COMP3161/COMP9161 Supplementary Lecture Notes
TinyC

Gabriele Keller

May 13, 2013

1 TinyC

We use a small fragment of the C language for our discussion of an imperative programming language as defined in the following grammar:

$$
\begin{array}{lll}
\mathit{prgm} & ::= & \mathit{gdecs}\ \mathit{stmt} \\
\mathit{gdecs} & ::= & \epsilon \mid \mathit{gdec}\ \mathit{gdecs} \\
\mathit{gdec} & ::= & \mathit{fdec} \mid \mathit{vdec} \\
\mathit{vdecs} & ::= & \epsilon \mid \mathit{vdec}\ \mathit{vdecs} \\
\mathit{vdec} & ::= & \texttt{int}\ \mathit{Ident}\ \texttt{=}\ v\ \texttt{;} \\
\mathit{fdec} & ::= & \texttt{int}\ \mathit{Ident}_2\ \texttt{(}\mathit{arguments}\texttt{)}\ \mathit{stmt} \\
\mathit{stmt} & ::= & \mathit{expr}\ \texttt{;} \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2; \mid \texttt{return}\ \mathit{expr}\ \texttt{;} \mid \\
 & & \{\ \mathit{vdecs}\ \mathit{stmts}\ \} \mid \texttt{while}\ \texttt{(}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt} \\
\mathit{stmts} & ::= & \epsilon \mid \mathit{stmt}\ \mathit{stmts} \\
\mathit{expr} & ::= & \mathit{Num} \mid \mathit{Ident} \mid \mathit{expr}_1\ \texttt{+}\ \mathit{expr}_2 \mid \mathit{expr}_1\ \texttt{-}\ \mathit{expr}_2 \mid \\
 & & \mathit{Ident}\ \texttt{=}\ \mathit{expr} \mid \mathit{Ident}\ \texttt{(}\mathit{exprs}\texttt{)} \\
\mathit{arguments} & ::= & \epsilon \mid \texttt{int}\ \mathit{Ident}_2\ \texttt{,}\ \mathit{arguments}
\end{array}
$$

Again, we only include a small set of built-in operations and only a single value type — the techniques we present can be extended to handle to handle a more extensive set of operations (easy) and types (more tricky). The most striking omission, though, are pointers. The semantics of pointers and pointer arithmetic are a complex subject in itself, and we will discuss it separately later in the course. Note that all variables must be initialised to a given constant value.

A program p consists of a sequence of global declaration *gdecs* and a statement s. Global declarations can be either variable declarations or function declarations. The statement is usually a block $\{\mathit{ldecs}\ \mathit{ss}\}$, that is, a sequence of local declarations followed by a sequence with statements

We distinguish between statements and functions: statements are predominantly about effect, where as expressions are about value. The distinction is not very clear, though, since expressions also can have effect on the state, and there is implicitly a value associated with each statement.

It is important to note the difference between TinyC's variables and MinHs'. MinHs variables are variables in the strict mathematical sense: they can be bound to a value, but once they are bound, their value cannot change. In contrast to this, TinyC variables represent memory locations, whose content can (and usually does) change during the execution of the program.

For example, the program listed below contains two global declarations - a variable and a function declaration, and the program statement (the 'main' function of our TinyC program), simply calls this function and assigns the return value to the global variable `result`:

```
int result = 0;
int div (int x, int y)
  int res = 0;
  while (x > y)
    x = x - y;
    res = res + 1;

  return res;


result = div (16, 5);
```

1.1 Static Semantics

Since we only have a single type, `int`, and the programmer has no way of defining new data types, there is not much to be done in TinyC in terms of type checking. We can check statically, though, if all the variables have been declared and initialised, and make sure that functions are always applied to the correct number of arguments.

We need to collect all the functions and variables declared globally and locally, and check the statements and expressions against these two sets:

- Variables $V = \{x_1, x_2, \ldots\}$
- Functions and their arity $F = \{f_1 : n_1, f_2 : n_2, \ldots\}$

We overload **decs** to denote a relation over global declarations and a pair V', F' and a relation over local declarations and a set V': **decs** :

- $V, F \vdash gdec$ **decs** V', F', if a global declaration is well-formed with respect to the given grammar and a set of previously defined functions F and variables V, and declares the set of variables V' and a set of functions F'.
- $V \vdash ldec$ **decs** V', if a local declaration is well-formed with respect to the given grammar and a set of previously declared variables V, and declares the set of variables V'.

