An Internalist Approach to Correct-by-Construction Compilers

Alberto Pardo
Universidad de la República
Montevideo, Uruguay
pardo@fing.edu.uy

Emmanuel Gunther
FAMAF – UNC and CONICET
Córdoba, Argentina
gunther@famaf.unc.edu.ar

Miguel Pagano
FAMAF - Universidad Nacional de Córdoba
Córdoba, Argentina
pagano@famaf.unc.edu.ar

Marcos Viera
Universidad de la República
Montevideo, Uruguay
mviera@fing.edu.uy

ABSTRACT

In this paper we present a methodology to organize the construction of a correct compiler, taking advantage of the power of full dependently type systems. The basic idea consists in decorating the abstract syntax of languages with their semantics, allowing to express the correctness of the compiler at type level. We show our methodology in a first small example and then explore how it can be promoted to more realistic languages, realizing that our internalistic approach is feasible for defining a correct-by-construction compiler from an imperative language with conditional iteration to a stack based intermediate language. We also show how this methodology can be combined with the externalist approach, compiling from the intermediate language to an assembly-like low level code and separately proving its correctness.

CCS CONCEPTS

• **Theory of computation** → *Logic and verification*; *Program verification*; • **Software and its engineering** → *Compilers*;

KEYWORDS

compiler design, dependent types, dependently-typed programming indexed families, correct-by-construction, ornaments

ACM Reference Format:
Alberto Pardo, Emmanuel Gunther, Miguel Pagano, and Marcos Viera. 2018. An Internalist Approach to Correct-by-Construction Compilers. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236950.3236965

1 INTRODUCTION

As hardly anyone programs in a machine language, one crucial tool of every computer system is the compiler (or interpreter). Therefore, a bug in the compiler can invalidate functional properties proven in high-level programs as they are compiled to a low-level language. There is an enormous literature about how to prove a compiler correct; pioneered by McCarthy and Painter [13] in 1967, and with a complete certified compiler for a large subset of C [7] as an outstanding milestone, usually the compiler is proven correct after being defined as a function mapping well-formed (this may imply well-typedness) programs to the target language. This relatively simply minded specification of the compiler leaves open the possibility of involuntary mistakes, that would not be caught by type-checking the compiler, but only on the verification process. Our proposal is to take full-advantage of dependently typed languages to give a more informative specification, that is a more fine grained type for the compiler: its type will encode its correctness.

One possibility to refine the specification of the compiler is a mapping from source programs to low-level code *together* with a proof of preservation of semantics. Let us remark that this is not so different than verifying the compiler after writing it, one just do both steps at the same time.

McBride introduced, in his ICFP 2012 keynote [12], an alternative to avoid the explicit proof of correctness, by building a type system for the low-level code in such a way that the type of an instruction is decorated by its operational semantics. If one can relate the semantics of the high-level language with this type system, then the specification of the compiler simply maps high-level programs to typed instructions. The main goal of this paper is to show this route is a feasible methodology to structure the development of correct-by-construction compilers.

In Section 2 we introduce our approach through the construction of a correct compiler from a very simple expressions language to a stack machine language. Then, in Section 3, we extend the language to an imperative language with iteration and variables, and apply our internalist methodology on it. In Section 4 we apply an hybrid approach in order to be able to compile to a low-level target language whose semantics can not be defined in a syntax directed way. Finally, in Section 5 we relate our approach to the theory of ornaments and conclude.

2 THE APPROACH

In this section we introduce the basis of our approach to compiler correctness by showing the development of a compiler and its correctness proof for a simple language of expressions. This first example provides the sufficient elements to explain, in rather simple terms, the essentials of our approach and its differences

$$\vdash n : \mathsf{nat} \qquad \frac{\vdash e_1 : \mathsf{nat} \qquad \vdash e_2 : \mathsf{nat}}{\vdash e_1 \oplus e_2 : \mathsf{nat}} \qquad \frac{\vdash e_1 : \mathsf{nat} \qquad \vdash e_2 : \mathsf{nat}}{\vdash e_1 \overset{\circ}{=} e_2 : \mathsf{bool}}$$

Figure 1: Typing rules for the source language

with the usual approaches to compiler correctness. We perform our development in Agda.

2.1 The languages

We want to develop a compiler for the following language of expressions:

$$e ::= \; n \;\mid\; e_1 \oplus e_2 \;\mid\; e_1 \overset{\circ}{=} e_2$$

According to this abstract syntax definition, an expression can be a constant, or the addition or equality comparison of two expressions. The following is its Agda representation:

```
data Expr : Set where
  |_|   : ℕ ⟶ Expr
  _⊕_ : (e₁ : Expr) ⟶ (e₂ : Expr) ⟶ Expr
  _≗_ : (e₁ : Expr) ⟶ (e₂ : Expr) ⟶ Expr
```

We define a notion of typing associated to this language where expressions can be of two types: nat and bool. We then impose natural type restrictions by defining the type system shown in Figure 1. The type system itself can be represented in Agda by defining the following relation:

```
data Type : Set where
  nat : Type
  bool : Type

data ⊢_:_ : Expr ⟶ Type ⟶ Set where
  tnat  : ∀ {n} ⟶ ⊢ | n | : nat
  tplus : ∀ {e₁ e₂} ⟶ ⊢ e₁ : nat ⟶ ⊢ e₂ : nat ⟶ ⊢ e₁ ⊕ e₂ : nat
  teq   : ∀ {e₁ e₂} ⟶ ⊢ e₁ : nat ⟶ ⊢ e₂ : nat ⟶ ⊢ e₁ ≗ e₂ : bool
```

Notice the use of the Type datatype for type representations.

Compiler correctness is usually understood as the problem of proving the semantic preservation of programs through compilation. This naturally leads to the provision of a formal semantics definition for each of the languages. In the case of the source language the definition of a formal semantics is a trivial task and we give it in terms of a functional semantics. We start with an interpretation for the representation of types:

```
𝒯⟦_⟧ : Type ⟶ Set
𝒯⟦ nat ⟧  = ℕ
𝒯⟦ bool ⟧ = Bool
```

To define the semantics for expressions we can benefit from having defined a type system and assign meaning only for those expressions that are well typed. We then define the functional semantics as a type-indexed function that requires the provision of a proof that the input expression is well typed.

$$st \vdash \mathsf{push}\ n \rightsquigarrow \mathsf{nat} :: st$$

$$\mathsf{nat} :: \mathsf{nat} :: st \vdash \mathsf{add} \rightsquigarrow \mathsf{nat} :: st$$

$$\mathsf{nat} :: \mathsf{nat} :: st \vdash \mathsf{eq} \rightsquigarrow \mathsf{bool} :: st$$

$$\frac{st \vdash c_1 \rightsquigarrow st' \qquad st' \vdash c_2 \rightsquigarrow st''}{st \vdash c_1, c_2 \rightsquigarrow st''}$$

Figure 2: Type system for target language

```
ℰ⟦_⟧_ : ∀ {t} ⟶ (e : Expr) ⟶ ⊢ e : t ⟶ 𝒯⟦ t ⟧
ℰ⟦ | n | ⟧ tnat            = n
ℰ⟦ e₁ ⊕ e₂ ⟧ (tplus p₁ p₂) = ℰ⟦ e₁ ⟧ p₁ + ℰ⟦ e₂ ⟧ p₂
ℰ⟦ e₁ ≗ e₂ ⟧ (teq p₁ p₂)   = ℰ⟦ e₁ ⟧ p₁ ==ₙ ℰ⟦ e₂ ⟧ p₂
```

Now we turn to the description of the target language of our compiler which is a standard low-level code that runs on a stack machine. The set of instructions is given by the following abstract syntax:

$$c ::= \; \mathsf{push}\ v \;\mid\; \mathsf{add} \;\mid\; \mathsf{eq} \;\mid\; c_1, c_2$$

The machine is able to push a value on top of the runtime stack and to add or compare for equality the two natural numbers located at the top of the stack. Instructions can also be composed in sequence. The following is the Agda representation of the abstract syntax:

```
data Code : Set where
  push : ℕ ⟶ Code
  add  : Code
  eq   : Code
  _,_  : Code ⟶ Code ⟶ Code
```

Like for the source language, we define a type system for the machine code (Figure 2). Besides imposing some natural type restrictions on a code this type system has the important feature of checking *stack-safety* in order to prevent stack underflows. For instance, the operations of addition and equality can only be performed when there are two natural numbers at the top of the stack.

