Advanced Topics in Programming Languages **Spring Semester, 2012**

Lecture 1: March 9, 2012

Lecturer: Mooly Sagiv *Scribe: Ghila Castelnuovo and Omer Tripp*

1.1 Course Themes

This course is concerned with the basic concepts that appear in modern programming languages. It aims to develop tools for understanding

1. the design choices governing a programming language;
2. how programming languages compare and relate to each other at the *conceptual* level (and not at the syntactic level); and
3. how ideas and concepts in programming languages have evolved historically.

A primary goal of this course is to prepare students for research in programming languages. The concepts and materials presented throughout the semester should allow students to read and understand theoretical papers published in leading conferences in this area, such as POPL and ICFP.

1.1.1 Language Goals and Tradeoffs

An important theme is the tradeoff the programming language embodies between *expressiveness* and *efficiency*: There are many situations where it is convenient to assign the programming language the responsibility for doing something automatically, but this often comes at a cost. For example, the Lisp run-time system, as well as the Java virtual machine, use garbage collection to detect and reclaim memory locations that are no longer in use. This facilitates writing programs, and makes programming less prone to errors. In some cases, however, automatic garbage collection may result in slowdown and/or lack of predictability in program execution, which is problematic for real-time programming.

Interestingly, most of the programming languages that are considered successful and popular were originally developed for a specific purpose. A notable example is the C programming language, which was designed and implemented from 1969 to 1973 by Dennis Ritchie in Bell Laboratories as part of the Unix operating system. Significant changes in C occurred from 1977 to 1979. These changes were motivated by the push to achieve portability of the Unix system.

The important connection between C and the Unix operating system is a key reason for the success of C: First, when programs are written to run under Unix, all of the basic system calls are immediately available in C. It is therefore easier to write many Unix applications in C than in other programming languages. Another related reason for the popularity of C is that it has a distinctive memory model that is close to the underlying hardware.

Another example is JavaScript, which was originally developed by Brendan Eich of Netscape under the name Mocha, later renamed to LiveScript, and finally to JavaScript (solely for commercial purposes, to ride on the popularity of Java). JavaScript has become one of the most popular programming languages on the web. Initially, however, many professional programmers denigrated this language because its target audience was web authors. The advent of AJAX returned JavaScript to the spotlight and brought more professional programming attention. The result was a proliferation of comprehensive frameworks and libraries, improved JavaScript programming practices, and increased usage of JavaScript outside of web browsers, as evidenced by the proliferation of server-side JavaScript platforms.

In general, the success of a programming language is influenced by various factors, including:

Motivating Application The language is designed so that programs belonging in a specific application domain can (mostly) be written more easily (as exemplified above).

Execution Model The language design is specific about how all basic operations are done. For example, Fortran prescribes a flat register machine with no stacks and no recursion. A systematic, predictable machine model enables developing robust and efficient compilers and run-time environments for the language, and these are critical for its success.

Tooling Diagnostic tools, testing infrastructure and other libraries that facilitate writing and maintaining code in the programming language—thereby making the programmer more productive—are key to the success of the language.

1.1.2 Adoption of Futuristic Ideas

There are a number of examples of how “futuristic” concepts, originally conceived to solve specific problems, influenced the design of programming languages (typically much faster than their doubters expected). Recursion is one such example. In the mid-1970s, all “serious” programmers used Fortran, which (as mentioned above) did not allow recursion. Recursion was generally regarded as too inefficient to be of practical value for “real programming”. Today, however, there is wide consensus on the utility of recursion in production code.

A similar example of a futuristic idea is object-oriented programming, which in the 1980s many people considered too inefficient and clumsy for real programming. This changed in the 1990s, as object-oriented programming became more widely accepted and used.

Yet another example is higher-order functions. Introduced in Lisp in the 1960s, they have existed for decades in functional languages, but gained wide attention only in recent years. One reason for this is the rising interest in parallel programming for multi-core architectures (e.g., Google's MapReduce technology) and lock-free synchronization techniques. Pizza and Scala are two (relatively) recent extensions of the Java language (Scala succeeding Pizza and being heavily influenced by it) where higher-order functions are supported (using lightweight syntax). Below is an example in Scala syntax, where function `apply` takes another function `f` and a value `v` and applies `f` to `v`:

```
def apply(f: Int => String, v: Int) = f(v)
```

A related example is garbage collection [3, 7], which was invented by John McCarthy around 1959 to solve problems in Lisp. For years, garbage collection was considered infeasible for imperative programming languages. The growing interest in object orientation, where a key principle is the encapsulation of abstractions into objects that communicate through clearly defined interfaces, changed this: Programmer-controlled storage management inhibits the modularity prescribed by object orientation. For this reason, most modern object-oriented languages, such as Smalltalk, Eiffel, Java, C# and Dylan, are supported by garbage collection. Today, even languages used in part for systems programming, such as Modula-3 and Oberon, provide garbage collection for these sound but pragmatic reasons. Garbage-collecting libraries are even available for such uncooperative languages as C and C++.

