Christoph Lüth

Programmiersprachen

Wintersemester 2023/24

Lecture Notes

Last revision: December 21, 2023.

Vorlesung vom 19.10.2023: Auswertung von Ausdrücken

1 Preliminaries

1.1 Partial Maps

A finite partial map from X to Y, written $f : X \rightharpoonup Y$, is a *right-unique* relations:

$$f \subseteq X \times Y \text{ such that } (x,y) \in f \wedge (x,z) \in f \Longrightarrow y = z$$

For $f \in X \rightharpoonup Y$, $x \in X$ we define

$$f(x) = y \iff (x,y) \in f \tag{1}$$

The right-uniquess makes $f(x)$ well-defined (*i.e.* there is at most one y such that $f(x) = y$), but it may not be defined. In this case, we write $f(x) = \bot$.

Notation: we write *e.g.* $\langle x \mapsto 3, y \mapsto 5, z \mapsto 7\rangle$ for $\{(x,3),(y,5),(z,7)\}$.

The *domain* of f is the set of all x where $f(x)$ is defined:

$$\mathrm{dom}(f) = \{x \mid \exists y.\, (x,y) \in f\} \tag{2}$$

We also define pointwise erasure and update (for a finite partial map f):

$$f \setminus x \stackrel{def}{=} \{(y,a) \mid (y,a) \in f \wedge y \neq x\} \tag{3}$$

$$f[x \mapsto y] \stackrel{def}{=} (f \setminus x) \cup \{(x,y)\} \tag{4}$$

Note how the update removes any existing map of x. Clearly, if f is right-unique so is $f \setminus x$ for any $x \in X$. Further, $\mathrm{dom}(f \setminus x) = \mathrm{dom}\, f \setminus \{x\}$. Hence,if f is right-unique so is $f[x \mapsto y]$.

We also have the following equations between update, lookup and remove:

$$(f[x_1 \mapsto y])(x_2) = \begin{cases} y & x_1 = x_2 \\ f(x_2) & x_1 \neq x_2 \end{cases} \tag{5}$$

$$(f \setminus x_1)(x_2) = \begin{cases} f(x_2) & x_1 \neq x_2 \\ \bot & x_1 = x_2 \end{cases} \tag{6}$$

$$(f[x_1 \mapsto y])[x_2 \mapsto z] = \begin{cases} (f[x_2 \mapsto z])[x_2 \mapsto y] & x_1 \neq x_2 \\ f[x_2 \mapsto z] & x_1 = x_2 \end{cases} \tag{7}$$

$$\tag{8}$$

Readers are invited to find the equations describing the interaction between $f[x \mapsto y]$ and $f \setminus z$, *i.e.* $(f[x_1 \mapsto y]) \setminus x_2 =?$ and $(f \setminus x_1)[x_2 \mapsto y] =?$.

Finally, the empty set $\emptyset \subseteq X \times Y$ is the empty map which is undefined everywhere. It is obviously right-unique, and $\mathrm{dom}(\emptyset) = \emptyset$.

2 Expressions

2.1 Abstract Syntax

The set **Exp** of all expressions is given as

$$\begin{aligned} e ::= \mathbb{Z} \mid l \mid \mathit{true} \mid \mathit{false} \\ \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\ \mid e_1 == e_2 \mid e_1 < e_2 \\ \mid\ !e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \end{aligned}$$

- This is *abstract syntax*, so there are not parentheses or operator precedences.
- We also allow the following as *syntactic sugar* (*i.e.* abbreviations):

  $$e ::= e_1 \neq e_2 \mid e_1 \leq e_2 \mid e_1 > e_2 \mid e_1 \geq e_2$$

  with the obvious translations: [1]

  $$\begin{aligned} e_1 \neq e_2 &\stackrel{def}{=} !(e_1 == e_2) \\ e_1 \leq e_2 &\stackrel{def}{=} !(e_2 < e_1) \\ e_1 > e_2 &\stackrel{def}{=} e_2 < e_1 \\ e_1 \geq e_2 &\stackrel{def}{=} !(e_1 < e_2) \end{aligned}$$

- We include denotations of literals directly into the syntax, *i.e.* we do not distinguish betwen the character sequence `34755` und the number $34755 \in \mathbb{Z}$; similary, we take the set of boolean values to be $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$.
- No function calls (yet).
- Note how terms in e can be represented as *trees*, so they always have a top symbol and (optionally) child nodes.

2.2 Evaluation

- Evaluation takes a *state* σ and an expression e, and reduces it ultimately to a *value* v.
- Values are either integers, or booleans: $\mathbf{V} \stackrel{def}{=} \mathbb{Z} \uplus \mathbb{B}$.
- However, the set of all states is $\Sigma = \mathbf{Idt} \rightharpoonup \mathbb{Z}$ (we only habe integer-value variables in the state).
- Evaluation is defined inductively in Figure 1 as a relation $\rightarrow_{Exp} \subseteq \mathbf{Exp} \times \Sigma \times \mathbf{V}$, written as $\langle e, \sigma \rangle \rightarrow v$ iff $((e, \sigma, v) \in \rightarrow_{Exp}$.
- Do all expressions evaluate?
- Evaluation can get "stuck". This corresponds to undefined expressions.

[1] Note that these translations only evaluate their arguments once, as opposed to *e.g.* $e_1 \leq e_2 \stackrel{def}{=} e_1 < e_2 \,||\, e_1 == e_2$.

$$\frac{i \in \mathbb{Z}}{\langle i, \sigma \rangle \to_{Exp} i} \qquad \frac{b \in \mathbb{B}}{\langle b, \sigma \rangle \to_{Exp} b} \qquad \frac{x \in \mathbf{Idt}, x \in \mathrm{dom}(\sigma), \sigma(x) = v}{\langle x, \sigma \rangle \to_{Exp} v}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}}{\langle e_1 + e_2, \sigma \rangle \to_{Exp} n_1 + n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}}{\langle e_1 - e_2, \sigma \rangle \to_{Exp} n_1 - n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}}{\langle e_1 * e_2, \sigma \rangle \to_{Exp} n_1 \cdot n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_2 \neq 0}{\langle e_1 / e_2, \sigma \rangle \to_{Exp} n_1 \div n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 = n_2}{\langle e_1 == e_2, \sigma \rangle \to_{Lexp} \mathit{true}}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 \neq n_2}{\langle e_1 == e_2, \sigma \rangle \to_{Lexp} \mathit{false}}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 < n_2}{\langle e_1 < e_2, \sigma \rangle \to_{Lexp} \mathit{true}}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 \geq n_2}{\langle e_1 < e_2, \sigma \rangle \to_{Lexp} \mathit{false}}$$

$$\frac{\langle e, \sigma \rangle \to_{Lexp} \mathit{true}}{\langle !\, e, \sigma \rangle \to_{Lexp} \mathit{false}} \qquad \frac{\langle e, \sigma \rangle \to_{Lexp} \mathit{false}}{\langle !\, e, \sigma \rangle \to_{Lexp} \mathit{true}}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Lexp} \mathit{false}}{\langle e_1 \,\&\&\, e_2, \sigma \rangle \to_{Lexp} \mathit{false}} \qquad \frac{\langle e_1, \sigma \rangle \to_{Lexp} \mathit{true} \qquad \langle e_2, \sigma \rangle \to_{Lexp} t}{\langle e_1 \,\&\&\, e_2, \sigma \rangle \to_{Lexp} t}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Lexp} \mathit{true}}{\langle e_1 \,||\, e_2, \sigma \rangle \to_{Lexp} \mathit{true}} \qquad \frac{\langle e_1, \sigma \rangle \to_{Lexp} \mathit{false} \qquad \langle e_2, \sigma \rangle \to_{Lexp} t}{\langle e_1 \,||\, e_2, \sigma \rangle \to_{Lexp} t}$$

Figure 1: Rules to evaluate expressions

$$\frac{}{\Gamma \vdash n : \textbf{int}} \qquad \frac{x \in \mathrm{dom}(\Gamma), \Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}} \qquad \frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 - e_2 : \textbf{int}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 * e_2 : \textbf{int}} \qquad \frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 / e_2 : \textbf{int}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 == e_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 < e_2 : \textbf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash e_1 \,\&\&\, e_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash e_1 \,||\, e_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash\, !e : \textbf{bool}}$$

Figure 2: Typing judgements for expresssions

- Note conjunction and disjunction: if first argument evaluates to *false* (*true*) second argument is not considered (short-circuit evaluation).
- Corresponds semantically to *non-strictness* in second argument: undefinedness is not propagated.
- The semantics is *deterministic*, *i.e.* a given expression evaluates to *at most* one value:

$$\langle e, \sigma \rangle \to v_1 \wedge \langle e, \sigma \rangle \to v_2 \Longrightarrow v_1 = v_2 \tag{9}$$

2.3 Linear Notation

The mathematical notation to draw inference trees gets large very quickly, therefore we use a more spatially economic *linear* notation, where the tree structure is represented by the indentation level.

$$\begin{aligned}
\langle 3, \sigma \rangle &\to 3 \\
\quad \langle 2, \sigma \rangle &\to 2 \\
\quad \langle x, \sigma \rangle &\to 5 \\
\langle 2 * x, \sigma \rangle &\to 2 \cdot 5 = 7 \\
\langle 3 + 2 * x, \sigma \rangle &\to 3 + 10 = 13
\end{aligned}$$

Vorlesung vom 23.10.2023: Simple Type Systems

2.4 Simple Type Systems

- Evaluation can get stuck for many reasons:

  - Variables not defined.
  - Wrong values, *e.g.* $x + (y == z)$ or $true = false$.
  - Division by zero

- We want to prevent this by typing before we run the program, so we split the evaluation of the program into a *static analysis* which is decidable (and performed at compile time), and *dynamic evaluation*, which corresponds to run-time.
- Typing associates expressions with types, presently only integers and booleans; we will later extend this substantially. The set of all types is written as

$$\mathcal{T}ypes \stackrel{def}{=} \{\mathbf{int}, \mathbf{bool}\}$$

- Typing needs a *type context* (to keep track of the types of the variables) $\mathscr{C} \stackrel{def}{=} \mathbf{Idt} \rightharpoonup \mathcal{T}ypes$. At the moment, the typing context is just given, later on it will be built from declarations (obviously).
- The typing relation $: \subseteq \mathscr{C} \times \mathbf{Exp} \times \mathcal{T}ypes$ (yes, it is called :, funny name I know, blame the parents), written as $\Gamma \vdash e : t$ for $(\Gamma, e, t) \in :$, is defined in Figure 2.
- Why does the following not hold ("Well-typed programs don't go wrong."):

$$\forall \Gamma, \sigma, e, t. \Gamma \vdash e : t \Longrightarrow \exists v. \langle e, \sigma \rangle \to v \tag{10}$$

- σ has to be well-typed, *i.e.* $\operatorname{dom}\Gamma = \operatorname{dom}\sigma$
- But even if σ is well-typed, this does not hold because of division by 0 — how to fix this?
- We will introduce error elements and exceptions later, but it is important to realize that partiality is a *feature*, not a bug — Turing-equivalent languages *have* to be partial.
- We do not have (10), but we have the following.[2]

$$\Gamma \vdash e : \mathbf{int} \wedge \langle \sigma, e \rangle \to v \Longrightarrow v \in \mathbb{Z} \tag{11}$$

  This means we do not have to keep track of types at run-time and is sometimes referred to as *type erasure*.

[2] This can only be written so concisely if all variables are of type **int**; otherwise the context and the state would have to agree on types.

Vorlesung vom 23.10.2023: Simple Types, Commands and Side Effects

3 A Simple Imperative Language: L_0

3.1 Abstract Syntax

$$c ::= \textbf{Idt} := \textbf{Exp} \mid \textbf{if}\ (e)\ \textbf{then}\, c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ (e)\ c \mid c_1; c_2 \mid \textbf{nil}$$

- Note we have concatenation and empty sequence – enough to build sequences, but easier.