The typing judgement is of the form $st \vdash c \rightsquigarrow st'$, where st, st' are stack types (lists with the types of the elements contained in the runtime stack). A typing judgement $st \vdash c \rightsquigarrow st'$ states the typing requirements on st, the stack type before the execution of a code c, and which is the stack type st' that results from the execution of that code. The type system can be represented in Agda by the following relation:

```
StackType : Set
StackType = List Type

data _⊢_⇝_ : StackType ⟶ Code ⟶ StackType ⟶ Set where
  tpush : ∀ {st} {n : ℕ} ⟶ st ⊢ push n ⇝ (nat :: st)
  tadd  : ∀ {st} ⟶ (nat :: nat :: st) ⊢ add ⇝ (nat :: st)
  teq   : ∀ {st} ⟶ (nat :: nat :: st) ⊢ eq ⇝ (bool :: st)
  tseq  : ∀ {st st' st''} {c₁ c₂} ⟶
            st ⊢ c₁ ⇝ st' ⟶ st' ⊢ c₂ ⇝ st'' ⟶ st ⊢ c₁ , c₂ ⇝ st''
```

$$\langle \text{push}\, n, s\rangle \Downarrow_{\mathcal{M}} n \rhd s$$

$$\langle \text{add}, n \rhd m \rhd s\rangle \Downarrow_{\mathcal{M}} (n + m) \rhd s$$

$$\langle \text{eq}, n \rhd m \rhd s\rangle \Downarrow_{\mathcal{M}} (n \equiv m) \rhd s$$

$$\frac{\langle c_1, s\rangle \Downarrow_{\mathcal{M}} s' \qquad \langle c_2, s'\rangle \Downarrow_{\mathcal{M}} s''}{\langle c_1, c_2, s\rangle \Downarrow_{\mathcal{M}} s''}$$

Figure 3: Operational semantics of the target language

An operational semantics of the machine code is presented in Figure 3. The transition relation is written as

$$\langle c, s\rangle \Downarrow_{\mathcal{M}} s'$$

meaning that the evaluation of code c in an inital stack s terminates with a final stack s'. Pushing a value v on a stack s is written as $v \rhd s$. Notice that the runtime stack may contain both natural numbers and boolean values.

The transition relation is partial as it is defined only for codes and stacks that satisfy the stack requirements. Therefore, when representing the semantics in Agda we will proceed as with expressions and associate meaning only to those instructions that are well typed.

Since they hold both natural numbers and boolean values, stacks can be seen as a sort of heterogeneous lists. One possible implementation of such a structure is to keep information about the type of each element contained in the stack. We will do so by indexing the type of stacks with a stack type.

```
data Stack : (st : StackType) → Set where
  ε : Stack []
  _▷_ : ∀ {t} {st} → 𝒯⟦ t ⟧ → Stack st → Stack (t ::  st)
```

The semantics relation can be implemented as follows:

```
data SemC : ∀ {st st'} → (c : Code) → st ⊢ c ⇝ st' →
                          Stack st → Stack st' → Set where
  pushR : ∀ {n st} {s : Stack st} →
             SemC (push n) tpush s (n ▷ s)
  addR : ∀  {m n st} {s : Stack st} →
             SemC add tadd (m ▷ (n ▷ s)) ((m + n) ▷ s)
  eqR   : ∀  {m n st} {s : Stack st} →
             SemC eq teq (m ▷ (n ▷ s)) ((m ==ₙ n) ▷ s)
  seqR  : ∀  {c₁ c₂ st st' st''} {p1 p2}
             {s : Stack st} {s' : Stack st'} {s'' : Stack st''} →
             SemC c₁ p1 s s' → SemC c₂ p2 s' s'' →
             SemC (c₁ , c₂) (tseq p1 p2) s s''
```

2.2 The compiler and its correctness proof

After having introduced the source and target languages we are ready to define the compiler and formulate its correctness property.[1] If the source language is a typed language, like is our case, then it is reasonable to only consider well-typed input programs for compilation. We then proceed as with the semantics and require the provision of a proof witnessing the well-typedness of the expression to be compiled. We can think of this requirement as the result of a previous phase of type checking.

```
compile : ∀ {t} (e : Expr) → ⊢ e :  t → Code
compile ∣ n ∣ tnat = push n
compile (e1 ⊕ e₂) (tplus p1 p2)
  = compile e1 p1 , (compile e₂ p2 , add)
compile (e1 ≗ e₂) (teq p1 p2)
  = compile e1 p1 , (compile e₂ p2 , eq)
```

This compiler simply describes the standard translation of expressions into reverse polish notation. It satisfies the following type-preservation property:

```
typepres : ∀ {t} {st} {e : Expr} →
             (tp : ⊢ e :  t) → st ⊢ compile e tp ⇝ (t ::  st)
```

This property states two relevant facts: (i) the compilation of an expression e of type t results in a code, which is typable in any input stack type st; and (ii) the resulting code leaves a value of the same type t at the top of the outcoming stack.

Based on this property the correctness of the compiler can then be formulated as follows:

```
correct : ∀ {t st} {e : Expr} {tp : ⊢ e :  t} {s : Stack st} →
            SemC (compile e tp) (typepres tp) s (ℰ⟦ e ⟧ tp ▷ s)
```

This states that the result of executing the code obtained from compiling a well-typed expression e starting with an arbitrary stack s is the final stack v ▷ s, where v is the result of evaluating e. This property is proved by a straightforward induction on e.

Let us now analyze the methodological approach we followed for designing this simple compiler and formulating its correctness. The definition of the building blocks of the compiler (abstract syntax of the languages, typing relations, semantic definitions, the compilation function itself) was completely standard. As usual, each building block was implemented by a separate definition, which of course may depend on the others (e.g., the typing relation uses the abstract syntax, the semantics uses the abstract syntax and the typing relation, etc.). An effect of this standard design is that both type-preservation and correctness need to be formulated as *external* predicates that relate the compilation function with the type systems of the languages in the former (typepres) and with their semantic definitions in the latter (correct). This means that these two properties are not being enforced by construction. On the contrary, they can only be checked once the compiler is completed. In other words, proving these properties is a sort of *program verification*. This approach to program design is usually referred to as an *externalist approach* [6].

In the following subsections we will present the different steps towards an *internalist approach* to compiler design in which type-preservation and correctness are enforced by construction.

2.3 Internalization of typing

In this subsection we start the description of a different approach to program design, usually referred to as the *internalist approach* [6], in which invariants are internalized into type definitions. We will

[1]We will sometimes refer to it as *semantic correctness* in order to stress that we mean correctness in the sense of semantic preservation of programs by the compiler.

refer to this process also as *ornamentation* as it has a strong relationship with the notion of *ornaments* introduced by McBride [5, 11], also referred to as *refinement* by other authors [2, 3, 6]. As a first approximation we deal with the internalization of the typing relation into the abstract syntax. This allows us to write a compiler that satisfies type-preservation by construction.

Concerning the source language, we can easily internalize the typing relation into the abstract syntax by defining the type of *well-typed expressions*. This type is given by an indexed datatype Expr$_t$ whose index represents the type of the expression being modeled by the term of the datatype.

```
data Exprₜ : Type ⟶ Set where
  |_|   : ℕ ⟶ Exprₜ nat
  _⊕_  : (e₁ : Exprₜ nat) ⟶ (e₂ : Exprₜ nat) ⟶ Exprₜ nat
  _≗_  : (e₁ : Exprₜ nat) ⟶ (e₂ : Exprₜ nat) ⟶ Exprₜ bool
```

The typing judgement e : Expr$_t$ t in Agda then corresponds to the judgement ⊢ e : t in the formal type system. We say that Expr$_t$ is an ornamented version of Expr. The definition of Expr$_t$ is immediate because the type system defined for expressions (Figure 1) is syntax-directed. This means that for each constructor in Expr$_t$ there is a corresponding typing rule. This is a key aspect for the success of the internalization process.

The internalization of the typing relation can then be formalized by the following lifting function:

```
_↑ₜ_ : ∀ {t} ⟶ (e : Expr) ⟶ ⊢ e : t ⟶ Exprₜ t
| x | ↑ₜ tnat              = | x |
(e₁ ⊕ e₂) ↑ₜ tplus p₁ p₂ = (e₁ ↑ₜ p₁) ⊕ (e₂ ↑ₜ p₂)
(e₁ ≗ e₂) ↑ₜ teq p₁ p₂   = (e₁ ↑ₜ p₁) ≗ (e₂ ↑ₜ p₂)
```

We can now redefine the functional semantics to work directly on well-typed expressions:

```
ℰₜ⟦_⟧ : ∀ {t} ⟶ Exprₜ t ⟶ 𝒯⟦ t ⟧
ℰₜ⟦ | n | ⟧     = fnat n
ℰₜ⟦ e₁ ⊕ e₂ ⟧ = fplus ℰₜ⟦ e₁ ⟧ ℰₜ⟦ e₂ ⟧
ℰₜ⟦ e₁ ≗ e₂ ⟧ = feq   ℰₜ⟦ e₁ ⟧ ℰₜ⟦ e₂ ⟧
```

