An Approach to Correct-by-Construction Compilers

Emmanuel Gunther[+] Miguel Pagano[+]

Alberto Pardo[*] Marcos Viera[*]

[+]FAMAF
Universidad Nacional de Córdoba
Argentina

[*]Instituto de Computación
Universidad de la República
Uruguay

Compiler correctness

- Compiler correctness has been the subject of research for a long time (algebraic approaches, categorical, calculational, type-theoretical, etc).
- Usual compiler correctness follows an *externalist* approach: first develop the languages and their semantics, write the compiler and finally establish its correctness.
- In this talk we present an *internalist* approach in the sense that we develop the compiler and its correctness proof simultaneously.
- Our development is in the context of dependently typed programming, using Agda.

Outline

1. We first develop a compiler for expressions following the usual *externalist* approach.
2. Then we show how the same compiler can be developed in an internalist way.
3. Finally, we present a correct-by-construction compiler for a simple While language.

Source language

Simple expressions

- Abstract syntax.

$$e ::= \quad n \quad | \quad e_1 \oplus e_2 \quad | \quad e_1 \mathrel{\mathring{=}} e_2$$

- Type system.

$$\vdash n : \mathsf{nat}$$

$$\frac{\vdash e_1 : \mathsf{nat} \qquad \vdash e_2 : \mathsf{nat}}{\vdash e_1 \oplus e_2 : \mathsf{nat}}$$

$$\frac{\vdash e_1 : \mathsf{nat} \qquad \vdash e_2 : \mathsf{nat}}{\vdash e_1 \mathrel{\mathring{=}} e_2 : \mathsf{bool}}$$

Agda representation

- Abstract syntax.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Expr}\ :\ \mathsf{Set}\ \textbf{where} \\
\quad |\_|\quad : \mathbb{N} \rightarrow \mathsf{Expr} \\
\quad \_\oplus\_\ : (e_1\ :\ \mathsf{Expr}) \rightarrow (e_2\ :\ \mathsf{Expr}) \rightarrow \mathsf{Expr} \\
\quad \_\mathrel{\hat{=}}\_\ : (e_1\ :\ \mathsf{Expr}) \rightarrow (e_2\ :\ \mathsf{Expr}) \rightarrow \mathsf{Expr}
\end{array}
$$

- Type system.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Type}\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{nat}\ :\ \mathsf{Type} \\
\quad \mathsf{bool}\ :\ \mathsf{Type}
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \vdash\_:\_\ :\ \mathsf{Expr} \rightarrow \mathsf{Type} \rightarrow \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{tnat}\ \ :\ \forall\ \{n\} \rightarrow\ \vdash |\ n\ | : \mathsf{nat} \\
\quad \mathsf{tplus}\ :\ \forall\ \{e_1\ e_2\} \rightarrow\ \vdash e_1 : \mathsf{nat} \rightarrow\ \vdash e_2 : \mathsf{nat} \rightarrow\ \vdash e_1 \oplus e_2 : \mathsf{nat} \\
\quad \mathsf{teq}\ \ \ :\ \forall\ \{e_1\ e_2\} \rightarrow\ \vdash e_1 : \mathsf{nat} \rightarrow\ \vdash e_2 : \mathsf{nat} \rightarrow\ \vdash e_1 \mathrel{\hat{=}} e_2 : \mathsf{bool}
\end{array}
$$

Typed expressions

- We can decorate the expression type with the type of expressions.

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Expr_t} & : & \mathsf{Type} \rightarrow \mathsf{Set}\ \textbf{where} \\
\quad |\_| & : & \mathbb{N} \rightarrow \mathsf{Expr_t}\ \mathsf{nat} \\
\quad \_\oplus\_ & : & (e_1 : \mathsf{Expr_t}\ \mathsf{nat}) \rightarrow (e_2 : \mathsf{Expr_t}\ \mathsf{nat}) \rightarrow \mathsf{Expr_t}\ \mathsf{nat} \\
\quad \_\mathrel{\hat{=}}\_ & : & (e_1 : \mathsf{Expr_t}\ \mathsf{nat}) \rightarrow (e_2 : \mathsf{Expr_t}\ \mathsf{nat}) \rightarrow \mathsf{Expr_t}\ \mathsf{bool}
\end{array}
$$

$$\vdash e : \sigma \quad \overset{\mathit{rep-by}}{\rightsquigarrow} \quad e : \mathsf{Expr_t}\ \sigma$$

Typed expressions

- We can decorate the expression type with the type of expressions.

