Verified Technology Mapping in an Agda DSL for Circuit Design

Circuit refinement through gate and data concretisation

João Paulo Pizani Flor
Universiteit Utrecht
Utrecht, The Netherlands
paulopizani@posteo.net

Wouter Swierstra
Universiteit Utrecht
Utrecht, The Netherlands
W.S.Swierstra@uu.nl

ABSTRACT

The use of mechanized proofs for verification of programming language metatheory is a well-established field of study, as is the application of analogous results to the design of digital circuits. Our interest resides in the use of dependent types to formalize and verify circuit transformations. In this specific paper we focus on the technology mapping step of the circuit design flow, which can be seen as a well-typed substitution of syntax for (primitive) semantics. We formalize the technology mapping refinement and show that it indeed preserves state-transition semantics, since it is compositional.

ACM Reference Format:
João Paulo Pizani Flor and Wouter Swierstra. 2022. Verified Technology Mapping in an Agda DSL for Circuit Design: Circuit refinement through gate and data concretisation. In *Proceedings of IFL2022.* ACM, New York, NY, USA, 13 pages.

1 INTRODUCTION

Like software, new hardware is rarely designed at once. Instead, the path from a specification to an application-specific integrated circuit is a long one, with several steps and with each step involving many design decisions and trade-offs. One of these design steps offering the greatest freedom is that of *technology mapping*—where *implementations* are picked for the primitives used in the design of the system until this point, and they are checked for compliance with functional and non-functional requirements.

This paper aims to define a formal framework for describing this process of technology mapping, as part of a bigger effort to verify the digital circuit design process. To this goal, we define an Embedded Domain-Specific Language (EDSL) in a host language with dependent types (Agda), using the dependent types of the host in order to guarantee type-safety properties of the circuits and using Agda as a proof assistant to show preservation properties related to technology mapping.

Starting from a high-level specification of a circuit in terms of its input-output behaviour, designers may start implementing parts of a circuit in terms of lower-level components, ultimately mapping these components down to individual gates. In this paper we formalize this notion of *refinement*, as essentially the (well-typed) substitution of syntax for semantics, where a high-level primitive component specified only by its semantics is replaced by an implementation, without changing the meaning of the overall circuit. In particular, we make the following contributions:

- We present λπ-Ware, a typed domain-specific language for the description, specification, simulation and synthesis of digital circuits embedded in the dependently typed programming language Agda. Crucially, we show how to use Agda's module system to parameterize λπ-Ware developments by the choice of primitive *gates* used in circuit definitions, as well as a base type carried over individual circuit wires.
- We define denotation semantics for the types and values that flow over circuit wires as well as for circuits themselves. By comparing semantics we can establish several *refinement* relations: between circuit wire types, between circuit wire values, between a gate and a circuit implementing it, between two gate libraries, and finally between a given circuit and a lower-level equivalent where each gate is mapped to an implementation in terms of lower-level gates.
- The key proven fact in the development is that substituting all primitive gates in a given circuit by respective implementations *preserves* semantics, as long as the implementation of each gate satisfies its specification.
- Finally, we demonstrate this approach by means of a case study, in which we describe the specification of a small Arithmetic-Logic Unit (ALU), and then refine it step-by-step using the relations and proofs developed in this paper. The resulting low-level circuit is defined in terms of elementary boolean gates.

The code of the hardware Domain-Specific Language (DSL) and assorted examples can be found online at https://gitlab.com/joaopizani/lambda1-hdl/tree/gate-refinement-paper.

2 SYNTAX OVERVIEW

Part of this work has been the creation of a Hardware Description Language (HDL) based on the typed lambda calculus. This language, λπ-Ware, is embedded into Agda [Norell 2007], a general-purpose dependently-typed programming language and proof assistant.

In this section we give an overview of the syntax and types of circuits described in λπ-Ware, by means of a running example of simple arithmetic circuits. Given the widely known basis of the language, our primary focus in this overview lies on the peculiar features of λπ-Ware originating from the desire to implement its terms in actual hardware.

IFL2022, Symposium on Implementation and Application of Functional Languages, August 2022

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

2.1 Universe of circuit types

The λπ-Ware language has well-typed circuits, done with the usual Agda technique of parameterizing the datatype definition of the EDSL with a *type universe*, representing those types that our circuits may take as inputs or produce as outputs.

The syntax of circuit types is presented below:

```
data U (B : Set) : Set where
  𝟙 ι       : U B
  _⊗_ _⊕_ : (σ τ : U B) → U B
  vec       : (τ : U B) (n : ℕ) → U B
```

We parameterize the universe of circuit types by a Set (B) [1]. As such we allow for choosing what is the type of basic data flowing over the wires: it could be booleans, numbers, or something else entirely chosen by the hardware designer. A value of such base type B is denoted by the *iota* (ι) constructor.

There is a code for the unit type (𝟙) in U. Its presence is necessary since it is frequently used as base case in the definitions of generic recursive circuit combinators. Furthermore, circuit types comprise binary products and coproducts, along with homogeneous fixed-size vectors. [2]. The arrow type is conspicuously missing, and in fact this restriction will be mirrored in the term syntax of λπ-Ware.

We chose to restrict ourselves to a first-order term language because higher-order functions do not have an immediate translation to hardware. More advanced techniques, like *defunctionalisation*, could be used as a synthesis step but are outside the scope of this study. Ultimately, this restriction is not so bad, because we can always use abstraction in the host language (Agda), hence allowing the user to write general definitions and avoid repeating themselves.

Keeping with our usage of a Simply-Typed Lambda Calculus (STLC) variation to represent circuits, we describe a circuit's inputs as *contexts*, which are essentially lists of types. The empty context is denoted by ε and the prepend operator by ◃ [3].

For example, the list of input types for a ripple-carry adder of width n can be written as follows, where Ctx Bool indicates a context with Bool as base type in the universe:

```
inputsAddN : (n : ℕ) → Ctx Bool
inputsAddN n = ι ◃ vec ι n ◃ vec ι n ◃ ε
```

That is, a ripple-carry adder of width n has three inputs: one carry-in bit and two bit vectors of size n each. As output for such an adder we have a pair of the carry-out bit and the summed vector:

```
outputAddN : (n : ℕ) → U Bool
outputAddN n = ι ⊗ vec ι n
```

Having seen how to describe the types that flow through circuit's wires and how to describe a given circuit's output and input ports, we now go on to describe the syntax of circuits themselves. We carry on using adders as an example to illustrate the definitions.

[1] Notation: we denote base types by capital Latin letters B and C, with subscripts for disambiguation.

[2] Notation: we denote circuit types by lower-case Greek letters towards the end of the alphabet (σ, ρ, τ, υ, ν), with subscripts for disambiguation.

[3] Notation: we denote contexts by the upper-case Greek letters Γ and Δ, with subscripts for disambiguation.

2.2 Syntax of circuits

Our DSL for hardware is deeply-embedded in the host Agda, and as such there is one core datatype modelling the syntax of circuits in λπ-Ware, called λB. Following the usual deep embedding of the STLC, the λB type is *indexed* by both the lambda term's context (inputs) and type (output). As such, our running example (n-bit wide binary adder) has the following Agda type:

```
addN : (n : ℕ) → λB (inputsAddN n) (outputAddN n)
```

Notice how addN is not a circuit, but a *function defining a family of circuits*: for each value of n, it gives a corresponding circuit (addN n). Now, let us introduce the constructors of the λB datatype, and after that give the definition of addN.

The most straightforward constructors of λB are those for variable binding, and they come from the usual (intrinsically-typed) embedding of the STLC in Agda, with a few peculiarities.

```
data λB : (Γ : Ctx) (τ : U) → Set where
  var  : (i : Γ ∋ τ)                    → λB Γ τ
  let′ : (x : λB Γ σ) (b : λB (σ ◃ Γ) τ) → λB Γ τ
  loop : (c : λB (σ ◃ Γ) (σ ⊗ τ))        → λB Γ τ
```

The parameter of the var constructor (of type Γ ∋ τ) can be thought of as an index into a context: it proves that there is an element of type τ in context Γ, and at which position it lies. The let′ constructor binds a term of type σ which becomes available for use in the context of the body b.

Peculiarly, our DSL has a *distinct and dedicated* constructor for looping (denoted loop), where one of the outputs of the computation (of type σ) is *fed back* again into the circuit. Due to the possible presence of loops in circuits, we give terms of λB a state-transition semantics, where loop is interpreted as a piece of memory containing state of type σ.

There are also the expected constructors for introduction and elimination forms of products, coproducts and vectors (Nil, Cons, Head, Tail). But we don't further expand on them since there is nothing specific to the hardware application.

Gate libraries. One critical piece of the puzzle is missing in the syntax of λB, namely the atomic components that actually perform computation in a circuit (with no further subparts). Instead of having any fixed set of gates (AND, NOT, OR, etc.), we have the whole development parameterized by a *gate library* G, through Agda's module system.

