An Internalist 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).

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.

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 where we develop the compiler and its correctness proof simultaneously.

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 where 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.

Externalist compiler

Source language: expressions

- Abstract syntax.

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

- Type system.

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

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

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

Agda representation

- Abstract syntax.

```
data Expr : Set where
  |_|   : ℕ → Expr
  _⊕_   : (e₁ : Expr) → (e₂ : Expr) → Expr
  _≟̊_   : (e₁ : Expr) → (e₂ : Expr) → Expr
```

- Type system.

```
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
```

Semantics

- Interpretation of types.

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

- Semantics of well-typed terms

$$\begin{aligned}
&\mathcal{E}[\![_]\!] : \forall\,\{\mathsf{t}\} \to (\mathsf{e} : \mathsf{Expr}) \to \vdash \mathsf{e} : \mathsf{t} \to \mathcal{T}[\![\ \mathsf{t}\ ]\!] \\
&\mathcal{E}[\![\ |\ \mathsf{n}\ |\ ]\!]\ \mathsf{tnat} = \mathsf{n} \\
&\mathcal{E}[\![\ \mathsf{e}_1 \oplus \mathsf{e}_2\ ]\!]\ (\mathsf{tplus}\ \mathsf{p}_1\ \mathsf{p}_2) = \mathcal{E}[\![\ \mathsf{e}_1\ ]\!]\ \mathsf{p}_1 + \mathcal{E}[\![\ \mathsf{e}_2\ ]\!]\ \mathsf{p}_2 \\
&\mathcal{E}[\![\ \mathsf{e}_1 \doteq \mathsf{e}_2\ ]\!]\ (\mathsf{teq}\ \mathsf{p}_1\ \mathsf{p}_2) = \mathcal{E}[\![\ \mathsf{e}_1\ ]\!]\ \mathsf{p}_1 ==_n \mathcal{E}[\![\ \mathsf{e}_2\ ]\!]\ \mathsf{p}_2
\end{aligned}$$

Target language: stack machine

We compile expressions into reverse polish notation.

- **Syntax:**

$$c ::= \mathit{push}\ n \mid \mathit{add} \mid \mathit{eq} \mid c_1, c_2$$

Target language: stack machine

We compile expressions into reverse polish notation.

- **Syntax:**

$$c ::= \mathit{push}\ n \mid \mathit{add} \mid \mathit{eq} \mid c_1, c_2$$

- **Agda representation**

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

Type system

Besides imposing type restrictions we also check stack safety.

```
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''
```

Semantics

$$
\begin{aligned}
&\textbf{data}\ \mathsf{SemC}\ :\ \forall\ \{\mathsf{st}\ \mathsf{st}'\} \to (\mathsf{c}\ :\ \mathsf{Code}) \to \\
&\qquad\qquad \mathsf{st} \vdash \mathsf{c} \rightsquigarrow \mathsf{st}' \to \\
&\qquad\qquad \mathsf{Stack\ st} \to \mathsf{Stack\ st}' \to \mathsf{Set}\ \textbf{where} \\
&\quad \mathsf{pushR}\ :\ \forall\ \{\mathsf{n}\ \mathsf{st}\}\ \{\mathsf{s}\ :\ \mathsf{Stack\ st}\} \to \\
&\qquad\qquad \mathsf{SemC}\ (\mathsf{push\ n})\ \mathsf{tpush}\ \mathsf{s}\ (\mathsf{n} \rhd \mathsf{s}) \\
&\quad \mathsf{addR}\ \ :\ \forall\ \{\mathsf{m}\ \mathsf{n}\ \mathsf{st}\}\ \{\mathsf{s}\ :\ \mathsf{Stack\ st}\} \to \\
&\qquad\qquad \mathsf{SemC\ add\ tadd}\ (\mathsf{m} \rhd (\mathsf{n} \rhd \mathsf{s}))\ ((\mathsf{m} + \mathsf{n}) \rhd \mathsf{s}) \\
&\quad \mathsf{eqR}\ \ \ :\ \forall\ \{\mathsf{m}\ \mathsf{n}\ \mathsf{st}\}\ \{\mathsf{s}\ :\ \mathsf{Stack\ st}\} \to \\
&\qquad\qquad \mathsf{SemC\ eq\ teq}\ (\mathsf{m} \rhd (\mathsf{n} \rhd \mathsf{s}))\ ((\mathsf{m} ==_n \mathsf{n}) \rhd \mathsf{s}) \\
&\quad \mathsf{seqR}\ \ \ :\ \forall\ \{\mathsf{c}_1\ \mathsf{c}_2\ \mathsf{st}\ \mathsf{st}'\ \mathsf{st}''\}\ \{\mathsf{p1}\ \mathsf{p2}\} \\
&\qquad\qquad \{\mathsf{s}\ :\ \mathsf{Stack\ st}\}\ \{\mathsf{s}'\ :\ \mathsf{Stack\ st}'\}\ \{\mathsf{s}''\ :\ \mathsf{Stack\ st}''\} \to \\
&\qquad\qquad \mathsf{SemC}\ \mathsf{c}_1\ \mathsf{p1}\ \mathsf{s}\ \mathsf{s}' \to \\
&\qquad\qquad \mathsf{SemC}\ \mathsf{c}_2\ \mathsf{p2}\ \mathsf{s}'\ \mathsf{s}'' \to \\
&\qquad\qquad \mathsf{SemC}\ (\mathsf{c}_1\ ,\ \mathsf{c}_2)\ (\mathsf{tseq\ p1\ p2})\ \mathsf{s}\ \mathsf{s}''
\end{aligned}
$$

Compilation of well-typed expressions

$$\begin{aligned}
&\mathsf{compile} : \forall\,\{\mathsf{t}\}\,(\mathsf{e} : \mathsf{Expr}) \to \vdash \mathsf{e} : \mathsf{t} \to \mathsf{Code} \\
&\mathsf{compile} \mid \mathsf{n} \mid \mathsf{tnat} \\
&\quad = \mathsf{push}\ \mathsf{n} \\
&\mathsf{compile}\ (\mathsf{e1} \oplus \mathsf{e}_2)\ (\mathsf{tplus}\ \mathsf{p1}\ \mathsf{p2}) \\
&\quad = \mathsf{compile}\ \mathsf{e1}\ \mathsf{p1}\ ,\ \mathsf{compile}\ \mathsf{e}_2\ \mathsf{p2}\ ,\ \mathsf{add} \\
&\mathsf{compile}\ (\mathsf{e1} \mathrel{\mathring{=}} \mathsf{e}_2)\ (\mathsf{teq}\ \mathsf{p1}\ \mathsf{p2}) \\
&\quad = \mathsf{compile}\ \mathsf{e1}\ \mathsf{p1}\ ,\ \mathsf{compile}\ \mathsf{e}_2\ \mathsf{p2}\ ,\ \mathsf{eq}
\end{aligned}$$

