Typing

Brigitte Pientka

April 14, 2007

So far, we only have considered syntax (i.e. what expressions are syntactically well-formed?) and operational semantics (i.e. how are we executing an expression?). However, there are many expressions which cannot be evaluated and our interpreter would get stuck. Examples of such expressions include : $\mathsf{if}\ 0\ \mathsf{then}\ 45\ \mathsf{else}\ 33$, $0 + \mathsf{true}$, $(\mathsf{fn}\ x \Rightarrow x + 1\ )\ \mathsf{true}$, etc.

How can we statically check whether an expressions would potentially lead to a runtime error? – We already encountered types in the language SML. Types approximate the runtime behavior and provide an effective light-weight tool for reasoning about programs. They allow us to detect errors statically and early on the development cycle. More generally, a type system is a tractable syntactic method for proving the absence of certain program behaviors. This is done by classifying expressions according to the kinds of values they compute.

In this note, we will briefly describe how this can be done, and what issues arise. We will start by considering our Nano-ML language which consisted of numbers, booleans, if-expressions and primitive operations. Next, we will considering an extension where we have variables and let-expressions. Finally, we will include functions and function application.

1 Basic Types

We begin by considering Nano-ML which includes numbers, booleans, if-expressions and primitive operations. Recall our inductive definition of these expressions:

$$\begin{array}{lll} \text{Operations } \mathsf{op} & ::= & + \mid - \mid * \mid < \mid = \\ \text{Expressions } e & ::= & n \mid e_1\ \mathsf{op}\ e_2 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{array}$$

This grammar inductively specifies well-formed expressions. We will now concentrate on the questions: What expressions do we consider well-typed? How do we assign an expression a type?

As mentioned earlier, types classify expressions according to their value. So what are the values in Nano-ML? Values in Nano-ML are numbers and booleans. Therefore we will only have two basic types which suffice to approximate the run-time behavior of NanoML expressions. The type int characterizes number and the type bool describes the booleans true and false. In other words, the expressions true and false are elements of the type $\textsc{bool}$, and number n are elements of the type int. We also sometimes say true and false inhabit the type bool, or similarly number n inhabit the type int. We will write the capital letter T for types.

$$\text{Types } T \ ::= \ \mathsf{int} \mid \mathsf{bool}$$

Next, we formally describe when an expression is well-typed using the following judgment:

$$e : T \quad \text{expression } e \text{ has type } T$$

We will define when an expression is well-typed inductively on the structure of the expression. A term is typable if there is some type T s.t. $e : T$. We start by considering numbers and booleans.

$$\frac{}{n : \mathsf{int}}\ \text{T-NUM} \qquad \frac{}{\mathsf{true} : \mathsf{bool}}\ \text{T-TRUE} \qquad \frac{}{\mathsf{false} : \mathsf{bool}}\ \text{T-FALSE}$$

Next, let us consider the if-expression. When is $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ well-typed? And what should its type be? – Intuitively, we can assign $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ a type T, if

- expression e has type bool (i.e. expression e evaluates eventually to a boolean)
- expression e_1 has some type T
- expression e_2 has the same type T

This can be formalized using inference rules as follows:

$$\frac{e : \mathsf{bool} \quad e_1 : T \quad e_2 : T}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T}\ \text{T-IF}$$

Finally, we consider the rules for primitive operations. We only show the rules for addition, multiplication, equality but the others are straightforward and follow similar principle.

$$\frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 + e_2 : \mathsf{int}}\ \text{T+} \qquad \frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 * e_2 : \mathsf{int}}\ \text{T*} \qquad \frac{e_1 : T \quad e_2 : T}{e_1 = e_2 : \mathsf{bool}}\ \text{T=}$$

2 Typing for tuples and projections

In a similar fashion, we can introduce types for tuples and projections.

$$\text{Expressions } e \ ::= \ \ldots \mid (e_1, e_2) \mid \mathsf{fst}\ e \ \mid \mathsf{snd}\ e$$

We can create tuples via $(e_1,\ e_2)$ and we can take them apart via $\mathsf{fst}\ e$ which extracts the first component of a tupel e, and $\mathsf{snd}\ e$ will extract the second component of a tupel e. Considering the evaluation rules for tuples, it is clear we have now not only numbers and booleans as values, but a tuple is also a possible value.

The type classifying tuples is called *product* and written as $T_1 \times T_2$. The rules for tuples and projections are then in fact straightforward.

$$\frac{e_1 : T_1 \quad e_2 : T_2}{(e_1,\ e_2) : T_1 \times T_2}\ \text{T-PAIR} \qquad \frac{e : T_1 \times T_2}{\mathsf{fst}\ e\ : T_1}\ \text{T-FST} \qquad \frac{e : T_1 \times T_2}{\mathsf{snd}\ e\ : T_2}\ \text{T-SND}$$

3 Typing for variables and let-expressions

In this section, we would like to extend our typing rules to handle variables and let-expressions. The first observation is that we need to be able to reason with assumptions about variables. For example, consider the following let-expressions: $\mathsf{let}\ x = 5\ \mathsf{in}\ x + 3\ \mathsf{end}$. We would like to argue as follows:

- 5 has int. Therefore the variable x will be bound at runtime to a value of type int.
- Assuming that x has type int, $x + 3$ has type int, because each subexpression has type int.

