Extending the Specification of the Toy Calculator

Big Step (Natural, Kahn-style) Operational Semantics

We seem to have bypassed a stage in our development — the relational (input-output) operational *specification* of the calculator. Let us extend the previous specification which we had given in Kahn's Natural semantics style to deal with the new forms that we have introduced. Recall that the calculator specifies a program that operates on abstract syntactic forms, returning canonical syntactic answers. We assume you are familiar with the presentation of inference rules with provisos, and will not provide any commentary with the rules.

(CalcNum) $\dfrac{}{\underline{N} \Longrightarrow \underline{N}}$

(CalcBool) $\dfrac{}{\underline{B} \Longrightarrow \underline{B}}$

(CalcPlus) $\dfrac{\underline{E_1} \Longrightarrow \underline{N_1} \quad \underline{E_2} \Longrightarrow \underline{N_2}}{\underline{E_1 + E_2} \Longrightarrow \underline{N}}$ provided $PLUS(\underline{N_1}, \underline{N_2}, \underline{N})$

(CalcTimes) $\dfrac{\underline{E_1} \Longrightarrow \underline{N_1} \quad \underline{E_2} \Longrightarrow \underline{N_2}}{\underline{E_1 * E_2} \Longrightarrow \underline{N}}$ provided $TIMES(\underline{N_1}, \underline{N_2}, \underline{N})$

(CalcNot) $\dfrac{\underline{E_1} \Longrightarrow \underline{B_1}}{\underline{\neg E_1} \Longrightarrow \underline{B}}$ provided $NOT(\underline{B_1}, \underline{B})$

(CalcAnd) $\dfrac{\underline{E_1} \Longrightarrow \underline{B_1} \quad \underline{E_2} \Longrightarrow \underline{B_2}}{\underline{E_1 \wedge E_2} \Longrightarrow \underline{B}}$ provided $AND(\underline{B_1}, \underline{B_2}, \underline{B})$

CalcOr) $\dfrac{\underline{E_1} \Longrightarrow \underline{B_1} \quad \underline{E_2} \Longrightarrow \underline{B_2}}{\underline{E_1 \vee E_2} \Longrightarrow \underline{B}}$ provided $OR(\underline{B_1}, \underline{B_2}, \underline{B})$

(CalcEq) $\dfrac{\underline{E_1} \Longrightarrow \underline{N_1} \quad \underline{E_2} \Longrightarrow \underline{N_2}}{\underline{E_1 = E_2} \Longrightarrow \underline{B}}$ provided $EQ(\underline{N_1}, \underline{N_2}, \underline{B})$

(CalcGt) $\dfrac{\underline{E_1} \Longrightarrow \underline{N_1} \quad \underline{E_2} \Longrightarrow \underline{N_2}}{\underline{E_1 > E_2} \Longrightarrow \underline{B}}$ provided $GT(\underline{N_1}, \underline{N_2}, \underline{B})$

Ensuring that Expressions are Well-formed

We have to ensure that the expressions which we provide to the calculator are not ill-formed, such as adding a numeric expression to a boolean, etc. The mechanism to do so

is to use a "type system", and specify a checker that weeds out the badly formed expressions.

Let us define (toy language) types, reflecting numeric and boolean expressions. Let us call these (unimaginatively) IntT and BoolT. Note that these are just syntactic names for the types we are creating in out language. [As we introduce more types and type constructions to our toy language, we will introduce more toy language type names/type operations.]

We specify a *relation* $\vdash E : T$ to say that (according to the rules of the language) "*expression E has type T.*" The turnstile and the colon are just markers/separators, but we will expect you to adhere to this notation. This typing relation is recursively defined, and again we use inference rules that follow the structure of the expression E.

(NumT) $\dfrac{}{\vdash N : \text{IntT}}$ All numerals N have type IntT.

(BoolT) $\dfrac{}{\vdash B : \text{BoolT}}$ All boolean (`myBool`) constants B have type BoolT.

(PlusT) $\dfrac{\vdash E_1 : \text{IntT} \qquad \vdash E_2 : \text{IntT}}{\vdash E_1 + E_2 : \text{IntT}}$

All addition expressions $E_1 + E_2$ have type IntT, provided the subexpressions E_1 and E_2 both have type IntT

(TimesT) $\dfrac{\vdash E_1 : \text{IntT} \qquad \vdash E_2 : \text{IntT}}{\vdash E_1 * E_2 : \text{IntT}}$

All multiplication expressions $E_1 * E_2$ have type IntT, provided the subexpressions E_1 and E_2 both have type IntT

(NotT) $\dfrac{\vdash E_1 : \text{BoolT}}{\vdash \neg E_1 : \text{BoolT}}$

All negation expressions $\neg E_1$ have type BoolT, provided the subexpressions E_1 have type BoolT

(AndT) $\dfrac{\vdash E_1 : \text{BoolT} \qquad \vdash E_2 : \text{BoolT}}{\vdash E_1 \wedge E_2 : \text{BoolT}}$