Compiler correctness

- Type preservation:

$$\begin{aligned}\mathsf{typepres} \;:\; &\forall\,\{\mathsf{t}\}\,\{\mathsf{st}\}\,\{\mathsf{e}\;:\;\mathsf{Expr}\} \to \\ &(\mathsf{tp}\;:\;\vdash \mathsf{e} : \mathsf{t}) \to \mathsf{st} \vdash \mathsf{compile}\ \mathsf{e}\ \mathsf{tp} \rightsquigarrow (\mathsf{t} :: \mathsf{st})\end{aligned}$$

Compiler correctness

- Type preservation:

$$\begin{aligned}\mathsf{typepres} \;:\; &\forall\,\{\mathsf{t}\}\,\{\mathsf{st}\}\,\{\mathsf{e}\;:\;\mathsf{Expr}\} \to \\ &(\mathsf{tp}\;:\;\vdash \mathsf{e} : \mathsf{t}) \to \mathsf{st} \vdash \mathsf{compile}\ \mathsf{e}\ \mathsf{tp} \rightsquigarrow (\mathsf{t} :: \mathsf{st})\end{aligned}$$

- Semantics preservation

$$\begin{aligned}\mathsf{correct} \;:\; &\forall\,\{\mathsf{t}\ \mathsf{st}\}\,\{\mathsf{e}\;:\;\mathsf{Expr}\}\,\{\mathsf{tp}\;:\;\vdash \mathsf{e} : \mathsf{t}\}\,\{\mathsf{s}\;:\;\mathsf{Stack}\ \mathsf{st}\} \to \\ &\mathsf{SemC}\ (\mathsf{compile}\ \mathsf{e}\ \mathsf{tp})\ (\mathsf{typepres}\ \mathsf{tp})\ \mathsf{s}\ (\mathcal{E}[\![\ \mathsf{e}\ ]\!]\ \mathsf{tp} \rhd \mathsf{s})\end{aligned}$$

Compiler correctness

- Type preservation:

$$\begin{aligned}\mathsf{typepres} \;:\; &\forall\,\{\mathsf{t}\}\,\{\mathsf{st}\}\,\{\mathsf{e}\;:\;\mathsf{Expr}\} \to \\ &(\mathsf{tp}\;:\;\vdash \mathsf{e} : \mathsf{t}) \to \mathsf{st} \vdash \mathsf{compile}\ \mathsf{e}\ \mathsf{tp} \rightsquigarrow (\mathsf{t} :: \mathsf{st})\end{aligned}$$

- Semantics preservation

$$\begin{aligned}\mathsf{correct} \;:\; &\forall\,\{\mathsf{t}\ \mathsf{st}\}\,\{\mathsf{e}\;:\;\mathsf{Expr}\}\,\{\mathsf{tp}\;:\;\vdash \mathsf{e} : \mathsf{t}\}\,\{\mathsf{s}\;:\;\mathsf{Stack}\ \mathsf{st}\} \to \\ &\mathsf{SemC}\ (\mathsf{compile}\ \mathsf{e}\ \mathsf{tp})\ (\mathsf{typepres}\ \mathsf{tp})\ \mathsf{s}\ (\mathcal{E}[\![\ \mathsf{e}\ ]\!]\ \mathsf{tp} \rhd \mathsf{s})\end{aligned}$$

Proving correctness corresponds to program verification.

Internalization of typing

Typed expressions

- We decorate type Expr with the type of the expression.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Expr}_\mathsf{t}\ :\ \mathsf{Type} \to \mathsf{Set}\ \textbf{where}\\
&\quad |\_| \quad\ : \mathbb{N} \to \mathsf{Expr}_\mathsf{t}\ \mathsf{nat}\\
&\quad \_\oplus\_ \ : (\mathsf{e}_1\ :\ \mathsf{Expr}_\mathsf{t}\ \mathsf{nat}) \to (\mathsf{e}_2\ :\ \mathsf{Expr}_\mathsf{t}\ \mathsf{nat}) \to \mathsf{Expr}_\mathsf{t}\ \mathsf{nat}\\
&\quad \_\hat{=}\_ \ : (\mathsf{e}_1\ :\ \mathsf{Expr}_\mathsf{t}\ \mathsf{nat}) \to (\mathsf{e}_2\ :\ \mathsf{Expr}_\mathsf{t}\ \mathsf{nat}) \to \mathsf{Expr}_\mathsf{t}\ \mathsf{bool}
\end{aligned}
$$

$$\vdash \mathsf{e}\ :\ \sigma \quad \overset{rep-by}{\rightsquigarrow} \quad \ulcorner \mathsf{e} \urcorner\ :\ \mathsf{Expr}_\mathsf{t}\ \sigma$$

Typed expressions

- We decorate type Expr with the type of the expression.

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

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

- Lifting function:

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

Semantics of typed expressions

- The Semantics.

$$
\begin{aligned}
&\mathcal{E}_t[\![\_]\!] : \forall\,\{\mathsf{t}\} \to \mathsf{Expr}_t\ \mathsf{t} \to \mathcal{T}[\![\ \mathsf{t}\ ]\!] \\
&\mathcal{E}_t[\![\ |\ \mathsf{n}\ |\ ]\!] \quad = \mathsf{fnat}\ \mathsf{n} \\
&\mathcal{E}_t[\![\ \mathsf{e}_1 \oplus \mathsf{e}_2\ ]\!] = \mathsf{fplus}\ \mathcal{E}_t[\![\ \mathsf{e}_1\ ]\!]\ \mathcal{E}_t[\![\ \mathsf{e}_2\ ]\!] \\
&\mathcal{E}_t[\![\ \mathsf{e}_1 \doteq \mathsf{e}_2\ ]\!] = \mathsf{feq}\ \ \mathcal{E}_t[\![\ \mathsf{e}_1\ ]\!]\ \mathcal{E}_t[\![\ \mathsf{e}_2\ ]\!]
\end{aligned}
$$

- Semantic algebra:

$$
\begin{aligned}
&\mathsf{fnat} : \mathbb{N} \to \mathbb{N} \\
&\mathsf{fnat}\ \mathsf{n} = \mathsf{n} \\
&\mathsf{fplus} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathsf{fplus}\ \mathsf{n}_1\ \mathsf{n}_2 = \mathsf{n}_1 + \mathsf{n}_2 \\
&\mathsf{feq} : \mathbb{N} \to \mathbb{N} \to \mathsf{Bool} \\
&\mathsf{feq}\ \mathsf{n}_1\ \mathsf{n}_2 = \mathsf{n}_1 ==_n \mathsf{n}_2
\end{aligned}
$$

Decorated version of code

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

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

Decorated version of code

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

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

- Lifting function:

$$
\_\uparrow_t\_\ :\ \forall\,\{\mathsf{st}\ \mathsf{st'}\} \to (\mathsf{c}\ :\ \mathsf{Code}) \to \mathsf{st} \vdash \mathsf{c} \rightsquigarrow \mathsf{st'} \to \mathsf{Code_t}\ \mathsf{st}\ \mathsf{st'}
$$