3.2 Evaluation

- Evaluation of statement works with the same state as evaluation of expressions, $\Sigma = \textbf{Idt} \rightharpoonup \textbf{V}$. But the important difference is that evaluation of statements returns a new state, not a value.
- Thus, evaluation of statements is a relation $\rightarrow_{Stmt} \subseteq \textbf{Stmt} \times \Sigma \times \Sigma$, written as $\langle s, \sigma\rangle \rightarrow_{Stmt} \sigma'$. It is defined inductively in Figure 3.
- Assignment does not require the variable x to be in the domain of the state σ. This means either variables in the state are created by assignment (as in Python), or we leave it to the type inference (see below) to prevent this situation.
- In general, we have

  $$\forall \sigma, c, \sigma'. \langle c, \sigma\rangle \rightarrow_{Stmt} \sigma' \Longrightarrow \mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$$

  (*i.e.* variables are not removed from the state).
- Evaluation may not terminate for the while loop. Moreover, it is undecidable wether the evaluation will eventually terminate. This partiality is part and parcel of Turing-completeness, as opposed to division by zero which we can work around (we will later see how).
- As given, evaluation is *deterministic:*

  $$\langle c, \sigma\rangle \rightarrow_{Stmt} \sigma_1 \wedge \langle c, \sigma\rangle \rightarrow_{Stmt} \sigma_2 \Longrightarrow \sigma_1 = \sigma_2$$

  In other words, a commands evaluates a state σ to *at most* one successor state σ'. (This holds for expressions as well.)

3.3 Typing

- Typing of statements is pretty basic, we just have to make sure that the arguments of if and while are booleans. The type of statements is written **unit**; the rules are given in Figure 4.
- As given, we only allow integer variables, and we require variables to be declared before assignment. How would we change the first rule so that we can create integer variables by just assigning to them? And what would be needed to have variables of boolean type as well?

$$\frac{\langle e, \sigma\rangle \to_{Exp} n \in \mathbb{Z}}{\langle x := e, \sigma\rangle \to_{Stmt} \sigma[x \mapsto n]} \qquad \frac{}{\langle \sigma, \textbf{nil}\rangle \to_{Stmt} \sigma}$$

$$\frac{\langle c_1, \sigma\rangle \to_{Stmt} \sigma' \qquad \langle c_2, \sigma'\rangle \to_{Stmt} \sigma''}{\langle c_1; c_2, \sigma\rangle \to_{Stmt} \sigma''}$$

$$\frac{\langle b, \sigma\rangle \to_{Lexp} \mathit{true} \qquad \langle c_1, \sigma\rangle \to_{Stmt} \sigma'}{\langle \textbf{if}\ (b)\ \textbf{then}\, c_1\ \textbf{else}\ c_2, \sigma\rangle \to_{Stmt} \sigma'} \qquad \frac{\langle b, \sigma\rangle \to_{Lexp} \mathit{false} \qquad \langle c_2, \sigma\rangle \to_{Stmt} \sigma'}{\langle \textbf{if}\ (b)\ \textbf{then}\, c_1\ \textbf{else}\ c_2, \sigma\rangle \to_{Stmt} \sigma'}$$

$$\frac{\langle b, \sigma\rangle \to_{Lexp} \mathit{false}}{\langle \textbf{while}\ (b)\ c, \sigma\rangle \to_{Stmt} \sigma}$$

$$\frac{\langle b, \sigma\rangle \to_{Lexp} \mathit{true} \qquad \langle c, \sigma\rangle \to_{Stmt} \sigma' \qquad \langle \textbf{while}\ (b)\ c, \sigma'\rangle \to_{Stmt} \sigma''}{\langle \textbf{while}\ (b)\ c, \sigma\rangle \to_{Stmt} \sigma''}$$

Figure 3: Rules to evaluate statements

$$\frac{x \in \mathrm{dom}(\Gamma) \qquad \Gamma \vdash e : \textbf{int}}{\Gamma \vdash x := e : \textbf{unit}}$$

$$\frac{\Gamma \vdash b : \textbf{bool} \qquad \Gamma \vdash c : \textbf{unit}}{\Gamma \vdash \textbf{while}\ (b)\ c : \textbf{unit}}$$

$$\frac{\Gamma \vdash b : \textbf{bool} \qquad \Gamma \vdash c_1 : \textbf{unit} \qquad \Gamma \vdash c_2 : \textbf{unit}}{\Gamma \vdash \textbf{if}\ (b)\ \textbf{then}\, c_1\ \textbf{else}\ c_2 : \textbf{unit}}$$

$$\frac{\Gamma \vdash c_1 : \textbf{unit} \qquad \Gamma \vdash c_2 : \textbf{unit}}{\Gamma \vdash c_1; c_2 : \textbf{unit}} \qquad \frac{}{\Gamma \vdash \textbf{nil} : \textbf{unit}}$$

Figure 4: Typing rules for statements

3.4 Expressions with Side Effects

Many programming languages (except for Haskell) allow expressions with side effects. How do we model evaluation of these?

In order to study this phenomenon, we need to alter our language slightly.

3.4.1 Abstract Syntax

$$
\begin{aligned}
e ::= {} & \mathbb{Z} \mid l \mid \mathit{true} \mid \mathit{false} \\
& \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
& \mid e_1 == e_2 \mid e_1 < e_2 \\
& \mid\ !e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \\
& \mid i := e
\end{aligned}
$$

$$
c ::= e \mid \textbf{if}\ (e)\ \textbf{then}\, c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ (e)\ c \mid c_1; c_2 \mid \textbf{nil}
$$

- Assignment is now an expression, and an expression is also a statement. So, all previous programs are still valid.

3.4.2 Evaluation

- Evaluation of an expression gives a *tuple* of resulting value, and a new state: $\langle e, \sigma \rangle \to_{Exp} \langle v, \sigma' \rangle$.
- Possible rules are given in Figure 5. We need to add the state in all rules for the expression, we need a rule for the assignment expression, and a new rule for an expression as a statement.
- The rules for the other statements stay as they are in Figure 3.
- Evaluating the assignment expressions changes the state as before, but returns the value of the right-hand-side of the expression.
- This is like in C, and allows to write *chain assignments* as in `a= b= c= 3*x+ 5`. Interestingly, in Rust assigments are an expression as well, but with type `()` (and value `()`, to make it impossible to write chain assignments, and to keep the programmer from confusing `=` and `==`.
- When evaluating an expression as a statement, we just discard the value of the expression; we evaluate it purely for its side effect (the changed state σ').
- The rules in Figure 5 evaluate the arguments of binary operators and logical connectors left-to-right. However, here different languages use different definitions. To evaluate right-to-left, the state needs to be passed the other way around (we only show the addition rule here; the same applies to subtraction, multiplication, division, and the two predicates $==$ and $<$):

$$
\frac{\langle e_1, \sigma_1 \rangle \to_{Exp} \langle n_1, \sigma_2 \rangle \qquad \langle e_2, \sigma \rangle \to_{Exp} \langle n_2, \sigma_1 \rangle \qquad n_i \in \mathbb{Z}}{\langle e_1 + e_2, \sigma \rangle \to_{Exp} \langle n_1 + n_2, \sigma_2 \rangle}
$$

  The C language does not specify in which order operands are evaluated. To model this behaviour, one takesq *both* rules, leading to a *non-deterministic* semantics.[3]

[3] The truth is even more complicated, C allows the side effects of the operands to be combined arbitrarily, so when we evaluate

$$\frac{i \in \mathbb{Z}}{\langle i, \sigma\rangle \to_{Exp} \langle i, \sigma\rangle} \qquad \frac{b \in \mathbb{B}}{\langle b, \sigma\rangle \to_{Exp} \langle b, \sigma\rangle} \qquad \frac{x \in \mathbf{Idt}, x \in \mathrm{dom}(\sigma), \sigma(x) = v}{\langle x, \sigma\rangle \to_{Exp} \langle v, \sigma\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle n_1, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \to_{Exp} \langle n_2, \sigma_2\rangle \qquad n_i \in \mathbb{Z}}{\langle e_1 + e_2, \sigma\rangle \to_{Exp} \langle n_1 + n_2, \sigma_2\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle n_1, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \to_{Exp} \langle n_2, \sigma_2\rangle \qquad n_i \in \mathbb{Z}}{\langle e_1 - e_2, \sigma\rangle \to_{Exp} \langle n_1 - n_2, \sigma_2\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle n_1, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \to_{Exp} \langle n_2, \sigma_2\rangle \qquad n_i \in \mathbb{Z}}{\langle e_1 * e_2, \sigma\rangle \to_{Exp} \langle n_1 \cdot n_2, \sigma_2\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle n_1, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \to_{Exp} \langle n_2, \sigma_2\rangle \qquad n_i \in \mathbb{Z}, n_2 \neq 0}{\langle e_1 / e_2, \sigma\rangle \to_{Exp} \langle n_1 \div n_2, \sigma_2\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle n_1, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \to_{Exp} \langle n_2, \sigma_2\rangle \qquad n_i \in \mathbb{Z}, n_1 = n_2}{\langle e_1 == e_2, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma_2\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle n_1, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \to_{Exp} \langle n_2, \sigma_2\rangle \qquad n_i \in \mathbb{Z}, n_1 \neq n_2}{\langle e_1 == e_2, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma_2\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle n_1, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \to_{Exp} \langle n_2, \sigma_2\rangle \qquad n_i \in \mathbb{Z}, n_1 < n_2}{\langle e_1 < e_2, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma_2\rangle}$$

$$\frac{\langle e_1, \sigma_1\rangle \to_{Exp} \langle n_1, \sigma_1\rangle \qquad \langle e_2, \sigma\rangle \to_{Exp} \langle n_2, \sigma_2\rangle \qquad n_i \in \mathbb{Z}, n_1 \geq n_2}{\langle e_1 < e_2, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma_2\rangle}$$

$$\frac{\langle e, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma_1\rangle}{\langle !\, e, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma_1\rangle} \qquad \frac{\langle e, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma_1\rangle}{\langle !\, e, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma_1\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma_1\rangle}{\langle e_1 \,\&\&\, e_2, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma_1\rangle} \qquad \frac{\langle e_1, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \to_{Exp} \langle t, \sigma_2\rangle}{\langle e_1 \,\&\&\, e_2, \sigma\rangle \to_{Exp} \langle t, \sigma_2\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma_1\rangle}{\langle e_1 \,||\, e_2, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma_1\rangle} \qquad \frac{\langle e_1, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma_1\rangle \qquad \langle e_2, \sigma_1\rangle \to_{Exp} \langle t, \sigma_2\rangle}{\langle e_1 \,||\, e_2, \sigma\rangle \to_{Exp} \langle t, \sigma_2\rangle}$$

$$\frac{\langle e, \sigma\rangle \to_{Exp} \langle n, \sigma'\rangle \qquad n \in \mathbb{Z}}{\langle x := e, \sigma\rangle \to_{Exp} \langle n, \sigma'[x \mapsto n]\rangle}$$

$$\ldots$$

$$\frac{\langle e, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle}{\langle e, \sigma\rangle \to_{Stmt} \sigma'}$$

$$\ldots$$

Figure 5: Rules to evaluate expressions with side effects.

the left (or right) operand, neither side effects may have taken place.

Vorlesung vom 30.10.2023: Variables and Memory Models

4 Names

We study local names in expressions first. This demonstrates how to handle names, and thus explores the concept of *scope*, without mutability or any aspects of life-time.

4.1 Abstract Syntax

$$\begin{aligned} e ::= {} & \mathbb{Z} \mid i \mid \mathit{true} \mid \mathit{false} \\ & \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\ & \mid e_1 == e_2 \mid e_1 < e_2 \\ & \mid\ !e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \\ & \mid i := e \\ & \mid \textbf{let } x = e_1 \textbf{ in } e_2 \end{aligned}$$

The typing is pretty easy: derive a type for e_1, and give that type to x while typing e_2.

$$\frac{\Gamma \vdash e_1 : \alpha \qquad \Gamma[x \mapsto \alpha] \vdash e_2 : \beta}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \beta}$$

Here, $\alpha, \beta \in \mathit{Types}$ are variables standing for arbitrary types (yes, both of them), so we allow names of type **int** and **bool**.

4.2 Evaluation

The big difference between local names and (mutable) variables like below are that names can be handled at the syntactic level. In other words, an expression like

let $x = 4 + 2 * y$ **in** $y < x$

should be evaluated as if we substitute $4 + 2 * y$ for x in the expression $y < x$, yielding $y < 4 + 2 * y$. This substitution is written as $(y < x)[4 + 2 * y/x]$, or in general $e_1[e_2/x]$ for "in the expression e_1, substitute the expression e_2 for the variable x".