The following rules define the relation judgments:

- An empty sequence of local declarations is well-formed and declares no variables:

$$\frac{}{V \vdash \circ \ \mathbf{decs}\ \emptyset}$$

- A local declaration is well-formed if the variable has not been declared previously:

$$\frac{x \notin V \qquad V \cup \{x\} \vdash ldecs\ \mathbf{decs}\ V'}{V \vdash \texttt{int}\ x = v;\ ldecs\ \mathbf{decs}\ V' \cup \{x\}}$$

- An empty sequence of global declarations is well-formed and declares no variables:

$$\frac{}{V, F \vdash \circ \ \mathbf{decs}\ \emptyset}$$

- A global value declaration is well-formed if the variable has not been declared previously:

$$\frac{V \vdash \texttt{int}\ x = v\ \mathbf{decs}\ \{x\} \qquad V \cup \{x\}, F \vdash gdecs\ \mathbf{decs}\ V', F'}{V, F \vdash \texttt{int}\ x = v; gdecs\ \mathbf{decs}\ V' \cup \{x\}, F'}$$

- A global function declaration is well-formed if the function has not been declared previously:

$$\frac{f \notin F \qquad V, F \cup \{f : n\} \vdash \mathit{gdecs}\ \mathbf{decs}\ V', F'}{V, F \vdash \texttt{int}\ f(x_1, \dots, x_n) = \mathit{stmt};\mathit{gdecs}\ \mathbf{decs}\ V', F' \cup \{f : n\}}$$

Now we have to define a relation to check if a program is well-formed, and if an expression or statement is well-formed with respect to a given set of variable and function declarations. Again, we use a single symbol, **ok**, to express the relation on expressions, statements as well as programs, and write

- *prg* **ok** if a program is well-formed
- $V,\ F\ \vdash\ \mathit{expr}$ **ok**, if an expression is well-formed with respect to the current environments V and F
- $V,\ F\ \vdash\ \mathit{stmt}$ **ok**, if a statement is well-formed with respect to the current environments v and F

The following rules define the judgement:

- Statements and expressions are well-formed, if all of their components are well-formed:

$$\frac{V, F \vdash \mathit{expr}\ \mathbf{ok} \qquad V, F \vdash \mathit{stmt}\ \mathbf{ok}}{V, F \vdash \texttt{while(}\mathit{expr}\texttt{)}\ \mathit{stmt}\ \mathbf{ok}}$$

$$\frac{V, F \vdash \mathit{expr}\ \mathbf{ok} \qquad V, F \vdash \mathit{stmt}_1\ \mathbf{ok} \qquad V, F \vdash \mathit{stmt}_2\ \mathbf{ok}}{V, F \vdash \texttt{if(}\mathit{expr}\texttt{)}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \mathbf{ok}}$$

$$\frac{V, F \vdash \mathit{expr}_1\ \mathbf{ok} \qquad V, F \vdash \mathit{expr}_2\ \mathbf{ok}}{V, F \vdash \mathit{expr}_1 + \mathit{expr}_2\ \mathbf{ok}}$$

- A block is well-formed under F and V if the local declarations are well-formed and the statements are well-formed with respect to the current and block local declarations:

$$\frac{V, F \vdash \mathit{ldecs}\ \mathbf{decs}\ V' \qquad V \cup V', F \vdash \mathit{stmts}\ \mathbf{ok}}{V, F \vdash \{\mathit{ldecs}\ \mathit{stmts}\}\ \ \mathbf{ok}}$$

- A variable is well-formed if it is declared:

$$\frac{x \in V}{F, V \vdash x\ \mathbf{ok}}$$

- An assignment is well-formed if the expression is well-formed and the variable declared:

$$\frac{F, V \vdash \mathit{expr}\ \mathbf{ok} \quad x \in V}{F, V \vdash x = \mathit{expr}\ \mathbf{ok}}$$