For reasons that will be clarified later, we make explicit the algebraic nature of the interpretation by defining it in terms of a semantic algebra:

```
fnat : ℕ ⟶ ℕ
fnat n = n

fplus : ℕ ⟶ ℕ ⟶ ℕ
fplus n₁ n₂ = n₁ + n₂

feq : ℕ ⟶ ℕ ⟶ Bool
feq n₁ n₂ = n₁ ==ₙ n₂
```

It is immediate to see that the following property holds:

```
semequivExpr : ∀ {t} {e : Expr} {p : ⊢ e : t} ⟶ ℰ⟦ e ⟧ p ≡ ℰₜ⟦ e ↑ₜ p ⟧
```

Now we apply the same procedure to the low-level code and define an ornamented version of Code, an indexed datatype Code$_t$ that captures well-typed instructions. In agreement with the typing relation this datatype is indexed by two stack types representing the stack types before and after the execution of each instruction. Therefore, in this new datatype we can only express instructions that satisfy the typing requirements and thus are stack-safe.

```
data Codeₜ : StackType ⟶ StackType ⟶ Set where
  push : ∀ {st} ⟶ ℕ ⟶ Codeₜ st (nat :: st)
  add  : ∀ {st} ⟶ Codeₜ (nat :: nat :: st) (nat :: st)
  eq   : ∀ {st} ⟶ Codeₜ (nat :: nat :: st) (bool :: st)
  _,_  : ∀ {st st' st''} ⟶
           Codeₜ st st' ⟶ Codeₜ st' st'' ⟶ Codeₜ st st''
```

Again, this definition is possible because the typing relation (Figure 2) is syntax-directed.

Like for expressions, we can express the internalization process by a lifting function:

```
_↑ₜ_ : ∀ {st st'} ⟶ (c : Code) ⟶ st ⊢ c ⇝ st' ⟶ Codeₜ st st'
push n ↑ₜ tpush = push n
add    ↑ₜ tadd  = add
eq     ↑ₜ teq   = eq
(c₁ , c₂) ↑ₜ (tseq p₁ p₂) = c₁ ↑ₜ p₁ , c₂ ↑ₜ p₂
```

The last step is to adapt the semantics so it works on well-typed instructions directly:

```
data SemCₜ : ∀ {st st'} ⟶ Codeₜ st st' ⟶
                 Stack st ⟶ Stack st' ⟶ Set where
  push : ∀ {st n} {s : Stack st} ⟶
             SemCₜ (push n) s (n ▷ s)
  add  : ∀ {m n st} {s : Stack st} ⟶
             SemCₜ add (m ▷ (n ▷ s)) ((m + n) ▷ s)
  eq   : ∀ {m n st} {s : Stack st} ⟶
             SemCₜ eq (m ▷ (n ▷ s)) ((m ==ₙ n) ▷ s)
  seq  : ∀ {st st' st'' c₁ c₂}
           {s : Stack st} {s' : Stack st'} {s'' : Stack st''} ⟶
           SemCₜ c₁ s s' ⟶ SemCₜ c₂ s' s'' ⟶ SemCₜ (c₁ , c₂) s s''
```

The following property then holds:

```
semequivCode : ∀ {st st'} {c : Code} {tp : st ⊢ c ⇝ st'}
                 {s : Stack st} {s' : Stack st'} ⟶
                 SemC c tp s s' ⟶ SemCₜ (c ↑ₜ tp) s s'
```

Once we have internalized the typing relation in the definition of both languages we are in conditions to redefine the compiler such that now it focuses on the translation between the types of well-typed terms:

```
compileₜ : ∀ {t} {st} ⟶ Exprₜ t ⟶ Codeₜ st (t :: st)
compileₜ | n |       = push n
compileₜ (e₁ ⊕ e₂) = compileₜ e₁ , (compileₜ e₂ , add)
compileₜ (e₁ ≗ e₂) = compileₜ e₁ , (compileₜ e₂ , eq)
```

It is interesting to see that, apart from expressing the compilation action itself, the function compile$_t$ stands for an inductive proof of *type preservation*. In fact, its type corresponds to the formulation of type preservation between the ornamented types of the languages. The type of compile$_t$ states, in addition, that the compiler generates stack-safe code as it is the only kind of code accepted by Code$_t$.

The internalization of the typing relations in the abstract syntax of the languages was an important step forward towards achieving the design of a correct-by-construction compiler. However, it is not enough as it just allows us to enforce type-preservation by construction. The type of $\mathsf{compile}_t$ does not prevent that we define, for example, the following erroneous compilation rule for addition as it is still type-preserving:

```
compileₜ (e₁ ⊕ e₂) = compileₜ e₁ , compileₜ e₂ , add , push 1 , add
```

Again, the problem is that the correctness predicate is external to the compilation function.

```
correctₜ : ∀ {t} {st} {e : Exprₜ t} {s : Stack st} ⟶
           SemCₜ (compileₜ e) s (ℰₜ⟦ e ⟧ ▷ s)
```

In the next subsection we take a further step and deal with the internalization of the semantics. This will be the final stage in the design of a correct-by-construction compiler.

2.4 Internalization of semantics

Our next step is to internalize the semantics with the aim at directly enforcing semantic correctness during compilation. We will proceed systematically as we did with the internalization of the typing relation. An interesting aspect of the methodology is that at each new stage the ornamentation is based on definitions given in the previous stage. It happened with the typing relation and now it will be the case with the definition of the semantics. It is not less important that the semantics of both languages has been given by syntax-directed definitions, something essential for ornamentation.

In the case of expressions we internalize the semantics by ornamenting the type of each well-typed expression with a value of the semantic domain (a natural number). The value attached at each constructor is computed by applying the corresponding operation of the semantic algebra.

```
data Exprₛ : ∀ {t} ⟶ 𝒯⟦ t ⟧ ⟶ Set where
  |_|  : (n : ℕ) ⟶ Exprₛ (fnat n)
  _⊕_ : ∀ {n₁ n₂} ⟶
          (e₁ : Exprₛ n₁) ⟶
          (e₂ : Exprₛ n₂) ⟶ Exprₛ (fplus n₁ n₂)
  _≗_ : ∀ {n₁ n₂} ⟶
          (e₁ : Exprₛ n₁) ⟶
          (e₂ : Exprₛ n₂) ⟶ Exprₛ (feq n₁ n₂)
```

The lifting function makes precise the meaning of the ornamentation.

```
_↑ₛ : ∀ {t} ⟶ (e : Exprₜ t) ⟶ Exprₛ ℰₜ⟦ e ⟧
| x |       ↑ₛ = | x |
(e₁ ⊕ e₂) ↑ₛ = (e₁ ↑ₛ) ⊕ (e₂ ↑ₛ)
(e₁ ≗ e₂) ↑ₛ = (e₁ ↑ₛ) ≗ (e₂ ↑ₛ)
```

A generic characterization of this kind of internalization using the algebra of a fold over the datatype to be ornamented is treated in [2, 3] under the name of a *refinement*.

Concerning the machine code, we decorate the type of each instruction with a pair of stacks representing the relation between the stacks before and after the execution of the instruction.

```
data Codeₛ : ∀ {st st'} ⟶ Stack st ⟶ Stack st' ⟶ Set where
  push : ∀ {st} {s : Stack st} ⟶ (n : ℕ) ⟶ Codeₛ s (n ▷ s)
  add  : ∀ {st} {s : Stack st} {m n} ⟶
           Codeₛ (m ▷ (n ▷ s)) ((m + n) ▷ s)
```

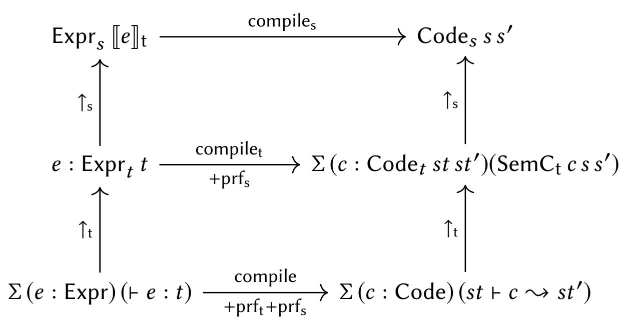
$$\begin{tikzcd}
\mathsf{Expr}_s\,[\![e]\!]_t \arrow[rr, "\mathsf{compile}_s"] & & \mathsf{Code}_s\,s\,s' \\
e : \mathsf{Expr}_t\,t \arrow[u, "\uparrow_s"] \arrow[rr, "\mathsf{compile}_t", "+\mathsf{prf}_s"'] & & \Sigma\,(c : \mathsf{Code}_t\,st\,st')(\mathsf{SemC}_t\,c\,s\,s') \arrow[u, "\uparrow_s"'] \\
\Sigma\,(e : \mathsf{Expr})\,(\vdash e : t) \arrow[u, "\uparrow_t"] \arrow[rr, "\mathsf{compile}", "+\mathsf{prf}_t{+}\mathsf{prf}_s"'] & & \Sigma\,(c : \mathsf{Code})\,(st \vdash c \rightsquigarrow st') \arrow[u, "\uparrow_t"']
\end{tikzcd}$$