Semantics of decorated code

$$
\begin{aligned}
&\textbf{data}\ \mathsf{SemC}_t\ :\ \forall\,\{\mathsf{st}\ \mathsf{st}'\} \to \mathsf{Code}_t\ \mathsf{st}\ \mathsf{st}' \to \\
&\qquad\qquad \mathsf{Stack}\ \mathsf{st} \to \mathsf{Stack}\ \mathsf{st}' \to \mathsf{Set}\ \textbf{where} \\
&\quad \mathsf{push}\ :\ \forall\,\{\mathsf{st}\ \mathsf{n}\}\,\{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\} \to \\
&\qquad \mathsf{SemC}_t\ (\mathsf{push}\ \mathsf{n})\ \mathsf{s}\ (\mathsf{n} \rhd \mathsf{s}) \\
&\quad \mathsf{add}\ \ :\ \forall\,\{\mathsf{m}\ \mathsf{n}\ \mathsf{st}\}\,\{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\} \to \\
&\qquad \mathsf{SemC}_t\ \mathsf{add}\ (\mathsf{m} \rhd (\mathsf{n} \rhd \mathsf{s}))\ ((\mathsf{m} + \mathsf{n}) \rhd \mathsf{s}) \\
&\quad \mathsf{eq}\ \ \ :\ \forall\,\{\mathsf{m}\ \mathsf{n}\ \mathsf{st}\}\,\{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\} \to \\
&\qquad \mathsf{SemC}_t\ \mathsf{eq}\ (\mathsf{m} \rhd (\mathsf{n} \rhd \mathsf{s}))\ ((\mathsf{m} ==_n \mathsf{n}) \rhd \mathsf{s}) \\
&\quad \mathsf{seq}\ \ \ :\ \forall\,\{\mathsf{st}\ \mathsf{st}'\ \mathsf{st}''\ \mathsf{c}_1\ \mathsf{c}_2\} \\
&\qquad \{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\}\,\{\mathsf{s}'\ :\ \mathsf{Stack}\ \mathsf{st}'\}\,\{\mathsf{s}''\ :\ \mathsf{Stack}\ \mathsf{st}''\} \to \\
&\qquad \mathsf{SemC}_t\ \mathsf{c}_1\ \mathsf{s}\ \mathsf{s}' \to \\
&\qquad \mathsf{SemC}_t\ \mathsf{c}_2\ \mathsf{s}'\ \mathsf{s}'' \to \\
&\qquad \mathsf{SemC}_t\ (\mathsf{c}_1\ ,\ \mathsf{c}_2)\ \mathsf{s}\ \mathsf{s}''
\end{aligned}
$$

Compiler between decorated ASTs

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

$$
\begin{aligned}
&\mathsf{compile}_t \;:\; \forall\,\{\mathsf{t}\}\,\{\mathsf{st}\} \to \mathsf{Expr}_t\;\mathsf{t} \to \mathsf{Code}_t\;\mathsf{st}\;(\mathsf{t} :: \mathsf{st}) \\
&\mathsf{compile}_t\;|\;\mathsf{n}\;| \;=\; \mathsf{push}\;\mathsf{n} \\
&\mathsf{compile}_t\;(\mathsf{e}_1 \oplus \mathsf{e}_2) \;=\; \mathsf{compile}_t\;\mathsf{e}_1\;,\;\mathsf{compile}_t\;\mathsf{e}_2\;,\;\mathsf{add} \\
&\mathsf{compile}_t\;(\mathsf{e}_1 \mathrel{\mathring{=}} \mathsf{e}_2) \;=\; \mathsf{compile}_t\;\mathsf{e}_1\;,\;\mathsf{compile}_t\;\mathsf{e}_2\;,\;\mathsf{eq}
\end{aligned}
$$

Correctness

$$\begin{aligned}\mathsf{correct}_t\ :\ \forall\ \{\mathsf{t}\}\ \{\mathsf{st}\}\ \{\mathsf{e}\ :\ \mathsf{Expr}_t\ \mathsf{t}\}\ \{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\} \rightarrow \\ \mathsf{SemC}_t\ (\mathsf{compile}_t\ \mathsf{e})\ \mathsf{s}\ (\mathcal{E}_t[\![\ \mathsf{e}\ ]\!] \rhd \mathsf{s})\end{aligned}$$

Correctness

$$\begin{aligned}\mathsf{correct}_t\ :\ \forall\ \{\mathsf{t}\}\ \{\mathsf{st}\}\ \{\mathsf{e}\ :\ \mathsf{Expr}_t\ \mathsf{t}\}\ \{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\} \rightarrow \\ \mathsf{SemC}_t\ (\mathsf{compile}_t\ \mathsf{e})\ \mathsf{s}\ (\mathcal{E}_t[\![\ \mathsf{e}\ ]\!] \rhd \mathsf{s})\end{aligned}$$

- Proving correctness still corresponds to program verification.
- The correctness predicate continues being external to the compilation function.
- The type of $\mathsf{compile}_t$ does not prevent that we define type-preserving but semantically erroneous rules like:

  $\mathsf{compile}\ (\mathsf{e}_1 \oplus \mathsf{e}_2)\ =\ \mathsf{compile}\ \mathsf{e}_1\ ,\ \mathsf{compile}\ \mathsf{e}_2\ ,\ \mathsf{add},\ \mathsf{push}\ 1\ ,\ \mathsf{add}$

Internalization of semantics

Semantic decoration of expressions

- We want to enforce semantics preservation during compilation.
- The type of an expression is now decorated with its own value.
- Those values are computed by the operations of the algebra.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Expr}_s\ :\ \forall\,\{\mathsf{t}\} \to \mathcal{T}[\![\ \mathsf{t}\ ]\!] \to \mathsf{Set}\ \textbf{where}\\
&\quad |\_| \quad\ :\ (\mathsf{n}\ :\ \mathbb{N}) \to \mathsf{Expr}_s\ (\mathsf{fnat}\ \mathsf{n})\\
&\quad \_\oplus\_\ :\ \forall\,\{\mathsf{n}_1\ \mathsf{n}_2\} \to\\
&\qquad\qquad (\mathsf{e}_1\ :\ \mathsf{Expr}_s\ \mathsf{n}_1) \to\\
&\qquad\qquad (\mathsf{e}_2\ :\ \mathsf{Expr}_s\ \mathsf{n}_2) \to \mathsf{Expr}_s\ (\mathsf{fplus}\ \mathsf{n}_1\ \mathsf{n}_2)\\
&\quad \_\hat{=}\_\ :\ \forall\,\{\mathsf{n}_1\ \mathsf{n}_2\} \to\\
&\qquad\qquad (\mathsf{e}_1\ :\ \mathsf{Expr}_s\ \mathsf{n}_1) \to\\
&\qquad\qquad (\mathsf{e}_2\ :\ \mathsf{Expr}_s\ \mathsf{n}_2) \to \mathsf{Expr}_s\ (\mathsf{feq}\ \mathsf{n}_1\ \mathsf{n}_2)
\end{aligned}
$$