- Do we evaluate first, and then substitute, or the other way around?
- How to define substitution? More delicate than it appears at first sight.
- Why? Consider

```
let x= 5 in
  let y= 2*x in
    (let x= 3 in x+y)+ x
```

$$
\begin{aligned}
\mathrm{FV}(n) &= \emptyset \\
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(\mathit{true}) &= \emptyset \\
\mathrm{FV}(\mathit{false}) &= \emptyset \\
\mathrm{FV}(e_1 + e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 - e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 * e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 / e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 == e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 < e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(!\,e) &= \mathrm{FV}(e) \\
\mathrm{FV}(e_1 \,\&\&\, e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 \,||\, e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(i := e) &= \mathrm{FV}(e) \\
\mathrm{FV}(\textbf{let } x = e_1 \textbf{ in } e_2) &= \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) \setminus \{x\})
\end{aligned}
$$

$$
\begin{aligned}
n[e/x] &= n \\
y[e/x] &= \begin{cases} e & x = y \\ y & \text{otherwise} \end{cases} \\
\mathit{true}[e/x] &= \mathit{true} \\
\mathit{false}[e/x] &= \mathit{false} \\
(e_1 + e_2)[e/x] &= (e_1[e/x]) + (e_2[e/x]) \\
(e_1 - e_2)[e/x] &= (e_1[e/x]) - (e_2[e/x]) \\
(e_1 * e_2)[e/x] &= (e_1[e/x]) * (e_2[e/x]) \\
(e_1 / e_2)[e/x] &= (e_1[e/x]) / (e_2[e/x]) \\
(e_1 == e_2)[e/x] &= (e_1[e/x]) == (e_2[e/x]) \\
(e_1 < e_2)[e/x] &= (e_1[e/x]) < (e_2[e/x]) \\
(!\,e_1)[e_2/x] &= !(e_1[e_2/x]) \\
(e_1 \,\&\&\, e_2)[e_3/x] &= (e_1[e_3/x]) \,\&\&\, (e_2[e_3/x]) \\
(e_1 \,||\, e_2)[e_3/x] &= (e_1[e_3/x]) \,||\, (e_2[e_3/x]) \\
(x := e_1)[e_2/y] &= (x := (e_1[e_2/y])) \\
(\textbf{let } x = e_1 \textbf{ in } e_2)[e_3/y] &= \begin{cases} \textbf{let } x = e_1[e_3/y] \textbf{ in } e_2 & x = y \\ \textbf{let } x = e_1[e_3/y] \textbf{ in } e_2[e_3/y] & x \neq y, x \notin \mathrm{FV}(e_3) \\ \textbf{let } z = e_1[e_3/y] \textbf{ in } (e_2[z/x])[e_3/y] & x \neq y, x \in \mathrm{FV}(e_3), z \notin \mathrm{FV}(e_3) \cup \mathrm{FV}(e_2) \end{cases}
\end{aligned}
$$

Figure 6: Definition of the substitution function.

- This *binding* occurrences of x and y are the essence of local names and scope.
- This needs concept of *free variables*.
- Figure 6 shows definition of substitution, and definition of free variables.
- The z in the final clause is a "fresh" variable. We can pick any z that satisfies $z \notin \mathrm{FV}(e_3) \cup \mathrm{FV}(e_2)$. If follows that $z \notin \mathrm{FV}(e_3)$, and $x \in \mathrm{FV}(e_3)$, hence $z \neq x$ (*i.e.* we cannot pick x).
- Note that variables on the left of assignments are considered completely different names. In other words, in **let** $x = 5$ **in** $x := x + 5$ the x in the rhs of the assignment is local name x, and the x on the lhs is the mutable variable x; subsequently, this expression evaluates to 10, and sets x to that value, rather than increasing the variable x by 5 and returning that value.

We can now define two rules which define the evaluation semantics of the names. Important: these are *alternatives*, we can pick one or the other, but not both!

$$\frac{\langle e_2[e_1/x], \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle}{\langle \textbf{let } x = e_1 \textbf{ in } e_2, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle} \tag{12}$$

$$\frac{\langle e_1, \sigma\rangle \to_{Exp} \langle v_1, \sigma'\rangle \qquad \langle e_2[v_1/x], \sigma'\rangle \to_{Exp} \langle v_2, \sigma''\rangle}{\langle \textbf{let } x = e_1 \textbf{ in } e_2, \sigma\rangle \to_{Exp} \langle v_2, \sigma''\rangle} \tag{13}$$

(14)

- (12) is non-strict; the effect is that e_2 is evaluated when needed (maybe multiple times, but not at all if not needed).
- (13) is strict: e_1 is evaluated exactly once.

5 Variables and Memory Models

- In general, a memory is something mapping *locations* to values: $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}$
- Faithful model of the machine/ISA level:

  $$\mathbf{Loc} = [0, \ldots, 2^W] \quad \mathbf{V} = [0, \ldots, 2^8]$$

  where W is the architecture word width (*e.g.* 32 bits).
- Slightly more abstract:

  $$\mathbf{Loc} = \mathbb{N}, \mathbf{V} = \mathbb{Z}$$

- We will look at composite values (arrays, structures etc.) later. First, get the basics right.
- Memory model has to explain house-keeping of variables:
  - Allocation
  - Deallocation
  - *Life cycle*

- When a variable is allocated, it does not (necessarily) have a value; we allow *uninitialised* memory locations (like in C and Rust; in Java, every memory cell has a default value). So our values are either integers, or a value $\bot$ for "undefined". This is written as $\mathbb{Z}_\bot = \mathbb{Z} \uplus \{\bot\}$.
- Hence, our memory model is

$$\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}, \mathbf{Loc} = \mathbb{N}, \mathbf{V} = \mathbb{Z}_\bot$$

- Note difference between "undefined" and "uninitialised".

5.1 Abstract Syntax

We add a single command to declare new variables:

$$\begin{aligned} c ::= {} & e \\ & \mid \textbf{if } (e) \textbf{ then } c_1 \textbf{ else } c_2 \\ & \mid \textbf{while } (e)\ c \\ & \mid c_1; c_2 \\ & \mid \textbf{nil} \\ & \mid \textbf{new } x : t \textbf{ in } c \end{aligned}$$

- Some languages distinguish declaration and commands (*e.g.* C), others do not (*e.g.* Java)
- Sequential declarations are syntactic sugar:

  `int x; int y; int z; ...` = **new** $x : int$ **in new** $y : int$ **in new** $z : int$ **in** $\ldots$
- Simultaneous (collateral) declarations are not allowed (what would that mean?)
- Declaration with initializations are syntactic sugar:

$$\textbf{new } x : t = i \textbf{ in } c = \textbf{new } x : t \textbf{ in } x := i; c$$

5.2 Typing

$$\frac{\Gamma[x \mapsto t] \vdash c : \textbf{unit}}{\Gamma \vdash \textbf{new } x : t \textbf{ in } c : \textbf{unit}}$$

- Declarations introduce new variables into the environment.

5.3 Evaluation

- Identifiers at compile-time mapped to locations.

$$\frac{x \in \mathbf{Idt}, x \in \mathrm{dom}(\Gamma), \sigma(\Gamma(x)) = v}{\langle x, \sigma\rangle \to_{Exp} \langle v, \sigma\rangle}$$

$$\frac{\Gamma \vdash \langle e, \sigma\rangle \to_{Exp} \langle n, \sigma'\rangle \qquad n \in \mathbb{Z}}{\Gamma \vdash \langle x := e, \sigma\rangle \to_{Stmt} \sigma'[\Gamma(x) \mapsto n]}$$

$$\frac{\Gamma \vdash \langle c_1, \sigma\rangle \to_{Stmt} \sigma' \qquad \Gamma \vdash \langle c_2, \sigma'\rangle \to_{Stmt} \sigma''}{\Gamma \vdash \langle c_1; c_2, \sigma\rangle \to_{Stmt} \sigma''}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma'\rangle \qquad \Gamma \vdash \langle c_1, \sigma'\rangle \to_{Stmt} \sigma''}{\Gamma \vdash \langle \mathbf{if}\ (b)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2, \sigma\rangle \to_{Stmt} \sigma''}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma'\rangle \qquad \Gamma \vdash \langle c_2, \sigma'\rangle \to_{Stmt} \sigma''}{\Gamma \vdash \langle \mathbf{if}\ (b)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2, \sigma''\rangle \to_{Stmt} s}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle \mathit{false}, \sigma'\rangle}{\Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma'\rangle \to_{Stmt} \sigma'}$$

$$\frac{\Gamma \vdash \langle b, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma'\rangle \qquad \Gamma \vdash \langle c, \sigma'\rangle \to_{Stmt} \sigma'' \qquad \Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma''\rangle \to_{Stmt} \sigma'''}{\Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma\rangle \to_{Stmt} \sigma'''}$$

$$\frac{\Gamma[x \mapsto l] \vdash \langle c, \sigma[l \mapsto \bot]\rangle \to_{Stmt} \sigma' \qquad l \notin \mathrm{dom}(\sigma)}{\Gamma \vdash \langle \mathbf{new}\ x : t\ \mathbf{in}\ c, \sigma\rangle \to_{Stmt} \sigma' \setminus l}$$

Figure 7: Rules to evaluate statements.

- Rules to evaluate statements in presence of new memory model are given in Figure 7. Notation:

$$\begin{aligned}\Gamma =& \mathbf{Idt} \rightharpoonup \mathbf{Loc}\\ \sigma =& \mathbf{Loc} \rightharpoonup V\\ \Gamma \vdash& \langle e, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle\\ \Gamma \vdash& \langle c, \sigma\rangle \to_{Stmt} \sigma'\end{aligned}$$

  Γ is called a static *environment*. The rules for $\Gamma \vdash \langle e, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle$ have the environment Γ added but apart from that are unchanged.

- Locations can be l-values or r-values, but if they appear as r-values, we refer to the value stored at that location rather than the location itself (this occurs in first rule as $\sigma(\Gamma(x)) = v$.)

- Implicit assumption is that $\Gamma \vdash c : \mathbf{unit}$; this guarantees all identifiers are declared.

- There is an invariant on the state:

$$\Gamma \vdash c : \mathbf{unit} \wedge \mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\Gamma) \wedge \Gamma \vdash \langle c, \sigma\rangle \to_{Stmt} \sigma' \Longrightarrow \mathrm{dom}(\sigma') \subseteq \mathrm{dom}(\Gamma)$$

  This guarantees the rules to handle identifiers as l-values and r-values always succeed for well-typed programs.

- Most interesting is the rule for declarations which shows the memory management:
  - $l \notin \mathrm{dom}(\sigma)$ is a new (fresh) location, previously unused.
  - $\Gamma[x \mapsto l]$ adds the map x to l to the environment. Note how Γ is not changed anywhere else.
  - $\sigma[l \mapsto \bot]$ adds the uninitialized location to the environment. Note that $l \in \mathrm{dom}(\sigma[l \mapsto \bot])$, but $(\sigma[l \mapsto \bot])(l) \notin \mathbb{Z}$.
  - Hence, one problem which may still occur is access of unitialized variables. (We can fix this later. Or not.)

- Note also how subsequent declarations will overshadow earlier ones, and how the scoping is modelled by the way the environment Γ i handed down.

5.4 Dynamic Memory Management

- Our simple language cannot handle dynamic memory management, nor does it have to.

- In order to do so, we would need locations as values which we can handle (even store in memory).

- To sketch this, assume we have an *operation* alloc() which allocates a new location, and a *statement* **free**(l) which deallocates the location. Obviously, alloc has a side effect, so it would need to b evaluated as an expression with a side effect:

$$\frac{l \notin \mathrm{dom}(\sigma)}{\Gamma \vdash \langle \mathrm{alloc}(), \sigma\rangle \to_{Exp} \langle l, \sigma[l \mapsto \bot]\rangle}$$

$$\frac{}{\Gamma \vdash \langle \mathbf{free}(l), \sigma\rangle \to_{Stmt} \sigma \setminus l}$$

- This neatly decomposes the **new** $x : t$ **in** s rule into two rules.

Vorlesung vom Aggregate Types: 06.11.2023

6 Aggregate Types

Aggregate types (also known as compound or composite types) build structured values from simple types (**int** and **bool**, which are as also called scalar types).