Figure 4: The internalist approach

```
  eq  : ∀ {st} {s : Stack st} {m n} ⟶
          Codeₛ (m ▷ (n ▷ s)) ((m ==ₙ n) ▷ s)
  _,_ : ∀ {st st' st''}
          {s : Stack st} {s' : Stack st'} {s'' : Stack st''} ⟶
          Codeₛ s s' ⟶ Codeₛ s' s'' ⟶ Codeₛ s s''
```

The lifting of a well-typed code then results in a code with the same shape and whose semantics is reflected on its own type.

```
_↑ₛ_ : ∀ {st st'} {s : Stack st} {s' : Stack st'} ⟶
        (c : Codeₜ st st') ⟶ SemCₜ c s s' ⟶ Codeₛ s s'
push n ↑ₛ push     = push n
add    ↑ₛ add      = add
eq     ↑ₛ eq       = eq
(c₁ , c₂) ↑ₛ seq t₁ t₂ = c₁ ↑ₛ t₁ , c₂ ↑ₛ t₂
```

The benefit of this second degree of internalization is that it allows us to deal with the semantics of the represented terms at the type level. With this new feature at hand we can now express the correctness property of the compiler in its own type.

```
compileₛ : ∀ {t} {st} {v : 𝒯⟦ t ⟧} {s : Stack st} ⟶
             (e : Exprₛ v) ⟶ Codeₛ s (v ▷ s)
compileₛ | n |      = push n
compileₛ (e₁ ⊕ e₂) = compileₛ e₂ , compileₛ e₁ , add
compileₛ (e₁ ≗ e₂) = compileₛ e₂ , compileₛ e₁ , eq
```

Similar to $\mathsf{compile}_t$, in addition to expressing the obvious term translation between the languages, $\mathsf{compile}_s$ represents the steps of an inductive proof of the correctness of the compiler. A remarkable aspect of the type level encoding of this property is that *the verification of correctness is reduced to type checking*. Effectively, it is the type-checker of Agda who certifies that this property is being satisfied by each of the equations of the compiler.

Lifting the semantics to the type level has a methodological impact in compiler design because it allows us to reason about the correctness of the compiler during its own construction. Supported by the features of Agda for developing programs interactively, it is thus possible to detect (either typing or semantic) mistakes performed at early stages of the development as they will reduce to type errors identified by Agda's type system.

Observe that the internalization of the semantics (as it already occurred with the typing relation) did not have any effect on the shape of the compilation equations. Apart from being defined on different data types (that only differ in their decorations), the Agda code of the three versions of the compilation function looks the same. This is not a specific attribute of this particular compiler, but a general consequence of the ornamentation method. Modulo some adjustments that each particular case may require, the same will happen for more complex compilers.

Figure 4 summarizes the relationship between the different versions of the compiler where we explicit the additional, external proof obligations regarding typing (prf_t) and correctness (prf_s) that some of them require. The coherence of this diagram relies heavily on the correctness of the lifting functions which are the ones that ultimately explain the effect of each ornamentation. Those functions structurally traverse the terms of datatypes in order to compute their ornamentations. They are essentially copy functions at the value level which compute the type-level decorations in a compositional way by using information contained in the own traversed terms together with supplementary data (e.g. a proof term of a typing relation, etc.) with identical structure, if necessary. They satisfy to be a bijection between the product of the original datatype and the supplementary data and the ornamented datatype; the inverse of a lifting is a forgetful map that includes a function that erases the type-level decorations. Their structural nature make lifting functions feasible to be derived automatically (see e.g. [5, 6]).

Summing up, the application of the internalist approach to compiler design described in this section allows us to pass from a design methodology based on the development of a *verified compiler* to one based on the development of a *correct-by-construction compiler*.

3 COMPILING CORRECTLY A SIMPLE IMPERATIVE LANGUAGE

In this section we apply the methodology presented in the previous section to construct a correct compiler for a simple imperative language. One could think that nothing is to be learned passing from the toy language of expressions to the toy language of statements, but there is something to learn: we have to deal with non-termination and functional indexing.

3.1 High-Level Language

The expressions language is extended to include variables.

```
data Expr : Set where
  |_| : ℕ → Expr
  _⊕_ : (e1 : Expr) → (e2 : Expr) → Expr
  _≗_ : (e1 : Expr) → (e2 : Expr) → Expr
  var  : Var → Expr
```

The statements are standard: assignment, sequencing and a *while* constructor.

```
data Stmt : Set where
  _:=_       : Var → Expr → Stmt
  _,_        : Stmt → Stmt → Stmt
  while_do_ : Expr → Stmt → Stmt
```

3.2 Typing the High-Level Language

The typing relation for expressions is extended to include the new construction. For simplicity we only have nat variables.

```
data ⊢_:_ : Expr → Type → Set where
  tnat  : ∀ {n} → ⊢ | n | : nat
  tplus : ∀ {e1 e2} → ⊢ e1 : nat → ⊢ e2 : nat →
                      ⊢ e1 ⊕ e2 : nat
  teq   : ∀ {e1 e2} → ⊢ e1 : nat → ⊢ e2 : nat →
                      ⊢ e1 ≗ e2 : bool
  tvar  : ∀ {x} → ⊢ var x : nat
```

In the case of statements, the typing relation states that expressions assigned to variables have to be of type nat and that the condition expression of a while has to be of type bool.

```
data ⊢_ : Stmt → Set where
  tassgn : ∀ {x} {e} → ⊢ e : nat → ⊢ (x := e)
  tseq    : ∀ {stmt1 stmt2} → ⊢ stmt1 → ⊢ stmt2 →
                               ⊢ (stmt1 , stmt2)
  twhile : ∀ {e} {stmt} → ⊢ e : bool → ⊢ stmt →
                          ⊢ (while e do stmt)
```

As in the previous section, we ornament both categories of the abstract syntax with the typing information.

```
data Expr_t : Type → Set where
  |_| : ℕ → Expr_t nat
  _⊕_ : (e1 : Expr_t nat) → (e2 : Expr_t nat) → Expr_t nat
  _≗_ : (e1 : Expr_t nat) → (e2 : Expr_t nat) → Expr_t bool
  var  : (x : Var) → Expr_t nat

data Stmt_t : Set where
  _:=_        : Var → Expr_t nat → Stmt_t
  while_do_ : Expr_t bool → Stmt_t → Stmt_t
  _,_         : Stmt_t → Stmt_t → Stmt_t
```

The well-typedness of statements consists of their well-formedness, therefore the Stmt_t type is not indexed. The lifting functions are straightforward.

```
_↑_t_ : ∀ {t} → (e : Expr) → ⊢ e : t → Expr_t t
| x | ↑_t tnat = | x |
(e1 ⊕ e2) ↑_t tplus p1 p2 = (e1 ↑_t p1) ⊕ (e2 ↑_t p2)
(e1 ≗ e2) ↑_t teq p1 p2   = (e1 ↑_t p1) ≗ (e2 ↑_t p2)
var x      ↑_t tvar         = var x

_↑stmt_t_ : (stmt : Stmt) → ⊢ stmt → Stmt_t
_↑stmt_t_ (x := e) (tassgn enat) = x := (e ↑_t enat)
_↑stmt_t_ (while e do stmt) (twhile ebool p)
                          = while e ↑_t ebool do (stmt ↑stmt_t p)
_↑stmt_t_ (stmt1 , stmt2)     (tseq p1 p2)
                          = (stmt1 ↑stmt_t p1) , (stmt2 ↑stmt_t p2)
```

3.3 Semantics of the High-Level Language

Given that the expressions of the imperative language include variables, the semantics has to be defined in terms of a state mapping variables to naturals.

```
State : Set
State = Var → ℕ
```

```
_[_⟶_] : State ⟶ Var ⟶ ℕ ⟶ State
σ [ x ⟶ n ] = λ y ⟶ if y ==_s x then n else σ y
```

The semantic domain of an expression of type t is defined as a function from a state s to $\mathcal{T}[\![\text{t}]\!]$, the interpretation of the type.

```
DomE_s : Type ⟶ Set
DomE_s t = (σ : State) ⟶ 𝒯⟦ t ⟧
```

The functions defining the semantic algebra have to consider a state, which is explicitly used to return the values of variables.

```
fcnat : ℕ ⟶ DomE_s nat
fcnat n = const n

fplus : DomE_s nat ⟶ DomE_s nat ⟶ DomE_s nat
fplus f1 f2 = λ σ ⟶ f1 σ + f2 σ

f== : DomE_s nat ⟶ DomE_s nat ⟶ DomE_s bool
f== f1 f2 = λ σ ⟶ f1 σ ==_n f2 σ

fvar : Var ⟶ DomE_s nat
fvar x = λ σ ⟶ σ x
```