Semantic decoration of expressions

- We want to enforce semantics preservation during compilation.
- The type of an expression is now decorated with its own value.
- Those values are computed by the operations of the algebra.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Expr}_s\ :\ \forall\,\{\mathsf{t}\} \to \mathcal{T}[\![\ \mathsf{t}\ ]\!] \to \mathsf{Set}\ \textbf{where}\\
&\quad |\_| \quad\ :\ (\mathsf{n}\ :\ \mathbb{N}) \to \mathsf{Expr}_s\ (\mathsf{fnat}\ \mathsf{n})\\
&\quad \_\oplus\_\ :\ \forall\,\{\mathsf{n}_1\ \mathsf{n}_2\} \to\\
&\qquad\qquad (\mathsf{e}_1\ :\ \mathsf{Expr}_s\ \mathsf{n}_1) \to\\
&\qquad\qquad (\mathsf{e}_2\ :\ \mathsf{Expr}_s\ \mathsf{n}_2) \to \mathsf{Expr}_s\ (\mathsf{fplus}\ \mathsf{n}_1\ \mathsf{n}_2)\\
&\quad \_\mathrel{\mathring{=}}\_\ :\ \forall\,\{\mathsf{n}_1\ \mathsf{n}_2\} \to\\
&\qquad\qquad (\mathsf{e}_1\ :\ \mathsf{Expr}_s\ \mathsf{n}_1) \to\\
&\qquad\qquad (\mathsf{e}_2\ :\ \mathsf{Expr}_s\ \mathsf{n}_2) \to \mathsf{Expr}_s\ (\mathsf{feq}\ \mathsf{n}_1\ \mathsf{n}_2)
\end{aligned}
$$

- Lifting function:

$$\_\uparrow_s\ :\ \forall\,\{\mathsf{t}\} \to (\mathsf{e}\ :\ \mathsf{Expr}_t\ \mathsf{t}) \to \mathsf{Expr}_s\ \mathcal{E}_t[\![\ \mathsf{e}\ ]\!]$$

Semantic decoration of code

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Code}_s\ :\ \forall\ \{\mathsf{st}\ \mathsf{st}'\} \to \mathsf{Stack}\ \mathsf{st} \to \mathsf{Stack}\ \mathsf{st}' \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{push}\ :\ \forall\ \{\mathsf{st}\}\ \{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\} \to (\mathsf{n}\ :\ \mathbb{N}) \to \mathsf{Code}_s\ \mathsf{s}\ (\mathsf{n} \rhd \mathsf{s})\\
&\quad \mathsf{add}\ \ :\ \forall\ \{\mathsf{st}\}\ \{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\}\ \{\mathsf{m}\ \mathsf{n}\} \to\\
&\qquad\qquad \mathsf{Code}_s\ (\mathsf{m} \rhd (\mathsf{n} \rhd \mathsf{s}))\ ((\mathsf{m} + \mathsf{n}) \rhd \mathsf{s})\\
&\quad \mathsf{eq}\ \ \ :\ \forall\ \{\mathsf{st}\}\ \{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\}\ \{\mathsf{m}\ \mathsf{n}\} \to\\
&\qquad\qquad \mathsf{Code}_s\ (\mathsf{m} \rhd (\mathsf{n} \rhd \mathsf{s}))\ ((\mathsf{m} ==_n \mathsf{n}) \rhd \mathsf{s})\\
&\quad \_,\_\ :\ \forall\ \{\mathsf{st}\ \mathsf{st}'\ \mathsf{st}''\}\\
&\qquad\qquad \{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\}\ \{\mathsf{s}'\ :\ \mathsf{Stack}\ \mathsf{st}'\}\ \{\mathsf{s}''\ :\ \mathsf{Stack}\ \mathsf{st}''\} \to\\
&\qquad\qquad \mathsf{Code}_s\ \mathsf{s}\ \mathsf{s}' \to \mathsf{Code}_s\ \mathsf{s}'\ \mathsf{s}'' \to \mathsf{Code}_s\ \mathsf{s}\ \mathsf{s}''
\end{aligned}
$$

Semantic decoration of code

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Code}_s\ :\ \forall\,\{\mathsf{st}\ \mathsf{st}'\} \to \mathsf{Stack}\ \mathsf{st} \to \mathsf{Stack}\ \mathsf{st}' \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{push}\ :\ \forall\,\{\mathsf{st}\}\,\{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\} \to (\mathsf{n}\ :\ \mathbb{N}) \to \mathsf{Code}_s\ \mathsf{s}\ (\mathsf{n} \rhd \mathsf{s})\\
&\quad \mathsf{add}\ \ :\ \forall\,\{\mathsf{st}\}\,\{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\}\,\{\mathsf{m}\ \mathsf{n}\} \to\\
&\qquad\qquad \mathsf{Code}_s\ (\mathsf{m} \rhd (\mathsf{n} \rhd \mathsf{s}))\ ((\mathsf{m} + \mathsf{n}) \rhd \mathsf{s})\\
&\quad \mathsf{eq}\ \ \ :\ \forall\,\{\mathsf{st}\}\,\{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\}\,\{\mathsf{m}\ \mathsf{n}\} \to\\
&\qquad\qquad \mathsf{Code}_s\ (\mathsf{m} \rhd (\mathsf{n} \rhd \mathsf{s}))\ ((\mathsf{m} ==_n \mathsf{n}) \rhd \mathsf{s})\\
&\quad \_,\_\ :\ \forall\,\{\mathsf{st}\ \mathsf{st}'\ \mathsf{st}''\}\\
&\qquad\qquad \{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\}\,\{\mathsf{s}'\ :\ \mathsf{Stack}\ \mathsf{st}'\}\,\{\mathsf{s}''\ :\ \mathsf{Stack}\ \mathsf{st}''\} \to\\
&\qquad\qquad \mathsf{Code}_s\ \mathsf{s}\ \mathsf{s}' \to \mathsf{Code}_s\ \mathsf{s}'\ \mathsf{s}'' \to \mathsf{Code}_s\ \mathsf{s}\ \mathsf{s}''
\end{aligned}
$$

Lifting function:

$$
\begin{aligned}
&\_\uparrow_s\_\ :\ \forall\,\{\mathsf{st}\ \mathsf{st}'\}\,\{\mathsf{s}\ :\ \mathsf{Stack}\ \mathsf{st}\}\,\{\mathsf{s}'\ :\ \mathsf{Stack}\ \mathsf{st}'\} \to\\
&\qquad\qquad (\mathsf{c}\ :\ \mathsf{Code}_t\ \mathsf{st}\ \mathsf{st}') \to \mathsf{SemC}_t\ \mathsf{c}\ \mathsf{s}\ \mathsf{s}' \to \mathsf{Code}_s\ \mathsf{s}\ \mathsf{s}'
\end{aligned}
$$

