3.2 and an ordered `Semigroup` in Example 3.3, we can define the exemplar Boolean expressions

```
1   b1 : BExp Nat semigroupNat (𝒳, ℐ, 𝒴)
2   b1 = Eq (Var x₂) (Val 5)
3
4   b2 : BExp Nat ordsgpNat (𝒳, ℐ, 𝒴)
5   b2 = LTE (Val 42) (Var x₂) OrdSemigroup
```

Here, we express $x_2 \simeq 5$ in `b1` and the inequality $42 \leq x_2$ in `b2`. The latter requires the proof, `OrdSemigroup`, that `ordsgpNat` is equipped with a total ordering.

3.3.3 Statements. Statements comprise numeric variable assignment, index variable assignment and increment, array declaration and update, for-loops, statement composition, and CSL assertions.

```
1   data Stmt : (c : Type) -> (cnst : Struct c set kind)
2            -> (𝒳, ℐ, 𝒴) -> Type where
3     Assn : (var : 𝒳) -> (a : AExp c cnst (MkVarSet ns is as fs))
4         -> Stmt c cnst vs
5     Idxd : (idx : ℐ) -> Stmt c cnst (MkVarSet ns is as fs)
6     Idxi : (idx : ℐ) -> Stmt c cnst (MkVarSet ns is as fs)
7     Aryd : (var^len : 𝒴)
8         -> (lenNZ : NotZero len)
9         -> Stmt c cnst vs
```

```
10    Aryu : (var^len : 𝒴)
11        -> (lenNZ : NotZero len)
12        -> (idx : ℐ)
13        -> (a : AExp c cnst vs)
14        -> Stmt c cnst vs
15    Iter : (var : 𝒳)
16        -> (var^len : 𝒴)
17        -> (lenNZ : NotZero len)
18        -> (s : Stmt c cnst vs)
19        -> Stmt c cnst vs
20    Comp : (s_1 : Stmt c cnst vs) -> (s_2 : Stmt c cnst vs)
21        -> Stmt c cnst vs
22    Cert : (b : BExp c cnst vs) -> Stmt c cnst vs
```

Unlike in arithmetic and Boolean expressions, all statements can be used with all algebraic structures. We assume that any capture annotations have been reified to numeric assignment expressions with literal values prior to the representation in Imp. This leaves assertions, denoted `Cert` in order to avoid confusion with numeric assignment statements, as the sole CSL construct that extends Imp. To simplify our presentation, we use Boolean expressions (Section 3.3.2) as our assertion language. In principle, our the assertion language could be different to Boolean expressions.

Example 3.6 (Statements for Natural Numbers). Given the definitions for natural numbers as a `Semigroup` in Example 3.2, we can define an exemplar statement.

```
1  stmt : Stmt Nat semigroupNat (𝒳, ℐ, 𝒴)
2  stmt =
3    Comp (Assn x₂ (Val 42))
4         (Comp (Cert (Eq (Var x₂) (Val 42)))
5               (Cert (Eq (Var x₂) (Val 5))))
```

Here, `stmt` is the composition of three statements: the assignment of x_2 to `42`, the assertion that x_2 is equal to `42`, and the assertion that x_2 is equal to `5`.

3.4 Well-Formedness

Since our syntax definitions in Section 3.3 ensure that programs in Imp are type-safe, our well-formedness property is principally concerned with avoiding the occurrence of undeclared variables and out-of-bounds array accesses. Determining well-formedness of arbitrary programs is a two-stage process for expressions and a four-stage process for statements. As a precursor to checking for out-of-bounds array accesses and occurrences of numeric and array variables that have not been declared, we first determine the validity of index variable occurrences and reify them according to a context. Reification produces an equivalent representation of the program with no index variable symbols. For statements, prior to reification, we first convert the statements into a continuation passing style and unroll for-loops. We are able to unroll all possible loops, since loops are defined to iterate over a (finite) fixed-length array.

3.4.1 Environment. An environment, `Env`, represents the variable symbols that are in scope at a given point in a program.

```
1  data Env : (𝒳, ℐ, 𝒴) -> Type where
2    MkEnv : (nums' : Vect n' 𝒳)
3         -> (idxs' : Vect k' ℐ)
4         -> (arys' : Vect m' 𝒴)
5         -> Env vs
```

In order to simplify our presentation, we use Γ to represent `Env`. While the terms *environment* and *context* are usually used interchangeably, in this paper, we will use *environment* to refer to the (sub)set(s) of variable symbols that are in scope, and *context* to refer to functions that map variable symbols to (ground) values.

Example 3.7 (Environment). We define an exemplar environment,

```
1  Γ : Env (𝒳, ℐ, 𝒴)
2  Γ = MkEnv [There Here]
3            [Here]
4            [(Here, (1 ** MkNotZero)),
5             (Here, (3 ** MkNotZero)),
6             (There Here, (5 ** MkNotZero))]
```

which states that one numeric variable, one index variable, and three array variables have been declared, given the sets of variable symbols $(\mathcal{X}, \mathcal{I}, \mathcal{Y})$. Array variable symbols are, in effect, a pair comprising an element of type `Elem AryVar as` and the length of the array. We consider two array variables to be distinct even when the element of type `Elem AryVar as` is the same but their defined lengths differ. Thus, in the above example, `(Here, (1 ** MkNotZero))` and `(Here, (3 ** MkNotZero))` are considered different variables. For convenience and clarity of presentation, we equivalently denote Γ as the triple:

$$\Gamma = (\{x_2\}, \{i_1\}, \{\alpha_1^1, \alpha_1^3, \alpha_2^5\})$$

3.4.2 Index Context. Index variables occur only in array access expressions and are incremented in statements. We define an index context, Υ, to be a function from index variable symbols to natural numbers. In our implementation, we represent this using the data type `IdxCtx`.

```
1  data IdxCtx : (env : Env vs) -> Type where
2    MkIdxCtx : {vs : VarSet nums idxs arys}
3             -> {idxs' : Vect k' (Elem IdxVar idxs)}
4             -> (ubs : Vect k' Nat)
5             -> IdxCtx (MkEnv nums' idxs' arys')
```