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Expr_t} & : & \mathsf{Type} \to \mathsf{Set}\ \textbf{where} \\
\quad |\_| & : & \mathbb{N} \to \mathsf{Expr_t\ nat} \\
\quad \_\oplus\_ & : & (e_1 : \mathsf{Expr_t\ nat}) \to (e_2 : \mathsf{Expr_t\ nat}) \to \mathsf{Expr_t\ nat} \\
\quad \_\circeq\_ & : & (e_1 : \mathsf{Expr_t\ nat}) \to (e_2 : \mathsf{Expr_t\ nat}) \to \mathsf{Expr_t\ bool}
\end{array}
$$

$$\vdash e : \sigma \quad \overset{rep-by}{\rightsquigarrow} \quad e : \mathsf{Expr_t}\ \sigma$$

- Well-typed expressions can then be translated into typed expressions.

$$
\begin{array}{lll}
\_ \uparrow_t \_ : \forall \{t\} \to (e : \mathsf{Expr}) \to \vdash e : t \to \mathsf{Expr_t}\ t & & \\
|\,x\,| \uparrow_t \mathsf{tnat} & = & |\,x\,| \\
(e_1 \oplus e_2) \uparrow_t \mathsf{tplus}\ p_1\ p_2 & = & (e_1 \uparrow_t p_1) \oplus (e_2 \uparrow_t p_2) \\
(e_1 \circeq e_2) \uparrow_t \mathsf{teq}\ p_1\ p_2 & = & (e_1 \uparrow_t p_1) \circeq (e_2 \uparrow_t p_2)
\end{array}
$$

Semantics of expressions

- Interpretation of types.

$$
\begin{aligned}
[\![\_]\!]_T &: \text{Type} \to \text{Set} \\
[\![\text{nat}]\!]_T &= \mathbb{N} \\
[\![\text{bool}]\!]_T &= \text{Bool}
\end{aligned}
$$

- The Semantics.

$$
\begin{aligned}
[\![\_]\!] &: \forall \{t\} \to \text{Expr}_t\ t \to [\![t]\!]_T \\
[\![|\ n\ |]\!] &= \text{fnat}\ n \\
[\![e_1 \oplus e_2]\!] &= \text{fplus}\ [\![e_1]\!]\ [\![e_2]\!] \\
[\![e_1 \stackrel{\circ}{=} e_2]\!] &= \text{feq}\quad [\![e_1]\!]\ [\![e_2]\!]
\end{aligned}
$$

- Semantic algebra:

$$
\begin{aligned}
\text{fnat}\ n &= n \\
\text{fplus}\ n_1\ n_2 &= n_1 + n_2 \\
\text{feq}\quad n_1\ n_2 &= n_1 \equiv n_2
\end{aligned}
$$

Target language

Machine code

- We compile expressions into reverse polish notation.
- Syntax:

$$c ::= \quad push \;\; n \quad | \;\; add \;\; | \;\; eq \;\; | \;\; c_1 \, , c_2$$

- Type system: states well-typed code and stack safety.

  Judgement: $st \vdash c \rightsquigarrow st'$ (st, st' are stack types)