Correct-by-construction compiler

Now the correctness property is directly enforced by the type of the compiler.

$$
\begin{aligned}
&\mathsf{compile}_s \;:\; \forall\,\{\mathsf{t}\}\,\{\mathsf{st}\}\,\{\mathsf{v} \;:\; \mathcal{T}[\![\,\mathsf{t}\,]\!]\}\,\{\mathsf{s} \;:\; \mathsf{Stack}\;\mathsf{st}\} \rightarrow \\
&\qquad\qquad (\mathsf{e} \;:\; \mathsf{Expr}_s\;\mathsf{v}) \rightarrow \mathsf{Code}_s\;\mathsf{s}\;(\mathsf{v} \rhd \mathsf{s}) \\
&\mathsf{compile}_s\;|\,\mathsf{n}\,| \;=\; \mathsf{push}\;\mathsf{n} \\
&\mathsf{compile}_s\;(\mathsf{e}_1 \oplus \mathsf{e}_2) \;=\; \mathsf{compile}_s\;\mathsf{e}_2\;,\;\mathsf{compile}_s\;\mathsf{e}_1\;,\;\mathsf{add} \\
&\mathsf{compile}_s\;(\mathsf{e}_1 \stackrel{\circ}{=} \mathsf{e}_2) \;=\; \mathsf{compile}_s\;\mathsf{e}_2\;,\;\mathsf{compile}_s\;\mathsf{e}_1\;,\;\mathsf{eq}
\end{aligned}
$$

A compiler for a While language

Extended languages

- We extend our source language with variables and statements.

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

  Variables are of type nat.

Extended languages

- We extend our source language with variables and statements.

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

  Variables are of type nat.

- We also extend the target language with new instructions.

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

Source language: 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
```

Source language: type system

$$
\begin{aligned}
&\textbf{data}\ \vdash\_:\_\ :\ \mathsf{Expr} \to \mathsf{Type} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{tnat}\ :\ \forall\,\{n\} \to \vdash |\,n\,| : \mathsf{nat}\\
&\quad \mathsf{tplus}\ :\ \forall\,\{e_1\ e_2\} \to\\
&\qquad\qquad \vdash e_1 : \mathsf{nat} \to \vdash e_2 : \mathsf{nat} \to \vdash e_1 \oplus e_2 : \mathsf{nat}\\
&\quad \mathsf{teq}\ \ :\ \forall\,\{e_1\ e_2\} \to\\
&\qquad\qquad \vdash e_1 : \mathsf{nat} \to \vdash e_2 : \mathsf{nat} \to \vdash e_1 \mathrel{\hat{=}} e_2 : \mathsf{bool}\\
&\quad \mathsf{tvar}\ :\ \forall\,\{x\} \to \vdash \mathsf{var}\ x : \mathsf{nat}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data}\ \vdash\_\ :\ \mathsf{Stmt} \to \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{tassign}\ :\ \forall\,\{x\}\,\{e\} \to \vdash e : \mathsf{nat} \to \vdash (x := e)\\
&\quad \mathsf{twhile}\ \ :\ \forall\,\{e\}\,\{\mathsf{stmt}\} \to\\
&\qquad\qquad \vdash e : \mathsf{bool} \to \vdash \mathsf{stmt} \to \vdash (\mathsf{while}\ e\ \mathsf{do}\ \mathsf{stmt})\\
&\quad \mathsf{tseq}\ \ \ :\ \forall\,\{\mathsf{stmt}_1\ \mathsf{stmt}_2\} \to\\
&\qquad\qquad \vdash \mathsf{stmt}_1 \to \vdash \mathsf{stmt}_2 \to \vdash (\mathsf{stmt}_1\ ,\ \mathsf{stmt}_2)
\end{aligned}
$$

Semantics of statements

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

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

$$\begin{aligned}
&[\![_]\!] \ : \ \mathsf{Stmt_t} \to \mathsf{Dom_S} \\
&[\![\mathsf{x} := \mathsf{e}]\!] \ = \ \mathsf{fassign\ x}\ (\mathcal{E}_t[\![\ \mathsf{e}\ ]\!]) \\
&[\![\mathsf{while\ e\ do\ stmt}]\!] \ = \ \mathsf{fwhile}\ (\mathcal{E}_t[\![\ \mathsf{e}\ ]\!])\ [\![\mathsf{stmt}]\!] \\
&[\![\mathsf{stmt}_1 ; \mathsf{stmt}_2]\!] \ = \ \mathsf{fseq}\ [\![\mathsf{stmt}_1]\!]\ [\![\mathsf{stmt}_2]\!]
\end{aligned}$$

- When the clock reaches the value 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

$$\mathsf{fassign} \;:\; \mathsf{Var} \to (\mathsf{State} \to \mathbb{N}) \to \mathsf{Dom_S}$$
$$\mathsf{fassign}\ \mathsf{x}\ \mathsf{fe} \;=\; \lambda\, \mathsf{clock}\ \sigma \to \mathsf{just}\ (\sigma\ [\ \mathsf{x} \leftarrow \mathsf{fe}\ \sigma\ ])$$

$$\mathsf{fwhile} \;:\; (\mathsf{State} \to \mathsf{Bool}) \to \mathsf{Dom_S} \to \mathsf{Dom_S}$$
$$\mathsf{fwhile}\ \mathsf{fb}\ \mathsf{fc}\ \mathsf{zero}\ \sigma \;=\; \mathsf{nothing}$$
$$\mathsf{fwhile}\ \mathsf{fb}\ \mathsf{fc}\ (\mathsf{suc}\ \mathsf{clock})\ \sigma$$
$$= \;\mathsf{if}\ \mathsf{fb}\ \sigma\ \mathsf{then}\ \mathsf{fc}\ (\mathsf{suc}\ \mathsf{clock})\ \sigma \;\gg\!=\; \mathsf{fwhile}\ \mathsf{fb}\ \mathsf{fc}\ \mathsf{clock}$$
$$\mathsf{else}\ \mathsf{just}\ \sigma$$

$$\mathsf{fseq} \;:\; \mathsf{Dom_S} \to \mathsf{Dom_S} \to \mathsf{Dom_S}$$
$$\mathsf{fseq}\ \mathsf{f_1}\ \mathsf{f_2} \;=\; \lambda\, \mathsf{clock}\ \sigma \to \mathsf{f_1}\ \mathsf{clock}\ \sigma \;\gg\!=\; \mathsf{f_2}\ \mathsf{clock}$$

Semantic decoration of statements

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

Lifting function:

$$\_\uparrow_s\ :\ (\mathsf{stmt}\ :\ \mathsf{Stmt_t}) \to \mathsf{Stmt_s}\ [\![\mathsf{stmt}]\!]$$

Target language: abstract syntax

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Code}\ :\ \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{push}\ :\ (\mathsf{n}\ :\ \mathbb{N}) \to \mathsf{Code}\\
&\quad \mathsf{add}\ \ :\ \mathsf{Code}\\
&\quad \mathsf{eq}\ \ \ :\ \mathsf{Code}\\
&\quad \mathsf{load}\ \ :\ (\mathsf{x}\ :\ \mathsf{Var}) \to \mathsf{Code}\\
&\quad \mathsf{store}\ :\ (\mathsf{x}\ :\ \mathsf{Var}) \to \mathsf{Code}\\
&\quad \mathsf{loop}\ \ :\ (\mathsf{c}_1\ :\ \mathsf{Code}) \to (\mathsf{c}_2\ :\ \mathsf{Code}) \to \mathsf{Code}\\
&\quad \_,\_\ \ :\ (\mathsf{c}_1\ :\ \mathsf{Code}) \to (\mathsf{c}_2\ :\ \mathsf{Code}) \to \mathsf{Code}
\end{aligned}
$$

Target language: type system