The single gate constructor (written ⟨_⟩) is what denotes the choice of an appropriate gate to use in a certain place in a circuit's syntax.

```
⟨_⟩ : (g : gIx G) {p : Δ ⊇ gCtx g} → λB Δ (gOut g)
```

Notice how the parameter g of ⟨_⟩ can only come from one of the gates in the library (gIx G), and that both the context (gCtx G g) and the output type (gOut G g) *depend* on the gate chosen. These functions are exactly what is packed into a record called a *gate library*: [4]

[4] Notation: we denote gate libraries by the upper-case Latin letters G, H, L. Gate indices are values of type gIx G, and variables with such type are denoted by lower-case Latin letters g, h, l. Subscripts are used for disambiguation.

```
record Gates (B : Set) : Set₁ where
  field
    gIx   : Set
    gCtx  : (g : gIx) → Ctx B
    gOut  : (g : gIx) → U B
```

Gate input weakening. Finally, notice how the gate constructors allow usage of a gate in a *larger* context, where more inputs than strictly necessary are available. The argument p chooses which inputs — among all available in context Δ — to route to the gate. Passing this weakening as *implicit argument* gives extra convenience: Often the difference between a gate's context and the larger one in which it is inserted is obvious, and can be deduced via unification.

Not only do we allow for the weakenings to be passed as argument of the gate constructors, but these are the *only* places where weakenings can occur. This restriction makes more convenient the comparison between the behaviour of a gate and of a given circuit implementing it, as we will see on Section 4.

Running example. Having seen how to define a circuit's syntax, we can give further details on our running example: an n-bit wide binary adder. The base type flowing over the wires is booleans (Bool), and we use a gate library with 4 basic gates: [5]

```
data B4Ix : Set where
  AND OR NOT XOR : B4Ix

B4Ctx NOT  = ι ◃ ε
B4Ctx _    = ι ◃ ι ◃ ε   -- AND, OR, XOR

B4 : Gates Bool
B4 = record {gIx  = B4Ix
            ;gCtx = B4Ctx
            ;gOut = const ι}   -- all gates have 1-bit output
```

The labels for the gates come simply from an enumeration type (B4Ix), and we define appropriately the shapes and sizes of contexts and outputs for each gate. In the following definitions, the circuit type λB should be always understood to have been applied to a module parameter B4 (the gate library for this running example).

```
ha : λB (ι ◃ ι ◃ ε) (ι {-cout-} ⊗ ι {-sum-})
ha = ⟨ AND ⟩ , ⟨ XOR ⟩

fa : λB (ι {-cin-} ◃ ι ◃ ι ◃ ε) (ι {-cout-} ⊗ ι {-sum-})
fa = ⟨ OR ⟩ , part
  where part = wkn ha {...}

addN : (n : ℕ) → λB (inputsAddN n) (outputAddN n)
addN zero    = var ix₀ , nil
addN (suc n) = MapAcc-par {n} fa ...
```

In the sketch of definitions above, a half-adder (ha) is used to build a full-adder (fa), and the n-bit adder (addN) is built as an *accumulating map* (MapAcc-par), essentially a row of n copies of the fa circuit, side-by-side.

[5]Notation: We name both specific gates (not variables) and gate libraries in *ALLCAPS* format.

3 SEMANTICS / SIMULATION

In this section we give semantics for all the language constructs discussed so far, namely, circuit types, gates and finally, circuits themselves, although we have simplified the presentation somewhat. In the full codebase of λπ-Ware we make the majority of these definitions more general, expressing semantics as a catamorphism on circuits. This allows us to define semantics preservation properties and congruence properties with respect to circuit transformations in general. However, due to the focus of this paper on simulation and in the interest of brevity, we give all types and definitions instantiated only to the case of simulation and remark along the way where the definitions can be generalised further.

Before diving into the details of the simulation semantics, notice that circuits in λπ-Ware are possibly sequential, that is, they may contain loops. Thus we can't simply map circuits to pure Agda functions. Instead the simulation semantics will map each circuit to a stateful function, in which a well-typed state value is threaded through.

3.1 Semantics of types and input/output values

Given that our circuit DSL is (intrinsically) typed, we must first establish semantics for circuit types, before doing so for circuits themselves. In this paper we give special attention to simulation semantics, as we want to discuss what happens when circuits are executed and, as such, we choose to map types in our DSL to types in the host language (Agda).

The simulation semantics for our types is called Val: it simply maps each constructor of the universe datatype (U B) into its corresponding Agda type. This is entirely straightforward: the ι constructor is mapped to the base type B; the 𝟙 type is mapped to Agda's unit type ⊤; etc.

```
Val : ∀ (τ : U B) → Set
Val 𝟙       = ⊤
Val ι       = B
Val (φ ⊗ ω) = Val φ × Val ω
...
```

With the semantics of circuit types at hand, we can define *environments* next. Each environment represents a heterogeneous list of values, indexed by a type context. Agda's standard library provides an implementation of this idea, Env, that we re-use here:

```
Ctx : Set → Set
Ctx B = List (U B)

Env : (U B → Set) → Ctx B → Set
```

In the simulation semantics, a context Γ denotes the listing of a circuit's *input types*, and thus an environment γ gives one possible list of *well-typed input values*. In other words, an environment (γ : Env Val Γ) ensures *by construction* that each of its elements v_k *has type* Val τ_k, where τ_k is the corresponding element in Γ [6].

```
Z : (σ ◃ Γ) ∋ σ
S : Γ ∋ τ → (σ ◃ Γ) ∋ τ
```

[6]Notation: we denote environments by lower-case Greek letters γ and δ, with subscripts for disambiguation.

The functions Z and S serve as pointers into a context, giving evidence for the location of a given type in a given context. While Z shows that the type is at the head of the list, the S ("successor") function gives the location for a type in the whole context, given the location for it somewhere in the tail.

3.2 Semantics of gates

In our language, *gates* are leaves of the circuit's Abstract Syntax Tree (AST), representing atomic computation primitives. Thus, we must give semantics to all gates in a circuit in order to give semantics to the circuit itself. Furthermore, given our interest in comparing the behaviour of a gate to that of an implementing circuit, it is useful to make the types of semantics align as much as possible.

Syntactical information about gates used in a circuit is packed in a *gate library* record. Aside from all the fields of Gates already mentioned in Section 2.2, we also need a field gSt, which gives the *state type* for each gate in the library. [7]

```
record Gates (B : Set) : Set₁ where
  field gIx      : Set
        gCtx     : (g : gIx) → Ctx B
        gSt gOut : (g : gIx) → U B
```

All these functions from a gate library are necessary ingredients for a gate library semantics, whose type is shown below:

```
gS : (G : Gates B) (g : gIx G)
  → Val (gSt g) → Env Val (gCtx g)
  → Val (gSt g) × Val (gOut g)
```

A gate library semantics maps a gate g in a library G to an Agda function: such function takes current state and input environment while returning next state and output. This type of simulation function corresponds to a common model of digital circuits known as *Mealy machines*.

We do not include the semantics as part of the gate library record, as the same library may have different notions of semantics associated with it, such as simulation or circuit area.[8].

One point we drive home further in the next sections is the great similarity between the type of gate semantics and the type of semantics for whole circuits. This is no coincidence, as we wish to relate the behaviour of a gate to that of a circuit, and define precisely when a circuit is said to *implement* such gate.

3.3 Semantics of circuits

Circuit state type. While the state of a gate is a *black box* and can be anything in the type universe U B, the type of state for a circuit follows the same structure as the circuit itself. The correspondence between a circuit and the type of its state is enforced by indexing the St datatype with the circuit c whose state is being represented. The only constructor of St with notable "content" is the one defining what is the state of a Loop, where the actual data storage happens:

```
data St {G : Gates B} : (c : λB[ G ] Γ τ) → Set where
  s⟨_⟩   : ∀ {g} (sg : Val (gSt g)) → St ⟨ g ⟩
  sLet   : ∀ {x e} (sx : St x)  (se : St e) → St (Let x e)
  sLoop  : ∀ {f}   (si : Val σ) (sf : St f) → St (Loop f)
  ...
```

The key detail here is the argument si : Val σ in the sLoop constructor, indicating that the state associated with a loop over σ consists of a Val σ, together with any further state used in the loop's body.

With all ingredients in hand, we can now express the semantics of whole circuits in our DSL. The two parameters of the function ⟦_⊨_⟧s are the library semantics S and circuit under evaluation c. This evaluates the circuit for a *step* of state transition.

```
S          : gS G
c          : λB[ G ] Γ τ
⟦ S ⊨ c ⟧s : St c → Env Val Γ → St c × Val τ
```

As said in Section 3.2, the type for circuit semantics (⟦ S ⊨ c ⟧s) is *very* similar to the type of gate semantics (gS G g), with the exception of the mismatch between gate state and circuit state types. This mismatch means that when we want to compare the behaviour of a gate and a circuit, we will need to convert between the respective state values.