Here, `ubs` is a vector of natural numbers. When an index variable occurs in the body of a loop, this represents the maximum value it is assigned to. We consider that the j^{th} element of `idxs'` is mapped to the j^{th} element in `ubs`. In order to determine the j^{th} element in `ubs`, we define the function `arrIdxBounds` that transforms a proof that some index variable is an element in `idxs'` into a proof that `ub` is the corresponding element in `ubs`.

```
1  data ArrIdxBounds : (idxs' : Vect k' (Elem IdxVar idxs))
2                   -> (ubs : Vect k' Nat)
3                   -> (idxinenv : Elem idx idxs')
4                   -> (ub : Nat)
5                   -> Type where
6    Here : ArrIdxBounds (idx :: idxs') (ub :: ubs) Here ub
7    There : (laterArr : ArrIdxBounds idxs' ubs later ub)
8         -> ArrIdxBounds (i :: idxs') (b :: ubs) (There later) ub
9
10 arrIdxBounds : (idxs' : Vect k' (Elem IdxVar idxs))
11             -> (ubs : Vect k' Nat)
12             -> (idxinenv : Elem idx idxs')
13             -> (ub : Nat ** ArrIdxBounds idxs' ubs idxinenv ub)
```

In order to simplify our presentation, we use Υ to represent `IdxCtx`, where $\Upsilon(i) = k$ states that the index variable i is mapped to some k in the context Υ.

Example 3.8 (Index Context). We define an exemplar index context,

```
ϒ : IdxCtx Γ
ϒ = MkIdxCtx [4]
```

assuming the environment Γ from Example 3.7, which states that there is a single declared index variable, i_1. For convenience and clarity of presentation, we equivalently denote ϒ using substitution notation:

$$\Upsilon = \{i_1 \mapsto 4\}$$

3.4.3 *Arithmetic Expressions.*

Index Reification. Since index reification pertains only to array access expressions, the definition of index-reified algebraic expressions only differs from `AExp` in the `Acc` case.

```
data RAExp : (c : Type) -> (cnst : Struct c set k)
          -> (𝒳, ℐ, 𝒴) -> Type where
    ...
    Acc : (var : 𝒴) -> (len : ℕ) -> (idx : ℕ)
       -> RAExp c cnst vs
    ...
```

Additionally, we define `ReifyIdx` to relate an algebraic expression with its reified form. Again, `Acc` is the only interesting case, where `idxisdecld` is a proof that `idx` is in scope, and `arridxub` relates `idxisdecld` with its upper bound `ub`.

```
data ReifyIdx : (a : AExp c cnst vs) -> Γ -> ϒ
             -> (a_r : RAExp c cnst vs) -> Type where
  ...
  Acc : (idxisdecld : Elem i Γ)
     -> (arridxub : ArrIdxBounds Γ ϒ idxisdecld ub)
     -> ReifyIdx (Acc α^n i) Γ ϒ (Acc α^n ub)
  ...

reifyIdx : (a : AExp c cnst vs) -> Γ -> ϒ
         -> Dec (a_r : RAExp c cnst vs ** ReifyIdx a Γ ϒ a_r)
```

We define `reifyIdx` to transform an algebraic expression, `a` into an equivalent reified algebraic expression, a_r, given some environment and index context. This is a decision procedure since `idx` may not be in scope.

Example 3.9 (Index Reification of Arithmetic Expressions for $\mathbb{N}$). Recall the simple arithmetic expression from Example 3.4,

```
a : AExp Nat semigroupNat (𝒳, 𝒴)
a = Add (Val 42) (Acc α₂⁵ i₁)
```

The index i_1 in `a` can be reified given an Index Context, ϒ. Assuming ϒ as defined in Example 3.8, the index reification, a_r, of `a` is:

```
a_r : RAExp Nat semigroupNat (𝒳, ℐ, 𝒴)
a_r = Add (Val 42) (Acc α₂⁵ 4)
```

Well-Formedness of Reified Arithmetic Expressions. For arithmetic expressions, `Var` and `Acc` are the interesting cases.

```
data WFAExp : (a : RAExp c cnst vs) -> Γ -> Type where
  ...
  Var : (varinenv : Elem x Γ) -> WFAExp (Var x) Γ
  Acc : (lenNZ : NotZero len)
     -> (varinenv : Elem α^n Γ)
     -> (ubLTlen : ub < len)
     -> WFAExp (Acc α^n ub) Γ
  ...

isWFAExp : (a : RAExp c cnst vs) -> Γ -> Dec (WFAExp a Γ)
```

Here, `Var` requires a proof that x is in scope, and `Acc` similarly requires a proof that α^n is in scope, but also that $n > 0$ and that the index is not out-of-bounds (i.e. `ub < len`). Constructs that have subexpressions (i.e. `Neg`, `Add`, and `Mul`) are well-formed only when their subexpressions are well-formed. Literals (`Val`) are trivially well-formed. The definition of the decision procedure for well-formedness, `isWFAExp`, is unsurprising.

Example 3.10 (Well-Formedness of Arithmetic Expressions for $\mathbb{N}$). We can produce a proof that the reified algebraic expression a_r in Example 3.9 is well-formed under the environment defined in Example 3.7.

3.4.4 *Boolean Expressions.*

Both the index-reification and well-formedness of Boolean expressions are trivial. For both equality and inequality expressions, (arithmetic) sub-expressions are themselves reified and checked for well-formedness.