- A function call is well-formed if the function is declared and called with the correct number of arguments:

$$\frac{F, V \vdash (\mathit{expr}_1, \dots, \mathit{expr}_n)\ \mathbf{ok} \quad f : n \in F}{F, V \vdash f(\mathit{expr}_1, \dots, \mathit{expr}_n)\ \mathbf{ok}}$$

1.2 Dynamic Semantics

We provide the dynamic semantics of TinyC in terms of a big-step semantics, similar to the one we used for MinHs. However, there is one major difference between the two languages: in TinyC, variables can (and generally do) change their value during execution. It is therefore not possible to deal with variables by replacing all occurrences with the value they are assigned to, as we did in MinHs. Instead, we have to keep a value environment, which contains all the currently visible variables together with their values. Since the scope of a function is the whole program, we also have to store the function definitions in the environment, although these definitions cannot change during execution.

1.2.1 The environment g

The environment has a similar form as the global declarations in the grammar: it consists of a set of function definitions of the form:

$\texttt{int}\, f(\texttt{int}\ x_1, \ldots)\ s$

and a ordered sequence of variable bindings $x = v$. Note that it is important that the variable bindings are ordered, since we have to be able to identify the binding of a variable put into the environment most recently.

We define the following operations on g

- **empty environment**: we write $\circ$ for the empty environment.
- **lookup:** We denote environment lookup using the infix operator @: so, $g \mathbin{@} x = 5$ means the current value of x is 5. If x is not in g, then $g@x$ is undefined.
- **update:** We write $g \mathbin{@} x \leftarrow 5$ to change the value of a variable x which already is in the environment to the value 5. Again, if x is not in g, $g \mathbin{@} x \leftarrow 5$ is undefined.
- **extending environment:** we write $g.\mathit{int}\ x = 4$ to add a new variable binding to an environment. If x is already in g, then the new binding overshadows the old binding (but does not delete it).

For example, $\circ .x = 5.x = 11 \mathbin{@} x$ returns 11 (the first binding of x is not accessible anymore), $\circ .x = 5.y = 11 \mathbin{@} x$ returns 11. Note that we do not need to define an operation to delete bindings from an environment.

1.2.2 Program Execution

A program can be executed without an environment, as all the variables and functions have to be declared in the program itself. For a program $p = \mathit{gdecsstmt}$ we write

$p \Downarrow (g',\ v;)$

If it evaluates to a value v and an environment g.

Given an environment g which contains bindings for all free variables in *stmt* and all functions, we write

$(g, \mathit{stmt}) \Downarrow (g',\ v)$

to express that *stmt* evaluates to the value v and the new environment g' under g, and similarly for expressions:

$(g,\ \mathit{expr}) \Downarrow (g', v)$

We start with the inference rules for if-statements. Apart from the environment g, which is threaded through the execution of all subexpressions and statements, they look almost the same as the MinHs rules (note that, in the absence of boolean values, we use 0 to represent false, and all other integer values to represent true):

$$\frac{(g, e) \Downarrow (g', 0) \qquad (g', s_2) \Downarrow (g'', rv)}{(g, \texttt{if(}e\texttt{) then } s_1 \texttt{ else } s_2) \Downarrow (g'', rv)}$$

$$\frac{(g, e) \Downarrow (g', v) \qquad (g', s_1) \Downarrow (g'', rv)}{(g, \texttt{if(}e\texttt{) then } s_1 \texttt{ else } s_2) \Downarrow (g'', rv)}$$

The evaluation of e, as well as the execution of all the statements, can change the environment — it is therefor important that we use the new environment g' for the evaluation of s_1 and s_2, respectively.

As with if-expressions, we have to evaluate the condition of the while-expression first, before we can decide how to continue. If it evaluates to 0 (false), the whole expression evaluates to 0, and we are finished:

$$\frac{(g, e) \Downarrow (g', 0)}{(g, \texttt{while(}e\texttt{)}s) \Downarrow (g', 0;)}$$

Otherwise, we have to execute the body of the while-expression, and then jump back to the while-loop:

$$\frac{(g, e) \Downarrow (g', v) \qquad (g', s;\ \texttt{while(}e\texttt{)}s) \Downarrow (g'', rv)}{(g, \texttt{while(}e\texttt{)}s) \Downarrow (g'', rv)}$$

Again, it is important that the new environment is always passed through.