6.1 Abstract Language

We need more types, and corresponding operations:

$$
\begin{aligned}
t ::= {} & \textbf{int} \mid \textbf{bool} \mid \textbf{array}\ n\ \textbf{of}\ t \mid \textbf{struct}(i:t)^* \\
l ::= {} & i \mid l.i \mid l[e] \\
e ::= {} & \mathbb{Z} \mid \mathit{true} \mid \mathit{false} \mid l \\
& \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
& \mid e_1 == e_2 \mid e_1 < e_2 \\
& \mid\ !e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \\
& \mid l := e \\
& \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2
\end{aligned}
$$

- L-values can now be more structured than just identifiers; they are denoted l above.
- The name is actual an misnomer, since they are l-expressions rather than values but the name is traditional so we'll stick with it.
- L-values can appear on the left side of an assignment, or inside an expression.
- The reader is invited to write down typing rules for the new l-values.

We need a few auxiliary functions: size : $\mathscr{T}ypes \to \mathbb{N}$ returns the size of (an element) of type t in basic memory units (we don't call them bytes here but really they are bytes or at least you can think of them as bytes):

$$
\begin{aligned}
\text{size}(\textbf{int}) &= 1 \\
\text{size}(\textbf{bool}) &= 1 \\
\text{size}(\textbf{array}\ n\ \textbf{of}\ t) &= n \cdot \text{size}(t) \\
\text{size}(\textbf{struct}(f_i : t_i)_{i=1,\dots,n}) &= \sum_{i=1}^{n} \text{size}(t_i)
\end{aligned}
$$

For a structure type $t = \textbf{struct}\, f_1 : t_1 \dots f_n : t_n$ and a label g which is contained in one of the labels $f_1, \dots, f_n$ (*i.e.* $f_l = g$ for $1 \le l \le n$), we define the *offset* of the field g:

$$
\text{offset}_{\textbf{struct}(f_i:t_i)_{i=1,\dots,n}}(g) = \sum_{k=1}^{l-1} \text{size}(t_k) \quad f_l = g
$$

6.2 Operational Semantics

When defining the operational semantics, we have a lot of leeway. First, the memory model is pretty clear: we can store integers, booleans and adresses. So our memory model is

$$\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V} \quad \text{with} \quad \mathbf{Loc} = \mathbb{N}, \mathbf{V} = (\mathbb{Z} + \mathbb{B} + \mathbb{N})_\perp$$

L-values evaluate to addresses. They can evaluate with side effects, because array index expressions may have a side effect (often seen in C as `a[i++]= ...`).

$$\Gamma \ldots \langle l, \sigma \rangle \to_{Lexp} \langle n, \sigma' \rangle$$

The main question here is what to do with expressions of aggregate type (structures and arrays). Here, we evaluate expressions of aggregate types to references to the underlying structure or array. This is similar to what is done in Java (Java has no structures, just objects), except that in Java a variable of object type is a pointer to that object. Figure 8 shows the relevant rules. (This also means we cannot calculate with arrays and structure, *e.g.* add or multiply them, as is possible in Python; readers are invited to devise their own semantics which allow such constructions.)

When we declare a variable of aggregate types, we allocate as many memory cells as the size of the type (size(t)); the auxiliary function mem_alloc(σ, n, k) allocates k cells starting with cell n (by setting them to $\perp$).

Assignments $l := e$ are handled differently depending on the type of the expression (and location). If the expression is of aggregate type, its value n is the reference pointing to where the aggregate value is in memory. Assignment copies the whole aggregate value (all structure fields, all arrays), using the auxiliary function mem_cp(σ, n, k, m) (which copies cells from addresses $m, \ldots, m+k-1$ to $n, \ldots, n+k-1$).

6.3 Reference Types

Alternatively, let us make references explicit:

$$\begin{aligned}
t &::= \mathbf{int} \mid \mathbf{bool} \mid \mathbf{array}\ n\ \mathbf{of}\ t \mid \mathbf{struct}(i:t)^* \mid \mathbf{ref}\ t \\
l &::= i \mid l.i \mid l[e] \mid \&e \\
e &::= \mathbb{Z} \mid \mathit{true} \mid \mathit{false} \mid l \\
&\quad \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
&\quad \mid e_1 == e_2 \mid e_1 < e_2 \\
&\quad \mid\ !e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \\
&\quad \mid l := e \\
&\quad \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \\
&\quad \mid *l
\end{aligned}$$

We introduce a new type, **ref** t for reference to t, and two operations $*l$ which dereference an l-value and $\&e$ which returns the address of an expression (turning an expression into an l-value). The rules for these two operators are as follows:

$$\frac{}{\Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle \Gamma(l), \sigma\rangle}$$

$$\frac{\Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle n, \sigma'\rangle \qquad \Gamma \vdash l : \textbf{array } n \textbf{ of } t \qquad \Gamma \vdash \langle e, \sigma'\rangle \to_{Exp} \langle v, \sigma''\rangle, v \in \mathbb{Z}}{\Gamma \vdash \langle l[e], \sigma\rangle \to_{Lexp} \langle l + \text{size}(t) \cdot v, \sigma''\rangle}$$

$$\frac{\Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle n, \sigma'\rangle \qquad \Gamma \vdash l : \textbf{struct}(f_1 : t_1 \dots f_n : t_n) \qquad \exists k.\, f_k = i}{\Gamma \vdash \langle l.i, \sigma\rangle \to_{Lexp} \langle n + \text{offset}_{\textbf{struct}(f_1:t_1 \dots f_n:t_n)}(i), \sigma'\rangle}$$

$$\frac{\Gamma \vdash l : \textbf{int} \text{ or } \Gamma \vdash l : \textbf{bool} \qquad \Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle n, \sigma'\rangle}{\Gamma \vdash \langle l, \sigma\rangle \to_{Exp} \langle \sigma'(n), \sigma'\rangle}$$

$$\frac{\Gamma \vdash l : \textbf{array } n \textbf{ of } t \text{ or } \Gamma \vdash l : \textbf{struct} \qquad \Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle n, \sigma'\rangle}{\Gamma \vdash \langle l, \sigma\rangle \to_{Exp} \langle n, \sigma'\rangle}$$

$$\frac{\Gamma \vdash e : \textbf{int} \text{ or } \Gamma \vdash e : \textbf{bool} \qquad \Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle n, \sigma'\rangle \qquad \Gamma \vdash \langle e, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle}{\Gamma \vdash \langle l := e, \sigma\rangle \to_{Exp} \langle v, \sigma'[n \mapsto v]\rangle}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t \text{ with } t = \textbf{array } n \textbf{ of } t_1 \text{ or } t = \textbf{struct} \\ \Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle n_1, \sigma'\rangle \qquad \Gamma \vdash \langle e, \sigma\rangle \to_{Exp} \langle n_2, \sigma'\rangle\end{array}}{\Gamma \vdash \langle l := e, \sigma\rangle \to_{Exp} \langle n_2, \text{mem\_cp}(\sigma', n_1, \text{size}(t) n_2,)\rangle}$$

$$\frac{\Gamma[x \mapsto l] \vdash \langle c, \text{mem\_alloc}(\sigma', l, \text{size}(t))\rangle \to_{Stmt} \sigma' \qquad \{l, \dots, l + \text{size}(t) - 1\} \cap \text{dom}(\sigma) = \emptyset}{\Gamma \vdash \langle \textbf{new } x : t \textbf{ in } c, \sigma\rangle \to_{Stmt} \sigma' \setminus \{l, \dots, l + \text{size}(t) - 1\}}$$

$$\text{mem\_cp}(\sigma, n, k, m) = \begin{cases} \sigma & k \le 0 \\ \text{mem\_cp}(\sigma[n \mapsto \sigma(m)], n+1, k-1, m+1) & k > 0 \end{cases}$$

$$\text{mem\_alloc}(\sigma, n, k) = \begin{cases} \sigma & k \le 0 \\ \text{mem\_alloc}(\sigma[n \mapsto \bot], n+1, k-1) & k > 0 \end{cases}$$

Figure 8: Rules to handle variables of aggregate types, together with two auxiliary functions.

$$\frac{\Gamma \vdash \langle e, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle \qquad v \in \mathbb{N}}{\Gamma \vdash \langle \& e, \sigma\rangle \to_{Lexp} \langle v, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle n, \sigma'\rangle \qquad n \in \mathbb{N}}{\Gamma \vdash \langle *l, \sigma\rangle \to_{Exp} \langle n, \sigma'\rangle}$$

References work best when combined with the dynamic memory management from subsection 5.4. This is a more or less faithful representation of the C memory model — less faithful, because we do not deal with issues such as memory alignment, and we do not have the many different scalar types (integers signed and unsigned, short and long; floats and doubles; characters); but it captures the way C handles references, and its memory management, and allows us to make the same mistakes that C allows, for example refering to addresses in the memory which have become invalid:

```
int y;
ref int p;
new x: int in
   p= & x;
y= 5+ *p; // (1)
```

At point (1), the address of `x` has become invalid, and the expression `*p` is not well-defined. This will show up here as `*p` evaluating to $n \in \mathbf{Loc}$ such that $n \notin \mathrm{dom}(\sigma)$ — try it!

$$\frac{x \in \mathbf{Idt}, x \notin \mathrm{dom}(\sigma)}{\langle x, \sigma\rangle \to_{Exp} \mathsf{E}}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle e_1/e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle} \qquad \frac{\Gamma \vdash \langle e_2, \sigma\rangle \to_{Exp} \langle 0, \sigma'\rangle}{\Gamma \vdash \langle e_1/e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle} \qquad \frac{\Gamma \vdash \langle e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle e_1/e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle e_1 + e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle} \qquad \frac{\Gamma \vdash \langle e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle e_1 + e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}$$

$$\ldots$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle e_1 == e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle} \qquad \frac{\Gamma \vdash \langle e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle e_1 == e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle e_1 < e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle} \qquad \frac{\Gamma \vdash \langle e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle e_1 < e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle !\, e_1, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle e_1 \,\&\&\, e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle} \qquad \frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}{\Gamma \vdash \langle e_1 \,||\, e_2, \sigma\rangle \to_{Exp} \langle \mathsf{E}, \sigma'\rangle}$$

Figure 9: Additional rules to handle runtime failure (undefinedness)

Vorlesung vom Errors and Exceptions: 13.11.2023

7 Errors and Exceptions

7.1 Runtime Errors

- A *runtime error* such as division by zero or illegal memory access results in evaluation to E.
- E is then propagated upwards: once evaluation produces a runtime error, further evaluation keeps the runtime error[4]
- Question: what is the new semantics?

$$\Sigma \rightharpoonup (\mathbf{V} \times \Sigma) + \mathsf{E} \tag{15}$$
$$\Sigma \rightharpoonup (\mathbf{V} + \mathsf{E}) \times \Sigma \tag{16}$$

- What is the difference?

[4]Except for non-strict conjunction/disjunction and case distinction.

$$r \in \{(), \mathsf{E}\}$$

$$\frac{}{\Gamma \vdash \langle \sigma, \mathbf{nil} \rangle \to_{Stmt} \langle (), \sigma \rangle}$$

$$\frac{\Gamma \vdash \langle c_1, \sigma \rangle \to_{Stmt} \langle (), \sigma' \rangle \qquad \Gamma \vdash \langle c_2, \sigma' \rangle \to_{Stmt} \langle r, \sigma'' \rangle}{\Gamma \vdash \langle c_1; c_2, \sigma \rangle \to_{Stmt} \langle r, \sigma'' \rangle} \qquad \frac{\Gamma \vdash \langle c_1, \sigma \rangle \to_{Stmt} \langle \mathsf{E}, \sigma' \rangle}{\Gamma \vdash \langle c_1; c_2, \sigma \rangle \to_{Stmt} \langle \mathsf{E}, \sigma' \rangle}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle \mathit{true}, \sigma' \rangle \qquad \Gamma \vdash \langle c_1, \sigma \rangle \to_{Stmt} \langle r, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{if}\ (b)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2, \sigma \rangle \to_{Stmt} \langle r, \sigma' \rangle}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle \mathit{false}, \sigma' \rangle \qquad \Gamma \vdash \langle c_2, \sigma \rangle \to_{Stmt} \langle r, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{if}\ (b)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2, \sigma \rangle \to_{Stmt} \langle r, \sigma' \rangle}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle \mathsf{E}, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{if}\ (b)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2, \sigma \rangle \to_{Stmt} \langle \mathsf{E}, \sigma' \rangle}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle \mathit{false}, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma \rangle \to_{Stmt} \langle (), \sigma' \rangle} \qquad \frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle \mathsf{E}, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma \rangle \to_{Stmt} \mathsf{E}\sigma'}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle \mathit{true}, \sigma_1 \rangle \qquad \Gamma \vdash \langle c, \sigma_1 \rangle \to_{Stmt} \langle (), \sigma_2 \rangle \qquad \Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma_2 \rangle \to_{Stmt} \langle r, \sigma_3 \rangle}{\Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma \rangle \to_{Stmt} \langle r, \sigma_3 \rangle}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle \mathit{true}, \sigma' \rangle \qquad \Gamma \vdash \langle c, \sigma' \rangle \to_{Stmt} \langle \mathsf{E}, \sigma'' \rangle}{\Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma \rangle \to_{Stmt} \langle \mathsf{E}, \sigma'' \rangle}$$

Figure 10: Rules to evaluate statements with runtime failure

- We choose (18), because it keeps the system state when the error occurs. This is what happens in almost all programming languages.
- We *augment* the existing rules by more rules which handle the error case (see Figure 9). These specify the propagation of E.
- What happens with statements? The same considerations apply, but we need a (token) value $()$ to pass along to signal "normal operation".

$$\Sigma \rightharpoonup \Sigma + \mathsf{E} \tag{17}$$
$$\Sigma \rightharpoonup (() + \mathsf{E}) \times \Sigma \tag{18}$$

- For statements, we need to add rules which define how E propagate (see Figure 10).