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

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

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

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

Agda representation

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

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''
```

Typed and stack-safe code

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Code_t} & : & \mathsf{StackType} \to \mathsf{StackType} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{push} & : & \forall\ \{\mathsf{st}\} \to (\mathsf{n}\ :\ \mathbb{N}) \to \mathsf{Code_t}\ \mathsf{st}\ (\mathsf{nat} :: \mathsf{st}) \\
\quad \mathsf{add} & : & \forall\ \{\mathsf{st}\} \to \mathsf{Code_t}\ (\mathsf{nat} :: \mathsf{nat} :: \mathsf{st})\ (\mathsf{nat} :: \mathsf{st}) \\
\quad \mathsf{eq} & : & \forall\ \{\mathsf{st}\} \to \mathsf{Code_t}\ (\mathsf{nat} :: \mathsf{nat} :: \mathsf{st})\ (\mathsf{bool} :: \mathsf{st}) \\
\quad \_,\_ & : & \forall\ \{\mathsf{st}\ \mathsf{st'}\ \mathsf{st''}\} \to \\
 & & \mathsf{Code_t}\ \mathsf{st}\ \mathsf{st'} \to \mathsf{Code_t}\ \mathsf{st'}\ \mathsf{st''} \to \mathsf{Code_t}\ \mathsf{st}\ \mathsf{st''}
\end{array}
$$

$$
\mathsf{st} \vdash \mathsf{c} \rightsquigarrow \mathsf{st'} \quad \overset{\textit{rep-by}}{\rightsquigarrow} \quad \mathsf{c}\ :\ \mathsf{Code_t}\ \mathsf{st}\ \mathsf{st'}
$$

Typed and stack-safe code

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Code_t} & : & \mathsf{StackType} \to \mathsf{StackType} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{push} & : & \forall \{\mathsf{st}\} \to (\mathsf{n} : \mathbb{N}) \to \mathsf{Code_t}\ \mathsf{st}\ (\mathsf{nat} :: \mathsf{st}) \\
\quad \mathsf{add} & : & \forall \{\mathsf{st}\} \to \mathsf{Code_t}\ (\mathsf{nat} :: \mathsf{nat} :: \mathsf{st})\ (\mathsf{nat} :: \mathsf{st}) \\
\quad \mathsf{eq} & : & \forall \{\mathsf{st}\} \to \mathsf{Code_t}\ (\mathsf{nat} :: \mathsf{nat} :: \mathsf{st})\ (\mathsf{bool} :: \mathsf{st}) \\
\quad \_,\_ & : & \forall \{\mathsf{st}\ \mathsf{st'}\ \mathsf{st''}\} \to \\
 & & \mathsf{Code_t}\ \mathsf{st}\ \mathsf{st'} \to \mathsf{Code_t}\ \mathsf{st'}\ \mathsf{st''} \to \mathsf{Code_t}\ \mathsf{st}\ \mathsf{st''}
\end{array}
$$

$$
\mathsf{st} \vdash \mathsf{c} \rightsquigarrow \mathsf{st'} \quad \overset{rep-by}{\rightsquigarrow} \quad \mathsf{c} : \mathsf{Code_t}\ \mathsf{st}\ \mathsf{st'}
$$

- From well-typed code to typed code.

$$
\begin{array}{llll}
\_ \uparrow_t \_ & : & \multicolumn{2}{l}{\forall \{\mathsf{st}\ \mathsf{st'}\} \to (\mathsf{c} : \mathsf{Code}) \to \mathsf{st} \vdash \mathsf{c} \rightsquigarrow \mathsf{st'} \to \mathsf{Code_t}\ \mathsf{st}\ \mathsf{st'}} \\
\mathsf{push}\ \mathsf{n} & \uparrow_t\ \mathsf{tpush} & = & \mathsf{push}\ \mathsf{n} \\
\mathsf{add} & \uparrow_t\ \mathsf{tadd} & = & \mathsf{add} \\
\mathsf{eq} & \uparrow_t\ \mathsf{teq} & = & \mathsf{eq} \\
(\mathsf{c}_1 , \mathsf{c}_2) & \uparrow_t\ \mathsf{tseq}\ \mathsf{p}_1\ \mathsf{p}_2 & = & (\mathsf{c}_1 \uparrow_t \mathsf{p}_1) , (\mathsf{c}_2 \uparrow_t \mathsf{p}_2)
\end{array}
$$

Big step semantics

Semantic relation: $\langle c, s\rangle \Downarrow_{\mathcal{M}} s'$ (s and s' are stacks)

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

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

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

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

Functional semantics in Agda

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

Functional semantics in Agda

$$\textbf{data}\ \text{Stack} : (\text{st} : \text{StackType}) \to \text{Set}\ \textbf{where}$$
$$\quad \epsilon : \text{Stack}\ []$$
$$\quad \_\triangleright\_ : \forall\ \{\text{t}\}\ \{\text{st}\} \to [\![\text{t}]\!]_T \to \text{Stack st} \to \text{Stack}\ (\text{t} :: \text{st})$$

$$\mathcal{C}[\![\_]\!] : \forall\ \{\text{st st'}\} \to \text{Code}_\text{t}\ \text{st st'} \to \text{Stack st} \to \text{Stack st'}$$
$$\mathcal{C}[\![\text{push n}]\!]\ \text{s} = \text{n} \triangleright \text{s}$$
$$\mathcal{C}[\![\text{add}]\!]\ (\text{n} \triangleright \text{m} \triangleright \text{s}) = \text{n} + \text{m} \triangleright \text{s}$$
$$\mathcal{C}[\![\text{eq}]\!]\ (\text{n} \triangleright \text{m} \triangleright \text{s}) = \text{n} \equiv \text{m} \triangleright \text{s}$$
$$\mathcal{C}[\![\text{c}_1 \,,\, \text{c}_2]\!]\ \text{s} = \mathcal{C}[\![\text{c}_2]\!]\ (\mathcal{C}[\![\text{c}_1]\!]\ \text{s})$$

Functional semantics in Agda

$$
\begin{aligned}
&\textbf{data}\ \text{Stack} : (\text{st} : \text{StackType}) \to \text{Set}\ \textbf{where}\\
&\quad \epsilon : \text{Stack}\ []\\
&\quad \_\triangleright\_ : \forall \{t\}\ \{st\} \to [\![t]\!]_T \to \text{Stack st} \to \text{Stack}\ (t :: st)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{C}[\![\_]\!] : \forall \{\text{st st'}\} \to \text{Code}_t\ \text{st st'} \to \text{Stack st} \to \text{Stack st'}\\
&\mathcal{C}[\![\text{push n}]\!]\ s &&= n \triangleright s\\
&\mathcal{C}[\![\text{add}]\!]\ (n \triangleright m \triangleright s) &&= n + m \triangleright s\\
&\mathcal{C}[\![\text{eq}]\!]\ (n \triangleright m \triangleright s) &&= n \equiv m \triangleright s\\
&\mathcal{C}[\![c_1\ ,\ c_2]\!]\ s &&= \mathcal{C}[\![c_2]\!]\ (\mathcal{C}[\![c_1]\!]\ s)
\end{aligned}
$$

- Semantic algebra:

$$
\begin{aligned}
&\text{fpush n} &&= \lambda\, s \to n \triangleright s\\
&\text{fadd} &&= \lambda\, (n \triangleright m \triangleright s) \to n + m \triangleright s\\
&\text{feq} &&= \lambda\, (n \triangleright m \triangleright s) \to n \equiv m \triangleright s\\
&\text{fseq}\ f_1\ f_2 &&= \lambda\, s \to f_2\ (f_1\ s)
\end{aligned}
$$

A type-correct compiler

We define a compiler from typed expressions to stack-decorated machine code.

$$
\begin{array}{lcl}
\text{compile} & : & \forall \{t\} \{st\} (e : \text{Expr}_t\ t) \rightarrow \text{Code}_t\ st\ (t :: st) \\
\text{compile}\ |\ n\ | & = & \text{push}\ n \\
\text{compile}\ (e_1 \oplus e_2) & = & \text{compile}\ e_1\ ,\ \text{compile}\ e_2\ ,\ \text{add} \\
\text{compile}\ (e_1 \mathrel{\hat{=}} e_1) & = & \text{compile}\ e_1\ ,\ \text{compile}\ e_2\ ,\ \text{eq}
\end{array}
$$

Type-preservation and stack-safety are enforced by construction.

Correctness

$$\forall \{t\} \{st\} \{e : \mathsf{Expr}_t\ t\} \{s : \mathsf{Stack}\ st\} \rightarrow \mathcal{C}[\![\mathsf{compile}\ e]\!]\ s \equiv [\![e]\!] \rhd s$$

Correctness

$$\forall \{t\} \{st\} \{e : \mathrm{Expr}_t\ t\} \{s : \mathrm{Stack}\ st\} \rightarrow \mathcal{C}[\![\mathrm{compile}\ e]\!]\ s \equiv [\![e]\!] \rhd s$$

- Proving correctness corresponds to *program verification*.
- In fact, under the *externalist approach* semantics preservation can be verified only after the compiler is finished.
- And therefore possible semantic mistakes are detected very late. For instance,

  $\mathrm{compile}\ (e_1 \oplus e_2) = \mathrm{compile}\ e_1\ ,\ \mathrm{compile}\ e_2\ ,\ \mathrm{add},\ \mathrm{push}\ 1\ ,\ \mathrm{add}$

The internalist approach

Semantic decoration: expressions

- In order to enforce semantics preservation during compiler construction we lift the semantics to the type level.
- The type of an expression is now indexed by a value of the semantic domain of the expression.
- Those values are calculated by applying the operations of the semantic algebra for expressions.