The simulation semantics for circuits is defined by simultaneous pattern matching on both the circuit and state value, since the type of state depends on the circuit. Definitions for the core clauses (first-order STLC-like) are quite unsurprising, with the usual extension and lookup of the environment in the clauses for variable binders and variable reference, respectively.

```
⟦ S ⊨ Let x b ⟧s (sLet sx sb) γ =
  let sx′ , rx = ⟦ S ⊨ x ⟧s sx γ
      sb′ , rb = ⟦ S ⊨ b ⟧s sb (↦ rx ◃ γ)
  in (sLet sx′ sb′) , rb

⟦ S ⊨ [ i ] ⟧s s       γ = s , (lookup i γ)
⟦ S ⊨ Val v ⟧s (sVal .v) _ = (sVal v) , v
...
```

More interesting and relevant for technology mapping is the evaluation of a gate (⟦ S ⊨ ⟨ g ⟩ ⟧s), in which we use the gate semantic function applied to index g. The gate itself takes an input environment (γ : Env Val Γ) potentially smaller than the one available at that gate's position in the circuit (δ). The context inclusion is described by gate constructor parameter p.

The Env⊇ (environment inclusion) helper function gives, from an inclusion (p : Δ ⊇ Γ), a mapping that takes the larger environment δ and gives back only the smaller environment included in it by p.

```
Env⊇ : (p : Δ ⊇ Γ) → (Env Val Δ → Env Val Γ)
⟦ S ⊨ ⟨ g ⟩ {p} ⟧s s⟨ sg ⟩ γ = s⟨ sg′ ⟩ , v
  where sg′ , v = (S g) sg (Env⊇ p γ)
```

The clause for loops is handled similarly to the one for let, evaluating the body in an enlargened context, but instead of obtaining the extra environment element from a bound term, it is assumed to come from the previous clock cycle. Notice that in a Loop, the body itself (f) may *also* be stateful (have loops), so we ensure that the next state of the body is also included in the next state of the whole Loop.

[7] Gate state: our circuits are stateful, and so are (possibly) the gates of which they are made.

[8] Notation: we denote gate library semantics by upper-case Latin letters S and T, with subscripts for disambiguation.

```
⟦ S ⊨ Loop f ⟧s (sLoop si sf) γ =
  let sf′ , (si′ , rl) = ⟦ S ⊨ f ⟧s sf (↦ si ◃ γ)
  in (sLoop si′ sf′) , rl
```

Besides these core constructs, the semantics function of course also handles the introduction and elimination forms for products, coproducts and homogeneous fixed-size vectors. However, in the interest of space, we omit these clauses here as their definitions are largely unsurprising.

In the following sections we will discuss the notion of circuit refinement, where each gate of a circuit is replaced by a corresponding implementing circuit. Because the gate constructor of our AST allows the gate to be used in a weakened context, when we want to compare a circuit and a gate we must thus *weaken the circuit*.

The wkn function transforms a circuit taking a given input context Γ into a circuit that takes a *larger* context Δ, as long as evidence of the inclusion (p : Δ ⊇ Γ) is provided. The definition consists of just "pushing" the weakening down the AST nodes, until the leaves (gates) are reached, when the given weakening is then combined with whatever weakening is inherently part of that gate.

```
wkn : (p : Δ ⊇ Γ) (c : λB[ G ] Γ τ) → λB[ G ] Δ τ
wkn p (⟨_⟩ g {q}) = ⟨_⟩ g {comp⊇ q p}
wkn p ([_] i)     = [_] (wknIx p i)
wkn p (Let x f)   = Let (wkn p x) (wkn (I p) f)
...
sWkn : (p : Δ ⊇ Γ) {c : λB[ G ] Γ τ} (s : St c)
       → St (wkn p c)
```

Furthermore, in constructors involving variable binding (such as Let), we take care to adapt the weakening accordingly as it is done in the body of the binder. For any definition of circuit semantics, it is expected that such semantics is *preserved* by weakening (wkn). In particular, our simulation semantics satisfies such property, expressed as follows:

```
wkn-pres : (S : gS G) (c : λB[ G ] Γ τ) (s : St c)
           (p : Δ ⊇ Γ) (δ : Env Val Δ)
  → map× (sWkn p) id (⟦ S ⊨ c ⟧s s (Env⊇ p δ))
    ≡ ⟦ S ⊨ wkn p c ⟧s (sWkn p s) δ
```

The proof derives from compositionality of the semantics and proceeds just by congruence with each element of the algebra, thus this property holds generally for any semantics expressed as a (dependent) fold.

4 TECHNOLOGY MAPPING REFINEMENT

Technology mapping is a process of taking a circuit model defined in terms of high-level primitives operating on high-level types, and obtaining a lower-level circuit operating on lower-level types. We are interested in showing that such transformation is *semantics-preserving* (in particular with respect to the circuit's input/output behaviour). Finding the right definition of semantics preservation is one of the key goals of this section.

As an example, imagine converting a high-level model of arithmetic unit into a lower-level. The higher-level model has adders and multipliers as primitives and operates on bounded naturals as base type, while the primitives of the lower-level are simple logic gates (AND, OR, NOT) and operate on single bits. The technology mapping refinement involves a chain of relations, where each definition relies on previous ones. We illustrate each of the concepts here by means of an example, before nailing down these precise relations in the subsequent subsections.

Data concretisation converting a (high-level) type to its (low-level) implementation, e.g., mapping bounded natural numbers into fixed width bitwords;

Gate implementation relating (high-level) primitives with an implementation in terms of (low-level) gates, e.g., implementing an adder in terms of only AND, OR, NOT;

Gate library refinement proving that such a gate implementation preserves semantics, e.g., bundling the implementation and correctness proof for each arithmetic gate (ADD, MUL, etc.)

Circuit refinement replacing all the (high-level) primitives in a circuit description with their (low-level) implementation, e.g., substituting each occurrence of adders and multipliers by a specific implementation using AND, OR, and NOT.

4.1 Data concretisation

Our notion of refinement not only substitutes atomic components for detailed implementations (with subcomponents), but crucially also allows these detailed implementations to operate over *more concrete* data than the non-refined version.

In order to have well-typed value concretisation, we first concretise types: a type in the universe U B is given and a new one in universe U C is produced, where each occurrence of the base type (leaf) is substituted by the given parameter ι′. The transformation can also be done over a whole context of types.

```
⇓τ : (ι′ : U C) (τ : U B) → U C
⇓τ ι′ 𝟙 = 𝟙
⇓τ ι′ ι = ι′
...
⇓Γ : (ι′ : U C) (Γ : Ctx B) → Ctx C
⇓Γ ι′ = map (⇓τ ι′)
```

Except for the interesting leaf clause (ι), all others proceed simply by structural recursion. Having concretised circuit types and contexts, we can then defined well-type concretisation for circuit values and environments [9].

```
⇓t : {τ : U B} {ι′ : U C} (⇓ι : Val ι → Val ι′)
     → Val τ → Val (⇓τ ι′ τ)
⇓t {τ = ι}       ⇓ι t = ⇓ι t
⇓t {τ = σ ⊗ ρ} ⇓ι (x , y) = (⇓t ⇓ι x) , (⇓t ⇓ι y)
⇓t ... = ...

⇓γ : {ι′ : U C} (⇓ι : Val ι → Val ι′)
     → Env Val Γ → Env Val (⇓Γ ι′ Γ)
⇓γ ⇓ι = mapAll (⇓t ⇓ι)
```

In the definition of ⇓t we highlight the clause where a real value translation happens, as all other clauses consist simply of structural recursion over the type τ.

[9] All concretisation functions are defined generally for any compositional semantics (not only simulation).

4.2 Gate implementation

When we talk about a circuit that *implements* a given gate, we are saying that, with *some adaptations*, this circuit can be used instead of that gate, i.e. we can substitute one for the other. The requirement for this substitution to be correct is that the semantics of gate and circuit are equivalent. Before showing all the details, let us have a quick idea of what this equivalence means and what supporting definitions are involved:

```
(⇓t ⇓ι) (proj₂ (S          g          s          γ))
≡          proj₂ (⟦ T ⊨_⟧s c (⇓gSt s) (⇓γ ⇓ι γ))
```

For simplicity the excerpt above only concerns itself with output values (hence the usage of proj_2), while the full relation must also take care of next states. On the left hand side, we run the gate g on the state s and environment γ (using library semantics S), before converting the result to a low-level type ($\Downarrow$t); on the right hand side, we convert the state s and environment γ to their low-level representation, before feeding them to the low-level circuit c.

