Cubex with Yields

Ross Tate

November 26, 2013

Yield is a control-flow construct that lets control flow jump out of a function like on a return, but later return to the point of the yield. They enable you to write functions that behave like iterators. These functions are also called generators. An example generator would look as follows (pseudocode similar to cubex, but not the actual extension):

$$\textbf{fun}\ \text{squares}()\ :\ \textbf{Iterable}\langle\textbf{Integer}\rangle =\ \textbf{for}\ (n\ \textbf{in}\ 1...)\{\textbf{yield}\ n * n;\};$$

1 Prerequisites

Your compiler and typechecker must be able to check and compile generic class defintions and a restricted form of inheritance: all classes only need to be able to extend $\textbf{Iterable}\langle T\rangle$ or $\top$.

2 Extensions to the Language Specification

The cubex grammar is changed as described in Figure 1. As before, calls to super can be omitted. The old grammar rule for classes can be added to allow for class and interface inheritance as usual (not forbidden, but also not required for this extensions).

We add another way of type-checking statements analogous to type-checking returns, but this time we check yields. The main difference between type-checking returns and yields is that we do not care whether a yield occurs (hence the boolean superscript is absent).

We modify the type-checking of classes according to the changes in the grammar. As in the grammar, you may keep your old way of type-checking inheritance of classes and interfaces, except for those classes that now inherit from **Iterable**. The main change here is that the **yielder** block must be checked with the new rule for type-checking yields.

$$\begin{array}{rrcl} \text{class} & c & ::= & \textbf{class}\ \nu_c\langle\Theta\rangle(\Gamma)\ \textbf{extends}\ \top\ \{s\ldots s\ \textbf{super}();\ \textbf{fun}\ \nu_v\sigma\ s\ldots\textbf{fun}\ \nu_v\sigma\ s\} \\ & & | & \textbf{class}\ \nu_c\langle\Theta\rangle(\Gamma)\ \textbf{extends}\ \textbf{Iterable}\langle\tau\rangle\ \{s\ldots s\ \textbf{super}();\ \textbf{yielder}\ s\ \textbf{fun}\ \nu_v\sigma\ s\ldots\textbf{fun}\ \nu_v\sigma\ s\} \end{array}$$

Figure 1: Extension to the Cubex core language grammar.

3 Semantics

Semantically, the yielder-block is used as follows: Whenever an instance of the class is used as the iterable of a for-loop, the yielder-block is executed until the first yield-statement is encountered. The result of the expression of that yield statement is used as the first value for the loop variable. The state of the program at the point of the yield statement (i.e. the stack and position in the code) is saved and the loop statement is executed. At the next iteration of the loop, the saved state of the yielder is restored, and execution continues from the last yield statement, until the next yield statement is encountered, and so on, and so forth. If the end of the yielder-block is ever reached, the iterable is finite, and the loop terminates (it can of course also terminate if there is a return-statement inside the loop).

4 Implementation Hints

The following is a sketch of a naïve implementation of yields. First, observe that all the state you need to save for later are the variables that are mutable inside of the yielder. Create a struct that contains these variables (and some more fields that we will get to), and read and write from and to that struct whenever you read and write from and to those mutable variables. Convert the yielder to a function that takes the this-object and the struct as arguments, and returns the struct. Convert each yield-statement into a return-statement, followed by a goto-label. Number the pairs of returns/goto-labels, and have a field in the struct that you return to save the number of the return you used (i.e. have an assignment to that field

$$\boxed{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau s : \hat{\Gamma}}$$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau s : \hat{\Gamma}' \qquad \Psi \mid \Theta \vdash \hat{\Gamma}' <: \hat{\Gamma}''}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau s : \hat{\Gamma}''} \qquad \frac{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau s : \hat{\Gamma}', \nu : \hat{\tau}, \nu' : \hat{\tau}', \hat{\Gamma}''}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau s : \hat{\Gamma}', \nu' : \hat{\tau}', \nu : \hat{\tau}, \hat{\Gamma}''}$$

$$\frac{\text{for all } i, \quad \Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma}_{i-1} \vdash_\tau s_i : \hat{\Gamma}_i}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma}_0 \vdash_\tau \{s_1 \dots s_n\} : \hat{\Gamma}_n}$$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma} \vdash e : \hat{\tau}}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau \nu \coloneqq e; : \hat{\Gamma}, \nu : \hat{\tau}} \qquad \frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma}, \nu : \hat{\tau}, \hat{\Gamma}' \vdash e : \hat{\tau}'}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma}, \nu : \hat{\tau}, \hat{\Gamma}' \vdash_\tau \nu \coloneqq e; : \hat{\Gamma}, \nu : \hat{\tau}', \hat{\Gamma}'}$$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma} \vdash e : \texttt{Boolean}\langle\rangle \qquad \Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau s_1 : \hat{\Gamma}' \qquad \Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau s_2 : \hat{\Gamma}'}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau \textbf{if } (e) \ s_1 \textbf{ else } s_2 : \hat{\Gamma}'}$$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma} \vdash e : \texttt{Boolean}\langle\rangle \qquad \Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau s : \hat{\Gamma}}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau \textbf{while } (e) \ s : \hat{\Gamma}}$$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma} \vdash e : \texttt{Iterable}\langle\hat{\tau}\rangle \qquad \Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma}, \nu : \hat{\tau} \vdash_\tau s : \hat{\Gamma}}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau \textbf{for } (\nu \textbf{ in } e) \ s : \hat{\Gamma}}$$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma} \vdash e : \tau}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash_\tau \textbf{yield } e; : \hat{\Gamma}}$$

Figure 2: Type Checking Yields

before the return). At the beginning of the function, have a switch statement that looks at the number in the struct given as an argument and jumps to the respective label. Finally, you of course also need to return the current value that you are yielding, so have a field in the struct that holds the current value. Don't forget to have some indicator of when no value is yielded because the end of the yielder is reached (you need to insert returns for that, too). Now, all that is left to do is write the right code to use the function and the struct in for-loops.

