ADVANCED PROGRAMMING LANGUAGES

Fall 2024, CM 0632. MSc in Computer Science and Information Technology, Ca' Foscari University of Venice

Introducing Types and Typed Systems

There is no universal agreement on a common definition for the notion of *type*. Instead of asking the question *What is a type?* we may therefore ask *why types are needed* in programming languages.

As note by Luca Cardelli and Peter Wegner in their famous paper, "*the road from untyped to typed universes has been followed many times in many different fields, and largely for the same reasons*". Consider, for example, the following untyped universes, where *untyped* means that there is in fact just one class of values:

- Bit strings in computer memory
- λ-expressions in the lambda-calculus
- Sets in set theory

Bit strings in computer memory are the most concrete of these universes, in which the only type (class of values) is the memory word, a bit string of fixed size in terms of which everything is represented, ultimately: characters, numbers, pointers, structured data, programs, etc.

In the (pure) lambda-calculus, as we have discussed, everything is (meant to represent) a function. Numbers, data structures and even bit strings can be represented by appropriate functions. Yet there is only one type: the type of functions from values to values, where all values are themselves functions.

In set theory, everything is a set, though the structure may vary significantly from set to set. To understand how untyped this universe is, one must remember that most of mathematics is represented in set theory by sets whose structural complexity reflects the complexity of the structures being represented. For example, functions are represented by possibly infinite sets of ordered pairs with unique first components, while integers are generally represented by sets of sets of sets whose level of nesting represents the cardinality of the integer:

$$
\begin{array}{lllll}
0 & = & \{\} & = & \emptyset \\
1 & = & \{0\} & = & \{\emptyset\} \\
2 & = & \{0,1\} & = & \{\emptyset,\{\emptyset\}\} \\
3 & = & \{0,1,2\} & = & \{\emptyset,\{\emptyset\}\,\{\emptyset,\{\emptyset\}\}\} \\
\ldots & & \ldots & & \ldots
\end{array}
$$

Typed domains

As soon as we start *working* in an untyped universe, types arise naturally.

- In computer memory, different operations presuppose different computer memory representations, for characters, integers, strings, . . .
- In set theory, as we have discussed, distinguishing among different set *orders* necessary to avoid paradoxes;
- In lambda calculus, as shown in our applied version, some functions are chosen to represent boolean values, others to represent integers, . . . and so on.

Untyped domains decompose thus into **subsets with uniform behavior**, which we may take as our definition of **types**. A type may be viewed as a set of dressed with clothing (or a suit of armor) that:

- provides values with a protective covering that hides the values' underlying representation,
- constrains the way values may be operated with by operators and functions.

For example, integers are presented as atomic values, thus with a a transparent representation (in languages that leave bitwise operators aside . . .) and exhibit uniform behavior by having the same set of applicable operations, Similarly, functions from integers to integers have a protected, internal representation and behave uniformly in that they apply to objects of a given type and produce values of a given type.

Of course, having defined the intended ways that values of a certain type should be operated upon, we need ways and tools to ensure that those intentions are complied with. Type systems provide one such tool, as their main purpose is to impose constraints which prevent objects / expressions from inconsistent interaction with other objects / expressions.

To understand what I mean by "inconsistent interaction", have a look at this video!

The code snippets shown in the video are (funny and) extreme, but they are conceptually not different from situations that arise in the lambda calculus as soon as we move from the untyped domain of the pure lambda calculus to a typed domain, i.e. one in which we are able to distinguish between terms whose result will be a numeric value, terms whose result will be a boolean, and terms whose result will be a lambda abstraction. In addition, we would like to provide such guarantees statically, by just inspecting the structure of expressions.

The basic components of type systems may be described as follows:

1. *Type binding*: the mechanism by which we associate a type with each of the basic components of a program, like constants and identifiers;
2. *Typing checking/inference*: the process by which each component of the program gets associated with a type, hence checking that operators and functions are applied in a type consistent way.

Typed Arithmetic Expressions

We exemplify the concepts discussed so far and introduce the basic concepts about type systems by looking at a minimal sub-language of boolean and arithmetic expressions. The language and its type system are nearly trivial, but precisely because of their simplicity they prove effective at illustrating the techniques and at exemplifying them them at work.