Here the function $\Downarrow$gSt maps each possible state value of a given gate g (with type Val (gSt g)) into state values of a corresponing circuit (with type St c).

```
⇓gSt : {g : gIx H} (ι´ : U C)
       {c : λB[ L ] (⇓Γ ι´ (gCtx g)) (⇓τ ι´ (gOut g))}
       → Val (gSt g) → St c
```

This function belongs to neither a gate nor a circuit on its own, but rather is *part of what defines the relation* between gate and circuit.

Having discussed concretisation for inputs, output and state, we are now almost ready to look into the full type and definition of the impBy relation between a gate and a circuit. As a last preparatory step, let us look at the similarities and differences between the *types* of semantic functions for gate and circuit. As always, our focus rests on *simulation* but the concepts are generalizable:

```
(gS H)     g
   : Val (gSt g) → Env Val (gCtx g) → Val (gSt g) × Val (gOut g)
(⟦ T ⊨_⟧s) c
   : St c          → Env Val Γ          → St c          × Val τ
```

Here, (gS H) g gives the simulation function for a given gate, picked from library H using index g. The semantics of a circuit (⟦ T ⊨_⟧s c) is *very* similar, mapping an initial state and environment to a next state and output.

There are only two key differences between the two types above:

- The *type of state* for a circuit depends on the circuit itself;
- The circuit semantics relies on a semantics for the gate library used (T : gS L).

The implementation relation between gate and circuit is called impBy. Before looking into its definition, let us first unravel its type, since it's already quite involved:

```
impBy : (S : gS H) (T : gS L) (⇓ι : Val ι → Val ι′) (g : gIx H)
        (c : λB[ L ] (⇓Γ ι′ (gCtx g)) (⇓τ ι′ (gOut g)))
        (⇓gSt : Val (gSt g) → St c)
        → Set
```

Here we are dealing with two gate libraries: a higher-level one (H, with base type B and semantics S), and a low-level one (L, with base type C and semantics T) From library H we pick a certain gate g, to form the claim that g is implemented by the circuit c.

The circuit's context and output types are related to those of gate g via the *concretisation* functions aforementioned. The choices made for data concretisation are described by the parameters ι' (implicit) and $\Downarrow\iota$. Each occurrence of the leaf constructor ι in the high level type is replaced by ι', and $\Downarrow\iota$ tells how to map high-level leaf *values* into low-level ones (semantic concretisation).

The last parameter, namely $\Downarrow$gSt, tells how to take a value of state for the gate g and make it into a value of state for the circuit (St c). Taking all the typing into account, the definition of impBy can be intuitively to state that first running the gate g on high-level inputs s and γ and then converting to the result to low-level outputs is the same as converting the inputs first and then passing these to the circuit c.

```
impBy S T ⇓ι g c ⇓gSt  =  ∀ s γ →
  map× ⇓gSt (⇓t ⇓ι) (S          g          s          γ)
  ≡                  ⟦ T ⊨_⟧s c (⇓gSt s) (⇓γ ⇓ι γ)
```

On the left-hand side of the equation, the low-level gate semantics (S g) is applied to well-typed current state and environment (S g s γ), and the resulting pair of next state and output are concretised. On the right-hand side, state and environment are first concretised before being fed to the circuit. The circuit c is simulated under the assumption of a low-level gate library semantics (T).

4.3 Gate library refinement

Commonly, the goal of technology mapping is to take a design using a certain library of primitive gates, and turn it into a design using "smaller" or "simpler" primitives.

If, for each gate g in some high-level library H, we have a circuit that implements it using only gates from a lower-level library L, then we can say there is a refinement relation *between the two gate libraries* H and L.

```
record ⇓g (S : gS H) {L : Gates C} (T : gS L)
          {ι´ : U C} (⇓ι : Val ι → Val ι´)
          (g : gIx H) : Set where
  field
    c    : λB[ L ] (⇓Γ ι´ (gCtx g)) (⇓τ ι´ (gOut g))
    ⇓gSt : Val (gSt g) → St c
    imp  : (impBy S T ⇓ι) g c ⇓gSt
```

First we have a record $\Downarrow$g which packages all the implementation details for one given gate: Such gate g from high-level library H (with semantics S) is said to have an implementation in a lower-level library L (with semantics T) whenever there is an appropriately-typed circuit using primitives from L, and such a circuit is shown to implement g. Furthermore, for each gate, the state concretisation function must be given, telling how to map the gate's state into an appropriately-structured circuit state.

A library can be seen as a collection of gates, thus the refinement of a gate library in terms of another is a function $\Downarrow$G returning the implementation record for each gate.

```
⇓G : (S : gS H) {L : Gates C} (T : gS L)
     {ι´ : U C} (⇓ι : Val ι → Val ι´) → Set
⇓G S T ⇓ι = λ g → ⇓g S T ⇓ι g
```

4.4 Circuit refinement

With the refinement of a whole gate library at hand, we can proceed to refine the definition of a circuit by means of *technology mapping*. In this operation, a circuit with gates coming from library H has *all* its gates replaced by subcircuits whose gates in turn come from library L. As all occurrences of the Gate constructor are affected, the resulting circuit contains *only* gates from the new library L; this property is expressed in the type of our technology mapping function.

The ⇓c function performs this operation, by using a *library refinement* (li) giving the implementation details for each gate g in H. Besides using a different library of gates, the resulting circuit also has its input context and output type appropriately concretised.

```
⇓c : {S : gS H} {L : Gates C} {T : gS L}
     {ι′ : U C} {⇓ι : Val ι → Val ι′} (li : ⇓G)
     → λB[ H ] Γ τ → λB[ L ] (⇓Γ ι′ Γ) (⇓τ ι′ τ)
```

In principle, ⇓c replaces *all* occurrences of gates in H by those in L. However, if desired, a *partial* refinement can be achieved using a lower library which is a tagged union of H with the "actually simpler" primitives, i.e. L = H ⊞ L₀ [10]. In such case the refinement is the identity for some gates (those in the image of the left injection), while it maps some other gates to lower-level implementing circuits.

Almost all clauses in the definition of ⇓c proceed trivially by induction, thus we focus on those clauses requiring additional lemmas. First the key clause for a gate (accompanied by weakening), denoted ⟨ g ⟩ {p}.

```
⇓c {ι′} li (⟨ g ⟩ {p}) = wkn (map⊇ (⇓τ ι′) p) ((li g) .c)
```

The heart of this definition replaces a gate g with its implementation as prescribed by the new gate library li. However, the gate constructor has two parts: the choice of gate g and the selection of its inputs, p. To ensure the corresponding inputs are still passed to the low level circuit, we weaken the resulting circuit using (the low level mapping determined by) p.

```
⇓c {ι′} li [ i ] = [_] (map∋ (⇓τ ι′) i)
```

Refinement of a circuit's state. By analogy to the gate refinement for a circuit we can define the gate refinement for a circuit's state. This function (⇓s) converts a high-level circuit state to the state associated with its low-level counterpart. It will be useful when talking about semantic comparisons between high- and low-level circuits.

```
⇓s : {S : gS H} {L : Gates C} {T : gS L}
     {ι′ : U C} {⇓ι : Val ι → Val ι′} (li : ⇓G)
     {c : λB[ H ] Γ τ} (s : St c) → St (⇓c li c)

⇓s li (s⟨ sg ⟩ {p}) = sWkn (map⊇ (⇓τ ι′) p) (((li _) .⇓gSt) sg)
⇓s li s[ ix ]       = s[_]  (map∋ (⇓τ ι′) ix)
⇓s li (sLoop si sf) = sLoop (⇓t ⇓ι si) (⇓s li sf)
```

[10] The _⊞_ function unites libraries via a tagged union of the index sets and wrapping gCtx, gOut, gSt appropriately.

Again, in the clause for the state of a gate (s⟨ sg ⟩) we must use the library implementation li for H, but now we utilise the ⇓gSt field of the ⇓G record, giving the state concretisation for a given gate. Furthermore, also in analogy to ⇓c, we must weaken this state, which is done by sK.

The state concretisation for a given gate cannot be completely defined by the gate's type and type of its implementing circuit, in other words, there are (possibly) multiple ways to concretise the state of a gate with a certain implementing circuit in mind. As such, the state concretisation is the hardware designer's choice and packed as a field in the ⇓G record.

The only other interesting clause in ⇓s is the one for the state of a loop, sLoop, where *value concretisation* takes place through the ⇓t function.

5 SEMANTIC PRESERVATION

After having formalized the notion of technology mapping by means of gate and data concretisation, we want to show that technology mapping preserves semantics. This is a key result of this paper, establishing that our notion of technology mapping is sound. In particular, we show here that this preservation property holds for the state-transition simulation semantics. This section breaks down the statement and proof of preservation in their most important clauses and lemmas, and gives some applications of the general property to useful specialized situations.