More generally, the expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$ has type T, if

- e_1 has type T_1
- assuming x has type T_1, the expression e_2 has type T.

To handle bound variables and be able to reason about them, we introduce a context Γ, which keeps track of our assumptions. The assumptions "variable x has type T" is written $x{:}T$. We can define a context Γ inductively as follows:

$$\text{Context } \Gamma \ := \ \cdot \mid \Gamma, x{:}T$$

$\cdot$ describes the empty context, i.e. there are no typing assumptions. If we have a context Γ then $\Gamma, x{:}T$ is also a context. Sometimes we also call $x{:}T$ a typing declaration. Every typing declaration $x{:}T$ occurs uniquely, i.e. for any two declarations $x{:}T$ and $x'{:}T'$, we have $x \neq x'$.

Next, we will generalize our typing judgment to take into account the context Γ (the set of assumptions available).

$$\Gamma \vdash e : T \quad \text{Expression } e \text{ has type } T \text{ using the typing declarations in } \Gamma$$

In the previous rules, the set of assumptions does not change, and hence we simply carry it as an extra parameter.

$$\dfrac{}{\Gamma \vdash n : \mathsf{int}}\ \text{T-NUM} \qquad \dfrac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \text{T-TRUE} \qquad \dfrac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \text{T-FALSE}$$

$$\dfrac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}\ \text{T+} \qquad \dfrac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 * e_2 : \mathsf{int}}\ \text{T*} \qquad \dfrac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 = e_2 : \mathsf{bool}}\ \text{T=}$$

$$\dfrac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T}\ \text{T-IF}$$

The more interesting question is how to formally deal with variables and describe the recipe for assigning a type to a let-expression. If we have an assumption $x{:}T$ in Γ, then we can conclude that a variable x has type T. The recipe for the let-expression is directly translated into an inference rule.

$$\dfrac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{T-VAR} \quad \dfrac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x{:}T_1 \vdash e_2 : T}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : T}\ \text{T-LET}\ \ x \text{ must be new}$$

We note that x must be new to ensure that the declaration $x{:}T_1$ does not clash with any other declaration in Γ. This can always be achieved by appropriately renaming the bound variable x in the expression e_2.

To check whether an expression e has a type T, we construct a proof for $\cdot \vdash e : T$. Here is an example, how we can check whether $\mathsf{let}\ x = 5\ \mathsf{in}\ x + 2\ \mathsf{end}$ has type int. We leave out the name of the rules so it fits on the page.

$$\dfrac{\dfrac{}{\cdot \vdash 5 : \mathsf{int}} \quad \dfrac{\dfrac{\dfrac{}{x{:}\mathsf{int} \vdash x : \mathsf{int}} \quad \dfrac{}{x{:}\mathsf{int} \vdash 2 : \mathsf{int}}}{x{:}\mathsf{int} \vdash x + 2 : \mathsf{int}} \quad \dfrac{\dfrac{}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash x : \mathsf{int}} \quad \dfrac{}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash y : \mathsf{int}}}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash x * y : \mathsf{int}}}{x{:}\mathsf{int} \vdash \mathsf{let}\ y = x + 2\ \mathsf{in}\ x * y\ \mathsf{end} : \mathsf{int}}}{\cdot \vdash \mathsf{let}\ x = 5\ \mathsf{in}\ \mathsf{let}\ y = x + 2\ \mathsf{in}\ x + y\ \mathsf{end}\ \mathsf{end} : \mathsf{int}}$$

Note that we can have unused assumptions in some branches.

Type-checking vs Type inference So far we mainly have talked about assigning a type to an expression. In practice however we often distinguish between type checking and type inference. In type checking, we give an expression e and a type T and we check whether $e : T$, i.e. expression e indeed has type T. However, we may not always have both the expression and the type. For example, in the rule T-LET, we cannot simply check whether expression e_1 has the type T_1, because we don't know what it is. We must infer a type for expression e_1.

In type inference, we give the expression e and infer a type T s.t. the expression has type T. The next question, we can ask is, whether we can always infer a unique type corresponding to an expression. In the language we have encountered so far, this is the case. This is a property about our type system and can be stated more formally as follows:

Uniqueness If $\Gamma \vdash e : T$ and $\Gamma \vdash e : T'$ then $T = T'$.

In fact it is quite easy to see that the typing rules are deterministic. At any given point, there is exactly one rule applicable for an expression.