```
data _⊢_⇝_ : StackType → Code → StackType → Set where
  rpush  : ...
  radd   : ...
  req    : ...
  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   : ...
```

Functional semantics

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

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

$$\mathcal{C}[\![\_]\!] : \forall\ \{\mathsf{st\ st'}\} \to \mathsf{Code_t\ st\ st'} \to \mathsf{Dom_C\ st\ st'}$$
$$\begin{aligned}
\mathcal{C}[\![\mathsf{push\ n}]\!] &= \mathsf{fpush\ n} \\
\mathcal{C}[\![\mathsf{add}]\!] &= \mathsf{fadd} \\
\mathcal{C}[\![\mathsf{eq}]\!] &= \mathsf{feq} \\
\mathcal{C}[\![\mathsf{load\ x}]\!] &= \mathsf{fload\ x} \\
\mathcal{C}[\![\mathsf{store\ x}]\!] &= \mathsf{fstore\ x} \\
\mathcal{C}[\![\mathsf{loop\ c_1\ c_2}]\!] &= \mathsf{floop}\ \mathcal{C}[\![\mathsf{c_1}]\!]\ \mathcal{C}[\![\mathsf{c_2}]\!] \\
\mathcal{C}[\![\mathsf{c_1\ ,\ c_2}]\!] &= \mathsf{fseqc}\ \mathcal{C}[\![\mathsf{c_1}]\!]\ \mathcal{C}[\![\mathsf{c_2}]\!]
\end{aligned}$$

Semantic algebra

$$\mathsf{fpush} \;:\; \forall\,\{\mathsf{st}\} \to (\mathsf{n} \;:\; \mathbb{N}) \to \mathsf{Dom_C}\ \mathsf{st}\ (\mathsf{nat} :: \mathsf{st})$$
$$\mathsf{fpush}\ \mathsf{n} \;=\; \lambda\,\{\mathsf{clock}\ (\mathsf{s}\ ,\ \sigma) \to \mathsf{just}\ ((\mathsf{n} \rhd \mathsf{s})\ ,\ \sigma)\}$$

$$\mathsf{fadd} \;:\; \forall\,\{\mathsf{st}\} \to \mathsf{Dom_C}\ (\mathsf{nat} :: \mathsf{nat} :: \mathsf{st})\ (\mathsf{nat} :: \mathsf{st})$$
$$\mathsf{fadd} \;=\; \lambda\,\{\mathsf{clock}\ ((\mathsf{n} \rhd (\mathsf{m} \rhd \mathsf{s}))\ ,\ \sigma) \to \mathsf{just}\ (((\mathsf{n} + \mathsf{m}) \rhd \mathsf{s})\ ,\ \sigma)\}$$

$$\mathsf{feq} \;:\; \forall\,\{\mathsf{st}\} \to \mathsf{Dom_C}\ (\mathsf{nat} :: \mathsf{nat} :: \mathsf{st})\ (\mathsf{bool} :: \mathsf{st})$$
$$\mathsf{feq} \;=\; \lambda\,\{\mathsf{clock}\ ((\mathsf{n} \rhd (\mathsf{m} \rhd \mathsf{s}))\ ,\ \sigma) \to \mathsf{just}\ (((\mathsf{n} \;\equiv\; \mathsf{m}) \rhd \mathsf{s})\ ,\ \sigma)\}$$

$$\mathsf{fload} \;:\; \forall\,\{\mathsf{st}\} \to (\mathsf{x} \;:\; \mathsf{Var}) \to \mathsf{Dom_C}\ \mathsf{st}\ (\mathsf{nat} :: \mathsf{st})$$
$$\mathsf{fload}\ \mathsf{x} \;=\; \lambda\,\{\mathsf{clock}\ (\mathsf{s}\ ,\ \sigma) \to \mathsf{just}\ ((\sigma\ \mathsf{x} \rhd \mathsf{s})\ ,\ \sigma)\}$$

$$\mathsf{fstore} \;:\; \forall\,\{\mathsf{st}\} \to (\mathsf{x} \;:\; \mathsf{Var}) \to \mathsf{Dom_C}\ (\mathsf{nat} :: \mathsf{st})\ \mathsf{st}$$
$$\mathsf{fstore}\ \mathsf{x} \;=\; \lambda\,\{\mathsf{clock}\ ((\mathsf{n} \rhd \mathsf{s})\ ,\ \sigma) \to \mathsf{just}\ (\mathsf{s}\ ,\ \sigma\ [\ \mathsf{x} \longrightarrow \mathsf{n}\ ])\}$$

Semantic algebra

$$
\begin{aligned}
&\mathsf{floop} \;:\; \forall\,\{\mathsf{st}\} \to \\
&\qquad \mathsf{Dom_C}\ \mathsf{st}\ (\mathsf{bool} :: \mathsf{st}) \to \mathsf{Dom_C}\ \mathsf{st}\ \mathsf{st} \to \mathsf{Dom_C}\ \mathsf{st}\ \mathsf{st} \\
&\mathsf{floop}\ \mathsf{fb}\ \mathsf{fc}\ \mathsf{zero}\ s\sigma \;=\; \mathsf{nothing} \\
&\mathsf{floop}\ \mathsf{fb}\ \mathsf{fc}\ (\mathsf{suc}\ \mathsf{clock})\ s\sigma \\
&\quad =\; \mathsf{fb}\ (\mathsf{suc}\ \mathsf{clock})\ s\sigma \;\gg\!= \\
&\qquad \lambda\,\{((\mathsf{b} \rhd \mathsf{s'})\ ,\ \sigma') \to \mathsf{if}\ \mathsf{b}\ \mathsf{then}\ \mathsf{fc}\ (\mathsf{suc}\ \mathsf{clock})\ (\mathsf{s'},\sigma') \;\gg\!= \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{floop}\ \mathsf{fb}\ \mathsf{fc}\ \mathsf{clock} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \mathsf{just}\ (\mathsf{s'}\ ,\ \sigma')\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{fseqc} \;:\; \forall\,\{\mathsf{st}\ \mathsf{st'}\ \mathsf{st''}\} \to \\
&\qquad \mathsf{Dom_C}\ \mathsf{st}\ \mathsf{st'} \to \mathsf{Dom_C}\ \mathsf{st'}\ \mathsf{st''} \to \mathsf{Dom_C}\ \mathsf{st}\ \mathsf{st''} \\