We can now ornament the abstract syntax (already ornamented with types) with the semantic information given by the semantic algebra.

```
data Expr_s : ∀ {t} ⟶ DomE_s t ⟶ Set where
  |_| : (n : ℕ) ⟶ Expr_s (fcnat n)
  _⊕_ : ∀ {f1 f2} ⟶ (e1 : Expr_s f1) ⟶ (e2 : Expr_s f2) ⟶
                     Expr_s (fplus f1 f2)
  _≗_ : ∀ {f1 f2} ⟶ (e1 : Expr_s f1) ⟶ (e2 : Expr_s f2) ⟶
                     Expr_s (f== f1 f2)
  var  : (x : Var) ⟶ Expr_s (fvar x)
```

As usual, the semantics of statements is defined as state transformers. The ornamented version of the low-level code of the previous section was indexed with a relation, relating the initial configuration to the configuration reached after executing the instruction. Let us now try to apply the same idea for statements by defining a big-step semantics. As can be seen it is a standard semantics; the only particularity is the auxiliary relation $\Rightarrow_w$ which is needed to have a syntax-directed definition. Notice that the auxiliary relation captures the usual two rules for the semantics of while.

```
mutual
  data _,_⇒_ : (s : Stmt_t) ⟶ State ⟶ State ⟶ Set where
    assgn : ∀ {x e σ} ⟶ (x := e) , σ ⇒ (σ [ x ⟶ (ℰ_t⟦ e ⟧ σ) ])
    seq   : ∀ {s1 s2 σ σ0 σ'} ⟶ s1 , σ ⇒ σ0 ⟶
                                 s2 , σ0 ⇒ σ' ⟶
                                 (s1 , s2) , σ ⇒ σ'
    while : ∀ {e s σ σ'} ⟶ ⟨ ℰ_t⟦ e ⟧ σ , e ⟩ s , σ ⇒_w σ' ⟶
                            (while e do s) , σ ⇒ σ'

  data ⟨_,_⟩_,_⇒_w_ : Bool ⟶ Expr_t bool ⟶ Stmt_t ⟶
                       State ⟶ State ⟶ Set where
    wFSem : ∀ {e s σ} ⟶ ⟨ false , e ⟩ s , σ ⇒_w σ
    wTSem : ∀ {s σ σ' σ'' e} ⟶ s , σ ⇒ σ' ⟶
                                 ⟨ ℰ_t⟦ e ⟧ σ' , e ⟩ s , σ' ⇒_w σ'' ⟶
                                 ⟨ true , e ⟩ s , σ ⇒_w σ''
```

As soon as we try to ornament the AST with the semantic relation we encounter problems; for a starter, how can we fill the hole ({! !}) in the definition of the while$_s$ constructor?

```
data Stmt_s : State ⟶ State ⟶ Set where
  _:=_   : ∀ {σ} {f : DomE_s nat} ⟶ (x : Var) ⟶ (e : Expr_s f) ⟶
             Stmt_s σ (σ [ x ⟶ (f σ) ])
  _,_    : ∀ {σ σ' σ''} ⟶
           Stmt_s σ σ' ⟶ Stmt_s σ' σ'' ⟶ Stmt_s σ σ''
  while_s : ∀ {σ σ'} {f : DomE_s bool} ⟶
           (e : Expr_s f) ⟶ {! !} ⟶ Stmt_s σ σ'
```

We cannot ornament the body with any pair of initial and final states, because we would be fixing them, when they are supposed to change in each iteration. An alternative is to alter the nice correspondence between the bare ASTs and their ornamented counterpart by having as a body of while$_s$ a bare term of type Stmt$_t$ and a proof for the semantics of the whole loop (from which we can extract the correct index σ'). We prefer to keep closer to the philosophy of ornaments and avoid this alternative, because it modifies the structure of the constructor and imposes the burden of carrying a proof as a subterm.

An important insight can be obtained if we prove that the semantics is functional (even though it is partial), hence we can move to a functional big-step semantics as advocated in [16] and ornament statements with that functional semantics. The partiality of the big-step relation arises from having general recursion and we must deal with non-termination also with the functional indices.

We use a finite approximation of the least fix-point whose accuracy is parameterized by a natural number (a *clock*), counting how many unfoldings of the loop will be done.[2] If the index comes to zero, then the semantics is undefined; it is only defined if the least fix-point is reached before the clock goes to zero. The partial approximation of the least fix-point can be defined in general using the Maybe type to represent undefinedness.

Thus, we define the semantic domain as a function from the clock and the initial state to a (possible) final state.

```
DomS_s : Set
DomS_s = (clock : ℕ) ⟶ (σ : State) ⟶ Maybe State
```

The denotational semantics of a statement is a state transformer with signature:

```
𝒮_t⟦_⟧_|_ : Stmt_t ⟶ DomS_s
```

defined in terms of the following semantic algebra.

```
fassgn : Var ⟶ DomE_s nat ⟶ DomS_s
fassgn x fe = λ clock σ ⟶ just (σ [ x ⟶ fe σ ])
```

In the case of assignment the state is updated by associating to the variable x the result of evaluating the expression e.

```
fseq : DomS_s ⟶ DomS_s ⟶ DomS_s
fseq f1 f2 = λ clock σ ⟶ f1 clock σ »= f2 clock
```

[2] Alternatively, one could use the Delay Monad to give the semantics of while using a fix-point operator as done by Benton et al.[4].

The semantics of the sequence is defined as the semantics of s_2 transforming the state resulting from the semantics of s_1.

```
fwhile : DomE_s bool → DomS_s → DomS_s
fwhile fb fc zero          σ = nothing
fwhile fb fc (suc clock) σ = if fb σ then fc (suc clock) σ »=
                                  fwhile fb fc clock
                              else  just σ
```

The definition of while is recursive, executing the body while the condition is true and the clock is greater than zero.

In order to internalize the semantics, the datatype is indexed with the semantic domain.

```
data Stmt_s : DomS_s → Set where
  _:=_     : ∀ {f} → (x : Var) → Expr_s f → Stmt_s (fassgn x f)
  _,_      : ∀ {f₁ f₂} → Stmt_s f₁ → Stmt_s f₂ → Stmt_s (fseq f₁ f₂)
  while_do_ : ∀ {fb} {f} → Expr_s fb → Stmt_s f →
                    Stmt_s (fwhile fb f)
```

Again, the lifting functions are straightforward. We only show their signatures.

```
_↑s : ∀ {t} → (e : Expr_t t) → Expr_s ℰ_t⟦ e ⟧
_↑stmt_s : (stmt : Stmt_t) → Stmt_s (𝒮_t⟦ stmt ⟧_|_)
```

And we can go from the original abstract syntax trees to the fully ornamented ones by composing the lifting functions.

```
_↑ts_ : ∀ {t} → (e : Expr) → (p : ⊢ e :  t) →
                Expr_s (ℰ_t⟦ e ↑t p ⟧)
e ↑ts p = (e ↑t p) ↑s

_↑stmt_ts_ : (stmt : Stmt) → (p : ⊢ stmt) →
             Stmt_s (𝒮_t⟦ stmt ↑stmt_t p ⟧_|_ )
stmt ↑stmt_ts p = (stmt ↑stmt_t p) ↑stmt_s
```

3.4 Intermediate Code

It is clear that the set of instructions for the arithmetic fragment cannot cope with the imperative language, so we add instructions for dealing with a store and loops.

$$c ::= \text{push } v \mid \text{add} \mid \text{equ} \mid c_1, c_2 \mid \text{load } x \mid \text{store } x \mid \text{loop}[c_1, c_2]$$

We include an instruction for loading the content of a variable to the top of the stack and its counterpart for storing the natural value at the top of the stack into a variable. We also add a looping instruction loop[c_1, c_2], where c_1 is a code that pushes a boolean at the top of the stack and c_2 is a code that represents the body of the loop. The *loop* executes c_1, removes the boolean from the top of the stack and stops if the boolean is *false*, or it executes c_2 and recursively loops otherwise.

```
data LCode : Set where
  push : (n : ℕ) → LCode
  add  : LCode
  eq   : LCode
  load : (x : Var) → LCode
  store : (x : Var) → LCode
  loop : (c₁ : LCode) → (c₂ : LCode) → LCode
  _,_  : (c₁ : LCode) → (c₂ : LCode) → LCode
```

Notice that in the low-level language there is no distinction between expressions and statements.