```
data Exprₛ : ∀ {t} → ⟦t⟧_T → Set where
  |_|   : (n : ℕ) → Exprₛ (fnat n)
  _⊕_   : ∀ {n₁ n₂} →
          (e₁ : Exprₛ n₁) →
          (e₂ : Exprₛ n₂) → Exprₛ (fplus n₁ n₂)
  _≗_   : ∀ {d₁ d₂} →
          (e₁ : Exprₛ n₁) →
          (e₂ : Exprₛ n₂) → Exprₛ (feq n₁ n₂)
```

Semantic decoration: expressions

- The lifting states that the type of an expression is decorated with the semantics of that expression.

$$
\begin{array}{lcl}
\_ \uparrow_s & : & \forall \{ t \} \rightarrow (e \; : \; \mathsf{Expr}_t \; t) \rightarrow \mathsf{Expr}_s \; [\![e]\!] \\
\mid x \mid \; \uparrow_s & = & \mid x \mid \\
(e_1 \oplus e_2) \; \uparrow_s & = & (e_1 \; \uparrow_s) \oplus (e_2 \; \uparrow_s) \\
(e_1 \doteq e_2) \; \uparrow_s & = & (e_1 \; \uparrow_s) \doteq (e_2 \; \uparrow_s)
\end{array}
$$

For code the type reflects the semantic action on stacks.

```
data Codeₛ : ∀ {st st'} → (Stack st → Stack st') → Set where
  push : ∀ {st} → (n : ℕ) → Codeₛ (fpush {st} n)
  add  : ∀ {st} → Codeₛ (fadd {st})
  eq   : ∀ {st} → Codeₛ (feq {st})
  _,_  : ∀ {st₀ st₁ st₂} {f₁} {f₂} →
         Codeₛ f₁ → Codeₛ f₂ →
         Codeₛ (fseq {st₀} {st₁} {st₂} f₁ f₂)
```