3.4.5 *Statements.*

Continuation Passing Style. In order to facilitate the inference of environments and contexts, statements are first transformed into an equivalent continuation-passing style.

```
data CPStmt : (c : Type) -> (cnst : Struct c set kind)
           -> (𝒳, ℐ, 𝒴) -> Type where
  Assn : (var : 𝒳)
      -> (a : AExp c cnst vs)
      -> (s_k : CPStmt c cnst vs)
      -> CPStmt c cnst vs
  ...
  Stop : CPStmt c cnst vs

(++) : (s1 : CPStmt c cnst vs) -> (s2 : CPStmt c cnst vs)
    -> CPStmt c cnst vs

data RelStmtCPS : (s : Stmt c cnst vs)
               -> (s_cps : CPStmt c cnst vs) -> Type where
  Assn : RelStmtCPS (Assn var a) (Assn var a Stop)
  ...
  Comp : (s1CPS : RelStmtCPS s1 s1_cps)
      -> (s2CPS : RelStmtCPS s2 s2_cps)
      -> RelStmtCPS (Comp s1 s2) (s1_cps ++ s2_cps)

relStmtCPS : (s : Stmt c cnst vs)
          -> (s_cps : CPStmt c cnst vs ** RelStmtCPS s s_cps)
```

Here, we define a new statement representation, `CPStmt`, and the type, `RelStmtCPS` that defines the relation between statements and CPS statements. Statement composition is the only interesting case: `s2` is appended to `s1`, where we define `(++)` to substitute the instance of `Stop` in `s1` for `s2` (occurrences of `Stop` in the body of for-statements are not substituted).

Loop Unrolling. Loops in CPS statements are then unrolled. We define a new statement representation, `LUStmt` that is the same as `CPStmt` but without loops.

```
data LUStmt : (c : Type) -> (cnst : Struct c set kind)
           -> (𝒳, ℐ, 𝒴) -> Type where

(++) : (s1 : LUStmt c cnst vs) -> (s2 : LUStmt c cnst vs)
    -> LUStmt c cnst vs

data UnrollCPStmt : (s : CPStmt c cnst (𝒳, ℐ, 𝒴)) -> Γ₀
                 -> (s_lu : LUStmt c cnst (𝒳, ℐ, 𝒴)) -> Γ_ω
                 -> Type where
  ...
  {- Γ₀' = (∅, ∅, A)-}
  {- Γ₀' = (∅, ∅, A ∪ {α^n})-}
```

```
    Aryd : (kLU : UnrollCPStmt s_k Γ₀′ k_lu Γ_ω)
        -> UnrollCPStmt (Aryd α^n lenNZ s_k) Γ₀
                        (Aryd α^n lenNZ k_lu) Γ_ω
    Iter : (aryinarys : Elem ary^len Γ)
        -> (sLU : UnrollCPStmt s Γ₀ s_lu Γ_κ)
        -> (kLU : UnrollCPStmt s_k Γ_κ k_lu Γ_ω)
        -> UnrollCPStmt (Iter var ary^len lenNZ s s_k) Γ₀
                        ((foldr (++) Stop (replicate len s_lu))
                         ++ k_lu) Γ_ω
    ...

unrollCPStmt : (s : CPStmt c cnst vs)
            -> (env : Env vs)
            -> Maybe (s_lu : LUStmt c cnst vs ** env_z : Env vs
       ** UnrollCPStmt s env s_lu env_z)
```

Similar to before, we additionally define a type that relates a CPS statement to its loop-unrolled equivalent. An environment is maintained between statements in order to know the length of the list being iterated over in the for-statement. Since we are only concerned with arrays, only array declarations update the environment. The other interesting case is `Iter`, which replaces the loop statement, e.g.

```
loop : Stmt Nat NatTestSyntax.sgpNat NatTestSyntax.testVs
loop =
  Comp (Aryd α² MkNotZero)
       (Iter x₁ i α² MkNotZero (Assn x₂ (Val 5)))
```

with a declaration of the index variable, $n-1$ repetitions of the loop body prepended with the assignment of the i^{th} element of α^2 to the given numeric variable and appended with an index increment statement, one final repetition of the loop body prepended with the assignment i^{th} element of α^2 but *without* the index increment, and finally the loop-unrolled continuation from the original loop. In terms of the above example, `loop` becomes

```
loop : LUStmt Nat semigroupNat (𝒳, ℐ, 𝒴)
loop =
  Aryd α² MkNotZero
  (Idxd i
  (Assn x₁ (Acc α² i)
  (Assn x₂ (Val 5)
  (Idxi i
  (Assn x₁ (Acc α² i)
  (Assn x₂ (Val 5)
  Stop))))))
```

We note that the covering function for `UnrollCPStmt` returns a `Maybe` instead of a `Dec` due to difficulties of the type checker to reduce the fold in proofs of contradiction in the `Iter` case. Since we are only interested in the proofs for assertions in well-formed programs, providing a proof of why a given program is not well-formed is not strictly necessary and will be left to future work.

Index Reification. Following loop-unrolling, index variables are reified. We define a new statement representation, `RStmt`, that is the same as `LUStmt` barring three changes: index declaration and increment constructors no longer take arguments, and all other occurrences of index variables are replaced with their values taken from some index-context.

```
data RStmt : (c : Type) -> (cnst : Struct c set kind)
          -> (𝒳, ℐ, 𝒴) -> Type where
  ...
  Idxd : (s_k : RStmt c cnst vs) -> RStmt c cnst vs
  Idxi : (s_k : RStmt c cnst vs) -> RStmt c cnst vs
  Aryu : (var^len : 𝒴)
      -> (lenNZ : NotZero len)
      -> (idx : Nat)
      -> (a : RAExp c cnst vs)
      -> (s_k : RStmt c cnst vs)
      -> RStmt c cnst vs
  ...

data ReifyIdxStmt : (s : LUStmt c cnst vs) -> Γ -> ϒ
                 -> (s_r : RStmt c cnst vs)
                 -> Type where
  ...
  Idxd : (kRI : ReifyIdxStmt s_k (Γ ∪ {i}) (ϒ ⊔ {i ↦ 0}) k_r
      -> ReifyIdxStmt (Idxd i s_k) Γ (MkIdxCtx ubs) (Idxd k_r)
  Idxi : (idxinenv : Elem i Γ)
      -> (arr : ArrIdxBounds Γ ϒ idxinenv ub)
      -> (kRI : ReifyIdxStmt s_k (Γ ∪ {i}) (ϒ ⊔ {i ↦ ub + 1}) k_r)
      -> ReifyIdxStmt (Idxi i s_k) Γ ϒ (Idxi k_r)
  Aryu : (aRI : ReifyIdxAExp a Γ ϒ a_r)
      -> (idxinenv : Elem i Γ)
      -> (arr : ArrIdxBounds Γ ϒ idxinenv ub)
      -> (kRI : ReifyIdxStmt s_k Γ ϒ k_r)
      -> ReifyIdxStmt (Aryu var^len lenNZ i a s_k) Γ ϒ
                      (Aryu var^len lenNZ ub a_r k_r)
  ...

reifyIdxStmt : (s : LUStmt c cnst vs) -> Γ -> ϒ
            -> Maybe (s_r : RStmt c cnst vs
                     ** ReifyIdxStmt s Γ ϒ s_r)
```