3.5 Stack-Safety

We extend the type system with the rules for the new operations. For load we state that the stack is extended with a nat. The operation store can only be applied if there is a nat at the top of the stack, which is removed after its execution. In the case of loop, the stack type is not modified, given that c_1 extends the stack with a bool and c_2 does not modify it. Notice that this is a quite restrictive condition, which is nevertheless fulfilled by compiled code.

```
data _⊢_⇝_ : StackType → LCode → StackType → Set where
  tpush : ∀ {st} {n : ℕ} → st ⊢ push n ⇝ (nat :: st)
  tadd  : ∀ {st} → (nat :: nat :: st) ⊢ add ⇝ (nat :: st)
  teq   : ∀ {st} → (nat :: nat :: st) ⊢ eq ⇝ (bool :: st)
  tload : ∀ {st} {x : Var} → st ⊢ load x ⇝ (nat :: st)
  tstore : ∀ {st} {x : Var} → (nat :: st) ⊢ store x ⇝ st
  tloop : ∀ {st} {c₁ c₂} → st ⊢ c₁ ⇝ (bool :: st) →
                  st ⊢ c₂ ⇝ st →
                  st ⊢ loop c₁ c₂ ⇝ st
  tseq  : ∀ {st st' st''} {c₁ c₂} → st ⊢ c₁ ⇝ st' →
                      st' ⊢ c₂ ⇝ st'' →
                      st ⊢ c₁ , c₂ ⇝ st''
```

The ornamented abstract syntax encoding the typing relation is extended with the new rules.

```
data LCode_t : StackType → StackType → Set where
  push : ∀ {st} → (n : ℕ) → LCode_t st (nat :: st)
  add  : ∀ {st} → LCode_t (nat :: nat :: st) (nat :: st)
  eq   : ∀ {st} → LCode_t (nat :: nat :: st) (bool :: st)
  load : ∀ {st} → (x : Var) → LCode_t st (nat :: st)
  store : ∀ {st} → (x : Var) → LCode_t (nat :: st) st
  loop : ∀ {st} → (c₁ : LCode_t st (bool :: st)) →
              (c₂ : LCode_t st st) →
              LCode_t st st
  _,_  : ∀ {st st' st''} → (c₁ : LCode_t st st') →
                  (c₂ : LCode_t st' st'') →
                  LCode_t st st''
```

We omit the definition of the lifting function because it is, again, straightforward.

```
_↑c_t_ : ∀ {st st'} → (c : LCode) → st ⊢ c ⇝ st' → LCode_t st st'
```

3.6 Semantics of the Intermediate Code

The configurations of the abstract machine now consist of pairs (s,σ), where σ : State is the state and s : Stack *st* is a stack typed by the stack-type *st*; thus configurations can also be thought as indexed by stack-types.

```
Conf : (st : StackType) → Set
Conf st = Stack st × State
```

As in the case of statements, we define denotational semantics with a clock to prevent infinite looping.

```
DomCₛ : (st : StackType) ⟶ (st' : StackType) ⟶ Set
DomCₛ st st' = (clock : ℕ) ⟶ (sσ : Conf st) ⟶ Maybe (Conf st')
```

The functions of the semantic algebra have to deal with the clock and the configurations.

```
fpush : ∀ {st} ⟶ (n : ℕ) ⟶ DomCₛ st (nat ∷ st)
fpush n = λ {clock (s , σ) ⟶ just ((n ▷ s) , σ)}
fadd : ∀ {st} ⟶ DomCₛ (nat ∷ nat ∷ st) (nat ∷ st)
fadd = λ {clock ((n ▷ (m ▷ s)) , σ) ⟶ just (((n + m) ▷ s) , σ)}
feq : ∀ {st} ⟶ DomCₛ (nat ∷ nat ∷ st) (bool ∷ st)
feq = λ {clock ((n ▷ (m ▷ s)) , σ) ⟶ just (((n ==ₙ m) ▷ s) , σ)}

fseqc : ∀ {st st' st''} ⟶ DomCₛ st st' ⟶ DomCₛ st' st'' ⟶
                          DomCₛ st st''
fseqc f₁ f₂ = λ clock sσ ⟶ f₁ clock sσ »= f₂ clock
```

The semantics of load consists on getting the value of the variable x in the state σ and pushing it on the stack. In the case of store, the natural value n is taken from the top of the stack and stored in the state.

```
fload : ∀ {st} ⟶ (x : Var) ⟶ DomCₛ st (nat ∷ st)
fload x = λ {clock (s , σ) ⟶ just ((σ x ▷ s) , σ)}

fstore : ∀ {st} ⟶ (x : Var) ⟶ DomCₛ (nat ∷ st) st
fstore x = λ {clock ((n ▷ s) , σ) ⟶ just (s , σ [ x ⟶ n ])}
```

The semantics of loop is defined recursively. When the clock is greater than zero the code of the condition is executed. If the condition pushes true on the top of the stack then the body is executed and loop is applied recursively with the clock decreased by one. If the condition pushes false then the current configuration is returned.

```
floop : ∀ {st} ⟶ DomCₛ st (bool ∷ st) ⟶ DomCₛ st st ⟶
                 DomCₛ st st
floop      fb fc zero        sσ = nothing
floop {st} fb fc (suc clock) sσ = fb (clock + 1) sσ »= fnext
  where fnext : Conf (bool ∷ st) ⟶ Maybe (Conf st)
        fnext (true ▷ s' , σ') = fc (clock + 1) (s' , σ') »=
                                  floop fb fc clock
        fnext (false ▷ s' , σ') = just (s' , σ')
```

In this case we add an extra constructor to the ornamented type, which allows us to change the index of an ornamented term with a function that is extensionally equal to the original index. This constructor, called substCₛ, is explained in the context of the compiler.

```
EqSem : ∀ {st st'} ⟶ DomCₛ st st' ⟶ DomCₛ st st' ⟶ Set _
EqSem f g = ∀ clock sσ ⟶ f clock sσ ≡ g clock sσ

data LCodeₛ : ∀ {st st'} ⟶ DomCₛ st st' ⟶ Set where
  push : ∀ {st} ⟶ (n : ℕ) ⟶ LCodeₛ (fpush {st} n)
  add  : ∀ {st} ⟶ LCodeₛ (fadd {st})
  eq   : ∀ {st} ⟶ LCodeₛ (feq {st})
  load  : ∀ {st} ⟶ (x : Var) ⟶ LCodeₛ (fload {st} x)
  store : ∀ {st} ⟶ (x : Var) ⟶ LCodeₛ (fstore {st} x)
  loop  : ∀ {st} {f₁ f₂} ⟶ (c₁ : LCodeₛ f₁) ⟶
                            (c₂ : LCodeₛ f₂) ⟶
                            LCodeₛ (floop {st} f₁ f₂)
  _,_   : ∀ {st st' st''} {f₁ f₂} ⟶
          (c₁ : LCodeₛ f₁) ⟶ (c₂ : LCodeₛ f₂) ⟶
          LCodeₛ (fseqc {st} {st'} {st''} f₁ f₂)
  substCₛ : ∀ {st st'} {f g} ⟶ LCodeₛ {st} {st'} f ⟶
                                EqSem f g ⟶ LCodeₛ g
```

3.7 The Correct Compiler

The compiler for expressions comes with no surprise with respect to the one defined in the previous section.

```
compExprₛ : ∀ {t} {f} {st} ⟶ Exprₛ {t} f ⟶
                LCodeₛ {st} (λ {clock (s , σ) ⟶ just ((f σ ▷ s) , σ) })
compExprₛ ∣ n ∣ = push n
compExprₛ (e₁ ⊕ e₂) = compExprₛ e₂ , (compExprₛ e₁ , add)
compExprₛ (e₁ ≐ e₂) = compExprₛ e₂ , (compExprₛ e₁ , eq)
compExprₛ (var x) = load x
```

The correctness of the compiler for statements is slightly more contrived, because the clock is relevant. The type of the compiledSem function describes a situation identified in the early literature about compiler correctness [15, 17]: correctness involves a homomorphism between the semantics of the respective languages.

```
compiledSem : ∀ {st} ⟶ DomSₛ ⟶ DomCₛ st st
compiledSem f clock (s , σ) = f clock σ »= (λ σ' ⟶ just (s , σ'))
```

In fact, compiledSem is the link between a semantic function of the source language and its counterpart in the semantics of the target language. This function encodes the correctness of the compiler: should it be nonsensical, correctness would not be correct.

```
compStmtₛ : ∀ {f} {st} ⟶ Stmtₛ f ⟶ LCodeₛ (compiledSem {st} f)
compStmtₛ (x := e) = compExprₛ e , store x
compStmtₛ (_,_ {f₁} {f₂} stmt₁ stmt₂)
  = (compStmtₛ stmt₁ , compStmtₛ stmt₂) *
  where _* : LCodeₛ _ ⟶ _
        c * = substCₛ c (eqseq f₁ f₂)
compStmtₛ (while_do_ {fb} {f} eb stmt)
  = (loop (compExprₛ eb) (compStmtₛ stmt)) *
  where _* : LCodeₛ _ ⟶ _
        c * = substCₛ c (eqloop fb f)
```

The substCₛ constructor also plays an important role in the definition of the compiler for statements. When we want to define the compiler and the translated index of a term is definitionally equal to the computed index of the generated code, the Agda type-checker transparently ensure the correction. But if those indices are not definitionally, but propositionally equal, we need to provide a witness for the equality. Therefore, we should prove that two semantic indices are equal, and as in general they are functions, there are two options: (a) we might postulate extensionality and uses the subst operator associated with the propositional equality of the standard library; or (b) we introduce a substitution constructor of propositionally equal indices. Note that in the high-level language we did not introduce such a constructor. This is because this language is not the target of any compilation; but one can imagine a first pass that perform some optimization at the high level, then we might have need to introduce a subst constructor also in Stmt$_s$.