7.2 Exceptions

- We choose a simple approach where there is an (arbitrary, finite) set of predefined exceptions $X \stackrel{def}{=} X_1, \ldots, X_n$. This is sufficient to show the basic principle of exceptions and exception handling.
- Additionally, we have a set of runtime errors $E = \{E_0, E_1\}$ where E_0 is for division by zero, and E_1 is for illegal memory access. (Both have been modelled by a single error value E above.)
- The reader is invited to extend the language so that the user can declare exceptions as an algebraic data type, with arguments and everything like in Java, Haskell and Python. What are the problems that arise?
- Exceptions are now essentially like the runtime errors from **??**, except that we raise them explicitly.

7.2.1 Extending the Language

We would like to catch exceptions both at the exception and the statement level, but mainly the latter, so we only have a catch statement. (Just like there is no case distinction expression, like `e ? f : g` in C or Java); this would be a simple addition.

$$\begin{aligned}
l &::= i \mid l.i \mid l[e] \\
e &::= \mathbb{Z} \mid \mathit{true} \mid \mathit{false} \mid l \\
&\quad \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
&\quad \mid e_1 == e_2 \mid e_1 < e_2 \\
&\quad \mid\ !e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \\
&\quad \mid l := e \\
&\quad \mid \textbf{throw}(x)
\end{aligned}$$

$$\begin{aligned}
c &::= e \mid \textbf{if}\ (e)\ \textbf{then}\, c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ (e)\ c \mid c_1; c_2 \mid \textbf{nil} \\
&\quad \mid \textbf{try}\ c_1\ \textbf{catch}\ x \to c_2
\end{aligned}$$

7.2.2 Typing Rules

$$\frac{x \in X}{\Gamma \vdash \textbf{throw}(x) : \alpha} \qquad \frac{\Gamma \vdash c_1 : \textbf{unit} \qquad \Gamma \vdash c_2 : \textbf{unit} \qquad x \in X + E}{\Gamma \vdash \textbf{try}\ c_1\ \textbf{catch}\ x \to c_2 : \textbf{unit}}$$

7.2.3 Operational Semantics

- Same considerations as above apply.
- The semantic domains are:
  - For expressions:
    $$\Sigma \rightharpoonup (\mathbf{V} + F) \times \Sigma$$
  - For statements:
    $$\Sigma \rightharpoonup (() + F) \times \Sigma$$
  - where $F = X + E$ in both cases.
- Those are a lot of rules! And we only give new ones...
- Still, I'd claim most of these are fairly stereotypical and the general mechanism is easy to understand.

7.3 More Non-Linear Control Flow

Exceptions are an example of *non-linear* control flow, in other words we have "jumps" where the control passes from one part of the porgram to another one, which is not adjacent (whatever that means precisely). The semantic mechanism of exceptions can be used to implement a break statement, which jumps out of a (nested) loop, or even return statements (as we shall see in the next section). This is in fact how it is really implemented in the JVM.

Here is a short sketch how the break statement can be implemented as syntactic sugar for throw and catch:

$$\textbf{label}\ x\ \textbf{in}\ c \equiv \textbf{try}\ c\ \textbf{catch}\ x \to \textbf{nil}$$
$$\textbf{break}\ x \equiv \textbf{throw}(x)$$

Here is an example how that works:

```
r := 1; x:= 0;
label x14 in
while (x< 99) {
  i := 1;
  while (i< 3) {
    r := r* i;
    if (r > 10) break x14;
    i := i+1
  }
  x := x+ 1
}
```

$$\frac{\Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle l.i, \sigma\rangle \to_{Lexp} \langle f, \sigma'\rangle} \qquad \frac{\Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle l[e], \sigma\rangle \to_{Lexp} \langle f, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle l[e], \sigma\rangle \to_{Lexp} \langle f, \sigma'\rangle}$$

$$\frac{\Gamma \vdash l : \textbf{int} \text{ or } \Gamma \vdash l : \textbf{bool} \qquad \Gamma \vdash \langle l, \sigma\rangle \to_{Lexp} \langle n, \sigma'\rangle \qquad n \notin \mathrm{dom}(\sigma')}{\Gamma \vdash \langle l, \sigma\rangle \to_{Exp} \langle E_1, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle e_1 / e_2, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle, v \in \mathbb{Z} \qquad \Gamma \vdash \langle e_2, \sigma'\rangle \to_{Exp} \langle f, \sigma''\rangle \qquad f \in X + E}{\Gamma \vdash \langle e_1 / e_2, \sigma\rangle \to_{Exp} \langle f, \sigma''\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle, v \in \mathbb{Z} \qquad \Gamma \vdash \langle e_2, \sigma'\rangle \to_{Exp} \langle 0, \sigma''\rangle}{\Gamma \vdash \langle e_1 / e_2, \sigma\rangle \to_{Exp} \langle E_0, \sigma''\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle e_1 + e_2, \sigma'\rangle \to_{Exp} \langle f, \sigma''\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle, v \in \mathbb{Z} \qquad \Gamma \vdash \langle e_2, \sigma'\rangle \to_{Exp} \langle f, \sigma''\rangle \qquad f \in X + E}{\Gamma \vdash \langle e_1 + e_2, \sigma\rangle \to_{Exp} \langle f, \sigma''\rangle}$$

$$\dots$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle e_1 == e_2, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle v \in \mathbb{Z} \qquad \Gamma \vdash \langle e_2, \sigma'\rangle \to_{Exp} \langle f, \sigma''\rangle \qquad f \in X + E}{\Gamma \vdash \langle e_1 == e_2, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle}$$

$$\dots$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle !\, e_1, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle \qquad f \in X + E}{\Gamma \vdash \langle e_1 \,\&\&\, e_2, \sigma\rangle \to_{Exp} \langle f, \sigma'\rangle}$$

$$\frac{\Gamma \vdash \langle e_1, \sigma\rangle \to_{Exp} \langle \mathit{true}, \sigma'\rangle \qquad \Gamma \vdash \langle e_2, \sigma'\rangle \to_{Exp} \langle f, \sigma''\rangle \qquad f \in X + E}{\Gamma \vdash \langle e_1 \,\&\&\, e_2, \sigma\rangle \to_{Exp} \langle f, \sigma''\rangle}$$

$$\dots$$

$$\frac{x \in X}{\Gamma \vdash \langle \textbf{throw}(x), \sigma\rangle \to_{Exp} \langle x, \sigma\rangle}$$

Figure 11: Rules to handle exceptions in expressions. Only new rules are given, and rules for $*$, $-$ and $<$, $||$ have been elided.

$$\frac{}{\Gamma \vdash \langle \sigma, \mathbf{nil} \rangle \to_{Stmt} \langle (), \sigma \rangle}$$

$$\frac{\Gamma \vdash \langle c_1, \sigma \rangle \to_{Stmt} \langle (), \sigma' \rangle \qquad \Gamma \vdash \langle c_2, \sigma' \rangle \to_{Stmt} \langle r, \sigma'' \rangle \qquad r \in \{()\} + X + E}{\Gamma \vdash \langle c_1; c_2, \sigma \rangle \to_{Stmt} \langle r, \sigma'' \rangle}$$

$$\frac{\Gamma \vdash \langle c_1, \sigma \rangle \to_{Stmt} \langle f, \sigma' \rangle \qquad f \in X + E}{\Gamma \vdash \langle c_1; c_2, \sigma \rangle \to_{Stmt} \langle f, \sigma' \rangle}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle true, \sigma' \rangle \qquad \Gamma \vdash \langle c_1, \sigma \rangle \to_{Stmt} \langle r, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{if}\ (b)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2, \sigma \rangle \to_{Stmt} \langle r, \sigma' \rangle}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle false, \sigma' \rangle \qquad \Gamma \vdash \langle c_2, \sigma \rangle \to_{Stmt} \langle r, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{if}\ (b)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2, \sigma \rangle \to_{Stmt} \langle r, \sigma' \rangle}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle f, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{if}\ (b)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2, \sigma \rangle \to_{Stmt} \langle f, \sigma' \rangle}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle false, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma \rangle \to_{Stmt} \langle (), \sigma' \rangle} \qquad \frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle f, \sigma' \rangle}{\Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma \rangle \to_{Stmt} f\sigma'}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle true, \sigma_1 \rangle \qquad \Gamma \vdash \langle c, \sigma_1 \rangle \to_{Stmt} \langle (), \sigma_2 \rangle \qquad \Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma_2 \rangle \to_{Stmt} \langle r, \sigma_3 \rangle}{\Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma \rangle \to_{Stmt} \langle r, \sigma_3 \rangle}$$

$$\frac{\Gamma \vdash \langle b, \sigma \rangle \to_{Exp} \langle true, \sigma' \rangle \qquad \Gamma \vdash \langle c, \sigma' \rangle \to_{Stmt} \langle f, \sigma'' \rangle}{\Gamma \vdash \langle \mathbf{while}\ (b)\ c, \sigma \rangle \to_{Stmt} \langle f, \sigma'' \rangle}$$

$$\frac{\Gamma \vdash \langle c_1, \sigma \rangle \to_{Stmt} \langle (), \sigma' \rangle}{\Gamma \vdash \langle \mathbf{try}\ c_1\ \mathbf{catch}\ x \to c_2, \sigma \rangle \to_{Stmt} \langle (), \sigma' \rangle} \qquad \frac{\Gamma \vdash \langle c_1, \sigma \rangle \to_{Stmt} \langle f, \sigma' \rangle \qquad e \neq f}{\Gamma \vdash \langle \mathbf{try}\ c_1\ \mathbf{catch}\ e \to c_2, \sigma \rangle \to_{Stmt} \langle f, \sigma' \rangle}$$

$$\frac{\Gamma \vdash \langle c_1, \sigma \rangle \to_{Stmt} \langle e, \sigma' \rangle \qquad \Gamma \vdash \langle c_2, \sigma' \rangle \to_{Stmt} \langle r, \sigma'' \rangle}{\Gamma \vdash \langle \mathbf{try}\ c_1\ \mathbf{catch}\ e \to c_2, \sigma \rangle \to_{Stmt} \langle r, \sigma'' \rangle}$$

Figure 12: Rules to handle exceptions in statements.

Vorlesung vom 20.11.2023: Procedures and Functions

8 Procedures and Functions

8.1 Syntax: Definition

- General format:
$$\textbf{fun}\ f(x_1,\ldots,x_n) = c$$
where c is a command.
- A *program* is just a collection of function definitions:
$$\begin{aligned}\phi \equiv\ & \textbf{fun}\ f_1(x_{1,1},\ldots,x_{1,n_1}) = b_1\\ & \textbf{fun}\ f_2(x_{2,1},\ldots,x_{2,n_2}) = b_2\\ & \ldots\\ & \textbf{fun}\ f_m(x_{m,1},\ldots,x_{m,n_m}) = b_m\end{aligned} \tag{19}$$
- Need to turn ϕ into an environment Γ, which maps function names (f_i above) to the paramters $x_{i,1},\ldots,x_{i,n_i}$.
- Better to introduce a new syntactic primitive: *parameterized blocks*
$$pb ::= \lambda i.pb \mid c$$
A parameterized block has either a formal parameter i, or it is just a command. Multiple parameters can be written as $\lambda x_1.\lambda x_2.\ldots\lambda x_n.\ c$ (we write this as $\lambda x_1,x_2,\ldots,x_n.c$).
- We define the function # (which returns the number of parameters) as
$$\begin{aligned}\#(\lambda x.b) &= 1+\#(b)\\ \#(c) &= 0\end{aligned}$$
- Parameterization is like the let-construct from earlier on, and in fact is the more basic construction. We shall see anon now to express let via parameterization, but we need substitution first. It is defined precisely like before, except it is now a statement rather than expression.
- The full definition is given in Figure 14. Compare it to the one in Figure 6 — where are the differences?