Here, `ReifyIdxStmt` relates a loop-unrolled statement with an equivalent index-reified statement. The environment, Γ, is only updated when an index variable is declared. Upon declaration of an index variable, the index context is also updated such that it maps the declared index variable, i, to 0. Should i already be in Γ, the new mapping replaces the old. In the above code, this is represented by $\sqcup$; i.e. $\Upsilon \sqcup \{i_j \mapsto m\} = \Upsilon \setminus \{i_j \mapsto 0, i_j \mapsto 1, i_j \mapsto 2, \ldots\} \cup \{i_j \mapsto m\}$. Index increment statements update the context analogously. In our implementation, we do not remove old instances in either the environment or context, but instead rely upon the definition of `isElem` returning the *first* occurrence of i in Γ. Future work will address this reliance upon implementation idiosyncrasies.

As in Section 3.4.5, the covering function, `reifyIdxStmt`, returns a `Maybe` value instead of a `Dec` due to impossible contradiction proof obligations. This is a consequence of the result that for any two proofs of vector membership, p_1, p_2 `: Elem x xs`, it does not hold that $p_1 = p_2$, for a given `x` and `xs`. Future work will address this.

Well-Formedness. The index-reification of statements enables the definition of a type expressing well-formedness; i.e. all variables are assigned/declared prior to occurrences in statements/subexpressions and that array accesses are never out-of-bounds.

```
data WFRStmt : (s : RStmt c cnst vs) -> Γ -> Type where
  Assn : (a_wf : WFRAExp a Γ)
      -> (k_wf : WFRStmt s_k (Γ ∪ {x}))
      -> WFRStmt (Assn x a s_k) Γ
  Aryd : (k_wf : WFRStmt s_k (Γ ∪ {α^n}))
      -> WFRStmt (Aryd α^n lenNZ s_k) Γ
  Aryu : (varinarys : Elem α^n Γ)
      -> (idxLTlen : LT idx len)
      -> (a_wf : WFRAExp a Γ)
      -> (k_wf : WFRStmt s_k Γ)
      -> WFRStmt (Aryu α^n lenNZ idx a s_k) Γ
  ...

isWFRStmt : (s : RStmt c cnst vs) -> (env : Env vs) -> Dec (
      WFRStmt s env)
```

Here, the interesting cases are: numeric variable assignment and array declaration, which adds the relevant variable to the environment; and array update, which requires that the array being accessed is in the environment and that the index is strictly less than the length of the array (arrays are considered to be zero-indexed).

3.5 Operational Semantics

We define a big-step operational semantics for well-formed programs in Imp. Since (finite) well-formed programs are intended to be both guaranteed to be ground and terminate, no error states are produced during evaluation/execution.

3.5.1 Contexts. Similarly to index contexts in Section 3.4.2, we define a context, Ψ, for numeric and array variables.

```
data Ctx : (c : Type) -> (env : Env vs) -> Type where
  MkCtx : {nums : Vect n (Elem NumVar ns)}
       -> {arys : Vect m (Elem AryVar as, (len ** NotZero len))}
       -> (nvals : Vect n c)
       -> (avals : Vect m (s ** Vect s c))
       -> (ok : StructSame arys avals)
       -> Ctx c (MkEnv nums idxs arys)
```

Here, a context takes a vector of numeric values, `nvals`, and a vector of array values, `avals`. Both vectors have the same length as the vector of numeric and array variables that are in the environment. The i^{th} element in `nvals` and `avals` are mapped to the i^{th} element in `nums` and `arys`, respectively. Each element in `avals` is itself a vector of numeric values. The proof term, `(ok : StructSame arys avals)`, requires that each vector of numeric values in `avals` is the same length as the declared array in `arys`. Upon declaration, each element of an array is set to the `zero` value defined in the given setoid over `c`.

Example 3.11 (Context). We might define an example context for the environment defined in Example 3.7, which has a single numeric variable, x_2, and three arrays, α_1^1, α_2^3, and α_2^5, in scope.

```
testAVals : Vect 3 (l : Nat ** Vect l Nat)
testAVals = [(1 ** [1]),(3 ** [2,3,4]),(5 ** [5,6,7,8,9])]

testAValsStructSame : StructSame {as=[AryVar, AryVar]} [(Here,
      (1 ** MkNotZero)), (Here, (3 ** MkNotZero)), (There Here,
      (5 ** MkNotZero))] NatTestEnv.testAVals
testAValsStructSame = Cons Refl (Cons Refl (Cons Refl Nil))

testCtx : Ctx Nat NatTestEnv.testEnv
testCtx = MkCtx [5] testAVals testAValsStructSame
```

Here, $x_2 = 5$; $\alpha_1^1 = [1]$; $\alpha_2^3 = [2,3,4]$; and $\alpha_2^5 = [5,6,7,8,9]$. The variable `testAValsStructSame` provides the proof that the list of arrays, `testAVals` is a vector of length three, with nested vectors of lengths 1, 3, and 5. We equivalently represent `testCtx` as the set of substitutions

$$\Psi = \{x_2 \mapsto 5, \alpha_1^1 \mapsto [1], \alpha_2^3 \mapsto [2,3,4], \alpha_2^5 \mapsto [5,6,7,8,9]\}$$

