10

Function Implementation

Exercises

10.1 a. The concrete syntax in Fig. 10.2 would be changed as follows:

$$\textit{ReturnStatement} \rightarrow \text{return [} \textit{Expression} \text{] ;}$$

The abstract syntax would not change.

b. Type rule 10.4 would change as follows:

> A *Return* statement with an *Expression* must appear in the `body` of every non-void function except `main`, and that *Expression* must have the same *Type* as that function.

Type rule 10.5 would change as follows:

> A *Return* statement without an *Expression* may optionally appear in the `body` of any void function.

c. Meaning Rule 10.2 would change as follows:

> The meaning of a *Return* with an *Expression* is computed by replacing the value of the `target` *Variable* (the name of the called function) in the current state by the value of the `result` *Expression*.

Note that no changes are required for a *Return* appearing in a void function; no result variable is placed in the activation record, and Meaning Rule 10.3 enables normal return to the caller as soon as the *Return* is reached.

10.2 Using the program in Fig. 10.5, here are some violations of:
Type Rule 10.6: replace `answer=Fibonacci(8);` by `Fibonacci(8);`.
Type Rule 10.7: replace `Fibonacci(8)` by `Fibonacci()`.
Type Rule 10.8: replace `Fibonacci(8)` by `Fibonacci(8.5)`.

10.3 a. This change raises neither a syntax error nor a type error, since the resulting program is syntactically correct and has no type rule violations.

b. The Semantics interpreter loops, since the `fib.cpp` program itself now loops (k is never decremented). Finally, the interpreter raises a Java exception StackOverflowError, since the meaning function for a Clite `while` statement continues to call itself.

c. The problem cannot be avoided; it is an example of the well-known Halting Problem.

10.4 The type validity functions V in Section 10.4.2 cover the same ideas as the Type Rules in Section 10.2. The section **Validity of a Function** corresponds to Type Rules 10.1-10.3, and the section **Validity of a Call and Return** corresponds to Type Rules 10.4-10.8. Additional validity functions formalize the need for a unique `main` function and the validity of each function's type map.

10.5 This restriction does not naturally belong in Meaning Rule 10.1. A side effect (changing the value of a global variable) can only occur during interpretation of a Clite *Assignment* statement. Thus, the meaning of an assignment (as defined in Chapter 8 (Meaning Rule 8.3) would need to be changed in order to eliminate side effects.

Of course, if side effects are eliminated, there's no need for global variables.

10.6 The difficulty here is that the first function M in Section 10.4.3 returns a *State*, while the second returns a *Value*. To merge these into a single rule, we must rename them, say M_1 and M_2, and define a new meaning function M that returns a *State-Value* pair as follows:

$$\begin{aligned} M &: Call \times Function \times State \rightarrow State \times Value \\ M(c, f, \sigma) &= (M_1(c, f, \sigma), undef) && \text{if f is void} \\ &= (\sigma, M_2(c, f, \sigma)) && \text{if f is non-void} \end{aligned}$$

The meaning function for an *Expression* with a non-void function call should be altered to use the second member of this *State-Value* pair, while the meaning function for a *Call* statement should use the first.

Note also that this definition eliminates side effects for non-void functions.

10.7 If students combine the two sets of V functions, they will have begun to answer this question. Also, extensions of V for *Statement* and *Expression* in Chapter 6 are needed to define the validity of a *Call* when it appears in either context, and the validity of a *Return* when it appears as a *Statement*.

10.8 If students combine the two sets of M functions, they will have begun to answer this question. Also, extensions of M for *Statement* and *Expression* in Chapter 8 are needed to define the meaning of a *Call* when it appears in either context.