8.2 Semantics

- The definition of a function in itself does not have any operational semantics.
- Firstly, a program definition like (20) above is written more precisely like
$$\begin{aligned}\phi \equiv\ & \textbf{fun}\ f_1 = \lambda x_{1,1},\ldots,x_{1,n_1}.b_1\\ & \textbf{fun}\ f_2 = \lambda x_{2,1},\ldots,x_{2,n_2}.b_2\\ & \ldots\\ & \textbf{fun}\ f_m = \lambda x_{m,1},\ldots,x_{m,n_m}.b_m\end{aligned} \tag{20}$$

$$
\begin{aligned}
\mathrm{FV}(n) &= \emptyset \\
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(\mathit{true}) &= \emptyset \\
\mathrm{FV}(\mathit{false}) &= \emptyset \\
\mathrm{FV}(e_1 + e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 - e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 * e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 / e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 == e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 < e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(!\,e) &= \mathrm{FV}(e) \\
\mathrm{FV}(e_1 \,\&\&\, e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(e_1 \,||\, e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(i := e) &= \mathrm{FV}(e) \\
\mathrm{FV}(\mathbf{nil}) &= \emptyset \\
\mathrm{FV}(c_1; c_2) &= \mathrm{FV}(c_1) \cup \mathrm{FV}(c_2) \\
\mathrm{FV}(\mathbf{if}\ (b)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2) &= \mathrm{FV}(b) \cup \mathrm{FV}(c_1) \cup \mathrm{FV}(c_2) \\
\mathrm{FV}(\mathbf{while}\ (b)\ c) &= \mathrm{FV}(b) \cup \mathrm{FV}(c) \\
\mathrm{FV}(\lambda x.\, c) &= \mathrm{FV}(c \setminus \{x\})
\end{aligned}
$$

Figure 13: The function FV returns the free variables in am expression, statement or parameterized block.

$$
\begin{aligned}
n[e/x] &= n \\
y[e/x] &= \begin{cases} e & x = y \\ y & \text{otherwise} \end{cases} \\
\mathit{true}[e/x] &= \mathit{true} \\
\mathit{false}[e/x] &= \mathit{false} \\
(e_1 + e_2)[e/x] &= (e_1[e/x]) + (e_2[e/x]) \\
(e_1 - e_2)[e/x] &= (e_1[e/x]) - (e_2[e/x]) \\
(e_1 * e_2)[e/x] &= (e_1[e/x]) * (e_2[e/x]) \\
(e_1 / e_2)[e/x] &= (e_1[e/x]) / (e_2[e/x]) \\
(e_1 == e_2)[e/x] &= (e_1[e/x]) == (e_2[e/x]) \\
(e_1 < e_2)[e/x] &= (e_1[e/x]) < (e_2[e/x]) \\
(!\, e_1)[e_2/x] &= !(e_1[e_2/x]) \\
(e_1 \,\&\&\, e_2)[e_3/x] &= (e_1[e_3/x]) \,\&\&\, (e_2[e_3/x]) \\
(e_1 \,||\, e_2)[e_3/x] &= (e_1[e_3/x]) \,||\, (e_2[e_3/x]) \\
(x := e_1)[e_2/y] &= (x := (e_1[e_2/y])) \\
\mathbf{nil}[e/x] &= \mathbf{nil} \\
(c_1; c_2)[e/x] &= (c_1[e/x]); (c_2[e/x]) \\
(\mathbf{if}\ (b)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2)[e/x] &= \mathbf{if}\ (b[e/x])\ \mathbf{then}\, c_1[e/x]\ \mathbf{else}\ c_2[e/x] \\
(\mathbf{while}\ (b)\ c)[e/x] &= \mathbf{while}\ (b[e/x])\ (c[e/x]) \\
(\lambda y.\, c)[e/x] &= \begin{cases} \lambda y.\, c & x = y \\ \lambda y.\, (c[e/x]) & x \neq y, y \notin \mathrm{FV}(e) \\ \lambda z.\, ((c[z/y])[e/x]) & x \neq y, y \in \mathrm{FV}(e), z \notin \mathrm{FV}(e) \cup \mathrm{FV}(c) \end{cases}
\end{aligned}
$$

Figure 14: Definition of the substitution function.

- Thus, a program is a list of pairs (f_i, pb_i) where f_i is an identifier and pb_i is a parameterized block.
- The *function environment* Γ_{fun} is a map $\mathbf{Idt} \rightharpoonup pb$ which maps function names to parameterized blocks. For a program $\phi = (f_i, pb_i)_{i=1,\dots,n}$ as above, we define $\Gamma_{\text{fun}}(\phi) = \{(f_i, pb_i) \mid (f_i, pb_i) \in \phi\}$. This is all a very roundabout way of saying we keep track which block the name f_i refers to.
- Note we do not distinguish between function names and variable names (just like Haskell), as this just complicates things in this simple exposition.
- I've forgotten to annotate the parameters $x_{i,j}$ above, and the function body with a type! This is not needed for the operational semantics, but for the type check (obviously).

8.3 Returning Values

- Before we can define the semantics of a function call, we need to be able to return a value. How would that work?
- A return statement breaks sequential program flow, just like an exception. And it returns to the point where the function was called, just like a catch statement! This suggests to model return by exception.
- We could not do that before, as exceptions could not contain a value, they were just flat values; and that was because we could not access the value when catching it.
- We extend the set E of exceptions as follows:

  $$E = \{E_R\} \times \mathbf{V} \cup \{E_0, E_1\}$$

  What does that mean? Exceptions are either constants E_0 or E_1 (for division by zero and illegal memory access), or pairs (E_R, v) where E_R is a constant and v is a value; these pairs signal a function returning value v

8.4 Extending the Languge

We extend our language with a function call statement (can also be a procedure call if it occurs as an expression statement), and a return statement.

$$
\begin{aligned}
l &::= i \mid l.i \mid l[e] \\
e &::= \mathbb{Z} \mid \mathit{true} \mid \mathit{false} \mid l \\
&\quad \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\
&\quad \mid e_1 == e_2 \mid e_1 < e_2 \\
&\quad \mid \,!\, e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \\
&\quad \mid l := e \mid \mathbf{throw}(x) \\
&\quad \mid f(e_1, \dots, e_n) \\
c &::= e \mid \mathbf{if}\ (e)\ \mathbf{then}\, c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ (e)\ c \mid c_1; c_2 \mid \mathbf{nil} \mid \mathbf{try}\ c_1\ \mathbf{catch}\ x \to c_2 \\
&\quad \mid \mathbf{return}\ e \\
pb &::= \lambda i.\, pb \mid c
\end{aligned}
$$

- Parameterized blocks *pb* do not have a semantics, as they are never evaluated. (In functional programming, unevaluated parameterized blocks such as these are refered to a as "chunks".) Their parameters $x_1, \ldots, x_n$ are first substituted with the correct number of arguments, and then the resulting command b is evaluated; this is what the function call rules will be doing.

We first look at function calls with one parameter. We get call-by-value if the argument is first evaluated to a value, which is then substituted for the parameter:

$$\frac{\Gamma(f) = \lambda x.b \qquad \Gamma \vdash \langle e, \sigma\rangle \to_{Exp} \langle v_1, \sigma'\rangle \qquad \langle b[v_1/x], \sigma'\rangle \to_{Stmt} \langle (E_R, v_2), \sigma'\rangle \qquad v_1, v_2 \in \mathbf{V}}{\Gamma \vdash \langle f(e), \sigma\rangle \to_{Exp} \langle v_2, \sigma''\rangle} \tag{21}$$

Remember from Figure 8 that if e is of aggregate type, it evaluates to the address $n \in \mathbf{Loc}$ in memory where the aggregate type (array, structure) is located, so this is very much like Java does it, and the way C handles arrays as parameters. We get call-by-need if the arguments are substituted as they are:

$$\frac{\Gamma(f) = \lambda x.b \qquad \langle b[e/x], \sigma\rangle \to_{Stmt} \langle (E_R, v), \sigma'\rangle \qquad v \in \mathbf{V}}{\Gamma \vdash \langle f(e), \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle} \tag{22}$$

If we want to have multiple parameters, we have to evaluate the arguments successively, passing along the state as we do so:

$$\frac{\begin{array}{c}\Gamma(f) = \lambda x_1 \ldots . \lambda x_n . b \qquad \Gamma \vdash \langle e_i, \sigma_{i-1}\rangle \to_{Exp} \langle v_i, \sigma_i\rangle \\ \Gamma \vdash \langle (b[v_1/x_1]) \ldots [v_n/x_n], \sigma_n\rangle \to_{Stmt} \langle (E_R, w), \sigma'\rangle \qquad i = 1, \ldots, n, v_i \in \mathbf{V}\end{array}}{\Gamma \vdash \langle f(e_1, \ldots, e_n), \sigma_0\rangle \to_{Exp} \langle w, \sigma'\rangle} \tag{23}$$

$$\frac{\Gamma(f) = \lambda x_1 \ldots . \lambda x_n . b \qquad \Gamma \vdash \langle (b[e_1/x_1]) \ldots [e_n/x_n], \sigma\rangle \to_{Stmt} \langle (E_R, v), \sigma'\rangle}{\Gamma \vdash \langle f(e_1, \ldots, e_n), \sigma\rangle \to_{Exp} \langle v, \sigma'\rangle} \tag{24}$$

What is missing here? Well, what happens when the function body does *not* return a value? That might happen if the control flow reaches the end of function body without a return statement; we can check that statically and make sure it does not happen, so we can omit that. But what happens if another exception is raised? Easy, we just pass it through (here only for cbv):

$$\frac{\begin{array}{c}\Gamma(f) = \lambda x_1 \ldots . \lambda x_n . b \qquad \Gamma \vdash \langle e_i, \sigma_{i-1}\rangle \to_{Exp} \langle v_i, \sigma_i\rangle \\ \Gamma \vdash \langle (b[v_1/x_1]) \ldots [v_n/x_n], \sigma_n\rangle \to_{Stmt} \langle f, \sigma'\rangle \qquad i = 1, \ldots, n, v_i \in \mathbf{V}, f \in \{E_0, E_1\} + X\end{array}}{\Gamma \vdash \langle f(e_1, \ldots, e_n), \sigma_0\rangle \to_{Exp} \langle f, \sigma'\rangle}$$

Finally, for cbv we have to handle the possiblity that an exception occurs when one of the arguments are evaluated. This reads like this (and is only needed for call-by-value):

$$\frac{\begin{array}{c}\Gamma(f) = \lambda x_1 \ldots . \lambda x_n . b \qquad \Gamma \vdash \langle e_i, \sigma_{i-1}\rangle \to_{Exp} \langle v_i, \sigma_i\rangle \\ \Gamma \vdash \langle e_k, \sigma_{k-1}\rangle \to_{Exp} \langle f, \sigma_k\rangle \qquad i = 1, ldots, k, k < leqn, v_i \in \mathbf{V}, f \in E + X\end{array}}{\Gamma \vdash \langle f(e_1, \ldots, e_n), \sigma_0\rangle \to_{Exp} \langle f, \sigma_k\rangle}$$

Here, if the k-th argument (for some $k \leq n$) evaluates to an exception f (and note this exception can be an E_R, meaning we return a value when passing an argument to another function — a strange thing to do, but entirely possible), then the whole function call evaluates to that exception, and the function body is

left untouched (unevaluated). Obviously, call-by-need does not need a rule like that (but you may want to ponder where exceptions raised when evaluating arguments can be handled — hint, not necessarily in the body).

The eagle-eyed reader may have spotted a minor inconsistency here: since $() \notin \mathbf{V}$, we cannot return unit values the way we have written things— we'd need $E = \{E_R \times (\mathbf{Loc} \cup \{()\})\}$, but that is easy to fix.