All conjunction expressions $E_1 \wedge E_2$ have type BoolT, provided the subexpressions E_1 and E_2 both have type BoolT

(OrT) $\dfrac{\vdash E_1 : \text{BoolT} \qquad \vdash E_2 : \text{BoolT}}{\vdash E_1 \vee E_2 : \text{BoolT}}$

All disjunction expressions $E_1 \vee E_2$ have type BoolT, provided the subexpressions E_1 and E_2 both have type BoolT

$$\textbf{(EqT)}\ \frac{\vdash E_1 : \text{IntT} \qquad \vdash E_2 : \text{IntT}}{\vdash E_1 = E_2 : \text{BoolT}}$$

All numeric equality expressions $E_1 = E_2$ have type BoolT, provided the subexpressions E_1 and E_2 both have type IntT

$$\textbf{(GtT)}\ \frac{\vdash E_1 : \text{IntT} \qquad \vdash E_2 : \text{IntT}}{\vdash E_1 > E_2 : \text{BoolT}}$$

All greater-than expressions $E_1 > E_2$ have type BoolT, provided the subexpressions E_1 and E_2 both have type IntT

Expressions that can be given a type under these rules are called "well-typed". Otherwise the expression is ill-formed, and so does not have a type ("ill-typed").

Type Preservation (version 1)

One of the nicest results in Programming Language theory is the following:

> **Theorem** (Type Preservation under $E \Longrightarrow A$)
> For all expressions E, A, for all types T, if $\vdash E : T$ and $E \Longrightarrow A$, then $\vdash A : T$

The importance of this result is that calculation does not change the type of a well-typed expression — the answer is of the same type. So we can type-check the expression once and for all (statically/ at compile time / before execution), and do not have to worry that a well-typed expression will become ill-typed any time during execution. From a pragmatic viewpoint, this means we need not insert (expensive) type checks in the abstract machine code during compilation just to ensure that the expression is well-typed.
[Note however that some checks may still have to be inserted. For example, the typing rules do not prevent division by zero].

but this isn't violation of types

A crucial assumption we will make is that the elementary operations are type-sound. For example, addition of two numerals returns a numeral and e.g., $GT(N_1, N_2, B)$ will return a boolean B (i.e., $\vdash B : \text{BoolT}$) for every pair of numerals N_1, N_2 (i.e., $\vdash N_1 : \text{IntT}$ and $\vdash N_2 : \text{IntT}$.)

Guess how we will prove this theorem? How are the relations $\vdash E : T$ and $E \Longrightarrow A$ defined? By induction on the structure of expressions E — so structural induction is the technique that is applicable.

Proof (By Induction on the structure/*ht* of E).
Base cases ($ht(E) = 0$)
Subcase ($E \equiv N$)
Assume $\vdash E : T$. Therefore $T = \text{IntT}$. Now $N \Longrightarrow N$. So trivially $\vdash N : \text{IntT}$

Subcase ($\underline{E} \equiv \underline{B}$)
Assume $\vdash \underline{E} : \underline{T}$. Therefore $\underline{T} = \underline{\text{BoolT}}$. Now $\underline{B} \Longrightarrow \underline{B}$. So trivially $\vdash \underline{B} : \underline{\text{BoolT}}$

Induction Hypothesis: Assume that for all expressions $\underline{E'}$, such that $ht(\underline{E'}) \leq k$, for all expressions $\underline{A'}$, for all types $\underline{T'}$, if $\vdash \underline{E'} : \underline{T'}$ and $\underline{E'} \Longrightarrow \underline{A'}$, then $\vdash \underline{A'} : \underline{T'}$

Induction Step ($ht(\underline{E}) = 1 + k$)
Subcase ($\underline{E} \equiv \underline{E_1 > E_2}$)
Assume $\vdash \underline{E} : \underline{T}$. Therefore by (**GtT**), $\underline{T} = \underline{\text{BoolT}}$ and $\vdash \underline{E_1} : \underline{\text{IntT}}$ and $\vdash \underline{E_2} : \underline{\text{IntT}}$.
Now by (**CalcGt**) $\underline{E_1 > E_2} \Longrightarrow \underline{B}$ provided $\underline{E_1} \Longrightarrow \underline{N_1}$, $\underline{E_2} \Longrightarrow \underline{N_2}$ and $GT(\underline{N_1}, \underline{N_2}, \underline{B})$ for some $\underline{N_1}, \underline{N_2}$.
By IH on $\underline{E_1}$, $\vdash \underline{N_1} : \underline{\text{IntT}}$ and by IH on $\underline{E_2}$, $\vdash \underline{N_2} : \underline{\text{IntT}}$
So by *type-soundness* of $GT(\underline{N_1}, \underline{N_2}, \underline{B})$, we have $\vdash \underline{B} : \underline{\text{BoolT}}$.

All other subcases are similar (and perhaps simpler). □

Exercise: Encode the type-checking relation $\vdash \underline{E} : \underline{T}$ in PROLOG as a predicate `hastype(E,T).`