3.5.2 Arithmetic Expressions. We define a big step operational semantics for algebraic expression via the type, `SRAExp`, which relates a given reified arithmetic expression to its value in `c`, given some context Ψ.

```
data SRAExp : (cnst: Struct c set kind)
           -> (a : RAExp c cnst vs) -> Γ
           -> (a_wf : WFRAExp a env) -> Ψ
           -> (val : c)
           -> Type where
  Val : SRAExp cnst (Val n) Γ Val Ψ n
  Var : (arr : ArrVarC Γ Ψ varinenv val)
     -> SRAExp cnst (Var var) Γ (Var varinenv) Ψ val
  Add : (a1_s : SRAExp cnst a1 Γ a1_wf Ψ a1_val)
     -> (a2_s : SRAExp cnst a2 Γ a2_wf Ψ a2_val)
     -> SRAExp cnst (Add a1 a2) Γ (Add a1_wf a2_wf) Ψ (π(+)
      a1_val a2_val)
  ...

sraexp : (cnst : Struct c set kind)
      -> (a : RAExp c cnst vs) -> Γ
      -> (a_wf : WFRAExp a env) -> Ψ
      -> (val : c ** SRAExp cnst a env a_wf ctx val)
```

Literal values evaluate to themselves; variables and array accesses are related to a value in the context by `ArrVarC` and `ArrVarIdxC`, respectively; and addition, negation, and multiplication are evaluated using the respective functions defined in the given algebraic structure.

3.5.3 Boolean Expressions. The semantics for boolean expressions are defined analogously to arithmetic expressions above. Numeric sub-expressions are evaluated, and the functions provided for definitional equality and inequalities are projected from the given algebraic structure.

3.5.4 Statements. We define the semantics of statements via the type, `SRStmt`. Unlike expressions, the result of executing the statements is the context, Ψ. Specifically, the final state of a given statement is the context of the final continuation; i.e. the context provided to a `Stop` statement.

```
data SRStmt : (cnst: Struct c set kind)
           -> (s : RStmt c cnst vs) -> Γ
           -> (s_wf : WFRStmt s env) -> Ψ
           -> Type where
  Assn : (a_s : SRAExp cnst a Γ a_wf Ψ a_val)
      -> (k_s : SRStmt cnst s_k (Γ ∪ x) k_wf (Ψ ⊔ {x ↦ a_val}))
      -> SRStmt cnst (Assn x a s_k) Γ (Assn a_wf k_wf) Ψ
  Aryu : (a_s : SRAExp cnst a Γ a_wf Ψ a_val)
      -> (arr : AryCtxUpdateAssn cnst Γ Ψ α^n lenNZ varinarys
                                 (natToFin idx n idxLTlen)
                                 a_var Ψ')
      -> (k_s : SRStmt cnst s_k Γ k_wf Ψ')
      -> SRStmt cnst (Aryu α^n lenNZ idx a s_k) Γ
                (Aryu varinarys idxLTlen a_wf k_wf) Ψ
  Cert : (b_s : SRBExp cnst b Γ b_wf Ψ val)
      -> (k_s : SRStmt cnst s_k Γ k_wf Ψ)
      -> SRStmt cnst (Cert b s_k) Γ (Cert b_wf k_wf) Ψ
  ...

srstmt : (cnst: Struct c set kind)
      -> (s : RStmt c cnst vs) -> Γ
      -> (s_wf : WFRStmt s env) -> Ψ
      -> SRStmt cnst s Γ s_wf Ψ
```

Here, the interesting cases are numeric variable assignment, array update, and CSL_{IMP} assertions. Assignments require the evaluation of the arithmetic expression, `a`, the result of which is used to update the context. As in Section 3.5.1, $\sqcup$ denotes the addition or replacement of a mapping in a given context. In our implementation, as before, we simply prepend `a_val` to the list of numeric values in the context. Array updates are similar, requiring the evaluation of the arithmetic expression, which is then used to update the relevant value in the context. Here, the update is defined via the type, `AryCtxUpdateAssn`. This relates the current context with a new context such that only the element at index `idx` of the array assigned to α^n is replaced by `a_val`. Array declarations are defined analogously, but such that a new vector of `zero` values are added

to the context. Finally, we define the semantics of CSL_{IMP} assertions to be equivalent to a typical `skip` statement. Assertions do not affect the execution of the program, but are instead used by the automatic proof inference system in order to determine whether the assertion holds. The argument, `b_s`, represents the evaluation of the boolean expression, and is used in order to determine the values of arithmetic sub-expressions when generating proofs for assertions.

4 AUTOMATICALLY PROVING ASSERTIONS

In this section, we define the type, `VCRStmt`, that extends assertion statements from CSL (see Section 2.1 and Listing 1 for an example of CSL assertions in C) with proofs (of contradition) that demonstrate that the assertion holds/does not hold true for the current context.

```
data VCRStmt : (cnst : Struct c set kind)
             -> (s : RStmt c cnst vs) -> Γ
             -> (s_wf : WFRStmt s env) -> Ψ
             -> (s_s : SRStmt cnst s env s_wf ctx)
             -> Type where
  ...
  Cert : (b_prf : Dec (VCRBExp cnst b Γ b_wf Ψ b_s))
      -> (k_vc : VCRStmt cnst s_k Γ k_wf Ψ k_s)
      -> VCRStmt cnst (Cert b s_k) Γ (Cert b_wf k_wf) Ψ
                  (Cert k_s)
```