&\mathsf{fseqc}\ \mathsf{f_1}\ \mathsf{f_2} \;=\; \lambda\ \mathsf{clock}\ s\sigma \to \mathsf{f_1}\ \mathsf{clock}\ s\sigma \;\gg\!=\; \mathsf{f_2}\ \mathsf{clock}
\end{aligned}
$$

Semantic decoration of code

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Code_s} : \forall\,\{\mathsf{st}\ \mathsf{st'}\} \to \mathsf{Dom_C}\ \mathsf{st}\ \mathsf{st'} \to \mathsf{Set}\ \textbf{where} \\
\quad \ldots \\
\quad \mathsf{load} \quad : \forall\,\{\mathsf{st}\} \to (\mathsf{x} : \mathsf{Var}) \to \mathsf{Code_s}\ (\mathsf{fload}\ \{\mathsf{st}\}\ \mathsf{x}) \\
\quad \mathsf{store} \quad : \forall\,\{\mathsf{st}\} \to (\mathsf{x} : \mathsf{Var}) \to \mathsf{Code_s}\ (\mathsf{fstore}\ \{\mathsf{st}\}\ \mathsf{x}) \\
\quad \mathsf{loop} \quad : \forall\,\{\mathsf{st}\}\ \{\mathsf{f_1}\ \mathsf{f_2}\} \to \\
\qquad\quad \mathsf{Code_s}\ \mathsf{f_1} \to \mathsf{Code_s}\ \mathsf{f_2} \to \mathsf{Code_s}\ (\mathsf{floop}\ \{\mathsf{st}\}\ \mathsf{f_1}\ \mathsf{f_2}) \\
\quad \_\text{,}\_ \quad : \forall\,\{\mathsf{st}\ \mathsf{st'}\ \mathsf{st''}\}\ \{\mathsf{f_1}\ \mathsf{f_2}\} \to \\
\qquad\quad \mathsf{Code_s}\ \mathsf{f_1} \to \mathsf{Code_s}\ \mathsf{f_2} \to \mathsf{Code_s}\ (\mathsf{fseqc}\ \{\mathsf{st}\}\ \{\mathsf{st'}\}\ \{\mathsf{st''}\}\ \mathsf{f_1}\ \mathsf{f_2}) \\
\quad \mathsf{subst_C} : \forall\,\{\mathsf{st}\ \mathsf{st'}\}\ \{\mathsf{f}\ \mathsf{g}\} \to \\
\qquad\quad \mathsf{Code_s}\ \{\mathsf{st}\}\ \{\mathsf{st'}\}\ \mathsf{f} \to \mathsf{f} \approx \mathsf{g} \to \mathsf{Code_s}\ \mathsf{g}
\end{array}
$$

Semantic decoration of code

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Code_s} : \forall\,\{\mathsf{st}\ \mathsf{st'}\} \to \mathsf{Dom_C}\ \mathsf{st}\ \mathsf{st'} \to \mathsf{Set}\ \textbf{where} \\
\quad \ldots \\
\quad \mathsf{load} \quad : \forall\,\{\mathsf{st}\} \to (\mathsf{x} : \mathsf{Var}) \to \mathsf{Code_s}\ (\mathsf{fload}\ \{\mathsf{st}\}\ \mathsf{x}) \\
\quad \mathsf{store} \quad : \forall\,\{\mathsf{st}\} \to (\mathsf{x} : \mathsf{Var}) \to \mathsf{Code_s}\ (\mathsf{fstore}\ \{\mathsf{st}\}\ \mathsf{x}) \\
\quad \mathsf{loop} \quad : \forall\,\{\mathsf{st}\}\ \{\mathsf{f_1}\ \mathsf{f_2}\} \to \\
\qquad\quad \mathsf{Code_s}\ \mathsf{f_1} \to \mathsf{Code_s}\ \mathsf{f_2} \to \mathsf{Code_s}\ (\mathsf{floop}\ \{\mathsf{st}\}\ \mathsf{f_1}\ \mathsf{f_2}) \\
\quad \_,\_ \quad : \forall\,\{\mathsf{st}\ \mathsf{st'}\ \mathsf{st''}\}\ \{\mathsf{f_1}\ \mathsf{f_2}\} \to \\
\qquad\quad \mathsf{Code_s}\ \mathsf{f_1} \to \mathsf{Code_s}\ \mathsf{f_2} \to \mathsf{Code_s}\ (\mathsf{fseqc}\ \{\mathsf{st}\}\ \{\mathsf{st'}\}\ \{\mathsf{st''}\}\ \mathsf{f_1}\ \mathsf{f_2}) \\
\quad \mathsf{subst_C} : \forall\,\{\mathsf{st}\ \mathsf{st'}\}\ \{\mathsf{f}\ \mathsf{g}\} \to \\
\qquad\quad \mathsf{Code_s}\ \{\mathsf{st}\}\ \{\mathsf{st'}\}\ \mathsf{f} \to \mathsf{f} \approx \mathsf{g} \to \mathsf{Code_s}\ \mathsf{g}
\end{array}
$$

$\mathsf{subst_C}$ is an extra constructor that aids the insertion of extensional equality proofs between semantic functions wherever necessary.

$$
\begin{array}{l}
\_\approx\_ : \forall\,\{\mathsf{st}\ \mathsf{st'}\} \to \mathsf{Dom_C}\ \mathsf{st}\ \mathsf{st'} \to \mathsf{Dom_C}\ \mathsf{st}\ \mathsf{st'} \to \mathsf{Set}\ \_ \\
\mathsf{f} \approx \mathsf{g} \ = \ \forall\ \mathsf{clock}\ \mathsf{s}\sigma \to \mathsf{f}\ \mathsf{clock}\ \mathsf{s}\sigma \equiv \mathsf{g}\ \mathsf{clock}\ \mathsf{s}\sigma
\end{array}
$$

Semantic decoration: code

The extra constructor is not reached by lifting.

$$\_\uparrow_s \; : \; \forall \, \{\mathsf{st}\; \mathsf{st'}\} \to (\mathsf{c} \; : \; \mathsf{Code_t}\; \mathsf{st}\; \mathsf{st'}) \to \mathsf{Code_s}\; (\mathcal{C}[\![\mathsf{c}]\!])$$

The semantics preserving compiler: expressions

$$
\begin{aligned}