5.1 Preservation property statement

The preservation property can be informally thought of as a *behavioural equivalence* between the high-level and low-level (refined) circuits, up to gate and data concretisation.

```
⇓c-pres : (li : ⇓G) (c : λB[ H ] Γ τ) (s : St c) (γ : Env Val Γ)
  → map× (⇓s li) (⇓t ⇓ι) (⟦ S ⊨ c ⟧s s γ)
    ≡ ⟦ T ⊨ ⇓c li c ⟧s (⇓s li s) (⇓γ ⇓ι γ)
```

Both key functions involved in this equality, namely the circuit semantics itself (⟦_⟧s) and the circuit refinement function (⇓c), are defined by induction over the circuit structure. Therefore the proof of ⇓c-pres proceeds by induction on the circuit (c : λB[H] Γ τ) and state (s : St c). Matching on a case of λB also forces the input/output types in certain ways, thus driving evaluation of the data concretisation functions (⇓t, ⇓γ).

Since our simulation semantics returns a product of the next state and output value, we need to prove the equality between two products. We do this by proving the equality of both the first and the second component separately.

```
⇓c-pres li c s γ = ×-≡,≡→≡ (⇓c-pres-st li c s γ)
                            (⇓c-pres-out li c s γ)
```

Thus we have reduced the soundness of our technology mapping to two keep lemmas, establishing that the outputs and states are preserved.

5.2 Proof sketch

Similarly to the proof of wkn-pres-out, the clauses in the proof of ⇓c-pres-out can be split in two categories: those who are "simply inductive" and the "base cases". The simply inductive clauses

follow by induction and by congruence with each of the functions in the algebra; For example, the semantics of Head involves Data.Vector.Base.head, so (⇓c-pres-out Ii (Head xs) (sHead xs) γ) reduces to:

```
(⇓t ⇓ι) (head (proj₂ (⟦ S ⊨       xs ⟧s       sxs          γ)))
≡       head (proj₂ (⟦ T ⊨ ⇓c Ii xs ⟧s (⇓s Ii sxs) (⇓γ ⇓ι γ)))
```

Notice how the head function "pops out" of the subexpression with the simulation semantics and sits between proj_2 and the output value concretisation (⇓t ⇓ι). Our goal is to rewrite the LHS by using the inductive hypothesis, but to do that we now need an additional commutativity lemma that *swaps* head and (⇓t ⇓ι) around.

```
⇓t-comm-head : (xs : Vec (Val τ) (suc n)) (⇓ι : Val ι → Val ι′)
             → (head ∘ ⇓t ⇓ι) xs ≡ (⇓t ⇓ι ∘ head) xs
```

The proof of this lemma follows just by definition of ⇓t and head (non-empty xs). An analogous commutativity lemma is needed for each of the algebra functions corresponding to each of the non-leaf constructors of λB.

With this lemma in hand, all that is needed to finish off the Head clause is congruence with head and induction. All other non-base cases of ⇓c-pres-out and ⇓c-pres-st follow this same pattern.

```
trans (⇓t-comm-head ⇓ι (proj₂ (⟦ S ⊨ xs ⟧s sxs γ)))
      (cong head (⇓c-pres-Val Ii xs sxs γ))
```

The "base cases" are the key to proving ⇓c-pres-out, and most important of all is the case for the gate constructor (⟨_⟩):

```
⇓c-pres-out-Gate :
  (Ii : ⇓G) (g : gIx H) (sg : Val (gSt H g)) (γ : Env Val Γ)
  → (⇓t ⇓ι) (proj₂ (⟦ S ⊨       ⟨ g ⟩ ⟧s       s⟨ sg ⟩       γ))
    ≡       proj₂ (⟦ T ⊨ (⇓c Ii) ⟨ g ⟩ ⟧s (⇓s Ii s⟨ sg ⟩) (⇓γ ⇓ι γ))
```

By applying the definitions of ⇓c, ⇓s and ⟦_⊨_⟧s to the case of gate we arrive at a form of ⇓c-pres-out-Gate in which we can see the opportunities to apply lemmas already at our disposal:

```
(⇓t ⇓ι) (proj₂ (S       g
                sg
                (Env⊇ q γ)))
≡       proj₂ (⟦ T ⊨_⟧s (wkn   (map⊇ (⇓τ ι′) q) (Ii g .c))
                        (sWkn (map⊇ (⇓τ ι′) p) ((Ii _ .⇓gSt) sg))
                        (⇓γ ⇓ι γ))
```

To solve this goal we will need of course the proof that gate g is implemented by circuit (Ii g .c), which is packaged in the library implementation record Ii. We will also need the wkn-pres lemmas to deal with the circuit and state weakenings involved. Furthermore we need a lemma that environment inclusion and environment concretisation are commutative. This lemma (⇓γ-Env⊇-comm) derives from the functoriality of ⇓γ (it's implemented as a map).

```
⇓γ-Env⊇-comm :
  (⇓ι : Val ι → Val ι′) (p : Δ ⊇ Γ) (δ : Env Val Δ)
  → Env⊇ (map⊇ (⇓τ ι′) p) (⇓γ ⇓ι δ) ≡ ⇓γ ⇓ι (Env⊇ p δ)
```

6 CASE STUDY

In this section we describe the minimalist example of an ALU for a simple processor. The goal is to illustrate the core ideas of refinement by means of gate and data concretisation.

We start the design process by using a coarse-grained library of gates (one for each operation in the ALU). Then we refine the design by concretising it into a library of lower-level logic gates (simple boolean logic).

6.1 High-level description

The high-level description of our example ALU consists of two major subcomponents: a Logic Unit (LU) and an Arithmetic Unit (AU). Each subcomponent works with its own gate library and base type for the type universe. In this subsection we give an overview of each subcomponent individually and what is necessary to "glue" them together in order to make an ALU.

High-level types. The base type of the Logic Unit is simply booleans. To avoid confusion between leaf types in universes with different bases, we give each an unique alias: the leaf type of universe U Bool is called $\mathbb{B}$.

```
𝔹 : U Bool
𝔹 = ι
```

The Arithmetic Unit subcomponent works with Fin n as base type, that is, the set of naturals from 0 up-to-but-excluding n.

```
ℕ_n : U (Fin n)
ℕ_n = ι
```

It does not denote a single base type but a family of base types, one for each n. In light of this, all related definitions from here on (of types, gates, circuits) are also generalized over n.

High-level gate syntax. The gate library is composed of two parts that are *united*: the arithmetic part and the logic part. The logic part is largely based on B4 as defined in Section 2.2, but each of the gates performs bitwise logical operations over *vectors* instead of single bits. We keep notation consistent by naming this library B4_k.

```
data B4ₖIx : Set where
  ANDₖ ORₖ NOTₖ XORₖ : B4ₖIx

B4ₖCtx : ∀ k → U Bool
B4ₖCtx k NOTₖ = vec 𝔹 k ◃ ε                -- NOTₖ is unary
B4ₖCtx k _    = vec 𝔹 k ◃ vec 𝔹 k ◃ ε      -- others are binary

B4ₖ : ∀ k → Gates Bool
B4ₖ k = record {gIx = B4ₖIx;  gCtx = B4ₖCtx k
               ; gSt = const 𝟙; gOut = const (vec 𝔹 k)}
```

As basis for the arithmetic part of the ALU, we have the FIN gate library, consisting of gates for performimg modular addition and multiplication on bounded naturals.

```
data FINIx : Set where ADD MUL : FINIx

FIN : ∀ n → Gates (Fin n)
FIN n = record {gIx = FINIx;  gCtx = const (ℕ_n ◃ ℕ_n ◃ ε)
               ; gSt = const 𝟙; gOut = const ℕ_n; }
```

Now we wish to build the *unified* gate library of our ALU by combining these two sublibraries (logic and arithmetic). To be able to combine them, both need to work over the *same base type*. The obvious alternative (which we choose) is to have Bool as base type. This means the arithmetic gates in the new unified library will have at the core the operations from FIN as above, but *wrapped* by (de)coding between Fin 2^k and vec $\mathbb{B}$ k.

This is not a performance issue since the high-level model serves only as specification and the low-level model is the one actually interesting to synthesize "in silicon". The whole point of technology mapping will be to show that both approaches (high and low) are equivalent:

```
Fin→W : Fin 2^k → vec 𝔹 k
W→Fin : vec 𝔹 k → Fin 2^k
```

In order to unite the two libraries under a common base type, as mentioned before, we first wrap the arithmetic gates with (de)coding so they can handle binary words. The result is a transformed arithmetic gate library, with booleans as base type. Crucially, as is shown later, the library semantics is transformed to apply the (de)coding functions.

```
FIN_k : ∀ k → Gates Bool
FIN_k k = record {gIx = FINIx; gSt = const 𝟙
                 ; gCtx = const (vec 𝔹 k ◃ vec 𝔹 k ◃ ε)
                 ; gOut = const (vec 𝔹 k); }
```

With this wrapped version of the arithmetic unit at hand, we can then build the total library by using the library union operator:

```
FINB4_k : ∀ k → Gates Bool
FINB4_k k = FIN_k k ⊞ B4_k k
```

A library FINB4$_k$ k has both of its subparts operating over the same base type (vectors of booleans). Internally, however, the operations of the arithmetic part are performed over Fin 2^k, wrapped by the required conversions.