```
↑ₛ : ∀ {st st'} → (c : Codeₜ st st') → Codeₛ (𝒞⟦c⟧)
push n ↑ₛ   = push n
add ↑ₛ      = add
eq ↑ₛ       = eq
(c₁ , c₂) ↑ₛ = (c₁ ↑ₛ , c₂ ↑ₛ)
```

The correct-by-construction compiler

The correctness property now is directly expressed by type of the compiler.

$$
\begin{array}{lcl}
\text{comp} : \forall \{t\} \{v : [\![t]\!]_T\} \{st\} \rightarrow & & \\
\quad (e : \text{Expr}_s\ v) \rightarrow \text{Code}_s \{st\} (\lambda\, s \rightarrow v \triangleright s) & & \\
\text{comp}\ |\ n\ | & = & \text{push}\ n \\
\text{comp}\ (e_1 \oplus e_2) & = & \text{comp}\ e_2\ ,\ \text{comp}\ e_1\ ,\ \text{add} \\
\text{comp}\ (e_1 \stackrel{\circ}{=} e_2) & = & \text{comp}\ e_2\ ,\ \text{comp}\ e_1\ ,\ \text{eq}
\end{array}
$$

A compiler for a While language

Extended language

- We extend our source language with variables and statements.

$$e ::= x \mid n \mid e_1 \oplus e_2 \mid e_1 \mathrel{\dot{=}} e_2$$

$$S ::= x := e \mid \textbf{while}\ e\ \textbf{do}\ S \mid S_1 ; S_2$$

Our variables are only of type nat.

Extended language

- We extend our source language with variables and statements.

$$\begin{array}{rcl} e & ::= & x \mid n \mid e_1 \oplus e_2 \mid e_1 \stackrel{\circ}{=} e_2 \\ S & ::= & x := e \mid \textbf{while}\ e\ \textbf{do}\ S \mid S_1 ; S_2 \end{array}$$

  Our variables are only of type nat.

- We also extend the target language with new instructions.

$$\begin{array}{rcl} c & ::= & \textbf{push}\ n \mid \textbf{add} \mid \textbf{eq} \mid c_1 , c_2 \\ & & \textbf{load}\ x \mid \textbf{store}\ x \mid \textbf{loop}(c_1 , c_2) \end{array}$$

  In $\textbf{loop}(c_1, c_2)$, the code c_1 corresponds to a condition that leaves a boolean value on top of the stack whereas c_2 is the body of the iteration.

Abstract syntax

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

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

Type system

```
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
```

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

Semantics of statements

- To cope with nontermination we add a clock to control the depth of iterations.

$$
\begin{aligned}
\mathsf{Dom}_S &: \mathsf{Set} \\
\mathsf{Dom}_S &= (\mathsf{clock} : \mathbb{N}) \to (\sigma : \mathsf{State}) \to \mathsf{Maybe\ State}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\_]\!] : \mathsf{Stmt}_t \to \mathsf{Dom}_S \\
&[\![x := e]\!] = \mathsf{fassign}\ x\ [\![e]\!] \\
&[\![\mathsf{while}\ e\ \mathsf{do}\ \mathsf{stmt}]\!] = \mathsf{fwhile}\ [\![e]\!]\ [\![\mathsf{stmt}]\!] \\
&[\![\mathsf{stmt}_1\ ,\ \mathsf{stmt}_2]\!] = \mathsf{fseq}\ [\![\mathsf{stmt}_1]\!]\ [\![\mathsf{stmt}_2]\!]
\end{aligned}
$$

- When the clock reaches zero it means timeout and Nothing is returned.

- A nonterminating program on a certain state σ is a program that returns Nothing for every clock.

Semantic algebra

$$\begin{aligned}
&\text{fassign} \;:\; \text{Var} \to (\text{State} \to \mathbb{N}) \to \text{Dom}_S \\
&\text{fassign x fe} \;=\; \lambda\, \text{clock}\; \sigma \to \text{just}\; (\sigma\, [\, \text{x} \;\leftarrow\; \text{fe}\; \sigma\,])
\end{aligned}$$

$$\begin{aligned}
&\text{fwhile} \;:\; (\text{State} \to \text{Bool}) \to \text{Dom}_S \to \text{Dom}_S \\
&\text{fwhile fb fc zero}\; \sigma \;=\; \text{nothing} \\
&\text{fwhile fb fc (suc clock)}\; \sigma \\
&\quad = \text{if fb}\; \sigma\; \text{then fc (suc clock)}\; \sigma \;\gg\!=\; \text{fwhile fb fc clock} \\
&\qquad\qquad\quad \text{else just}\; \sigma
\end{aligned}$$

$$\begin{aligned}
&\text{fseq} \;:\; \text{Dom}_S \to \text{Dom}_S \to \text{Dom}_S \\
&\text{fseq}\; f_1\; f_2 \;=\; \lambda\, \text{clock}\; \sigma \to f_1\; \text{clock}\; \sigma \;\gg\!=\; f_2\; \text{clock}
\end{aligned}$$

Semantic decoration: statements

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Stmt_s} : \mathsf{Dom_S} \to \mathsf{Set}\ \textbf{where} \\
\quad \_\!:=\!\_ \ : \forall \{f\} \to (x : \mathsf{Var}) \to \mathsf{Expr_s}\ f \to \mathsf{Stmt_s}\ (\mathsf{fassign}\ x\ f) \\
\quad \mathsf{while\_do\_} : \forall \{fb\}\ \{f\} \to \\
\qquad\qquad \mathsf{Expr_s}\ fb \to \mathsf{Stmt_s}\ f \to \mathsf{Stmt_s}\ (\mathsf{fwhile}\ fb\ f) \\
\quad \_,\_ : \forall \{f_1\ f_2\} \to \mathsf{Stmt_s}\ f_1 \to \mathsf{Stmt_s}\ f_2 \to \mathsf{Stmt_s}\ (\mathsf{fseq}\ f_1\ f_2)
\end{array}
$$

$$
\begin{array}{l}
\_\ \uparrow_s\ : (\mathsf{stmt} : \mathsf{Stmt_t}) \to \mathsf{Stmt_s}\ [\![\mathsf{stmt}]\!] \\
(x := e)\ \uparrow_s\ = x := (e\ \uparrow_s) \\
(\mathsf{while}\ e\ \mathsf{do}\ \mathsf{stmt})\ \uparrow_s\ = \mathsf{while}\ (e\ \uparrow_s)\ \mathsf{do}\ (\mathsf{stmt}\ \uparrow_s) \\
(\mathsf{stmt_1}\ ,\ \mathsf{stmt_2})\ \uparrow_s\ = (\mathsf{stmt_1}\ \uparrow_s\ ,\ \mathsf{stmt_2}\ \uparrow_s)
\end{array}
$$

Target language: abstract syntax

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Code} : \mathsf{Set}\ \textbf{where} & & \\
\quad \mathsf{push} & : & (n : \mathbb{N}) \rightarrow \mathsf{Code} \\
\quad \mathsf{add} & : & \mathsf{Code} \\
\quad \mathsf{eq} & : & \mathsf{Code} \\
\quad \mathsf{load} & : & (x : \mathsf{Var}) \rightarrow \mathsf{Code} \\
\quad \mathsf{store} & : & (x : \mathsf{Var}) \rightarrow \mathsf{Code} \\
\quad \mathsf{loop} & : & (c_1 : \mathsf{Code}) \rightarrow (c_2 : \mathsf{Code}) \rightarrow \mathsf{Code} \\
\quad \_,\_ & : & (c_1 : \mathsf{Code}) \rightarrow (c_2 : \mathsf{Code}) \rightarrow \mathsf{Code}
\end{array}
$$

Type system

