3 Typing Expressions

Normalisation for \(\Downarrow\) does not hold!

**Theorem** (Normalisation - does not hold for \(L_1\)). For any expression \(e\), then there exists a value \(v\) such that \(e \Downarrow v\)

\[\forall e. \exists v. e \Downarrow v\]

**Counter Examples**

- \(5 \&\& \text{true} \Downarrow ?\)
- \(5 + (3 \text{-true}) \Downarrow ?\)

Try them out!

---

**Slide 58**

Recall the Normalisation Theorem for \(L_1\), discussed briefly in Sec. 1.1 and now restated on Slide 58. This (failed) Theorem stated that every expression evaluates down to a value. Slide 58 provides two witness expressions proving that this property does not hold, but there are infinitely many other examples.

A closer inspection of these counter-examples leads us to conclude that the property fails to hold only for expression that are somehow *ill-formed*. For instance the expression \(5 \&\& \text{true}\) does not make much sense because the operator \(\&\&\) is appropriately defined only over two boolean expressions; recall that in rule \(\text{EAND}\) on Slide 11, we used metavariables \(b_1\) and \(b_2\), denoting boolean values. We also had a similar condition for rule \(\text{RAND1}\) on Slide 18. Thus our semantics for \(L_1\) do not state how the \(\&\&\) operator is supposed to behave when one of the operands is a numeral (and not a boolean). Similarly, the expression \(5 + (3 \text{-true})\) does not make sense because, in turn, the sub-expression \(3 \text{-true}\) does not make sense since “\(-\)” is defined only for numerals.

**Stuck Expressions**

**Definition 7** (Terminal Expressions). \(e \mapsto^* \overline{e}^\prime\). \(e \rightarrow e^\prime\)

**Definition 8** (Stuck Expressions). \((\text{stuck } e) \triangleq e \mapsto^* \text{ and } e \neq v\)

---

**Slide 59**

From a semantic perspective, if we (reasonably) expect that expressions reduce to values, then these ill-formed expressions can also be termed as *ill-behaved* since they violate this property. Conveniently, the reduction semantics for \(L_1\) provides us with a precise and intuitive way how to characterise these ill-behaved expressions. On Slide 59 we refer to these expressions as *stuck*, making use of the terminal expression definition of Slide 55. Thus, the predicate \((\text{stuck } e)\) states that a stuck expression is a terminal expression that is *not* a value.

In our semantics, stuck expressions denote expressions that do not reduce further. In an implementation faithful to our semantics, stuck expressions potentially denote expressions whose reduction is undefined, and
Runtime Errors

\[
\begin{align*}
&c_1 \Downarrow n \quad e_{Err1} \\
&c_1 \& c_2 \Downarrow \text{err} \\
&c_2 \Downarrow n \quad e_{Err2}
\end{align*}
\]

Examples of possible runtime errors:

- accessing data past the program allocated memory
- violate some security restriction.
- ...

Slide 60

may thus result in a runtime-error. In fact we could have chosen a more suggestive notation and defined a semantics where such expressions results in an actual runtime-error as shown by some example rules on Slide 60. Typical runtime errors in practice include segmentation faults and illegal instructions that violate some security policy. In what follows, we will not pursue such a definition of runtime-error further, but at the same time interchangeably refer to stuck expressions as runtime errors.

There are various ways how to deal with runtime errors. The method we shall investigate in this course is also the method adopted by most mainstream programming languages. In particular, we shall be concerned with defining mechanisms for analysing expressions without evaluating/reducing them and, through this analysis, determine which expressions are well-behaved and which are not. Stated otherwise, our analysis will enable us to filter out the ill-behaved expressions and, by not executing them, avert runtime errors during a program’s execution. Also, such an analysis allows us to limit the study of certain properties, such as the Normalisation Theorem on Slide 58, to the subset of well-behaved expressions.

In programming languages, such analysis is often carried out by a type system. A type system acts as a predicate over programs: it normally takes a program and returns a yes/no answer, depending on whether the analysis succeeds or not. If the analysis succeeds, then the type system determines that the program is well-typed; otherwise it determines that the program is ill-typed. A well-typed program should imply that it is also well-formed. Note however that the opposite need not be the case: we can have well-formed (or well-behaved) programs that are not well-typed.

Example 9. Consider the alternate (short-circuiting) evaluation semantics of Slide 16. There, the expression below does evaluate to a value:

\[ \text{true} \& \text{5} \Downarrow \text{true} \]

It may be the case that our type system determines that \text{true} \& \text{5} is ill-typed and thus rejects the example, even though it is well-behaved.