High-level gate semantics. Having defined the syntax and types of the higher-level description of our ALU, we now make the definition complete by giving it its semantics. Firstly, the modular arithmetic operations come straight out of Agda's standard library:

```
addN mulN : Fin n → Fin n → Fin n
```

We then wrap these operations to make them match the exact expected interface for each gate, and finally bundle the semantics of each gate into a semantic function for the whole FIN library (FINgS) [11].

```
specADD specMUL : Env Val (ℕ_n ◃ ℕ_n ◃ ε) → Val ℕ_n
specADD (↦ x ◃ ↦ y ◃ ε) = addN x y
specMUL (↦ x ◃ ↦ y ◃ ε) = mulN x y

FINgS ADD = idState specADD
FINgS MUL = idState specMUL
```

The definitions of the specification functions for ADD and MUL in the *binary-coded* FIN$_k$ library make use of specADD and specMUL from the basic FIN library, but wrapped with the appropriate (de)coding functions.

```
specADD_k specMUL_k : Env Val (vec 𝔹 k ◃ vec 𝔹 k ◃ ε)
                      → Val (vec 𝔹 k)
specADD_k = Fin→W ∘ specADD ∘ mapEnv W→Fin
specMUL_k = Fin→W ∘ specMUL ∘ mapEnv W→Fin

FIN_kgS ADD = idState specADD_k
FIN_kgS MUL = idState specMUL_k
```

In the interest of space, we do not show here the semantics for the gates in the B4$_k$ library, as they simply consist of application of the corresponding logic functions (and, or, and so forth). Finally, the semantics of the united gate library is the union of the semantics for each library individually.

```
FINB4_kgS : gS (FINB4_k k)
FINB4_kgS {k} = (FIN_k {k}) gS⊞ (B4_kgS {k})
```

High-level ALU. With all the gates and their semantics understood, let's look at the definition of our minimal ALU as a whole. It is composed of two parts: one arithmetical and one logical. Each of these parts just consists of all the relevant gates, side-by-side, along with multiplexing used to decide which operation to perform on the inputs.

For convenience of exposition, we fix the word length to be 8 bits in this example. However, the definitions in the source code are all parameterized by word length.

```
λb = λB[ B4_k 8 ]
W8 = vec 𝔹 8

LUcmd : U B   -- LU: "Logic Unit"
LUcmd = 𝟙 ⊕ 𝟙 ⊕ 𝟙 ⊕ 𝟙

LU : λb (LUcmd ◃ W8 ◃ W8 ◃ ε) W8
LU = Case⊕ #0 {-cmd -} of K_2 ⟨ NOT ⟩    -- use only 2nd data in
  or Case⊕ #0          of K_1 ⟨ AND ⟩
  or Case⊕ #0          of K_1 ⟨ OR ⟩
  or                     K_1 ⟨ XOR ⟩
```

The Logic Unit takes two data inputs and a *command* input. The command type is simply a tagged union of units, used to select the relevant operation. A more complex command type could potentially be used, but it may then require some sort of decoding unit, so we go for the simple alternative since command decoding is not this study's focus.

We combine all the separate parts by cascading case analysis on the command, with the detail that each subcircuit does *not* need the command itself (thus the K_1 in front), and the Not circuit in particular only uses the second data input.

The definitions forming the Arithmetic Unit follow the same general scheme, with appropriate command and data inputs:

```
λb = λB[ FIN_k 8 ]

AUcmd : U B   -- AU: "Arithmetic Unit"
AUcmd = 𝟙 ⊕ 𝟙

AU : λb (AUcmd ◃ W8 ◃ W8 ◃ ε) W8
AU = Case⊕ #0 of K_1 ⟨ ADD ⟩
              or K_1 ⟨ MUL ⟩
```

[11] The idState takes a combinational semantic function (i.e. one that takes no state) and makes it into a semantic function that takes any state and passes it through untouched.

With both subunits (Arithmetic and Logic) at hand, we can glue them together to form the ALU. The gate library is the joint library mentioned before. Finally, there is a joint command input which is a tagged union of either an LU or an AU command.

```
λb = λB[ FINB4_k 8 ]
ALUcmd : U B
ALUcmd = LUcmd ⊕ AUcmd
ALU : λb (ALUcmd ◃ W8 ◃ W8 ◃ ε) W8
ALU = Case⊕ #0 of LU or AU
```

6.2 Low-level description

In the low-level description of our ALU case study, we implement all primitive gates used in the high-level description in terms of *simpler* gates.

It is not necessary, however, to perform data concretisation, since both the high and the low-level descriptions use the same base type (booleans). We made this choice for equal base types both in the interest of simplicity, but also because it was convenient to unify the high-level Logic and Arithmetic subunits.

Low-level gate syntax and Logic Unit. First we choose the library of primitive gates to use in the low level description. Namely, we conveniently use the B4 library already defined in Section 2.2, containing NOT, AND, OR and XOR.

This choice is particularly convenient for the implementation of the low-level Logic Unit. That is because the high-level specifications for the bitwise logical operations are just applying Vector.map to each logical function. To make the low-level circuits that implement each gate, we use each appropriate gate from B4 and apply a mapping circuit combinator.

```
λb = λB[ B4 ]
Not8          : λb (W8 ◃ ε) W8
And8 Or8 Xor8 : λb (W8 ◃ W8 ◃ ε) W8
Not8 = Map-par ⟨ NOT ⟩
...
```

The Map-par combinator is defined generically by induction on the size n of input/output vector type, and its definition has some levels of indirection in the actual source code of λπ-Ware, but it can be expressed directly just using the basic λB vector constructors (Nil and Cons).

Low-level Arithmetic Unit. For the Arithmetic Unit, there is some care needed to ensure that there is a match between the input/output *types* of high-level and low-level descriptions. Also we define the implementing circuits carefully to make showing the equivalence between specification and implementation not unnecessarily difficult.

Notice first that in the core of the high-level arithmetic gate specifications (specADD, specMUL) there is a *modulo* operation, that is, we are dealing with *modular arithmetic*. This is mostly for convenience, since we wish the sizes of inputs and outputs of our ALU to be uniform. Since we are dealing always with mod 2^k and with natural numbers, we can perform the modulo by *truncating to k bits* (eight in the ALU specifically).

```
AddN : λb (𝔹 ◃ vec 𝔹 k ◃ vec 𝔹 k ◃ ε) (𝔹 ⊗ vec 𝔹 n)
MulN : λb (     vec 𝔹 k ◃ vec 𝔹 k ◃ ε) (     vec 𝔹 2 k)
Snd : λb Γ (τ1 ⊗ τ2) ⟶ λb Γ τ2
Add8 Mul8 : λb (W8 ◃ W8 ◃ ε) W8
Add8 = Let (Con false)    -- carry-in is zero
           (Snd AddN)     -- take second: ignore carry-out
Drop : ∀ k ⟶ λb Γ (vec τ (k + n)) ⟶ λb Γ (vec τ n)
Mul8 = Drop 8 MulN
```

In the case of addition, truncating to k bits means discarding the *carry-out* bit, since normally adding two k-bit would result in a k+1-bit output (k regular output bits plus carry). We reuse here the ripple-carry adder AddN as it has been defined in Section 2.2.

The multiplication of two k-bit natural numbers results in a number with up to 2 k bits. We then perform mod 2^k by simply discarding the k most-significant digits of the result.

A possible performance improvement to the multiplier would be to not even calculate the digits that are discarded, but we take the "calculate then discard" approach for two reasons. First, because it makes the proof of equivalence simpler. Furthermore, the general multiplier is more reusable, and an equivalence between more general and more efficient versions could be future work.

6.3 Refinement and semantic preservation

We now arrive at the core point of the case study: give a simple example of how verified technology mapping can be applied in a concrete circuit. In the previous subsections we gave an exposition of a toy example ALU both in terms of a high-level specification and a low-level implementation (with different primitive gates). Now we focus on what are the remaining ingredients to apply the *semantic preservation theorem* for technology mapping, and conclude by applying said theorem.

In the previous section we gave one circuit in the low-level corresponding to each gate in the high level, with matching types and environments. We must now also prove that each such circuit indeed *implements* the corresponding high-level gate.