&\mathsf{comp}_e \;:\; \forall\,\{\mathsf{t}\}\,\{\mathsf{f}\}\,\{\mathsf{st}\} \rightarrow \\
&\qquad\qquad \mathsf{Expr}_s\,\{\mathsf{t}\}\,\mathsf{f} \rightarrow \\
&\qquad\qquad \mathsf{Code}_s\,\{\mathsf{st}\}\,(\lambda\,\{\mathsf{clock}\,(\mathsf{s}\;,\;\sigma) \rightarrow \mathsf{just}\,((\mathsf{f}\;\sigma \rhd \mathsf{s})\;,\;\sigma)\}) \\
&\mathsf{comp}_e\;|\;\mathsf{n}\;| \;=\; \mathsf{push}\;\mathsf{n} \\
&\mathsf{comp}_e\;(\mathsf{e}_1 \oplus \mathsf{e}_2) \;=\; \mathsf{comp}_e\;\mathsf{e}_2\;,\;\mathsf{comp}_e\;\mathsf{e}_1\;,\;\mathsf{add} \\
&\mathsf{comp}_e\;(\mathsf{e}_1 \stackrel{\circ}{=} \mathsf{e}_2) \;=\; \mathsf{comp}_e\;\mathsf{e}_2\;,\;\mathsf{comp}_e\;\mathsf{e}_1\;,\;\mathsf{eq} \\
&\mathsf{comp}_e\;(\mathsf{var}\;\mathsf{x}) \;=\; \mathsf{load}\;\mathsf{x}
\end{aligned}
$$

The semantics preserving compiler: statements

$$\mathsf{comp_s} \;:\; \forall\,\{f\}\,\{st\} \to \mathsf{Stmt_s}\ f \to \mathsf{Code_s}\ (\mathsf{funCode}\ \{st\}\ f)$$

$$\mathsf{comp_s}\ (x := e) \;=\; \mathsf{comp}_e\ e\ ,\ \mathsf{store}\ x$$

$$\mathsf{comp_s}\ (\mathsf{while}\ \{fb\}\ \{f\}\ e\ stmt) \;=\; (\mathsf{loop}\ (\mathsf{comp}_e\ e)\ (\mathsf{comp_s}\ stmt))\ *$$

where $\_*\ :\ \mathsf{Code_s}\ \_ \to \_$

$$c\ * \;=\; \mathsf{subst_C}\ c\ (\mathsf{eqloop}\ fb\ f)$$

$$\mathsf{comp_s}\ (\_;\_\ \{f_1\}\ \{f_2\}\ stmt_1\ stmt_2) \;=\; (\mathsf{comp_s}\ stmt_1\ ,\ \mathsf{comp_s}\ stmt_2)\ *$$

where $\_*\ :\ \mathsf{Code_s}\ \_ \to \_$

$$c\ * \;=\; \mathsf{subst_C}\ c\ (\mathsf{eqseq}\ f_1\ f_2)$$

$$\mathsf{funCode}\ :\ \forall\,\{st\} \to \mathsf{Dom_S} \to \mathsf{Dom_C}\ st\ st$$

$$\mathsf{funCode}\ f\ \mathsf{clock}\ (s\ ,\ \sigma) \;=\; f\ \mathsf{clock}\ \sigma \;\gg\!=\; (\lambda\,\sigma' \to \mathsf{just}\ (s\ ,\ \sigma'))$$

Auxiliary proofs

$$
\begin{aligned}
&\mathsf{eqloop} : \forall\,\{\mathsf{st}\}\ \mathsf{fb}\ \mathsf{f} \to \\
&\qquad (\mathsf{floop}\ (\lambda\,\{\mathsf{clock}\ (\mathsf{s}\ ,\ \sigma) \to \mathsf{just}\ ((\mathsf{fb}\ \sigma \rhd \mathsf{s})\ ,\ \sigma)\})\\
&\qquad\quad (\mathsf{funCode}\ \mathsf{f}))\\
&\qquad\quad\ \approx\\
&\qquad (\mathsf{funCode}\ (\mathsf{fwhile}\ \mathsf{fb}\ \mathsf{f})))\\
&\mathsf{eqloop}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{zero}\ \mathsf{s}\sigma\ =\ \mathsf{refl}\\
&\mathsf{eqloop}\ \mathsf{f}_1\ \mathsf{f}_2\ (\mathsf{suc}\ \mathsf{clock})\ (\mathsf{s}\ ,\ \sigma)\ \mathbf{with}\ \mathsf{f}_1\ \sigma\\
&\mathsf{eqloop}\ \mathsf{f}_1\ \mathsf{f}_2\ (\mathsf{suc}\ \mathsf{clock})\ (\mathsf{s}\ ,\ \sigma)\ \mid\ \mathsf{false}\ =\ \mathsf{refl}\\
&\mathsf{eqloop}\ \mathsf{f}_1\ \mathsf{f}_2\ (\mathsf{suc}\ \mathsf{clock})\ (\mathsf{s}\ ,\ \sigma)\ \mid\ \mathsf{true}\ \mathbf{with}\ (\mathsf{f}_2\ (\mathsf{suc}\ \mathsf{clock})\ \sigma)\\
&\mathsf{eqloop}\ \mathsf{f}_1\ \mathsf{f}_2\ (\mathsf{suc}\ \mathsf{clock})\ (\mathsf{s}\ ,\ \sigma)\ \mid\ \mathsf{true}\ \mid\ \mathsf{nothing}\ =\ \mathsf{refl}\\
&\mathsf{eqloop}\ \mathsf{f}_1\ \mathsf{f}_2\ (\mathsf{suc}\ \mathsf{clock})\ (\mathsf{s}\ ,\ \sigma)\ \mid\ \mathsf{true}\ \mid\ \mathsf{just}\ \sigma'\\
&\quad =\ \mathsf{eqloop}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{clock}\ (\mathsf{s}\ ,\ \sigma')
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{funCode} : \forall\,\{\mathsf{st}\} \to \mathsf{Dom}_\mathsf{S} \to \mathsf{Dom}_\mathsf{C}\ \mathsf{st}\ \mathsf{st}\\
&\mathsf{funCode}\ \mathsf{f}\ \mathsf{clock}\ (\mathsf{s}\ ,\ \sigma)\ =\ \mathsf{f}\ \mathsf{clock}\ \sigma\ \gg\!=\ (\lambda\,\sigma' \to \mathsf{just}\ (\mathsf{s}\ ,\ \sigma'))
\end{aligned}
$$

Auxiliary proofs

```
eqseq : ∀ {st} f₁ f₂ →
          (fseqc (funCode f₁) (funCode f₂))
             ≈
          (funCode (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
```

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