```
data _⊢_⇝_ : StackType → Code → StackType → Set where
  rpush  : ∀ {st} {n : ℕ} → st ⊢ push n ⇝ (nat :: st)
  radd   : ∀ {st} → (nat :: nat :: st) ⊢ add ⇝ (nat :: st)
  req    : ∀ {st} → (nat :: nat :: st) ⊢ eq ⇝ (bool :: st)
  rload  : ∀ {st} {x : Var} → st ⊢ load x ⇝ (nat :: st)
  rstore : ∀ {st} {x : Var} → (nat :: st) ⊢ store x ⇝ st
  rloop  : ∀ {st} {c₁ c₂} →
           st ⊢ c₁ ⇝ (bool :: st) → st ⊢ c₂ ⇝ st →
           st ⊢ loop c₁ c₂ ⇝ st
  rseq   : ∀ {st st' st''} {c₁ c₂} →
           st ⊢ c₁ ⇝ st' → st' ⊢ c₂ ⇝ st'' → st ⊢ c₁ , c₂ ⇝ st''
```

Semantics

Like in the source language, we add a clock to control iteration.

$$\mathrm{Dom}_C : (st : StackType) \to (st' : StackType) \to Set$$
$$\mathrm{Dom}_C\ st\ st' = (clock : \mathbb{N}) \to (s\sigma : Conf\ st) \to Maybe\ (Conf\ st')$$

$$Conf : (st : StackType) \to Set$$
$$Conf\ st = Stack\ st \times State$$

$$\mathcal{C}[\![\_]\!] : \forall \{st\ st'\} \to Code_t\ st\ st' \to \mathrm{Dom}_C\ st\ st'$$
$$\mathcal{C}[\![push\ n]\!] = fpush\ n$$
$$\mathcal{C}[\![add]\!] = fadd$$
$$\mathcal{C}[\![eq]\!] = feq$$
$$\mathcal{C}[\![load\ x]\!] = fload\ x$$
$$\mathcal{C}[\![store\ x]\!] = fstore\ x$$
$$\mathcal{C}[\![loop\ c_1\ c_2]\!] = floop\ \mathcal{C}[\![c_1]\!]\ \mathcal{C}[\![c_2]\!]$$
$$\mathcal{C}[\![c_1\ ,\ c_2]\!] = fseqc\ \mathcal{C}[\![c_1]\!]\ \mathcal{C}[\![c_2]\!]$$

Semantic algebra

$$\text{fpush} : \forall \{st\} \rightarrow (n : \mathbb{N}) \rightarrow \text{Dom}_C \; st \; (nat :: st)$$
$$\text{fpush } n = \lambda \{\text{clock } (s , \sigma) \rightarrow \text{just } ((n \triangleright s) , \sigma)\}$$

$$\text{fadd} : \forall \{st\} \rightarrow \text{Dom}_C \; (nat :: nat :: st) \; (nat :: st)$$
$$\text{fadd} = \lambda \{\text{clock } ((n \triangleright (m \triangleright s)) , \sigma) \rightarrow \text{just } (((n + m) \triangleright s) , \sigma)\}$$

$$\text{feq} : \forall \{st\} \rightarrow \text{Dom}_C \; (nat :: nat :: st) \; (bool :: st)$$
$$\text{feq} = \lambda \{\text{clock } ((n \triangleright (m \triangleright s)) , \sigma) \rightarrow \text{just } (((n \equiv m) \triangleright s) , \sigma)\}$$

$$\text{fload} : \forall \{st\} \rightarrow (x : Var) \rightarrow \text{Dom}_C \; st \; (nat :: st)$$
$$\text{fload } x = \lambda \{\text{clock } (s , \sigma) \rightarrow \text{just } ((\sigma \; x \triangleright s) , \sigma)\}$$

$$\text{fstore} : \forall \{st\} \rightarrow (x : Var) \rightarrow \text{Dom}_C \; (nat :: st) \; st$$
$$\text{fstore } x = \lambda \{\text{clock } ((n \triangleright s) , \sigma) \rightarrow \text{just } (s , \sigma \, [\, x \longrightarrow n \,])\}$$

Semantic algebra