A more important aspect of type-systems is decidability, i.e., we do not mind (that much) if our analysis is at times imprecise and rejects well-formed/behaved programs, but we care more that our analysis is usable in practice and always returns an answer (a yes or a no). Another aspect of type systems is the complexity of the analysis: typically, we would not want our type analysis to have exponential complexity, as it would run the risk of taking a very long time before it returns an answer.

These days, type systems form an integral part of many mainstream programming languages. As a result, their use extends further from simply filtering out ill-behaved programs as outlined on Slide 62. For example, in Java, the type system (i.e., classes, interfaces, packages etc.) plays a central role in structuring a program written in that language. A type system can also influence language design; for instance, we may chose to
Type System

- acts as a predicate over programs
- has to have a decidable algorithm
- ... preferably not with exponential complexity.

allow certain programs to be written in our language (i.e., the language BNF), with the knowledge that we can then filter out illegal programs though our type system - this often leads to cleaner syntax descriptions of a language. Finally, and perhaps in conflict with the picture we presented thus far, types may sometimes also affect the dynamic semantics of certain languages. A case in point is the language C++. Consider the case where a class C2 (directly) extends some other class C1, overriding some method m() declared in C1 in the process. Then, the following code

```c
C1 x = new C2(); // create a new instance of C2, but declaring it as a C1 object
x.m(); // invoke method m()
```

exhibits a behaviour that is dependent on the types(classes) assigned. In particular, the method dispatched in the command x.m() is the one defined in the superclass C1, even though the object held at x is an instance of class C2. The rationale here is that in C++, runtime performance is of utmost importance and the language designers wanted to shift as much computation as possible to the compilation phase. Thus, what method to dispatch is determined statically, i.e., at compile time, and the type assigned to variable x is the criteria used by the compiler to determine this dispatch.

Type systems are broadly categorised as either static or dynamic. Typically static type systems analyse a program before it starts executing. On the other hand, in a dynamic type system, the analysis is woven in during the execution of a program.

There are a number of issues that one needs to consider when deciding whether to adopt a static or dynamic type system for a particular language. For example, static type-checking needs to be performed only once, typically before or during the compilation of the program. On the other hand, dynamic type-checking is performed every time a program is run and it is hard to tease apart the analysis from the program execution. This has a number of undesirable effects:

- In dynamic type-checking, if we run the same program twice, we perform the same number of checks twice. In static type-checking, these checks can be performed only once.

1Classes are not the same thing as types, but consider it to be the case for the sake of this example.
Roles of a Type System

- Preventing certain kinds of errors
- aid in structuring programs
- guiding language design.
- affect the runtime semantics of the typed program itself.

Static Versus Dynamic Type Systems

We have two places where we can perform these checks:

**Static**: all at once before computing the program.

**Dynamic**: bit by bit, as we compute the program.

Characteristics:

- Static type-checking does not affect the runtime performance of a program.
- Dynamic type-checking can be more precise.

Since dynamic checks are carried out during the execution of a program, dynamic type-checking affects the runtime performance of a program. Static type-checking need not affect the runtime performance of a program.

Dynamic type-checking does carry some advantages over static type-checking. For instance, it is is often the case that dynamic type-checking is more precise than static type-checking because some errors are hard to detect before a program executes i.e., the execution of the program thus far may provide additional information that can be used during type checks. Moreover, the amount of checks that are performed by a dynamic type-checker per program execution may end up being less that the amount of static checks performed for the same program execution, because the runtime checks are only performed for the current branch of execution — stated otherwise, a static type checker in general has no prior information which branch of execution will be taken, and as a result has to check them all.

Example 10. If one wants to check that an array `arr` of length 10 never gets accessed past its 10th cell then there is no general (decidable) way how to check that the expression `arr[e]`, for some complex expression `e`, does not violate this condition without executing the program and evaluating `e`.

Thus, such checks tend to be outside the reach of static type-checking but, conceivably, such a check can be introduced at runtime after `e` is evaluated. Because of these reasons, most programming languages nowadays prefer to shift as much type-checking as possible to the compile-time phase, and leave only a limited number of checks to the runtime phase.

Having said this, the recent popularity of rapid development languages such as Ruby and Python, together
with recent advances in verification, have reversed this tide lately, and we are seeing again a keen interest in dynamic typing by the programming languages community. Indeed it would be fair to say that in future we should be seeing more commercial programming languages where this clear-cut distinction will become more blurred. For this course however, we shall be focussing on static type systems.