Naturally, `Cert` is the only interesting case. It takes an argument that represents the proof of the (reified) Boolean expression that comprises the assertion for the environment and context at that particular statement. The environment and context are derived from, and defined by, the proofs of well-formedness and semantics that `VCRStmt` is indexed by. The proof itself is comprised of the type, `VCRBexp`, which represents only (propositionally) *true* Boolean expressions under the given context. Accordingly, and following convention, a proof that the assertion *does not* hold is represented by a function with the type `VCRBExp cnst b Γ b_wf Ψ b_s -> Void`.

```
data VCRBExp : (cnst : Struct c set kind)
             -> (b : RBExp c cnst vs) -> Γ
             -> (b_wf : WFRBExp b env) -> Ψ
             -> (b_s : SRBExp cnst b Γ b_wf Ψ val)
             -> Type where
  Eq : {set : Setoid c (≃)}
    -> {cnst : Struct c set kind}
    -> (prf : a1_val (≃) a2_val)
    -> {a1_s : SRAExp cnst a1 Γ a1_wf Ψ a1_val}
    -> {a2_s : SRAExp cnst a2 Γ a2_wf Ψ a2_val}
    -> OrdVCRBExp cnst (Eq a1 a2) Γ (Eq a1_wf a2_wf) Ψ
                  (Eq a1_s a2_s)
  LTE : {lte : c -> c -> Type}
     -> (prf : lte a1_val a2_val)
     -> {a1_s : SRAExp (OrdSemigroup ...)
                       a1 Γ a1_wf Ψ a1_val}
     -> {a2_s : SRAExp (OrdSemigroup ...)
                       a2 env a2_wf ctx a2_val}
     -> OrdVCRBExp (OrdSemigroup sgp
                                 (MkTotalOrder lte is_lte ...))
                   (LTE a1 a2 OrdSemigroup) Γ
                   (LTE a1_wf a2_wf) Ψ
                   (LTE a1_s a2_s)
```

Here, both equality and inequality cases take the respective type definition from the given algebraic structure that represents the boolean operation. Each case then require an element of that type applied to the evaluated arithmetic sub-expressions. `VCRBExp` can be extended according to the defined algebraic structures and operations. The decision procedure for `VCRBExp` is straightforward.

```
isVCRBExp : (cnst : Struct c set kind)
          -> (b : RBExp c cnst vs)
          -> Γ
          -> (b_wf : WFRBExp b Γ)
          -> Ψ
          -> (b_s : SRBExp cnst b Γ b_wf Ψ val)
          -> Dec (VCRBExp cnst b Γ b_wf Ψ b_s)
isVCRBExp {set=(MkSetoid ... is_cong ...)} cnst (Eq a1 a2) Γ
           (Eq a1_wf a2_wf) Ψ (Eq {a1_val} {a2_val} a1_s a2_s) =
  case is_cong a1_val a2_val of
    Yes prf => Yes (Eq prf)
    No contra => No (\(Eq prf) => contra prf)
...
```

Here, the relevant decision procedure is projected from the given algebraic structure and is used to produce a proof, which is then used as an argument to the concomitant `VCRStmt` constructor.

5 DEMONSTRATION

In order to illustrate our approach, we consider list summation, an example of summing an array of natural numbers to demonstrate the principles of our technique, where we manually apply each step; it is intended that this process will be fully automatic in the future. In C, we might have the summation function, `sumList`, that takes

```
1  int sumList(const int *xs) {
2    int sum = 0;
3    int loop_energy;
4    __csl_energy(&loop_energy);
5    for (i = 0; i < len(xs); i++){
6      sum = sum + xs[i];
7    }
8    __csl_assert(sum == 15);
9    __csl_assert(loop_energy < 10);
10   return sum;
11 }
```

an array as a parameter and returns the sum of its elements. Here, `sumList` has been annotated with a CSL capture annotation on Line 4 to measure, e.g., the estimated energy consumption of the for-statement on Lines 5–7. CSL assertions have also been introduced on Lines 8 & 9; the first represents a check that the result of the loop is correct, and the second represents that the cost of the loop is within a given upper-bound, *i.e. does not exceed 10 joules of energy*. The body of `sumList` can be represented in IMP for a specific value of `xs` and where we assume, for the sake of this example, that all values in `xs` are zero or greater (i.e. are natural numbers).

```
loopBody : Stmt Nat ordsgpNat (𝒳,ℐ,𝒴)
loopBody = Assn Here (Add (Var x₁) (Var x₂))

sumListNatOSg : Stmt Nat ordsgpNat (𝒳,ℐ,𝒴)
sumListNatOSg =
  Comp (Assn x₁ (Val 0))
  (Comp (Assn x₃ (Val 7))
  (Comp (Iter x₂ i₁ α₁⁵ MkNotZero loopBody)
  (Comp (Cert (Eq (Var x₁) (Val 15)))
  (Cert (LTE (Var x₃) (Val 10) OrdSemigroup)))))
```

We use the `OrdSemigroup` definition for the natural numbers from Example 3.3 and assume the existence of the array `[1,2,3,4,5]`, whose declaration and assignment statements are omitted, in the environment and context. Here, x_1 is the accumulator, x_2 is assigned to each element in the array over the course of all iterations of the loop, and x_3 represents the result of the CSL capture annotation in

Line 4 of the `sumList` definition. The assertion on Line 9 represents a functional check that the result of the summation is the expected value, and the assertion on Line 10 represents a check to ensure that the result of the capture annotation is within a certain bound. In order to generate proofs of these assertions, we first ensure that `sumListNatOSg` is well-formed. This process begins with a transformation into continuation-passing style by the `relStmtCPS` covering function.

```
1  s_cps : CPStmt Nat ordsgpNat (𝒳,ℐ,𝒴)
2  s_cps =
3    (Assn x₁ (Val 0)
4    (Assn x₃ (Val 7)
5    (Iter x₂ i₁ α₁⁵ MkNotZero
6      (Assn x₁ (Add (Var x₁) (Var x₂)) Stop)
7    (Cert (Eq (Var x₁) (Val 15)) (Cert (LTE (Var x₃)) (Val 10)
       OrdSemigroup) Stop))))
```

This is followed by loop-unrolling via the `unrollCPStmt`, which replicates the body of the loop five times.

```
1  s_lu : LUStmt Nat ordsgpNat (𝒳,ℐ,𝒴)
2  s_lu =
3    (Assn x₁ (Val 0)
4    (Assn x₃ (Val 7)
5    (Idxd i₁
6    (Assn x₂ (Acc α₁⁵ i₁)
7    (Assn x₁ (Add (Var x₁) (Var x₂))
8    (Idxi i₁
9    (Assn x₂ (Acc α₁⁵ i₁)
10   (Assn x₁ (Add (Var x₁) (Var x₂))
11   (Idxi i₁
12   ...
13   (Assn x₁ (Add (Var x₁) (Var x₂))
14   (Cert (Eq (Var x₁) (Val 15))
15   (Cert (LTE (Var x₃) (Val 10) OrdSemigroup) Stop)...)
```