Syntax. The expressions of our simple language are defined by the following productions:

$$
\begin{array}{rcll}
a, b, c, \in \mathrm{Const} & ::= & m, n, p & \text{numeric literals } 0, 1, 2, \ldots \\
& | & \mathsf{true} \mid \mathsf{false} & \text{boolean literals} \\
M, N, P \in \mathrm{Exp} & ::= & a, b, c & \text{constants} \\
& | & M + N & \text{addition} \\
& | & M = N & \text{equality test} \\
& | & \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P & \text{conditional}
\end{array}
$$

> Note: the legal expressions may be defined equivalently by a system of inductive rules for $M \in \mathrm{Exp}$

Evaluation. As we have shown in our previous lecture, all the expressions of this simple language (with the exception of the equality test) can be encoded as λ-terms. However, for the purpose of demonstrating the import of types and type systems, we build the syntactic forms from primitive constants and operators.

Evaluating the expressions of the new calculus can be defined as in the lambda calculus, as a syntactic transformation. However, instead of the β-reduction used in the λ-calculus, here we introduce a few ad-hoc rules to simplify the arithmetic and boolean operators. The resulting reduction is still denoted $\rightarrow$ and defined by the following *δ-rules*.

Numeric Reductions

$$
\frac{(m \text{ and } n \text{ numeric literals})}{n + m \rightarrow n +_N m} \qquad \frac{M \rightarrow M'}{M + m \rightarrow M' + m} \qquad \frac{M \rightarrow M'}{m + M \rightarrow m + M'}
$$

The first rule defines the reduction for the numeric *redex*, i.e. an addition whose operands are numeric constant. The two additional rules define the reduction of complex expressions, by reducing the enclosed redexes, from left to right.

Boolean Reductions

$$
\frac{(a \text{ and } b \text{ different constants})}{a = b \rightarrow \mathsf{false}} \qquad \frac{}{M = M \rightarrow \mathsf{true}} \qquad \frac{M \rightarrow M'}{M = N \rightarrow M' = N} \qquad \frac{M \rightarrow M'}{N = M \rightarrow N = M'}
$$

$$\frac{}{\textsf{if true then } M \textsf{ else } N \to M} \qquad \frac{}{\textsf{if false then } M \textsf{ else } N \to N}$$

$$\frac{M \to M'}{\textsf{if } M \textsf{ then } N \textsf{ else } P \to \textsf{if } M' \textsf{ then } N \textsf{ else } P}$$

The rationale is the same as for the numeric operators. We have four axioms defining the reductions for the corresponding redexes, and one inductive rule to reduce a complex expression by reducing the enclosed redex.

Values, stuck terms and errors

Having introduced the constants and the operators as primitives with ad-hoc reduction rules brings about a new question related with reduction and normal form.

In the pure lambda calculus we have defined a normal form as an expression without redexes. In that case, given that all (closed) expressions could be thought of as belonging to the same category, that of functions, normal forms are also the natural candidate for a notion of value.

In our present case we could take the same approach and base our definition of value of the notion of normal form, which for the calculus of expressions can be defined as an expression without δ -redex.

However, we see immediately there are normal forms that don't exactly look like values. Take for instance the terms $4 + \textsf{true}$ and $\textsf{if}(5 \textsf{ then M else N}$: they both are normal forms, but they can hardly be considered values. Rather, the very much look like the representation of something that went wrong in the computation.

To distinguish normal forms that we may effectively classify as values from normal forms that we should take as stuck / error terms we need a refined notion of value . We make this discussion formal below.

Definition - Values and *Stuck terms*. The *values* of our calculus of arithmetic expressions are dthe numeric and boolean constants. A term that is not a value and that does not contain any redex is a *stuck term*.

Inherent in our interpretation of stuck terms as computational errors is the understanding that operators should be applied to their arguments with a certain discipline and that when that disciplined is not complied with we have an error.

Now, we would like to be able to tell that its evaluation will not get stuck without actually evaluating a term. To do that, we need to be able to distinguish between terms whose result will be a numeric value and terms whose result will be a boolean: that's the task of the type system.

We introduce two types, Nat and Bool, for classifying terms in this way. The metavariables S, T, U, etc. will be used throughout the book to range over types. Saying that "a term M has type T" means that M we anticipate - statically - that M evaluates to a value of the appropriate form.