In our case, we have a proof obligation for sequences and one for loops:

```
eqseq : ∀ {st} f₁ f₂ ⟶ EqSem {st} {st}
          (fseqc (compiledSem f₁) (compiledSem f₂))
          (compiledSem (λ clock σ ⟶ f₁ clock σ »= f₂ clock))
eqloop : ∀ {st} fb f ⟶ EqSem {st} {st}
          (floop (λ { clock (s , σ) ⟶ just ((fb σ ▷ s) , σ) })
                      (compiledSem f))
          (compiledSem (fwhile fb f))
```

4 GENERATING LOW-LEVEL CODE

In this section we generate low-level code from the intermediate language of the previous section. Our target language is an idealized assembler for a machine that has a read-only memory for code. The semantics of the machine is given by a relation specifying the change of the program counter and the effect on the memory.

As we have stressed, the internalist approach can only be used when the property to ornament the data-type is syntax-directed; unfortunately, it is not clear if one can give a syntax-directed semantics for our low-level language. For this reason we cannot apply the methodology proposed in the previous sections; we show that it is still feasible to prove the correctness of the compiler with respect to the intermediate language ornamented by its semantics. In fact, we show that the mathematical functions defining the semantics of each constructor can be implemented in the low-level language. Although this may look as a shortcoming of our methodology, a realistic compiler is a composition of smaller compilers between intermediate languages. In this sense, the correctness of the first compiler to the intermediate LCode language makes the methodology worthy enough.

Low-Level Language. The instruction set is standard, although instead of named labels, the destinations of jumps are specified by relative indices to decrease or increase the program counter.

```
data Shift : Set where left right : ℕ ⟶ Shift
data Instr : Set where
  nop add eq : Instr
  push : ℕ ⟶ Instr
  jmp jz : (l : Shift) ⟶ Instr
  load store :   (x : Var) ⟶ Instr
Assm : Set
Assm = List Instr
```

Execution environment. A configuration of the machine consists of the program counter, an untyped stack (booleans are modelled as naturals), and a state. We do not include the code in the configuration because it is read-only.

```
Machine : Set
Machine = ℕ × State × List ℕ
pc : Machine ⟶ ℕ
pc (lbl , _ , _) = lbl
```

In the previous sections we used typing disciplines to prevent some classes of bad behaviours, for our Assm language fewer guarantees can be determined statically. The well-definedness of jumps (the target location is a defined index for the current code) is an example of a property that could be statically enforced. We opted to give a small-step semantics where jumping instructions are always successful, if the target is not defined, then the machine get stucks after the jump.

```
fjmp : Shift ⟶ Machine ⟶ Machine
fjmp (left i)   (k , σ , s) = k − i , σ , s
fjmp (right i) (k , σ , s) = k + i , σ , s
```

The following relation specifies the transition produced by a single instruction. Notice that the only instructions that cannot progress are those that try to use data from the stack.

```
data __⇝ᵢ_ : Instr ⟶ Machine ⟶ Machine ⟶ Set where
  nop : ∀ {k σ s} ⟶  nop  (k , σ , s) ⇝ᵢ (k +1 , σ , s)
  push : ∀ {k σ s n} ⟶  push n  (k , σ , s) ⇝ᵢ (k +1 , σ , n ::  s)
  add   : ∀ {k σ s n₁ n₂} ⟶
                add  (k , σ , n₁  ::  n₂  ::  s) ⇝ᵢ (k +1 , σ , n₁ + n₂ ::  s)
  eq     : ∀ {k σ s n₁ n₂} ⟶
                eq  (k , σ , n₁  ::  n₂  ::  s) ⇝ᵢ (k +1 , σ , n₁ ==ₙ n₂ ::  s)
  jmp   : ∀ {cf sh} ⟶  jmp sh  cf ⇝ᵢ (fjmp sh cf)
  jz0    : ∀ {k σ s sh} ⟶
                jz sh  (k , σ , 0  ::  s) ⇝ᵢ (fjmp sh (k , σ , s))
  jzs     : ∀ {k σ s sh n} ⟶
                jz sh  (k , σ , n + 1  ::  s) ⇝ᵢ (k +1 , σ , s)
  load   : ∀ {k σ s x} ⟶
                load x  (k , σ , s) ⇝ᵢ  (k +1 , σ , σ x ::  s)
  store : ∀ {k σ s x n} ⟶
                store x  (k , σ , n ::  s) ⇝ᵢ (k +1 , σ [ x ⟶ n ] , s)
```

Given a code and a configuration the machine makes a transition only if the instruction at the program counter can progress. The transition semantics is the transitive closure of the single step relation.

```
data __⇝_ (c : Assm) : Machine ⟶ Machine ⟶ Set where
  step : ∀ {i cf cf'} ⟶ (c ‼ pc cf ≡ just i) ⟶
             i  cf ⇝ᵢ  cf' ⟶  c  cf ⇝  cf'
__⇝*_ : (is : Assm) ⟶ Machine ⟶ Machine ⟶ Set
 is  cf ⇝*  cf' = Tr ( is _⇝_) cf cf'
```

Given two code fragments corresponding respectively to the computation of the guard and the body of a loop, we can easily combine them to produce the code corresponding to the loop.

```
loopAssm isB isC = isB ++ [ jz le ] ++ isC ++ [ jmp li ]
  where le = right (2 + length isC)
            li  = left (1 + length isB + length isC)
```

$$\begin{tikzcd}
\mathsf{LCode_s}\ f \arrow[rrd, "\mathsf{comp{+}prf_{sj}}"] & & \\
c : \mathsf{LCode_t}\ st\ st' \arrow[u, "\uparrow_s"] & & \\
\Sigma\,(c : \mathsf{LCode})\,(st \vdash c \rightsquigarrow st') \arrow[u, "\uparrow_t"] \arrow[rr, "\mathsf{comp}", "+\mathsf{prf_{sj}}"'] & & \Sigma\,(is : \mathsf{Assm})\,(is \vdash cf \rightsquigarrow cf')
\end{tikzcd}$$

Figure 5: Generating low-level code

The affirmation that loopAssm *isB isC* corresponds to a looping combinator will be justified once we prove the correctness of the compiler.

Generating low-level code. In this part we use the semantic decoration of the intermediate ASTs of LCode in order to develop an externalistic correct compiler targeting Assm. We prove the denotational semantics of LCode to be computational adequate with respect to the small-step semantics of Assm; in fact, we prove a slightly more general result that allows us to combine code compositionally.

As can be seen in Figure 5 we have two choices to compile from LCode to Assm. Either the input are bare ASTs and we prove the correctness only for well-typed programs, which corresponds to the horizontal arrow; alternatively, the compiler only accepts decorated ASTs, the diagonal arrow. In both cases, we have to prove externally the correctness; we argue that the latter approach is cleaner.

In order to formally express the adequacy we need to match the booleans of LCode with its representation in Assm as naturals.

```
unty : {t : Type } ⟶ 𝒯⟦ t ⟧ ⟶ ℕ
unty {nat} n = n
unty {bool} false = 0
unty {bool} true = 1
unty* : ∀ {st} ⟶ Stack st ⟶ List ℕ
unty* ε = []
unty* (v ▷ stack) = unty v ::  unty* stack
```

We can say that a low-level code *realizes* a mathematical function if any proof that the function is defined for a given high-level configuration is mapped into traces from the initial configuration to the final configuration.

```
realizes : ∀ {st st'} ⟶ DomCₛ st st' ⟶ Assm ⟶ Set
realizes f is = ∀ {cl s s' σ σ'} ⟶ f cl (s , σ) ≡ just (s' , σ') ⟶
  is (0 , σ , unty* s) ⇝* (length is , σ' , unty* s')
```

Now we can understand that the obligation $\mathsf{prf_{sj}}$ of the diagonal arrow corresponds exactly to a proof that the Assm code generated by the compiler compL realizes the index of the ornamented ASTs.

```
compL : ∀ {st st'} {f : DomCₛ st st'} ⟶ LCodeₛ f ⟶ Assm
compL (push n)   = [ push n ]
compL add        = [ add ]
compL eq         = [ eq ]
compL (loop b c) = loopAssm (compL b) (compL c)
compL (c₀ , c₁)     = compL c₀ ++ compL c₁
compL (load x)      = [ load x ]
compL (store x)     = [ store x ]
compL (substCₛ c _) = compL c
```

We invite the reader to discover why one cannot prove the following naïve statement of correctness.

```
prfₛⱼ : ∀ {st st'} {f : DomCₛ st st'} ⟶ (lc : LCodeₛ f) ⟶
  realizes f (compL lc )
```

The remedy is evident, one needs to generalize the statement of realization to take into account some possible context on which the compiled code is generated.

```
ctx-realizes : ∀ {st st'} ⟶ DomCₛ st st' ⟶ Assm ⟶ Set
ctx-realizes f is = ∀ cl s {s'} σ {σ'} {γ γ'} ⟶
 f cl (s , σ) ≡ just (s' , σ') ⟶
  γ ++ is ++ γ' (length γ , σ , unty* s) ⇝*
                 (length (γ ++ is) , σ' , unty* s')
```

One may prove by induction on lc : LCodeₛ f that the generate code realizes contextually f.

```
ctx-prfₛⱼ : ∀ {st st'} {f : DomCₛ st st'} ⟶ (lc : LCodeₛ f) ⟶
  ctx-realizes f (compL lc)
```

This proof is mostly tedious because one needs several bureaucratic lemmas in order to combine traces parameterized by different (but propositional equal) codes. Obviously, the original version of correctness can be recovered by letting the prefix and the suffix be the empty list.