**Type Systems: How does it work?**

We define *approximations* for the behaviour of a program, by identifying a *type* for every kind of behaviour we wish to approximate.

\[ t \in \text{Typ} ::= \text{bool} \mid \text{int} \]

We then define a relation associating programs to these types (i.e., approximate behaviour).

\[ \vdash : \text{Exp} \times \text{Typ} \]

So how does a type system work? Its mechanics are based on the notion of a *type*, usually denoted by the metavariable \( t \), and in general terms, it attempts to *approximate* the behaviour of a (sub)program. The behaviour of (well-behaved) programs in \( L_1 \) can be characterised by the values they reduce to; an approximation to these values is given through the type definition on Slide 64. It is useful to think of types as (named) sets.

A typing relation, denoted by the symbol \( \vdash \) on Slide 64, defines the members of this set. More specifically for \( L_1 \), a definition for the relation \( \vdash \) would associate expressions to these types, according to the value the expression is expected to return. Note that since the set \( \text{Exp} \) is infinite (but finitely described), our definition populating \( \vdash \) must be inductively defined.

There are many possible value approximations that one can take for \( L_1 \). For instance, we could have defined a more refined type definition, such as:

\[ t \in \text{Typ} ::= \text{bool} \mid \text{BigInt} \mid \text{SmallInt} \]

But we have to be careful when we do so: important aspects of type systems are decidability and complexity, and behaviour approximations that are too fine may have adverse effects on these aspects. For instance, to assign expressions to the refined type definition above, we would need to somehow approximate whether a value returned by an expression is a big integer or a small integer, which may not be decidable in general.

The type judgement \( \vdash : \text{Exp} \times \text{Typ} \) is the least relation satisfying the rules on Slide 65. For instance,

- **rule tNum** states that all numeral expressions are of type \( \text{int} \).
- **rule tLEq** states that, assuming that \( e_1 \) is of type \( \text{int} \) and also that \( e_2 \) is of type \( \text{int} \) as well, then the expression \( e_1 \leq e_2 \) is of type \( \text{bool} \).

Despite the apparent resemblance, note the difference between the evaluation rules on Slides 10 and 11 and the typing rules of Slide 65. The latter rules are not concerned with computing a particular value, but instead determine an approximation for this value (e.g., instead of returning a particular value, say 5, for a particular expression, the type system returns \( \text{int} \), which approximates the value 5). In fact, compared to the evaluation rules, the typing rules do not have any side conditions (which is where computation occured on Slides 10 and 11).
As we saw before with inductively defined sets, we can use the typing relation as a proof system, to prove that a particular expression has a particular type. This process is often referred to as type-checking or type-inferencing, depending largely on how the typing question is phrased. Loosely speaking, type-checking refers to the case when we are given all the necessary facts and we have to find a proof justifying those facts. For example, in our case, a type-checking question would be

"Does the expression $(1 \cdot (2 + 3)) \leq 7$ have type `bool`?"

On the other hand, type-inferencing refers to the case when a subset of the facts are provided, and using these limited facts and the typing rules, we have to infer the remainder of the facts. Type-inferencing often requires some degree of guessing in the derivation, which is why it usually increases the computational complexity of typing. For example, in our case, a type-inference question would be

"What type does the expression $(1 \cdot (2 + 3)) \leq 7$ have?"

Interestingly for both these questions, the answer is given by the derivation on Slide 66. For the first question, the derivation justifies the claim that $(1 \cdot (2 + 3)) \leq 7$ has type `bool`, whereas for the second question, the derivation justifies the choice (inference) of `bool` for the expression $(1 \cdot (2 + 3)) \leq 7$.

Note also that for this specific case, the complexity of type-checking and type-inference is the same. Since we happen to have a single typing rule for every expression construct, we can use the expression to guide our search for the type-derivation (proof). We often refer to such systems as being syntax directed i.e., through the form of the expression, we know exactly which typing rule to apply. For instance, if we have an expression of the form $e_1 \cdot e_2$, we know that we can only apply rule $T\text{Add}$ to type-check it. This however may not necessarily be the case for other languages and type-systems.