4 Typing for functions and function application

Next we extend our type system to include functions and function application. We have seen that functions are first-class values, and since types classify values we will add a new type to our NanoML types, the function type $T_1 \to T_2$.

$$\text{Types } T \ ::= \ \mathsf{int} \mid \mathsf{bool} \mid T_1 \to T_2$$

A function $\mathsf{fn}\ x \Rightarrow e$ has type $T_1 \to T_2$ if assuming x has type T_1, we can show that the body e_2 has type T_2. An application $e_1\ e_2$ has type T if expression e_1 has type $T_2 \to T$ and e_2 has type T_2. In other words, expression e_2 is a suitable input to the function computed by e_1. More formally,

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \mathsf{fn}\ x \Rightarrow e\ : T_1 \to T_2}\ \text{T-FN} \qquad \frac{\Gamma \vdash e_1 : T_2 \to T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T}\ \text{T-APP}$$

In interesting question to ask whether it is still true that every function has a unique type. This is no longer true. For example,

$$\begin{array}{ll} \mathsf{fn}\ x \Rightarrow x & \text{has type } \mathsf{int} \to \mathsf{int} \\ \mathsf{fn}\ x \Rightarrow x & \text{has type } \mathsf{bool} \to \mathsf{bool} \\ \mathsf{fn}\ x \Rightarrow x & \text{has type } (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}) \\ & \ldots \end{array}$$

In fact, the identity function has infinitely many types! How can we recover that every expression has a unique type? – There are two solutions to this problem. The easiest is to allow type annotations to resolve ambiguity. As a consequence, we would write $\mathsf{fn}\ x{:}T \Rightarrow e$ and annotate the input variable x with its type. So to use the identity function as a function for booleans we would need to write $\mathsf{fn}\ x : \mathsf{bool} \Rightarrow x$ and when we would like to use it for integers we would need to write $\mathsf{fn}\ x : \mathsf{int} \Rightarrow x$. This is unfortunate, because the function is generic and can be executed with integers and with booleans. A remedy to this problem is to allow type variables α. Instead of requiring that every expression has a unique type, we allow that an expression may have multiple types, but it must have a *principal type*, a type which is more general from which all others can be derived. We will come back to how to infer that a given expression has a principal type T later. So our types include now:

$$\text{Types } T \ ::= \ \mathsf{int} \mid \mathsf{bool} \mid T_1 \to T_2 \mid \alpha$$

5 References

In this section, we briefly highlight how to extend our syntax and types to include references.

$$\begin{array}{lcl} \text{expression } e & ::= & \ldots \mid e_1 := e_2 \mid !e \mid \mathsf{ref}\ e \mid () \\ \text{types } T & ::= & \ldots \mid T\ \mathsf{ref} \mid \mathsf{unit} \end{array}$$

Just to consider the typing for this extension is in fact fairly straightforward. However, we do not consider extending the operational semantics with references which would require us to model formally the heap.

$$\frac{\Gamma \vdash e_1 : T\ \mathsf{ref} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}} \qquad \frac{\Gamma \vdash e : T\ \mathsf{ref}}{\Gamma \vdash !e : T} \qquad \frac{\Gamma \vdash e : T}{\Gamma \vdash \mathsf{ref}\ e : T\mathsf{ref}} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}}$$

6 Properties of type systems

As we mentioned earlier, type systems approximate the run-time behavior and we designed the typing rules in such a way that certain expressions which could potentially cause problems during run-time are considered as ill-typed. For example, $\mathsf{true} + 5$ would lead to an error during runtime and the type systems identifies this expression as ill-typed. The intention is that if an expression e has a type T, then evaluation of e will not get stuck. Moreover, if expression e has type T and e evaluates to some value v then v will have the same type T. In other words types are preserved during evaluation. Both of these two properties together are called *type safety*:

Safety If expression e has type T, then either e is already a value or we can evaluate it in one step to another expression e' and e' has type T.

Safety incorporates really two properties which we merged together in the previous statement. The first one, called *progress*, says if an expression e is well-typed, then either it is a value or we can evaluate it to some other expression e'. The second property, called *preservation*, says that if an expression e has type T and e evaluates some expression e' then e' has type T.

The operational semantics we have introduced previously is a big-step semantics, and evaluates expressions to values in one big step. Hence we do not have the ability to reason about intermediate expressions which may occur during evaluation. To prove progress, we need a more fine grained model of evaluation, namely a small-step semantics. However, on property we can even prove about our big-step semantics is *type preservation.* If an expression e has type T and e evaluates to some final value v, then v will have the same type T. More formally this can be stated as follows:

Preservation If $e \Downarrow v$ and $e : T$ then $v : T$.

These meta-theoretic properties ensure that in fact the typing rules do what we intended them to do. Or in other words, if a program type checks, then it will not go wrong during evaluation. Remarkably this property even holds in the presence of references, exceptions and other features we find in real programming languages. In the presence of references for example we can guarantee statically that we will never try to access a memory location which wasn't created appropriately earlier. This in essence guarantees that the program when executed will not core dump.