Notice that we can state the correctness of the end-to-end compiler by saying that for each well-typed program p the generated code *realizes* the semantics of p:

```
realizesWhile : (p : Stmt) ⟶ ⊢ p ⟶ Assm ⟶ Set
realizesWhile p tp is = ∀ {σ σ' cl} ⟶
 (⟦ p ↑stmtₜ tp ⟧ₛ cl | σ) ≡ just σ' ⟶
   is (0 , σ , []) ⇝* (length is , σ' , [])
```

In order to get the compiler from the high-level language targeting the low-level and its correctness proof we compose the correct compiler from Section 3 with the compiler from LCode to Assm. Notice that the evidence that the bare program p is well-typed can be generated by a type-checker.

```
compiler : (p : Stmt) ⟶ (tp : ⊢ p) ⟶
              Σ[ c ∈ Assm ] (realizesWhile p tp c)
compiler p tp = compL lcode , correct
  where
    lcode = compStmtₛ {st = []} (p ↑stmtₜₛ tp)
    semp = ⟦ p ↑stmtₜ tp ⟧ₛ_|_
    correct : realizesWhile p tp (compL lcode)
    correct equ = prfₛⱼ lcode (prfₛ semp equ)
```

The term $\mathsf{prf_s}$ ... construct a proof that compileSem semp is defined whenever semp is also defined.

At this point one can see that the end-to-end compiler consists of climbing up on the first column of Fig. 4, then use the internalist compiler $comp_s$ and finally get correct Assembler code using the diagonal arrow of Fig. 5.

5 CONCLUSION

Discussion. Ornaments were first proposed by McBride in [11] as a principled way to work with ADTs decorated by different properties. As shown by McBride in that article (elaborating an example of McKinna [14]), ornaments can be used to define a correct compiler without a separate proof of correctness. Based on this theory, Ko and Gibbons [6] characterized McBride's approach as *internalist* in contrast with the more usual *externalist*, which is closer to verification. In the latter, constraints are defined as separate predicates, while in the former one uses inductive families embedding the constraint as indices of each constructor. The isomorphism between an externalist data-type (together with a predicate) and its internalist counterpart is called a *refinement*.

Ko and Gibbons proposed a methodology to obtain for free both versions instead of defining independently each of them. Indeed, neither the original datatype nor the internalized version are declared as indexed families; one *describes* both the data-type and their ornamentations by a universe of descriptions, along the lines of Martin-Löf's definition of a universe à la Tarski[10]. From these descriptions one freely gets the data-type, the predicate defining the constraint (for example, sortedness for lists), and also the refinement between the internal and the external versions of the data-type. Dagand and McBride [5] showed that functions defined over bare data-types can be lifted across ornaments.

Working with descriptions of data-types and ornaments is fairly tedious, as can be assessed by comparing the whole formalizations of compiling arithmetic expressions without using descriptions and using them which are available in the repository.[3] Transporting functions across ornaments is an incentive to use this formal device. Williams and his coauthors [18, 19] have proposed to extend ML with native support for ornaments, thus alleviating the syntactical burden.

Our methodology to construct correct compilers is based on decorating the *types* of the ASTs using the semantics of the corresponding language; i.e. for each language we define an inductive family indexed by its semantics. Having the semantic value of each constructor directly available, the compiler can be defined between those families: its well-typedness guarantees the preservation of semantics. The key point of our methodology is the isomorphism between the externalist and the internalist version of the datatypes, which ensures that we are decorating correctly.

Future work. We plan to study the possibility of giving an end-to-end compiler using ornaments. We also project to target a low language closer to a realistic assembler language. Another interesting future work line is to use our approach to implement optimizations. In this case the source and target languages are the same.

We have explored the use of a delay monad [1] as an alternative way of dealing with non-termination; it would be interesting to complete the whole development using it.

Those lines are all concerned with the preservation of semantics; we also think that the internalistic approach can be also applied fruitfully to get compiler preserving intensional aspects. The first author [9] has experienced using Haskell's GADTs to internalize non-interference in a simple language, and, more recently, Manzino [8] used Agda to define a non-interference compiler of a While language.

[3]https://gitlab.fing.edu.uy/CorrectCompiler/ppdp18.git

REFERENCES

[1] Andreas Abel and James Chapman. 2014. Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014. (EPTCS)*, Paul Levy and Neel Krishnaswami (Eds.), Vol. 153. 51–67. https://doi.org/10.4204/EPTCS.153.4

[2] Robert Atkey, Patricia Johann, and Neil Ghani. 2011. When Is a Type Refinement an Inductive Type?. In *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science)*, Martin Hofmann (Ed.), Vol. 6604. Springer, 72–87. https://doi.org/10.1007/978-3-642-19805-2_6

[3] Robert Atkey, Patricia Johann, and Neil Ghani. 2012. Refining Inductive Types. *Logical Methods in Computer Science* 8, 2 (2012). https://doi.org/10.2168/LMCS-8(2:9)2012

[4] Nick Benton, Andrew Kennedy, and Carsten Varming. 2009. Some Domain Theory and Denotational Semantics in Coq. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Lecture Notes in Computer Science, Vol. 5674. Springer Berlin Heidelberg, 115–130.

[5] Pierre-Évariste Dagand and Conor McBride. 2014. Transporting functions across ornaments. *J. Funct. Program.* 24, 2-3 (2014), 316–383. https://doi.org/10.1017/S0956796814000069

[6] Hsiang-Shang Ko and Jeremy Gibbons. 2011. Modularising Inductive Families. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming (WGP '11)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2036918.2036921

[7] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. *SIGPLAN Not.* 41, 1 (Jan. 2006), 42–54. https://doi.org/10.1145/1111320.1111042

[8] Cecilia Manzino. 2018. *Security preserving program translations.* Master's thesis. PEDECIBA Informática, Universidad de la República, Uruguay.

[9] Cecilia Manzino and Alberto Pardo. 2014. A Security Types Preserving Compiler in Haskell. In *Programming Languages - 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings (Lecture Notes in Computer Science)*, Fernando Magno Quintão Pereira (Ed.), Vol. 8771. Springer, 16–30. https://doi.org/10.1007/978-3-319-11863-5_2

[10] Per Martin-Löf. 1984. *Intuitionistic Type Theory.* Bibliopolis.

[11] Conor McBride. 2011. Ornamental algebras, algebraic ornaments. *unpublished* (2011).

[12] Conor Thomas McBride. 2012. Agda-curious?: An Exploration of Programming with Dependent Types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/2364527.2364529

[13] John McCarthy and James Painter. 1967. Correctness of a compiler for arithmetic expressions. American Mathematical Society, 33–41.

[14] James McKinna and Joel Wright. 2006. A type-correct, stack-safe, provably correct, expression compiler. *Submitted to the Journal of Functional Programming* (2006).

[15] F Lockwood Morris. 1973. Advice on structuring compilers and proving them correct. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, 144–152.

[16] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

[17] James W Thatcher, Eric G Wagner, and Jesse B Wright. 1980. More on advice on structuring compilers and proving them correct. In *International Workshop on Semantics-Directed Compiler Generation.* Springer, 165–188.

[18] Thomas Williams, Pierre-Évariste Dagand, and Didier Rémy. 2014. Ornaments in practice. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, José Pedro Magalhães and Tiark Rompf (Eds.). ACM, 15–24. https://doi.org/10.1145/2633628.2633631

[19] Thomas Williams and Didier Rémy. 2018. A principled approach to ornamentation in ML. *PACMPL* 2, POPL (2018), 21:1–21:30. https://doi.org/10.1145/3158109