```
floop : ∀ {st} →
          Dom_C st (bool :: st) → Dom_C st st → Dom_C st st
floop fb fc zero sσ  =  nothing
floop fb fc (suc clock) sσ
   =  fb (suc clock) sσ  >>=
     (λ {((b ▷ s') , σ') → if b then fc (suc clock) (s',σ')  >>=
                                        floop fb fc clock
                                   else just (s' , σ')})

fseqc : ∀ {st st' st''} →
          Dom_C st st' → Dom_C st' st'' → Dom_C st st''
fseqc f1 f2  =  λ clock sσ → f1 clock sσ  >>=  f2 clock
```

```
data Codes : ∀ {st st'} → DomC st st' → Set where
  ...
  load    : ∀ {st} → (x : Var) → Codes (fload {st} x)
  store   : ∀ {st} → (x : Var) → Codes (fstore {st} x)
  loop    : ∀   {st} {f1 f2} →
                Codes f1 → Codes f2 → Codes (floop {st} f1 f2)
  _,_     : ∀   {st st' st''} {f1 f2} → Codes f1 → Codes f2 →
                Codes (fseqc {st} {st'} {st''} f1 f2)
  substC  : ∀ {st st'} {f g} →
                Codes {st} {st'} f → EqSem f g → Codes g
```

Semantic decoration: code