Logic Unit gate implementation. We will show only one example of such gate/circuit implementation proof for a Logic Unit gate, as the others follow exactly the same pattern. First let us just look at the statement of the implementation property for the AND_k gate and And_8 circuit:

```
impBy B4_kgS B4gS id AND_k And8 (const sAND8) =
  ∀ (s : Val (gSt AND_k) {-unit-} ) γ ⟶
    map× (const sAnd8) (⇓t id)
      (B4_kgS      AND_k s                γ)
  ≡   ⟦ B4gS ⊨_⟧s And8  (const sAnd8 s) (⇓γ id γ)
```

There are two crucial characteristics of And_8 that facilitate the proof of the above statement:

- It is combinational, the *state is irrelevant* to correctness. So the state concretisation function can simply be const sAnd_8
- High and low-level base types are equal, so the function for data concretisation is simply the identity

The statement can be simplified by using the definitions of $\Downarrow$t ($\Downarrow$t id = id), const (const $sAnd_8$ s = $sAnd_8$) and $\Downarrow\gamma$ ($\Downarrow\gamma$ id γ = γ). Afterwards, the goal of equality between two pairs is split into two subgoals, one for each projection.

$sAnd_8 \equiv proj_1\ (\llbracket\ B4gS \vDash And_8\ \rrbracket s\ sAnd_8\ \gamma)$

The state equality is simplest, since And_8 is built from a *combinational* circuit combinator (Map-par), which passes the state through unmodified. Thus the initial state is equal to the final state, and is equal to the subgoal's LHS.

$$\begin{aligned} & proj_2\ (B4_k gS \quad AND_k\ s \quad \gamma) \\ \equiv\ & proj_2\ (\llbracket\ B4gS \vDash \_\rrbracket s\ And_8\ sAnd_8\ \gamma) \end{aligned}$$

The equality between the results (second projection) will ultimately (within a few more reduction steps) come to rely on an equivalence lemma which expresses essentially the *specification* of the Map-par combinator.

$$\begin{aligned} & proj_2\ (\llbracket\ S \vDash \_\rrbracket s\ (Map\text{-}par\ f)\ \_\ (\mapsto xs \triangleleft \gamma)) \\ \equiv\ & map\ (\lambda\ x \rightarrow proj_2\ (\llbracket\ S \vDash \_\rrbracket s\ f \quad \_\ (\mapsto x \triangleleft \gamma)))\ xs \end{aligned}$$

For conciseness we omit details of the initial state values above, since the circuit built by Map-par just passes through (unmodified) the state of each copy of f. The proof for this lemma proceeds by induction on the vector size and structural induction on the vector xs. The induction over n drives reduction via the semantics of Map-par on the LHS, while induction over xs drives reduction via the definition of map on the RHS.

As all other LU bitwise operators are also defined via Map-par, the proof for each of them proceeds in an identical manner.

Arithmetic Unit gate implementation. In the case of the arithmetic unit there are more subtle differences between the high-level specification and low-level implementation, thus the implementation proof has more intermediate steps.

$$\begin{aligned} & impBy\ FIN_k gS\ B4gS\ id\ ADD\ Add_8\ (const\ sAdd_8) = \\ & \quad \forall\ (s : Val\ (gSt\ ADD)\ \{\text{-unit -}\}\ )\ \gamma \rightarrow \\ & \qquad map\times\ (const\ sAdd_8)\ (\Downarrow t\ id) \\ & \qquad\quad (FIN_k gS \quad ADD\ s \quad \gamma) \\ & \quad \equiv \quad \llbracket\ B4gS \vDash \_\rrbracket s\ Add_8\ (const\ sAdd_8\ s)\ (\Downarrow\gamma\ id\ \gamma) \end{aligned}$$

The same simplification steps can be taken here as were taken when discussing the implementation of AND_k, namely involving the definitions of $\Downarrow\gamma$, $\Downarrow$t and const. Also, the equality of pairs is again split into a pair of equalities, one for each projection.

$sAdd_8 \equiv proj_1\ (\llbracket\ B4gS \vDash Add_8\ \rrbracket s\ sAdd_8\ \gamma)$

Again, the state equality subgoal is simple to solve for the same reason as as in the LU cases: for a circuit built with *combinational* combinators, the state does not matter (is passed through). However the result equality subgoal has considerable subtleties to it.

$$\begin{aligned} & proj_2\ (FIN_k gS \quad ADD\ s \quad \gamma) \\ \equiv\ & proj_2\ (\llbracket\ B4gS \vDash \_\rrbracket s\ Add_8\ sAdd_8\ \gamma) \end{aligned}$$

First of all, there is the carry-out ignore wrapping that makes Add_8 out of AddN. Further reducing we will see that the subgoal ultimately comes to rely on the behaviours of (on the RHS) an *accumulating map circuit combinator* (MapAccL-par) and (on the LHS) the $specADD_k$ function with its binary (de)coding wrappers.

$$\begin{aligned} & addN\ :\ Fin\ 2^k \rightarrow Fin\ 2^k \rightarrow Fin\ 2^k\ \ \{\text{-special case n = } 2^k\ \text{-}\} \\ & specADD\ (\mapsto x \triangleleft \mapsto y \triangleleft \varepsilon)\ =\ addN\ x\ y \\ & \llbracket\ B4 \vDash AddN\ k\ \rrbracket s\ \ldots\ =\ \llbracket\ B4 \vDash MapAccL\text{-}par\ \{k\}\ Add\ \rrbracket\ \ldots \\ & specADD_k \quad \ldots\ =\ Fin{\rightarrow}\mathbb{W} \circ specADD \circ mapEnv\ \mathbb{W}{\rightarrow}Fin \end{aligned}$$

Notice how MapAccL-par is applied to input vector length k, and that the type of addN is also specialized to the case where $n = 2^k$. Thus to progress in this proof, induction over k is needed. The MapAccL-par side of the equation will reduce in such a way that allows quite direct application of the inductive hypothesis. To be able to make progress in the $specADD_k$ side, we will need to *split* a number of type Fin 2 ** (suc k), using properties of exponents along the way.

$$2\ **\ (suc\ k)\ =\ 2 \times 2^k\ =\ 2^k + 2^k$$

$$splitFin\ :\ Fin\ (2^k + 2^k) \rightarrow Bool \times Fin\ 2^k$$

A number of type Fin $(2^k + 2^k)$ can be either smaller than 2^k or larger. The splitFin function returns the boolean of whether the number is larger than 2^k, along with the remainder. In this way, it essentially *decodes one single bit* of the number. Via this splitting of Fin numbers, we are able to complete the proof.

Semantic preservation of ALU. With the semantic equivalence proofs for each pair of high-level gate and corresponding implementing circuit, we can apply the preservation theorem for the ALU circuit as a whole. First we just build the library gate refinement function $\Downarrow$A (for ALU), which bundles up the information about how to implement each gate.

$$\begin{aligned} \Downarrow A\ AND_k\ &=\ \textbf{record}\ \{c\ =\ And_8 \\ &\qquad ;\Downarrow gSt\ =\ const\ sAnd_8 \\ &\qquad ;imp\ =\ impAnd_8\ \} \\ \Downarrow A\ OR_k\ &=\ \textbf{record}\ \{c\ =\ Or_8 \\ &\qquad ;\Downarrow gSt\ =\ const\ sOr_8 \\ &\qquad ;imp\ =\ impOr_8\ \} \end{aligned}$$

...

Then we can finally apply the preservation theorem $\Downarrow$c-pres, with an initial state sALU which is of the correct shape but is irrelevant for computational behaviour (since the ALU is combinational). What we gain is the certainty that the whole low-level description (after applying $\Downarrow$c) is a *correct implementation* of the high-level model we used as *specification*, given that each of the gates is correct.

$$\begin{aligned} & \Downarrow c\text{-}pres\ \Downarrow A\ ALU\ sALU\ :\ (\gamma\ :\ Env\ Val\ \Gamma) \\ & \quad \rightarrow map\times\ (\Downarrow s\ \Downarrow A)\ id \\ & \qquad (\llbracket\ FINB4_k gS \vDash \quad ALU\ \rrbracket s \quad sALU \quad \gamma) \\ & \quad \equiv \quad \llbracket\ B4gS \quad \vDash (\Downarrow c\ \Downarrow A\ ALU)\ \rrbracket s\ (\Downarrow s\ \Downarrow A\ sALU)\ (\Downarrow\gamma\ id\ \gamma) \end{aligned}$$

This notion of *correctness w.r.t. a higher-level circuit model* is complementary to the more common notion of correctness: correctness of a circuit w.r.t. a host-language definition (Agda function). It is part of a chain of reasoning where a designer starts with trustworthy/verified Agda functions (maybe from a standard library) and through successive steps of refinement can arrive at a circuit model which they deem suitable for implementation in their underlying hardware technology of choice (e.g. synthesizable VHDL).