The Typing Relation

The typing relation is formalized by means of typing judgements of the form $M : T$ (read M has type T), and defined inductively by the following system of rules:

$$\frac{}{\text{true} : \text{Bool}} \qquad \frac{}{\text{false} : \text{Bool}} \qquad \frac{}{n : \text{Nat}}$$

$$\frac{M : \text{Nat} \quad N : \text{Nat}}{M + N : \text{Nat}} \qquad \frac{M : T \quad N : T}{M = N : \text{Bool}} \qquad \frac{M : \text{Bool} \quad N : T \quad P : T}{\text{if } M \text{ then } N \text{ else } P : T}$$

The system is pretty straightforward and mostly self-explanatory.

- The first block of rules (in fact, axioms) assigns the type Bool to the boolean constants true and false and the type Nat to all the numerals.
- In the second block, the first rule assigns the type Nat to expressions that apply the $+$ operator to two Nat expressions, and the type Bool to the equality expression provided that the terms equated are expressions of the same type (either Nat or Bool or, when we get to calculi with more interesting sets of types, any other type): the two uses of the single metavariable T express the constraint that the the types of the sub-terms M and N be the same.
- Finally, the last rule assigns a type to a conditional expression based on the types of its subexpressions: the guard M must have type Bool, while N and P must both evaluate to values of the same type. Again, two uses of the metavariable T constrain the types of the two branches to coincide.

Type Safety: *well-typed terms don't go wrong*

The fundamental property of type systems is *safety* (also called soundness), which we summarize with the slogan *well-typed terms do not go wrong.*. We have already chosen how to formalize what it means for a term to go wrong: it means to reach a *stuck term*. What we want to know, then, is that well-typed terms do not get stuck. We show this in two steps, commonly known as the *progress* and *preservation* theorems.

Theorem [Progress] If $M : T$ then M is a value or there exists N such that $M \to N$.

Theorem [Preservation] If $M : T$ and $M \to N$ then $N : T$.

The Progress theorem tells us that well-typed terms are not stuck: either they are values or they can reduce to another term; the Preservation theorem, in turn, guarantees that if a well-typed term takes a reduction step, then the resulting term is also well typed.

The combination of type preservation and progress ensures that all terms whose evaluation reaches a stuck state a rule out by the type system as *ill-typed*. Notice, however, that this does not mean that an ill-typed term will necessarily lead to a stuck state: for instance, $\mathsf{if\ true\ then\ 0\ else\ false}$ is an ill-typed term, but its evaluation reduces to a value, which is either $\mathsf{0}$ or false.

Thus being well-typed constitutes a sufficient - but not necessary - condition for the absence of stuck states during the evaluation of well-typed terms. That is not surprising indeed, as the set of terms whose evaluation does not get stuck is a proper subset of terms: given that the latter is a recursively enumerable set, by Rice's Theorem, we know that none of its subsets is decidable (hence, there is no way to characterize them with exact precision).

The purpose of a sound type system is, therefore, is to enforce — i.e., satisfy the type preservation and progress theorems — without being too conservative, i.e. it should assign types to most of the programs we actually care about writing.

Proofs of the main type system properties

We conclude the presentation by examining its main properties. We start with two basic theorems tabout typing derivations.

Lemma [Inversion of The Typing Relation]

If $\mathsf{true} : T$ then $T = \mathsf{Bool}$.
If $\mathsf{false} : T$ then $T = \mathsf{Bool}$.
If $\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P : T$ then $M : \mathsf{Bool}$, $N : T$ and $P : T$.
If $n : T$ then $T = \mathsf{Nat}$.
If $M + N : T$ then $T = \mathsf{Nat}$ and $M : \mathsf{Nat}$ and $N : \mathsf{Nat}$.
If $M = N : T$ then $T = \mathsf{Bool}$ and $M : S$ and $N : S$ for some type S.

Proof: Immediate from an inspection of the typing rules.

The inversion lemma is sometimes called *generation lemma* for the typing relation: given a valid typing statement, it shows how a proof of this statement could have been generated.

In fact, the inversion lemma provides us with an algorithm for calculating the types of terms: for each term, the lemma tells us how to calculate its type (if it has one) from the types of its sub-terms.

Theorem [Uniqueness of Types]
For each term M there exists at most one type T such that $M : T$. Moreover, there is just one

derivation of the typing judgement from the inference rules.