In order to maintain functional equivalence, the unrolled loop is preceded by a declaration of i_1 (Line 5), each repetition of the loop body is preceded by a statement assigning x_2 to the i^{th} element of α_1^5 (Lines 6 & 9), and finally, each repetition, excepting the last, of the loop body is followed by the incrementation of i_1 (Lines 8 & 11). Having unrolled the loop, it is now possible to reify `s_lu` using `reifyIdxStmt`.

```
1  s_r : RStmt Nat ordsgpNat (𝒳,ℐ,𝒴)
2  s_r =
3    (Assn x₁ (Val 0)
4    (Assn x₃ (Val 7)
5    (Idxd
6    (Assn x₂ (Acc α₁⁵ 0)
7    (Assn x₁ (Add (Var x₁) (Var x₂))
8    (Idxi
9    (Assn x₂ (Acc α₁⁵ 1)
10   (Assn x₁ (Add (Var x₁) (Var x₂))
11   (Idxi
12   ...
13   (Assn x₁ (Add (Var x₁) (Var x₂))
14   (Cert (Eq (Var x₁) (Val 15))
15   (Cert (LTE (Var x₃) (Val 10) OrdSemigroup) Stop)...)
```

Here, we observe that the occurrences of i_1 in Lines 6 & 9 in `s_lu` have been replaced with their relevant natural numbers in Lines 6 & 9 of `s_r`. We now determine the final stage of well-formedness by applying `isWFRStmt`.

```
1  sumListEnv : Env (𝒳,ℐ,𝒴)
2  sumListEnv = MkEnv [] [] [(Here, (5 ** MkNotZero))]
3
4  s_wf : WFRStmt s_r sumListEnv
5  s_wf =
6    (Assn Val (Assn Val (Idxd
7    (Assn (Acc MkNotZero Here (LTESucc LTEZero))
8    (Assn (Add (Var (There (There Here))) (Var Here))
9    (Idxi
10   (Assn (Acc MkNotZero Here (LTESucc (LTESucc LTEZero)))
11   ...
12   (Cert (Eq (Var Here) Val)
13   (Cert (LTE (Var (There ... (There Here))))))))))) Val)
14   Stop)...)
```

Since we assume that α_1^5 is already defined and its elements assigned, we specify the initial environment `sumListEnv`, which only contains an array of length 5. `s_wf` is then a proof that all variables are in the environment before they occur in assignment statements or subexpressions (e.g. on Lines 7–8, 10, & 12–13), and that array accesses are not out-of-bounds (e.g. on Lines 10 & 13). Given this proof that `s_r` is well-formed, and in order to determine whether the assertions hold, it is necessary to first derive a context for those assertions. We therefore apply `srstmt` to both `s_r` and `s_wf`. We note that this is equivalent to executing the program and using the values of x_1 and x_3 at the point of each assertion.

```
1  sumListCtx : Ctx Nat testListEnv
2  sumListCtx = MkCtx [] [(5 ** [1, 2, 3, 4, 5])] (Cons Refl [])
3
4  s_s : SRStmt ordsgpNat s_r sumListEnv s_wf sumListCtx
5  s_s =
6    Assn Val (Assn Val (Idxd (Assn (Acc Here) ... Stop)...)
7
8  Ψ_ω = MkCtx [15, 5, 10, 4, 6, 3, 3, 2, 1, 1, 7, 0]
9                        [(5 ** [1, 2, 3, 4, 5])]
10                       (Cons Refl []))
```

As before, we provide an initial context with our array already included. The result of `srstmt` is a witness to our semantics, `s_s`, and the inferred contexts at each continuation, where Ψ_ω is the final context. In Ψ_ω, we observe that the values for the arrays have not changed, which is expected. Excepting the final two elements of the vector in Line 8, which represents the values of numeric variables, each pair of elements (i.e. 15 & 5, 10 & 4, etc.) represent updated values of x_1 and x_2 respectively. We can now generate the proofs for both assertion statements by applying `vcrstmt` to `s_r`, `s_wf`, and `s_s`.

```
1  s_vc : VCRStmt ordsgpNat s_r sumListEnv s_wf sumListCtx s_s
2  s_vc =
3    Assn
4    ...
5    (Cert (Yes (Eq Refl))
6    (Cert (Yes (LTE (LTESucc ... (LTESucc LTEZero)...) Stop)...)
```

Here, `s_vc` represents the extension of assertions in `s_r` given the context at that point the program. Line 5 contains the proof that $x_1 \simeq 15$ holds true, where $x_1 = 1 + 2 + 3 + 4 + 5$. Similarly, Line 6 contains the proof that the cost value obtained from the capture annotation is less than the programmer-provided upper bound; i.e. $x_3 \leq 10$, where $x_3 = 7$.

6 RELATED WORK

In addition to the aforementioned related work throughout the paper, the calculation and bounding of resource usage is a topic of great interest in the programming language community, with approaches typically focussed on time, space, and type systems [8, 9, 15, 17, 18, 21, 32]. Other non-functional properties, such as information flow and leakage, have also been modelled using type

systems [7, 33]. Energy consumption, which is of increasing interest to embedded systems programming [24], has also been represented as a function of program arguments [13, 22]. These approaches typically depend upon specific type systems or languages; accordingly, applying them to other languages can prove non-trivial [6]. Abstract interpretation offers an alternative approach to the verification and debugging of programs in languages that may not necessarily be best equipped for the desired techniques [23]. Examples include debugging of both imperative and logical programs [3, 4], and approaches to verification by Cousot [10]. More recently, The Ciao Preprocessor system (CiaoPP) [16, 23, 29] models Java [14] and XC [1] programs as sequences of Horn Clauses in order to debug and certify programs, using resource usage information that the system derives. A high-level comparison between CiaoPP and *Drive* is given by Brown *et al.* [6].