Alternatively, we specify the semantics of a while-loop in a single rule, in terms of if-expressions:

$$\frac{(g, \texttt{if(}e\texttt{) then } \{s;\texttt{while(}e\texttt{)}s\} \texttt{ else } 0;) \Downarrow (g', rv;)}{(g, \texttt{while(}e\texttt{)}s) \Downarrow (g'', rv;)}$$

If a return statement contains just a value, it is not evaluated any further:

$$\frac{(g, e) \Downarrow (g', v)}{(g, \texttt{return(}e\texttt{);}) \Downarrow (g', \texttt{return(}v\texttt{);})}$$

Blocks introduce a set of new variable bindings into the environment:

$$\frac{(g.l, ss) \Downarrow (g'.l', rv)}{(g, \{l\ ss\}) \Downarrow (g', rv)}$$

Note that the statement sequence ss is executed under the old environment g plus the new local bindings. The resulting environment has the form $g'.l'$, meaning that bindings in g may have changed, and bindings in l may have changed, but no additional bindings have been introduced (it is necessary to look at all the inference rules to see why this is always the case). The resulting environment is g' — that is, we add the new bindings only temporary for the execution of ss, and remove them when leaving the block.

To evaluate a sequence of statements, the single statements are evaluated one after the other, passing on the resulting environment. If a statement evaluates to a return value, the remaining statements are ignored. Also, note how the results of non-return statements are discarded.

$$\frac{}{(g, \circ) \Downarrow (g, 0;)}$$

$$\frac{(g, s) \Downarrow (g', \texttt{return(}v\texttt{);})}{(g, s\ ss) \Downarrow (g', \texttt{return(}v\texttt{);})} \qquad \frac{(g, s) \Downarrow (g', v;) \quad (g', ss) \Downarrow (g'', v';)}{(g, s\ ss) \Downarrow (g'', v';)}$$

Variables are evaluated by looking up the binding in the environment, and assignment change the value of a variable in the environment:

$$\frac{g@x = v}{(g, x) \Downarrow (g, v)} \qquad \frac{(g, e) \Downarrow (g'.v)}{(g, x\texttt{=}e\texttt{;}) \Downarrow (g'@x \leftarrow v, v)}$$

The semantics of function application is expressed in terms of the semantics of block statements:

$$\frac{\begin{array}{c} g@f = \texttt{int}\ f\texttt{(}\texttt{int}\ x_1, \ldots\texttt{)}\ s \\ (g, (e_1, \ldots)) \Downarrow (g', (v_1, \ldots)) \quad (g', \{\texttt{int}\ x_1 = v_1; \ldots s\}) \Downarrow (g'', \texttt{return(}v\texttt{);}) \end{array}}{(g, f(e_1, \ldots)) \Downarrow (g'', v)}$$

$$\frac{\begin{array}{c} g@f = \texttt{int}\ f\texttt{(}\texttt{int}\ x_1, \ldots\texttt{)}\ s \\ (g, (e_1, \ldots)) \Downarrow (g', (v_1, \ldots)) \quad (g', \{\texttt{int}\ x_1 = v_1; \ldots s\}) \Downarrow (g'', v\texttt{;}) \end{array}}{(g, f(e_1, \ldots)) \Downarrow (g'', v)}$$

where

$$\frac{(g, e_1) \Downarrow (g_1, v_1), (g_1, e_2) \Downarrow (g_2, v_2) \ldots (g_{n-1}, e_n) \Downarrow (g_n, v_n)}{(g, (e_1, \ldots, e_n)) \Downarrow (g_n, (v_1, \ldots, v_n))}$$

Evaluation of function application is the only rule which "unwraps" a return statement. Combined with the rules for sequences of statements, this means that a return statement will effectively cause the evaluation to jump to the first statement after the innermost function call.