$\boxed{\Psi \mid \Delta \mid \Gamma \vdash c : \Psi \mid \Delta}$

$$\frac{\begin{array}{c}\Psi' = \textbf{class } \nu\langle\Theta\rangle \textbf{ extends } \top \{\nu_1\sigma_1; \ldots; \nu_n\sigma_n;\} \qquad \Delta' = \nu\langle\Theta\rangle(\hat{\Gamma}) : \nu\langle\Theta\rangle \\ \text{for all } i, \qquad \Psi, \Psi' \mid \Theta \vdash \sigma_i \\ \Psi, \Psi' \mid \Theta \vdash \hat{\Gamma} \qquad \hat{\Gamma}_0 = \hat{\Gamma} \qquad \text{for all } i, \qquad \Psi, \Psi' \mid \Theta \mid \Delta, \Delta' \mid \Gamma \mid \hat{\Gamma}_{i-1} \vdash s_i : \hat{\Gamma}_i \\ \Psi, \Psi' \mid \Theta \vdash \nu\langle\Theta\rangle : \Delta'' \\ \text{for all } \bar{\nu}\langle\bar{\Theta}\rangle(\bar{\Gamma}) : \bar{\tau} \text{ in } \Delta'', \qquad \text{for all } \bar{\nu}' \text{ in } \bar{\Theta}, \qquad \bar{\nu}' \text{ not in } \Theta \\ \text{for all } i, \quad \sigma_i = \langle\Theta_i\rangle(\Gamma_i) : \tau_i \quad \Psi, \Psi' \mid \Theta, \Theta_i \mid \Delta, \Delta', \Delta'' \mid \Gamma, \hat{\Gamma}_m \mid \Gamma_i \vdash^{\textbf{true}}_{\tau_i} \hat{s}_i : \Gamma'_i \\ \text{for all } \hat{\nu}\sigma \text{ in } \Delta'', \qquad \Psi, \Psi' \mid \Theta \vdash \nu\langle\Theta\rangle : \hat{\nu}\end{array}}{\Psi \mid \Delta \mid \Gamma \vdash \textbf{class } \nu\langle\Theta\rangle(\hat{\Gamma}) \textbf{ extends } \top \{s_1 \ldots s_m \textbf{ super}();\ \textbf{fun } \nu_1\sigma_1\ \hat{s}_1\ \ldots\ \textbf{fun } \nu_n\sigma_n\ \hat{s}_n\} : \Psi' \mid \Delta'}$$

$$\frac{\begin{array}{c}\Psi \mid \Theta \vdash \tau \\ \Psi' = \textbf{class } \nu\langle\Theta\rangle \textbf{ extends Iterable}\langle\tau\rangle \{\nu_1\sigma_1; \ldots; \nu_n\sigma_n;\} \qquad \Delta' = \nu\langle\Theta\rangle(\hat{\Gamma}) : \nu\langle\Theta\rangle \\ \text{for all } i, \qquad \Psi, \Psi' \mid \Theta \vdash \sigma_i \\ \Psi, \Psi' \mid \Theta \vdash \hat{\Gamma} \qquad \hat{\Gamma}_0 = \hat{\Gamma} \qquad \text{for all } i, \qquad \Psi, \Psi' \mid \Theta \mid \Delta, \Delta' \mid \Gamma \mid \hat{\Gamma}_{i-1} \vdash s_i : \hat{\Gamma}_i \\ \Psi, \Psi' \mid \Theta \vdash \nu\langle\Theta\rangle : \Delta'' \\ \text{for all } \bar{\nu}\langle\bar{\Theta}\rangle(\bar{\Gamma}) : \bar{\tau} \text{ in } \Delta'', \qquad \text{for all } \bar{\nu}' \text{ in } \bar{\Theta}, \qquad \bar{\nu}' \text{ not in } \Theta \\ \text{for all } i, \quad \sigma_i = \langle\Theta_i\rangle(\Gamma_i) : \tau_i \quad \Psi, \Psi' \mid \Theta, \Theta_i \mid \Delta, \Delta', \Delta'' \mid \Gamma, \hat{\Gamma}_m \mid \Gamma_i \vdash^{\textbf{true}}_{\tau_i} \hat{s}_i : \Gamma'_i \\ \Psi, \Psi' \mid \Theta \mid \Delta, \Delta', \Delta'' \mid \Gamma, \hat{\Gamma}_m \mid \varnothing \vdash_\tau \tilde{s} : \tilde{\Gamma} \\ \text{for all } \hat{\nu}\sigma \text{ in } \Delta'', \qquad \Psi, \Psi' \mid \Theta \vdash \nu\langle\Theta\rangle : \hat{\nu}\end{array}}{\Psi \mid \Delta \mid \Gamma \vdash \textbf{class } \nu\langle\Theta\rangle(\hat{\Gamma}) \textbf{ extends Iterable}\langle\nu_p\rangle \{s_1 \ldots s_m \textbf{ super}();\ \textbf{yielder } \tilde{s}\ \textbf{fun } \nu_1\sigma_1\ \hat{s}_1\ \ldots\ \textbf{fun } \nu_n\sigma_n\ \hat{s}_n\} : \Psi' \mid \Delta'}$$

Figure 3: Class Checking