Proof. By induction on the set of terms M, using the appropriate clause of the inversion lemma (plus the induction hypothesis) for each case. We look at the case for conditionals as a representative, taking $M = \mathsf{if}\ \hat{M}\ \mathsf{then}\ N\ \mathsf{else}\ P$.
If M has no type we are done. Otherwise, if $M : T$, the inversion lemma tells us that $\hat{M} : \mathsf{bool}$, $N : T$ and $P : T$. Moreover these types are unique. Then we conclude with an application of the rule for conditionals.

Lemma [Canonical Forms]

- If M is a value and $M : \mathsf{Bool}$ then M is either true or false;
- If M is a value and $M : \mathsf{Nat}$ then M is a numeric literal n.

Proof. Values can have three forms: true, false or a numeric literal. Then the proof follows directly from the uniqueness of types. In fact, $M : \mathsf{Bool}$ implies that M cannot have type Nat hence it must true or false, and the same applies dually when $M : \mathsf{Nat}$.

Proof of the Progress Theorem
We proceed by induction on the derivation of the typing judgement $M : T$. We proceed by cases on all the rules that may have derived the judgement in question, and assume (by rule induction) that the theorem holds for the judgements in the premises of the rule.

The cases of the axioms for the boolean and integer constants are immediate. since in all these cases M is a value. For the other cases, we take rule for conditionals as a representative: the remaining cases are similar.

If $M : T$ derives by the rule for conditionals, then M is of the form $\mathsf{if}\ M'\ \mathsf{then}\ N\ \mathsf{else}\ P : T$, derived from $M' : \mathsf{Bool}$. By the induction hypothesis we know that either M' is a value or else there is some term M'' such that $M' \to M''$. In the first case, by the canonical forms theorem M' is either true or false: hence either $M \to N$ or $M \to P$. If instead $M' \to M''$ then also $M \to \mathsf{if}\ M''\ \mathsf{then}\ N\ \mathsf{else}\ P$, as desired.

Proof of the Preservation Theorem
Wwe proceed by rule induction on the judgement $M : T$, reasoning by cases on the last rule applied in the derivation of the judgement. Again we show the case of conditionals as a representative for the other cases.

Assume M is the conditional expression $\mathsf{if}\ M'\ \mathsf{then}\ P\ \mathsf{else}\ Q$. Then, the judgement $M : T$ is derived from $M' : \mathsf{Bool}$, $P : T$ and $Q : T$. We have three possible subcases:

- $M \to N$ because $M' = \mathsf{true}$ and $N = P$.
  The proof follows directly from the hypothesis $P : T$.
- $M \to N$ because $M' = \mathsf{false}$ and $N = Q$.
  The proof follows directly from the hypothesis $Q : T$.

- $M \to N$ because $M' \to M''$ and $N = \mathsf{if}\ M''\ \mathsf{then}\ P\ \mathsf{else}\ Q$.
  From $M' : \mathsf{Bool}$ and $M' \to M''$, by the induction hypothesis we know that $M'' : \mathsf{Bool}$. From this and from $P : T$ and $Q : T$, we can apply the rule for conditional to derive $N : T$ as desired.

Exercises

1. Subject expansion? Having seen the subject reduction property, it is reasonable to wonder whether the opposite property -subject expansion-also holds. Is it always the case that, if $M \to N$ and $N : T$ then $M : T$? If so, prove it. If not, give a counterexample.

2. Typing of conditionals. As we argued above, the ill-typed terms of our expression language include terms like
$\mathsf{if\ true\ then\ 0\ else\ false}$ that actually behaves fine during evaluation. Show that if we change the typing of conditionals to make the previous term well-typed we can construct a well-typed term that gets stuck.

3. Typing of equality. In our typing rule for the equality term $M = N$ we require that M and N have the same type.

- How would you change the rule to relax that constraint (and admit the equality test on terms of different types)?
- Which of the properties of the original system would change with the new rule?
- In particular, would the new system still be sound? If so, prove it. Otherwise provide a counterexample.

CREDITS

The first section of this lecture is taken from Luca Cardelli and Peter Wegner's paper *On understanding types, data abstraction, and polymorphism.* Computing Surveys, 17(4):471–522, December 1985

The section on arithmetic expressions is adapted from Chapter 8 of Benjamin's Pierce's book on Types and Programming Languages. MIT Press. 2001.