8.5 Implementation Considerations

The rules above use substitution on the syntactic level. Surely, a Real Programming Language^{TM} (apart from Haskell) would not (and in fact, could not, once the function has been compiled to machine code) do that? Well, what happens in these langauges is expressed in our language as follows. A function

$$f(x_1 : t_1, \ldots, x_n : t_n)\{b\}$$

in C or Java is modelled in our language as

$$\textbf{fun}\ f = \lambda y_1, \ldots, y_n.\ \textbf{new}\ x_1 : t_n = y_1\ \textbf{in}\ \ldots \textbf{new}\ x_n : t_n = y_n\ \textbf{in}\ b$$

where $y_1, \ldots, y_n \notin \mathrm{FV}(b)$. This is precisely how C describes parameter passing: function parameters are local variables, initialised with the value that is passed to the function when it is called. Note that the parameters $y_1, \ldots, y_n$ do not appear in b, so $b[v_i/y_i] = b$ (for any v_i; and if v_i is just a value, it will have no free variables, so we do not need to worry about the ominuous third clause in Figure 14.

The reader is invited to write down the (simplified) version of rules (21) and (22) for these functions like these.

Vorlesung vom 27.11.2023: Fortgeschrittene Typsysteme

9 Type Inference

We study type inference with a very simple language first. This follows very closely the classical exposition in [3]. For a more accessible introduction, see [1].

9.1 A Very Simple Language

9.1.1 Expressions

$$e ::= \mathbf{Idt} \mid e_1(e_2) \mid \lambda x.e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$$

9.1.2 Types and Type Schemes

$$\tau ::= \alpha \mid \iota \mid \tau_1 \to \tau_2$$
$$\sigma ::= \tau \mid \forall \alpha.\sigma$$

- α are *type variables*, ι are predefined Types (*e.g.* **int**, **bool**, but also $[\alpha]$)
- Type schemes always look like $\sigma = \forall \alpha_1, \ldots \alpha_n.\tau$, *i.e.* a bunch of quantifiers around a type τ.
- $\mathrm{FV}(\sigma)$ are the *free variables* in a type scheme σ (*i.e.* the variables in τ minus the $\alpha_1, \ldots, \alpha_n$)

9.1.3 Substitutions

- A substitution is given by $S = [\tau_1/\alpha_1] \ldots [\tau_n/\alpha_n]$, substituting variables for types.
- The empty substitution $\emptyset$ acts as the identity. Substitutions can be composed, $S_2 \cdot S_1$ applies S_2 to all substituting terms in S_1 and also adds those variables which where not in the domain of S_1. Composing with the empty substitution gives the identity, *i.e.* $\emptyset \cdot S = S \cdot \emptyset = S$, and $\tau(S_2 \cdot S_1) = (\tau S_2)S_1$.
- It acts on type *scheme* σ, written σS, by replacing α_i with τ_i, *renaming bound variables as necessary* (we know how this works from above).
- Example:

$$\sigma = \forall \alpha.[alpha] \to (\beta \to \alpha)$$
$$S = [\alpha \to \mathbf{int}/\beta]$$
$$\sigma S = \forall \gamma.\ [gamma] \to ((\alpha \to \mathbf{int}) \to \gamma)$$

- σS is called instance of σ.

$$\frac{(x,\sigma)\in\Gamma}{\Gamma\vdash x:\sigma}\ [\textsc{Taut}] \qquad \frac{\Gamma\vdash e:\sigma \qquad \sigma<\sigma'}{\Gamma\vdash e:\sigma}\ [\textsc{Inst}] \qquad \frac{\Gamma\vdash e:\sigma \qquad \alpha\notin \mathrm{FV}(\Gamma)}{\Gamma\vdash e:\forall\alpha.\sigma}\ [\textsc{Gen}]$$

$$\frac{\Gamma\vdash e_1:\tau_1\to\tau_2 \qquad \Gamma\vdash e_2:\tau_1}{\Gamma\vdash e_1(e_2):\tau_2}\ [\textsc{Comb}] \qquad \frac{\Gamma+(x:\tau_1)\vdash e:\tau_2}{\Gamma\vdash \lambda x.e:\tau_1\to\tau_2}\ [\textsc{Abs}]$$

$$\frac{\Gamma\vdash e_1:\sigma \qquad \Gamma+(x:\sigma)\vdash e_2:\tau}{\Gamma\vdash \textbf{let } x=e_1 \textbf{ in } e_2:\tau}\ [\textsc{Let}]$$

Figure 15: Rules for Type Inference

- Given a type scheme $\sigma=\forall\alpha_1,\dots\alpha_n.\tau$, a *generic instance* is $\sigma'=\forall\beta_1,\dots,\beta_m.\tau'$ if $\tau'=\tau[\tau_i/\alpha_i]$ for some τ_i and $\beta_i\notin \mathrm{FV}\,\sigma$; we write this as $\sigma>\sigma'$. Note that generic instances act on *bound* variables, whereas instances act on free variables.

9.1.4 Type Inference Rules

We infer *type judgements* of the form

$$\Gamma\vdash e:\sigma \tag{25}$$

where Γ is a context, e is an expression, and σ a type scheme.

The rules in Figure 15 allow us to infer judgements like (25). However, unlike the rules for the operational semantics, they are not deterministic in the sense that it is always clear which rule to apply.

9.1.5 Type Unification

Given two types τ_1,τ_2, the type unification algorithm U calculates a substitution V such that

(i) $\tau_1V=\tau_2V$ (V unifies τ_1,τ_2)

(ii) If there is a substitution S such that $\tau_1S=\tau_2$, then there is another substitution R such that $S=R\cdot V$, *i.e.* S is an instance of V.

(iii) V only involves variables in τ_1,τ_2, $\mathrm{dom}(V)\subseteq \mathrm{FV}(\tau_1)\cup\mathrm{FV}(\tau_2)$

The algorithm proceeds by case distinction on τ_1 and τ_2:

- Case $\tau_1\equiv\alpha$: if $\alpha\notin\mathrm{FV}(\tau_2)$, return $S\stackrel{def}{=}[\tau_2/\alpha]$, otherwise fail.
- Case $\tau_2\equiv\alpha$: if $\alpha\notin\mathrm{FV}(\tau_1)$, return $S\stackrel{def}{=}[\tau_1/\alpha]$, otherwise fail.
- Case $\tau_1\equiv\iota$ and $\tau_2=\iota$: return $S\stackrel{def}{=}\emptyset$
- Case $\tau_1\equiv\tau_{11}\to\tau_{12}$ and $\tau_2\equiv\tau_{21}\to\tau_{22}$: Calculate $S_1=U(\tau_{11},\tau_{21})$ and $S_2=U(\tau_{12}S_1,\tau_{22}S_1)$ (either may fail; in that case, propagate failure). Then return $S\stackrel{def}{=}S_1\cdot S_2$.
- In all other cases fail.

9.1.6 Type Inference

This is the algorithm that drives type inference in Haskell and other languages with parametric polymorphism. First, the *closure* of a type τ with respect to a context Γ is given by

$$\overline{\Gamma}(\tau) \stackrel{def}{=} \forall \alpha_1, \ldots, \alpha_n . \tau$$

where $\alpha_1, \ldots, \alpha_n \in \mathrm{FV}(\tau) \setminus \mathrm{FV}(\Gamma)$ (*i.e.* α_i is free in τ but not in Γ); the closure is the result of applying rule GEN as often as allowed to τ in the context Γ.

For context Γ and expression e, algorithm W computes

$$W(\Gamma, e) = (S, \tau)$$

where S is a substutition, and τ a type such that $\Gamma S \vdash e : \tau$. It proceeds by case distinction on e:

- $e \equiv x$, $(x : \forall \alpha_1, \ldots, \alpha_n . \tau') \in \Gamma$.
  Then return $S \stackrel{def}{=} \emptyset$, $\tau \stackrel{def}{=} \tau'[\beta_1/\alpha_1] \ldots [\beta_n/\alpha_n]$ where β_i are fresh, *i.e.* $\beta_i \in \mathrm{FV}(\Gamma)$.
- $e \equiv e_1(e_2)$.
  Then let

  $$\begin{aligned} &W(\Gamma, e_1) = (S_1, \tau_1) \\ &W(\Gamma, e_2) = (S_2, \tau_2) \\ &U(\tau_1 S_2, \tau_2 \to \beta) = V \quad \text{with } \beta \notin \mathrm{FV}(\Gamma) \end{aligned}$$

  Note all of these may fail; in that case, propagate failure.
  If they do not fail, return $S \stackrel{def}{=} V \cdot S_2 \cdot S_1, \tau \stackrel{def}{=} \beta V$.
- $e \equiv \lambda x . e$
  Let $W(\Gamma + (x : \beta), e_1) = (S_1, \tau_1)$ with $\beta \notin \mathrm{FV}(\Gamma)$ (may fail; in that case, propagate failure).
  Then return $S \stackrel{def}{=} S_1, \tau \stackrel{def}{=} \beta S \to \tau_1$.
- $E \equiv$ **let** $x = e_1$ **in** e_2.
  Let $W(\Gamma, e_1) = (S_1, \tau_1)$ and $W(\Gamma S_1 + (x : \overline{\Gamma S_1}(\tau_1)), e_2) = (S_2, \tau_2)$ (may fail; you know what to do by now). Then return $S = S_2 \cdot S_1, \tau \stackrel{def}{=} \tau_2$.

The algorithm W is *sound*, *i.e.* if $W(\Gamma, e) = (S, \tau)$ then $\Gamma S \vdash e : \tau$, and it computes the *principal type*, *i.e.* $W(\Gamma, e) = (S, \tau)$ and $\Gamma \vdash e : \sigma'$, then $\sigma' = \tau S$.

Vorlesung vom 27.11.2023: Datenabstraktion

10 Data Abstraction

The following development follows the slogan "Abstract types have existential type" [4, 2]. The main observation is that just as parametric polymorphism is modelled by universal quantification over types, as in the previous lecture, abstract datatypes are modelled by existential quantification.

No knowledge of formal logic is required to follow the lecture, but an intuitive understanding of what $\exists x.\, \phi(x)$ means will help.

10.1 Preliminaries

We generalize our types further to model abstract datatypes (note that we lose the type inference as we do so). Wor types are now[5]

$$\begin{aligned}\tau ::= \;& \alpha \mid \iota \mid \tau_1 \to \tau_2 \mid \textbf{array}\ \tau \mid [\tau] \mid \\ & \tau_1 \wedge \tau_2 \mid \forall\alpha.\,\tau \mid \exists\alpha.\,\tau\end{aligned}$$

More types mean more expressions:

$$\begin{aligned}e ::= \;& \textbf{Idt} \mid e_1(e_2) \mid \lambda x.\, e \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \\ & \langle e_1, e_2\rangle \mid \textbf{abstype}\ \alpha\ \textbf{is}\ e \mid \textbf{open}\ e_1\ \textbf{in}\ e_2\end{aligned}$$

Note we now have explicit tupels in our language. We will need these to model signatures, *i.e.* types of modules. Essentially, our modules will be tuples of operations $\langle e_1, \ldots, e_n\rangle$ and the type of the module will be a type $\tau_1 \wedge \ldots \wedge \tau_n$, with $e_i : \tau_i$. (We write n-tuples like that with the tacit understanding they are just syntactic sugar for repeated binary tupels $\langle e_1, e_2\rangle$; I just do not want to have to type ellipses all the time.)

10.2 Rules

Let's get the boring stuff out of the way first. The rules for the product type are obvious:

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash f : \tau}{\Gamma \vdash \langle e, f\rangle : \sigma \wedge \tau} \qquad \frac{\Gamma \vdash \langle e, f\rangle : \sigma \wedge \tau}{\Gamma \vdash e : \sigma} \qquad \frac{\Gamma \vdash \langle e, f\rangle : \sigma \wedge \tau}{\Gamma \vdash f : \tau}$$

We use pattern matching to desconstruct tupels. Instead of introducing operations fst and snd to write, *e.g.*, $\lambda x.\ \mathrm{fst}(x) + \mathrm{snd}(y)$ to calculate the sum of a tuple, we write$\lambda\langle x, \rangle.\, x + y$. To deconstruct tupels in the context (assumptions), we use the (derived) rule

$$\frac{\Gamma + x : \sigma + y : \tau \vdash t : \rho}{\Gamma + \langle x, y\rangle : \sigma \wedge \tau \vdash t : \rho}\ [\textsc{ProdL}]$$

[5] We've added arrays and lists, as we need them for our examples below.