```
data Code_s : ∀ {st st'} → Dom_C st st' → Set where
  ...
  load   : ∀ {st} → (x : Var) → Code_s (fload {st} x)
  store  : ∀ {st} → (x : Var) → Code_s (fstore {st} x)
  loop   : ∀   {st} {f1 f2} →
               Code_s f1 → Code_s f2 → Code_s (floop {st} f1 f2)
  _,_    : ∀   {st st' st''} {f1 f2} → Code_s f1 → Code_s f2 →
               Code_s (fseqc {st} {st'} {st''} f1 f2)
  subst_C : ∀ {st st'} {f g} →
               Code_s {st} {st'} f → EqSem f g → Code_s g
```

subst_C is an extra constructor that aids the insertion of extensional equality proofs between semantic functions wherever necessary.

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

Semantic decoration: code

Notice that the extra constructor is not reached by lifting.

$$
\begin{array}{lcl}
\_ \uparrow_s & : & \forall \{\mathsf{st}\ \mathsf{st'}\} \to (\mathsf{c}\ :\ \mathsf{Code_t}\ \mathsf{st}\ \mathsf{st'}) \to \mathsf{Code_s}\ (\mathcal{C}[\![\mathsf{c}]\!]) \\
\mathsf{push}\ \mathsf{n}\ \uparrow_s & = & \mathsf{push}\ \mathsf{n} \\
\mathsf{add}\ \uparrow_s & = & \mathsf{add} \\
\mathsf{eq}\ \uparrow_s & = & \mathsf{eq} \\
\mathsf{load}\ \mathsf{x}\ \uparrow_s & = & \mathsf{load}\ \mathsf{x} \\
\mathsf{store}\ \mathsf{x}\ \uparrow_s & = & \mathsf{store}\ \mathsf{x} \\
\mathsf{loop}\ \mathsf{c_1}\ \mathsf{c_2}\ \uparrow_s & = & \mathsf{loop}\ (\mathsf{c_1}\ \uparrow_s)\ (\mathsf{c_2}\ \uparrow_s) \\
(\mathsf{c_1}\ ,\ \mathsf{c_2})\ \uparrow_s & = & (\mathsf{c_1}\ \uparrow_s)\ ,\mathsf{c}\ (\mathsf{c_2}\ \uparrow_s)
\end{array}
$$

The semantics preserving compiler: expressions

$$
\begin{array}{lcl}
\text{comp}_e & : & \forall \{t\}\ \{f\}\ \{st\} \rightarrow \\
& & \text{ExprSem}\ \{t\}\ f \rightarrow \\
& & \text{Code}_s\ \{st\}\ (\lambda\ \{\text{clock}\ (s\ ,\ \sigma) \rightarrow \text{just}\ ((f\ \sigma \triangleright s)\ ,\ \sigma)\}) \\
\text{comp}_e\ \mid n \mid & = & \text{push n} \\
\text{comp}_e\ (e_1 \oplus e_2) & = & \text{comp}_e\ e_2\ ,\ (\text{comp}_e\ e_1\ ,\ \text{add}) \\
\text{comp}_e\ (e_1 \mathrel{\mathring{=}} e_2) & = & \text{comp}_e\ e_2\ ,\ (\text{comp}_e\ e_1\ ,\ \text{eq}) \\
\text{comp}_e\ (\text{var}\ x) & = & \text{load}\ x
\end{array}
$$

The semantics preserving compiler: statements

$$
\begin{aligned}
&\mathsf{comp_s} \;:\; \forall\,\{\mathsf{f}\}\,\{\mathsf{st}\} \to \\
&\qquad\qquad \mathsf{Stmt_s\ f} \to \mathsf{Code_s}\ (\mathsf{correctCodeF}\ \{\mathsf{st}\}\ \mathsf{f}) \\
&\mathsf{comp_s}\ (\mathsf{x} := \mathsf{e}) \;=\; \mathsf{comp}_e\ \mathsf{e}\ ,\ \mathsf{store\ x} \\
&\mathsf{comp_s}\ (\mathsf{while}\ \{\mathsf{fb}\}\ \{\mathsf{f}\}\ \mathsf{e\ stmt}) \;=\; (\mathsf{loop}\ (\mathsf{comp}_e\ \mathsf{e})\ (\mathsf{comp_s\ stmt}))\ * \\
&\quad \mathbf{where}\ \_* \;:\; \mathsf{Code_s}\ \_ \to \_ \\
&\qquad\qquad \mathsf{c}\ * \;=\; \mathsf{subst_C\ c}\ (\mathsf{eqloop\ fb\ f}) \\
&\mathsf{comp_s}\ (\_,\_\ \{\mathsf{f_1}\}\ \{\mathsf{f_2}\}\ \mathsf{stmt_1\ stmt_2}) \;=\; (\mathsf{comp_s\ stmt_1}\ ,\ \mathsf{comp_s\ stmt_2})\ * \\
&\quad \mathbf{where}\ \_* \;:\; \mathsf{Code_s}\ \_ \to \_ \\
&\qquad\qquad \mathsf{c}\ * \;=\; \mathsf{subst_C\ c}\ (\mathsf{eqseq\ f_1\ f_2})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{correctCodeF} \;:\; \forall\,\{\mathsf{st}\} \to \mathsf{Dom_S} \to \mathsf{Dom_C\ st\ st} \\
&\mathsf{correctCodeF\ f\ clock}\ (\mathsf{s}\ ,\ \sigma) \;=\; \mathsf{f\ clock}\ \sigma \;\gg\!=\; (\lambda\,\sigma' \to \mathsf{just}\ (\mathsf{s}\ ,\ \sigma'))
\end{aligned}
$$

Auxiliary proofs

```
eqloop : ∀ {st} fb f →
           EqSem {st} {st}
             (floop (λ {clock (s , σ) → just ((fb σ ▷ s) , σ)})
                   (correctCodeF f))
             (correctCodeF (fwhile fb f))
eqloop f₁ f₂ zero sσ = refl
eqloop f₁ f₂ (suc clock) (s , σ) with f₁ σ
eqloop f₁ f₂ (suc clock) (s , σ) | false = refl
eqloop f₁ f₂ (suc clock) (s , σ) | true with (f₂ (suc clock) σ)
eqloop f₁ f₂ (suc clock) (s , σ) | true | nothing = refl
eqloop f₁ f₂ (suc clock) (s , σ) | true | just σ'
   = eqloop f₁ f₂ clock (s , σ')

correctCodeF : ∀ {st} → Dom_S → Dom_C st st
correctCodeF f clock (s , σ) = f clock σ ≫= (λ σ' → just (s , σ'))
```

Auxiliary proofs

```
eqseq  : ∀ {st} f₁ f₂ →
            EqSem {st} {st} (fseqc (correctCodeF f₁) (correctCodeF f₂))
              (correctCodeF (λ clock σ → f₁ clock σ  ≫=  f₂ clock))
eqseq f₁ f₂ clock (s , σ) with f₁ clock σ
eqseq f₁ f₂ clock (s , σ)  |  nothing  =  refl
eqseq f₁ f₂ clock (s , σ)  |  just σ'  =  refl

correctCodeF  :  ∀ {st} → Dom_S → Dom_C st st
correctCodeF f clock (s , σ)  =  f clock σ  ≫=  (λ σ' → just (s , σ'))
```