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented the small generic imperative language, Imp, representing a subset of C with CSL assertions. Using the dependently-typed language, Idris, we defined both the syntax and a big-step operational semantics for Imp. Imp is parameterised by a pointed carrier type and an algebraic structure, enabling a generic and formally-based framework for the expression of arithmetic and Boolean operations. Our semantics for Imp facilitate a robust context inference that, in turn, facilitates the automatic generation of proofs for CSL assertions. We demonstrate our approach on a representative example of summing an array, featuring both CSL assertions and capture annotations.

In the future, we will expand our evaluation to include the entirety of the BEEBs benchmark suite, demonstrating a range of non-functional properties, including both *energy* and *time*. Furthermore, we will extend our non-functional properties to include security, allowing our formalism to guide the programmer in preventing common security hacks, such as side-channel attacks. Finally, we will prove properties of both our semantics and abstract interpretation, including soundness, determinism, and confluence, in order to improve confidence in our approach.

ACKNOWLEDGEMENTS

This work was generously supported by the EU Horizon 2020 project, *TeamPlay* (https://www.teamplay-h2020.eu), grant number 779882, and UK EPSRC *Discovery*, grant number EP/P020631/1.

REFERENCES

[1] 2011. XC Specification ver. 1.0 (X5965A). (2011). https://www.xmos.com/developer/xc-specification.
[2] Elvira Albert, Puri Arenas, Germán Puebla, and Manuel V. Hermenegildo. 2012. Certificate size reduction in abstraction-carrying code. *TPLP* 12, 3 (2012), 283–318.
[3] Giovanni Bacci and Marco Comini. 2010. Abstract Diagnosis of First Order Functional Logic Programs. In *LOPSTR (Lecture Notes in Computer Science)*, Vol. 6564. Springer, 215–233.
[4] François Bourdoncle. 1993. Abstract Debugging of Higher-Order Imperative Languages. In *PLDI*. ACM, 46–55.
[5] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* 23, 5 (2013), 552–593.
[6] Christopher Brown, Adam D. Barwell, Yoann Marquer, Céline Minh, and Olivier Zendra. 2019. Type-Driven Verification of Non-functional Properties. In *PPDP*. Accepted for publication.
[7] Hongxu Chen, Alwen Tiu, Zhiwu Xu, and Yang Liu. 2018. A Permission-Dependent Type System for Secure Information Flow Analysis. In *CSF*. IEEE Computer Society, 218–232.
[8] Wei-Ngan Chin and Siau-Cheng Khoo. 2001. Calculating Sized Types. *Higher-Order and Symbolic Computation* 14, 2-3 (2001), 261–300.
[9] Ezgi Çiçek, Deepak Garg, and Umut A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *ESOP (Lecture Notes in Computer Science)*, Vol. 9032. Springer, 406–431.
[10] Patrick Cousot. 2003. Automatic Verification by Abstract Interpretation. In *VMCAI (Lecture Notes in Computer Science)*, Vol. 2575. Springer, 20–24.
[11] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.
[12] Heiko Falk and Paul Lokuciejewski. 2010. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems* 46, 2 (01 Oct 2010), 251–300. https://doi.org/10.1007/s11241-010-9101-x
[13] Kyriakos Georgiou, Steve Kerrison, Zbigniew Chamski, and Kerstin Eder. 2017. Energy Transparency for Deeply Embedded Programs. *TACO* 14, 1 (2017), 8:1–8:26.
[14] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.
[15] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*. ACM, 127–139.
[16] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Program.* 58, 1-2 (2005), 115–140.
[17] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 14:1–14:62.
[18] John Hughes and Lars Pareto. 1999. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *ICFP*. ACM, 70–81.
[19] Graham Hutton. 2007. *Programming in Haskell*. Cambridge University Press, New York, NY, USA.
[20] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO (Lecture Notes in Computer Science)*, Vol. 1109. Springer, 104–113.
[21] Ugo Dal Lago and Barbara Petit. 2013. The geometry of types. In *POPL*. ACM, 167–178.
[22] Umer Liqat, Steve Kerrison, Alejandro Serrano, Kyriakos Georgiou, Pedro López-García, Neville Grech, Manuel V. Hermenegildo, and Kerstin Eder. 2013. Energy Consumption Analysis of Programs Based on XMOS ISA-Level Models. In *LOPSTR (Lecture Notes in Computer Science)*, Vol. 8901. Springer, 72–90.
[23] Pedro López-García, Luthfi Darmawan, Maximiliano Klemen, Umer Liqat, Francisco Bueno, and Manuel V. Hermenegildo. 2018. Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. *TPLP* 18, 2 (2018), 167–223.
[24] Jeremy Morse, Steven Kerrison, and Kerstin Eder. 2018. On the limitations of analysing worst-case dynamic energy of processing. *ACM Transactions on Embedded Computing Systems* 17, 3 (2 2018). https://doi.org/10.1145/3173042
[25] George C. Necula. 1997. Proof-Carrying Code. In *POPL*. ACM Press, 106–119.
[26] Hanne Riis Nielson and Flemming Nielson. 2007. *Semantics with Applications: An Appetizer*. Springer.
[27] James Pallister, Simon J. Hollis, and Jeremy Bennett. 2013. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. *CoRR* abs/1308.5174 (2013). arXiv:1308.5174 http://arxiv.org/abs/1308.5174
[28] Christopher Schwaab, Ekaterina Komendantskaya, Alasdair Hill, Frantisek Farka, Ronald P. A. Petrick, Joe B. Wells, and Kevin Hammond. 2019. Proof-Carrying Plans. In *PADL (Lecture Notes in Computer Science)*, Vol. 11372. Springer, 204–220.
[29] Alejandro Serrano, Pedro López-García, and Manuel V. Hermenegildo. 2014. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *TPLP* 14, 4-5 (2014), 739–754.
[30] Franck Slama and Edwin Brady. 2017. Automatically Proving Equivalence by Type-Safe Reflection. In *CICM (Lecture Notes in Computer Science)*, Vol. 10383. Springer, 40–55.
[31] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.
[32] Pedro B. Vasconcelos and Kevin Hammond. 2003. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL (Lecture Notes in Computer Science)*, Vol. 3145. Springer, 86–101.
[33] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2/3 (1996), 167–188.