The rules for universal quantifications are straightforward:

$$\frac{\Gamma \vdash t : \sigma \qquad \alpha \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash t : \forall \alpha . \sigma} \qquad\qquad \frac{\Gamma \vdash t : \forall \alpha . \sigma}{\Gamma \vdash t : \sigma[\tau/\alpha]}$$

If a type variable does not appear in the assumptions, we can quantify over it. Once we have quantified over a type variable, we can replace it with any old type that we want. (For the definition of substitution on types, see Section 9.1.3.)

The rules for the existential type quantifier create an abstract datatype, and let us use one (written as **open**, corresponds to an "import" statement):

$$\frac{\Gamma \vdash e : \sigma[\tau/\alpha]}{\Gamma \vdash \textbf{abstype } \alpha \textbf{ is } e : \exists \alpha . \sigma} \text{ [ExI]} \qquad \frac{\Gamma \vdash \textbf{abstype } p \textbf{ is } e : \exists \alpha . \sigma \qquad \Gamma + e : \sigma[p/\alpha] \vdash f : \rho}{\Gamma \vdash \textbf{open } (\textbf{abstype } p \textbf{ is } e) \textbf{ in } f : \rho} \text{ [ExE]}$$

When we import the abstrat datatype, the type variable α in the type is replaced with the constant p (given by the name of the datatype); nothing can be assumed about p except for the operations available in the abstract datatype. These are added to the assumptions (the context Γ) in the second premise of the rule ExE, and can thus be used when typing f. Since an abstract datatype will always be a tuple of n operations (the cases of $n = 0$ and $n = 1$ are possible, but rarely make sense), it makes sense to combine rules ExE and ProdL into one rule:

$$\frac{\Gamma \vdash \textbf{abstype } \alpha \textbf{ is } \langle e_1, \ldots, e_n \rangle : \exists \alpha . \sigma_1 \wedge \ldots \wedge \sigma_n \qquad \Gamma + x_1 : \sigma_1[p/\alpha] + \ldots + x_n : \sigma_n[p/\alpha] \vdash f : \rho}{\Gamma \vdash \textbf{open } (\textbf{abstype } \alpha \textbf{ is } \langle e_1, \ldots, e_n \rangle) \textbf{ in } f : \rho} \text{ [ExE']}$$

Note how the only thing that is visible of the abstract datatype e from the outside is the type, *i.e.*. $\exists \alpha . \tau$ and if τ is a tuple $\sigma_1 \wedge \ldots \wedge \sigma_n$, we can refer to the components σ_i (we call them x_i above, the original names are lost behind the existential quantifier.)

10.3 Example Deriviations

10.3.1 Points

A classical example are points as an abstract datatype. To start with, let us model types as cartesian tupels, and three operations mk_point, let and get_x. This can be wrapped in one big module expression as follows:

$$\begin{aligned} \Gamma_0 \vdash \textbf{let} \quad \text{mk\_point} &= \lambda xy . \langle x, y \rangle \\ \text{len} &= \lambda \langle x, y \rangle . \ \mathrm{sqrt}(x \cdot x + y \cdot y) \\ \text{get\_x} &= \lambda \langle x, y \rangle . x \\ \textbf{in } \langle \text{mk\_point}, \text{len}, \text{get\_x} \rangle &: (\mathbb{R} \to \mathbb{R} \to \mathbb{R} \wedge \mathbb{R}) \wedge (\mathbb{R} \wedge \mathbb{R} \to \mathbb{R}) \wedge (\mathbb{R} \wedge \mathbb{R} \to \mathbb{R}) \end{aligned}$$

We can now apply rule ExI and obtain an existential type:

$$\begin{aligned} \Gamma_0 \vdash \textbf{let} \quad \text{mk\_point} &= \lambda xy . \langle x, y \rangle \\ \text{len} &= \lambda \langle x, y \rangle . \ \mathrm{sqrt}(x \cdot x + y \cdot y) \\ \text{get\_x} &= \lambda \langle x, y \rangle . x \\ \textbf{in } \langle \text{mk\_point}, \text{len}, \text{get\_x} \rangle &: \ (\mathbb{R} \to \mathbb{R} \to \mathbb{R} \wedge \mathbb{R}) \wedge (\mathbb{R} \wedge \mathbb{R} \to \mathbb{R}) \wedge (\mathbb{R} \wedge \mathbb{R} \to \mathbb{R}) \\ &\equiv (\mathbb{R} \to \mathbb{R} \to \alpha) \wedge (\alpha \to \mathbb{R}) \wedge (\alpha \to \mathbb{R})[\mathbb{R} \wedge \mathbb{R}/\alpha] \end{aligned}$$

$$\begin{aligned} \Gamma_0 \vdash \textbf{abstype } pt \textbf{ is } \textbf{let} \quad \text{mk\_point} &= \lambda xy . \langle x, y \rangle \\ \text{len} &= \lambda \langle x, y \rangle . \ \mathrm{sqrt}(x \cdot x + y \cdot y) \\ \text{get\_x} &= \lambda \langle x, y \rangle . x \\ \textbf{in } \langle \text{mk\_point}, \text{len}, \text{get\_x} \rangle &: \exists \alpha . (\mathbb{R} \to \mathbb{R} \to \alpha) \wedge (\alpha \to \mathbb{R}) \wedge (\alpha \to \mathbb{R}) \end{aligned}$$

We can now use the datatype to type an operation $\mathrm{len}(\mathrm{mk\_point}(3)(4))$ (this should evaluate to 5, but we are only interested in the type for now). To use len and mk_point, we must import them (*i.e.* open the datatype). Let us write P for the point ADT, then the above deriviation is

$$\mathscr{D} \stackrel{def}{=} \frac{\vdots}{\Gamma_0 \vdash P : \exists\alpha.\,(\mathbb{R}\to\mathbb{R}\to\alpha)\wedge(\alpha\to\mathbb{R})\wedge(\alpha\to\mathbb{R})} \tag{26}$$

We now want to infer the type of **open** P **in** $\mathrm{len}(\mathrm{mk\_point}(3)(4))$ in the context Γ_0. The first thing to note is that the names, len and $\mathrm{mk_point}$ are actually not visible from the outside of the datatype (from the usability perspective, bit of a disaster); all we know is these are the second and third operation of the datatype, they might as well be called foo and baz. Using rule ExE' from above, we write

$$\dfrac{\mathscr{D}\qquad \dfrac{\dfrac{\dfrac{}{\Gamma_1\vdash f_1:\mathbb{R}\to\mathbb{R}\to pt}\qquad\dfrac{}{\Gamma_1\vdash 3:\mathbb{R}}}{\Gamma_1\vdash f_1(3):\mathbb{R}\to pt}\qquad\dfrac{}{\Gamma_1\vdash 4:\mathbb{R}}}{\Gamma_1\vdash f_1(3)(4):pt}\qquad\dfrac{}{\Gamma_1\vdash f_2:pt\to\mathbb{R}}}{\Gamma_1\equiv\Gamma_0+f_1:\mathbb{R}\to\mathbb{R}\to pt+f_2:pt\to\mathbb{R}+f_3:pt\to\mathbb{R}\vdash f_2(f_1(3)(4)):\mathbb{R}}}{\Gamma_0\vdash\textbf{open}\ P\ \textbf{in}\ f_2(f_1(3)(4)):\mathbb{R}}$$

where $\mathscr{D}$ is the derivation from equation (26).

10.3.2 Stacks

Stacks are to abstract datatypes what *escherichia coli* is to microbiology — an obliging study object. We briefly show two different implementations of stacks.

The first one is the functional implementation of stacks as lists:

$$\dfrac{\dfrac{\Gamma_0\vdash\langle[],\mathrm{hd},\mathrm{tail},\mathrm{cons},\mathrm{nil}\rangle:[\alpha]\wedge([\alpha]\to\alpha)\wedge([\alpha]\to[\alpha])\wedge(\alpha\to[\alpha]\to[\alpha])\wedge([\alpha]\to\mathbb{B})}{\Gamma_0\vdash\textbf{abstype}\ st\ \textbf{is}\ \langle[],\mathrm{hd},\mathrm{tail},\mathrm{cons},\mathrm{null}\rangle:\exists\beta.\,\beta\wedge(\beta\to\alpha)\wedge(\beta\to\beta)\wedge(\alpha\to\beta\to\beta)\wedge(\beta\to\mathbb{B})}}{\Gamma_0\vdash\textbf{abstype}\ st\ \textbf{is}\ \langle[],\mathrm{hd},\mathrm{tail},\mathrm{cons},\mathrm{null}\rangle:\forall\alpha.\,\exists\beta.\,\beta\wedge(\beta\to\alpha)\wedge(\beta\to\beta)\wedge(\alpha\to\beta\to\beta)\wedge(\beta\to\mathbb{B})}$$

Here, hd, tail, cons, and null are the operations which return the head and the rest of the list, append a new element to the front (: in Haskell) and check wether the list is empty. This abstract datatype is parametrically polymorph (it has type $\forall\alpha.\exists\beta.\sigma$, so σ contains both α and β. When we open the datatype, we have to instantiate the type variable α, so we get stacks of a certain type. (We may instantiate α with another type variable at point of usage, but the point is that a stack has one specific type for each usage.)

The second implementation of stacks is as a tuple of arrays and stack pointer. We derive the same type as before. We give (via let) names to the operations now — we did not do so above — but that is more of a

convenience:

$$
\begin{array}{lrl}
\Gamma_0 \vdash & \textbf{let} & \text{empty} = \textbf{let } x = \text{alloc}(\textbf{array } \alpha, 1000) \textbf{ in } \langle x, 0\rangle \\
& & \text{top} = \lambda\langle a, p\rangle.\, a[p-1] \\
& & \text{pop} = \lambda\langle a, p\rangle.\, \langle a, p-1\rangle \\
& & \text{push} = \lambda b\langle a, p\rangle.\, \langle a[p] := b, p+1\rangle \\
& & \text{is\_empty} = \lambda\langle a, p\rangle.\, p = 0 \\
& \textbf{in} & \langle \text{empty}, \text{top}, \text{pop}, \text{push}, \text{is\_empty}\rangle : \\
& & (\textbf{array } \alpha \wedge \textbf{int}) \wedge ((\textbf{array } \alpha \wedge \textbf{int}) \to \alpha) \wedge ((\textbf{array } \alpha \wedge \textbf{int}) \to (\textbf{array } \alpha \wedge \textbf{int})) \wedge \\
& & (\alpha \to (\textbf{array } \alpha) \wedge (\textbf{int}) \to (\textbf{array } \alpha \wedge \textbf{int})) \wedge ((\textbf{array } \alpha \wedge \textbf{int})) \to \mathbb{B} \\
& & \equiv (\beta \wedge (\beta \to \alpha) \wedge (\beta \to \beta) \wedge (\alpha \to \beta \to \beta) \wedge (\beta \to \mathbb{B}))[\textbf{array } \alpha \wedge \textbf{int}/\beta]
\end{array}
$$

$$
\frac{}{\Gamma_0 \vdash \textbf{abstype } st \textbf{ is let } \ldots \textbf{ in } \ldots : \exists\beta.\, \beta \wedge (\beta \to \alpha) \wedge (\beta \to \beta) \wedge (\alpha \to \beta \to \beta) \wedge (\beta \to \mathbb{B})}
$$

$$
\frac{}{\Gamma_0 \vdash \textbf{abstype } st \textbf{ is let } \ldots \textbf{ in } \ldots : \forall\alpha.\, \exists\beta.\, \beta \wedge (\beta \to \alpha) \wedge (\beta \to \beta) \wedge (\alpha \to \beta \to \beta) \wedge (\beta \to \mathbb{B})}
$$

Let us call the stacks above S_1 and S_2. The reader is invited to show how stacks can be used, *e.g.* to type the expression $\text{top}(\text{push}(7, \text{push}(5, \text{empty})))$, which is more exactly written as

$$
\frac{?}{\Gamma_0 \vdash \textbf{open } S_1 \textbf{ in } \text{top}(\text{push}(7, \text{push}(5, \text{empty}))) : ?}
$$

References

[1] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, April 1987.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[3] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, January 1982. Association for Computing Machinery.

[4] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.