A syntax directed type system implies pleasing properties such as decidability wrt. to type checking and type inference. Decidability for a particular problem is usually satisfied if one can exhibit an algorithm that can decide that problem. Syntax directedness essentially means that the rules in Slide 65 define an algorithm for determining type checking and type inference. More precisely, if we have an expression of the form $e_1 \cdot e_2$, our algorithm would pattern match the structure of the expression with a typing rule matching that structure and if none are found, we abort i.e., return a negative answer. If a match is found, the rule
A proof that \( (1 + (2+3)) \leq 7 : \text{bool} \)

\[
\begin{array}{c}
\vdash 1 : \text{int} \\
\vdash 2 : \text{int} \\
\vdash 3 : \text{int} \\
\vdash (1 + (2+3)) : \text{int} \\
\vdash 7 : \text{int} \\
\vdash (1 + (2+3)) \leq 7 : \text{bool}
\end{array}
\]

Decidability of Type Checking and Type Inference

Theorem 11. Given \( e \) and \( t \) one can decide whether \( \vdash e : t \)

Proof. Follows from syntax directed rules the Type System. \( \square \)

Theorem 12. Given \( e \) one can decide whether for some \( t \) we have \( \vdash e : t \)

Proof. Follows from syntax directed rules the Type System. \( \square \)

itself would tell us whether to stop the algorithm (in the case of an axiom) or else give us new subgoals to pattern match (in the case of an inductive rule). Termination of this algorithm is guaranteed by the fact that the new subgoals would be getting smaller than the original typing goal and thus we would be getting closer to hitting an axiom. Crucially though, this process of pattern matching is necessarily deterministic as there is at most one rule that matches the expression, which means that this algorithm is fully specified; if, on the other hand, we had more than one rule that could be applied, we would need to give more detail in order to specify the algorithm i.e., which rule to choose. In our case, syntax directedness means that we are able to show Theorem 11 and Theorem 12 stated on Slide 67.

Note that the type of the expression \( (1 + (2+3)) \leq 7 \) is \text{bool}, which happens to correctly approximate the value the expression evaluates to, i.e., \text{true} (cf. derivation on Slide 13). Interestingly, the two ill-behaved expressions we saw earlier do not type-check (for any type), as outlined on Slide 68. These cases already give us a sense of correspondence between well-typed expressions (i.e., those that type-check for some type) and well-behaved expressions (i.e., those that reduce to a value of some sort). More specifically we seem to have the following inclusion:

\[ \text{the set of well-typed expressions} \subseteq \text{the set of well-behaved expressions}. \] (12)

We need to prove this property to make absolutely sure that it does indeed hold. Luckily we can reasonably hope to do this because everything thus far has been formally defined.

Example 13. The type system defined in Slide 65 rejects the program \text{true \&\& 5} discussed earlier in Example 9. This is because we cannot derive \( \vdash \text{true \&\& 5} : t \) for any \( t \in \{\text{int}, \text{bool}\} \).

Before we embark on the task of showing that the type system only accepts well-behaved programs, let us prove another property of the type system that will serve as a warm up exercise for the main proof in this
Expressions that fail to type-check

\[
\begin{align*}
\vdash 5 : \text{TNum} & \quad \vdash \text{true} : \text{TBool} \\
\vdash 5 \&\& \text{true} : ? \\
\vdash 5 \vdash 3 \text{int} & \quad \vdash \text{true} : \text{TBool} \\
\vdash 3 - \text{true} : ? \\
\vdash 5 + (3 - \text{true}) : ?
\end{align*}
\]

Slide 68

section. Slide 69 formalises the Uniqueness of Typing property, which states that for any expression we can at most assign one type in our type system. The proof is by induction on the structure of \( e \). Note however that we never need to use the inductive hypothesis in this proof because it turns out that the type assigned is never dependent on the type assigned to the premises in the type derivation. In future however we will encounter cases whereby this is not the case.

Let us now turn our attention back to proving the property outlined in (12). Earlier, in Definition 8 of Slide 59, we formalised what it means for an expression to be ill-behaved in terms of the reduction semantics of Section 1 i.e., it gets stuck before it reaches a value. Dually we can define what it means for an expression to be presently well-behaved, i.e., an expression that is not stuck.

\[
e \text{is well-behaved} \overset{\text{def}}{=} e = v \lor \exists e'. e \rightarrow e'
\]

Notice that, by itself, this property is not strong enough to describe well-behaved expressions.

**Example 16.** The expression \((4+2)+\text{true}\) satisfies property (13) since

\[
(4+2)+\text{true} \rightarrow 6+\text{true}.
\]

However this expression is ill-behaved because it does not reduce to a value. In fact we have \(6+\text{true} \not\rightarrow\).

Although this property is not strong enough to characterise well-behaved expressions by itself, it will suffice when used in conjunction with another property we present later on. We refer to this property as Progress. Theorem 17 in Slide 71 states that expressions that are typeable in our type system observe this property; we prove this theorem by induction on the typing derivation. Slide 16 only considers one inductive rule, namely \(\text{tLeq}\). For the proof to be complete however, we have to consider the other 2 axioms and 4 inductive rules. This is left as an exercise to the reader.