7 DISCUSSION

7.1 Related work

There is a rich tradition of using the functional programming paradigm to model digital circuits. Sheeran [2005] and Chen [2012] both give a historical overview of the different approaches and domain specific embedded languages that have been explored in the last decades, including μFP [Sheeran 1984], Ruby [Brown and Hutton 1994], Lava [Bjesse et al. 1999; Gill et al. 2009; Singh 2004], Wired [Axelsson et al. 2005], Forsyde [Sander and Jantsch 2004], Hawk [Launchbury et al. 1999; Matthews et al. 1998], and others. These approaches typically embed a domain specific language for hardware description in a strongly typed, general purpose functional language such as Haskell—just as we have embedded our DSL in Agda. One particularly interesting aspect of Lava is that it enables users to define parametrised circuits, such as an adder taking its width as an argument, and *connection patterns*, higher order functions that capture recurring structures in hardware design. When it comes to verification, however, this is typically done by *instantiating* parametrised circuits and calling an automatic solver on the result. In our approach, on the other hand, we can use our host language, Agda, to establish inductively the correctness of circuit generators—rather that verifying each instance separately. Furthermore, we can exploit the carefuly construed compositional structure of a circuit *during* verification, rather than only calling an automated solver on the final design. Finally, the main focus of this paper has been in formalising the *technology mapping* process whereas Lava employs a fixed set of primitive circuits and boolean types.

More recently, the work on Clash [Baaij 2015; **?**] has started to explore a new point in this design space. Where most existing approaches use Haskell as a host language to embed circuits, Clash instead generates hardware for (a fragment of) GHC's core language directly. This enables users to re-use many of the syntactic features, such as pattern matching, that our DSL is lacking; on the other hand, the focus of the work on Clash is very much defining circuits, rather than their verification in a proof assistant.

There is a long tradition of verifying hardware using proof assistants, in particular using Isabelle/HOL [Boulton et al. 1992; Melham 1993], where the higher order logic is used to model both circuits and their specifications. The approach outlined in this paper, makes essential use of *dependent types* to ensure that our circuits can be executed safely—or more precisely, assigned a denotational semantics as a Mealy machine. One important consequence of this, is that it allows us to safely replace a (high level) circuit with its (low level) implementation—precisely the property required for safe technology. Similar shallow embeddings have been done in proof assistants based on type theory before [Coupet-Grimal and Jakubiec 2004; Paulin-Mohring 1995]. Work by Brady et al. [2007] has shown how to record a circuit's semantics in its type, allowing for correct-by-construction circuit definitions. Unfortunately, this work only considers combinational circuits (without state), rather than the Mealy machines used in this paper.

Our approach is closest in spirit to other domain specific embedded languages using dependent types, such as Coquet [Braibant 2011] and PiWare [Pizani Flor et al. 2016]. Both these languages, however, focus exclusively on working at a single (low-level) of circuit design. The key innovation presented in this paper is parametrising circuits by the types they process, allowing for the (gradual) mapping from high specifications to (low level) implementatations.

7.2 Future work

Modular construction of gate libraries. In the case study we have built our high-level library of primitive gates by *uniting* two smaller libraries (arithmetic and logical). However, we had to work around the requirement of using a *single* base type for both sub-libraries.

We are still investigating which, if any, fundamental changes to the typing discipline, syntax and/or semantics of $\lambda\pi$-Ware may be required to allow combining these type-heterogeneous gate libraries. Even though this question does not belong to the core of technology mapping, such combined libraries with different types reflect the modular design that often occurs in reality and would increase part reuse. Our previous work on combining datatypes [Swierstra 2008] suggests one direction to tackle this issue.

Congruence of refinement with circuit transformations. A property of the technology mapping refinement that we wish to explore in the future is that it should be *compatible* with *compositional* circuit transformations. That means doing the transformation and then refining should have the same semantic effect as first refining and then applying a (modified) transformation on the lower level.

This could apply, for example, to the verified timing transforms that we explored in previous work. We expect this property to hold since the refinement's proof of semantic preservation is mostly a consequence of the *compositionality* of both the refinement function and the used circuit semantics. Thus any compositional transformation would be compatible.

Variable binding. The representation of bound variables, using well typed De Bruijn indices, is perfect for defining the semantics of a circuit; writing larger circuits by hand, however, is far too cumbersome. We want to investigate how to add a more human friendly frontend to our current implementation, for instance, exploiting existing techniques for converting between different approaches to variable binding [Atkey et al. 2009] or using Agda's macros and other metaprogramming features [Walt and Swierstra 2012].

ACKNOWLEDGMENTS

This work was supported by the Netherlands Organization for Scientific Research (NWO) project on *A Dependently Typed Language for Verified Hardware*.

Special mention goes to Xander van der Goot, who contributed to the beginnings of the case study applying formalized technology mapping to a simplified data unit of a microprocessor.

REFERENCES

Robert Atkey, Sam Lindley, and Jeremy Yallop. 2009. Unembedding Domain-specific Languages. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1596638.1596644

Emil Axelsson, Koen Claessen, and Mary Sheeran. 2005. Wired: Wire-Aware Circuit Design. In *Correct Hardware Design and Verification Methods*. Springer, Berlin, Heidelberg, 5–19. https://doi.org/10.1007/11560548_4

Christiaan Pieter Rudolf Baaij. 2015. *Digital circuits in ClaSH: functional specifications and type-directed synthesis*. info:eu-repo/semantics/doctoralThesis. University of Twente, Enschede. https://doi.org/10.3990/1.9789036538039

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1999. Lava: hardware design in Haskell. *ACM SIGPLAN Notices* 34, 1 (Jan. 1999), 174–184. https://doi.org/10.1145/291251.289440

Richard J Boulton, Andrew D Gordon, Michael JC Gordon, John Harrison, John Herbert, and John Van Tassel. 1992. Experience with embedding hardware description languages in HOL.. In *TPCD*, Vol. 10. 129–156.

Edwin Brady, James Mckinna, and Kevin Hammond. 2007. Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types. In *Trends in Functional Programming 2007*.

Thomas Braibant. 2011. Coquet: A Coq Library for Verifying Hardware. In *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Number 7086 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 330–345. http://link.springer.com/chapter/10.1007/978-3-642-25379-9_24

C. Brown and G. Hutton. 1994. Categories, allegories and circuit design. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*. 372–381. https://doi.org/10.1109/LICS.1994.316052

Gang Chen. 2012. A Short Historical Survey of Functional Hardware Languages. *International Scholarly Research Notices* 2012 (2012), 1–11.

Solange Coupet-Grimal and Line Jakubiec. 2004. Certifying circuits in type theory. *Formal aspects of computing* 16, 4 (2004), 352–373.

]retro G. Érdi. [n. d.]. *Retrocomputing with Clash: Haskell for FPGA Hardware Design*.

Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. 2009. Introducing Kansas Lava. In *Proceedings of the Symposium on Implementation and Application of Functional Languages (LNCS, Vol. 6041)*. Springer-Verlag.

John Launchbury, Jeffrey R. Lewis, and Byron Cook. 1999. On embedding a microarchitectural design language within Haskell. *ACM SIGPLAN Notices* 34, 9 (Sept. 1999), 60–69. https://doi.org/10.1145/317765.317784

John Matthews, Byron Cook, and John Launchbury. 1998. Microprocessor Specification in Hawk. In *In Proceedings of the 1998 International Conference on Computer Languages*. IEEE Computer Society, Los Alamitos, CA, USA, 90–101. https://doi.org/10.1109/ICCL.1998.674160

T. Melham. 1993. *Higher Order Logic and Hardware Verification*. Cambridge Tracts in Theoretical Computer Science, Vol. 31. Cambridge University Press. https://doi.org/10.1017/CBO9780511569845

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph. D. Dissertation. Chalmers University of Technology.

Christine Paulin-Mohring. 1995. Circuits as streams in Coq: Verification of a sequential multiplier. In *International Workshop on Types for Proofs and Programs*. Springer, 216–230.

Joao Paulo Pizani Flor, Yorick Sijsling, and Wouter Swierstra. 2016. Π-Ware : Hardware Description and Verification in Agda. In *21th International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Tarmo Uustalu (Ed.).

I Sander and A Jantsch. 2004. System modeling and transformational design refinement in ForSyDe [formal system design]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23, 1 (Jan. 2004), 17–32. https://doi.org/10.1109/TCAD.2003.819898

Mary Sheeran. 1984. muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM Press, 104–112. https://doi.org/10.1145/800055.802026

M Sheeran. 2005. Hardware Design and Functional Programming: a Perfect Match. (2005). http://www.jucs.org/jucs_11_7/hardware_design_and_functional/jucs_11_7_1135_1158_sheeran.pdf

S. Singh. 2004. Designing reconfigurable systems in Lava. In *VLSI Design, 2004. Proceedings. 17th International Conference on*. 299–306. https://doi.org/10.1109/ICVD.2004.1260941

Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Paul van der Walt and Wouter Swierstra. 2012. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*. Springer, 157–173.