The second property we consider that leads to our notion of soundness of the type system, (12), is called (type) Preservation. Intuitively this states that when an expression typechecks and this expression reduces to some other expression, the expression it reduces to also typechecks. In the case of our type system, we can actually show a stronger result: when an expression typechecks at a particular type \( t \), then the expression it reduces to must typecheck at that same exact type. In general, however, this can be too restrictive; in some type systems, the type at which the reduced-to-expression is typechecked may become more informative i.e., more precise - as is often the case in systems with subtyping.
Uniqueness of Typing

**Theorem 14.** \((\text{For all } e) \vdash e : t_1 \text{ and } \vdash e : t_2 \text{ implies } t_1 = t_2\)

**Proof.** By induction on the structure of \(e\). We here show one representative base case and one representative inductive case:

**Case** \(e = n\): The only typing rule that we could apply is \(\text{tNum}\), hence \(t_1 = \text{int} = t_2\).

**Case** \(e = e_1 <= e_2\): \(\vdash e_1 <= e_2 : t_1\) could have only been derived using \(\text{tLeq}\) and similarly for \(\vdash e_1 <= e_2 : t_2\), which implies \(t_1 = \text{bool} = t_2\).
Theorem 17. \( \vdash e : t \) implies \( e = v \) or \( \exists e'. e \rightarrow e' \)

Proof. By induction on the derivation of \( \vdash e : t \). We here outline the case for rule TLeq. We know that \( e = e_1 \leq e_2, t = \text{bool} \) and that

\[
\begin{align*}
\vdash e_1 : \text{int} & \quad (14) \\
\vdash e_2 : \text{int} & \quad (15)
\end{align*}
\]

By (14) and I.H. we have two subcases:

\( \exists e'_1, e_1 \rightarrow e'_1 \): Result follows from rLeq2 of Slide 18.

\( e_1 = v_1 \): By (14) and Lemma 15 \( v_1 = n_1 \). By (15) and I.H. we have two subcases:

\( \exists e'_2, e_2 \rightarrow e'_2 \): Result follows from rLeq3 of Slide 18.

\( e_2 = v_2 \): By (14) and Lemma 15 \( v_2 = n_2 \) and result follows by rLeq1 of Slide 18.

3.1 Exercises

1. Complete the proof for Theorem 14.
2. Complete the proof for Theorem 17.
3. Complete the proof for Theorem 18.
4. Complete the proof for Theorem 19.
5. Define (inductively) the function \( \text{op}(e) \) which returns the number of operations used in an expression. For instance, \( \text{op}(5) \) should return 0, \( \text{op}(5 \cdot 1) = 1 \) and \( \text{op}(\neg \text{true}) = 2 \).

(a) Prove that \( \text{op}(e) = 0 \) implies \( e = v \).
(b) Prove that \( e_1 \rightarrow e_2 \) implies \( \text{op}(e_2) < \text{op}(e_1) \).

6. Using the result of the previous two questions prove that every expression in \( L_1 \) is terminating.
Preservation

Theorem 18. ⊢ e : t and e → e' implies ⊢ e' : t

Proof. By induction on the derivation of e → e'.

rLeq1: We know

\[ e = n_1 \leq n_2 \quad (16) \]
\[ e' = b \text{ where } b = n_1 \leq n_2 \quad (17) \]

By (16) and TLEQ we know \( t = \text{bool} \) and by (17) and Lemma 15 we have \( \vdash b : \text{bool} \).

rLeq2: We know

\[ e = e_1 \leq e_2 \quad (18) \]
\[ e' = e_3 \leq e_2 \quad (19) \]
\[ e_1 \rightarrow e_3 \quad (20) \]

By (18) and TLEQ we know

\[ \vdash e_1 : \text{int} \quad (21) \]
\[ \vdash e_2 : \text{int} \quad (22) \]
\[ t = \text{bool} \quad (23) \]

and by (21), (20) and I.H. we deduce \( \vdash e_3 : \text{int} \). Result follows by (19)(22), (23) and TLEQ.

rLeq3: Analogous to previous case.

Safety (Soundness)

Theorem 19. ⊢ e : t and e →* e' implies ¬(stuck(e'))

Proof. A formal proof is by induction on \( n \) in \( e \rightarrow^* e' \).

\( n = 0 \): Immediate by Theorem 17 (Progress).

\( n = k + 1 \): Left as an exercise. (Hint, it follows if we can show that \( \vdash e' : t \))