1.1.3 New Trends in Programming Languages

The landscape of programming languages is quite dynamic. In recent years, there are new trends in several frontiers:

Commercial Type-safe languages, such as Java and C#, are gaining increasing momentum. One indication of this is a recent study of the popularity of programming languages based on data from Powell's Books [1], according to which three of the four highest-ranking languages are Java, Visual Basic and C#.

Interestingly, there is also evidence for the increasing popularity of languages for web programming, and in particular scripting languages such as PHP and JavaScript. The same study found, based on Craigslist, that three of the five languages appearing most frequently in job listings are PHP, JavaScript and SQL.

Teaching In university courses, Java took the place of C as the introductory language for students. This is because Java does not require the developer to manage the data representation in the memory and thus it is an easier language to learn and also due to the fact that Java has available free and friendly development environments (most notably, Eclipse)

Research In the research frontier, there have been several recent advances:

- **Modularity** New module features have been standardized recently. Most importantly, templates and STL (in C++) and generics (in Java, Scala and C#) improve the ability to write library code that is both (a) general and (b) amenable to specialization and (c) verifiable and (d) efficient. This is an improvement compared to standard OOP, since template-/generics-based code enables more optimizations, and restricts the behaviour of the library such that correctness of the client can be proved more easily.

  Despite the similar goals of generics and templates, there are several notable differences [4]. C++ templates use a compile-time model. When a template is used in a C++ program, the effect is as if a sophisticated macro processor had been used, and—after compilation—objects of a class with a different type parameter are different types at runtime. Generics, on the other hand, are not just a feature of the compiler, but also a feature of the runtime. A generic type, such as `List<T>`, maintains its specialized type after it has been compiled. In this way, the abstraction on the different type parameters is not lost after compilation.

  Another example of module features is Scala traits, which are similar to interfaces in Java, but permit the definition of default implementations for some methods.

- **Program Analysis** Nowadays there is more extensive use of automated tools for error detection. Some of these algorithms are integrated into the development environment (IDE) and/or into the build process.

- **Isolation and Security** Interest in language-based security is fast increasing. This is largely because of the security needs of web and mobile programming.

- **Web 2.0** Recent advances in web technology—most notably, Web 2.0—favor shifting responsibility to the client side of web applications.

1.2 Type Systems

Programming languages are designed to help programmers organize computational constructs and use them correctly. For this reason, many programming languages organize data and computations into collections called "types". In general, a *type* is a collection of computational entities that share some common property. Examples include the type `int` of integers and the type `int` $\rightarrow$ `int` of functions from integers to integers. More interesting examples are the Pascal subrange type [0..100] of integers in the range $1 - -100$ and the `int channel` type in concurrent ML denoting communication channels carrying integer values.

Pierce provides the following definition for a type system ([8], p.1):

A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

This form of verification is crucial, as the following example demonstrates: On Tuesday, June 4, 1996, four European Space Agency spacecraft were launched on the maiden flight of the Ariane 5 rocket. The launch ended in failure due to an error in the software design caused by inadequate protection from integer overflow. This resulted in the rocket veering off its flight path 37 seconds after launch, beginning to disintegrate under high aerodynamic forces, and finally self-destructing by its automated flight termination system. The failure has become known as one of the most infamous software bugs in history, the result being a loss of more than 370 million dollars.

The Ariane 5 software reused the specifications from the Ariane 4, but the Ariane 5's flight path was considerably different and beyond the range for which the reused computer program had been designed. Because of the different flight path, a data conversion from a 64-bit floating point to 16-bit signed integer value caused a hardware exception (more specifically, an arithmetic overflow, as the floating point number had a value too large to be represented by a 16-bit signed integer). Efficiency considerations had led to the disabling of the software handler (in Ada code) for this error trap, although other conversions of comparable variables in the code remained protected. This caused a cascade of problems, culminating in destruction of the entire flight.

The source code in Ada that caused the overflow is the following line, where the conversion from 64 bits to 16 bits unsigned is not protected:

```
P_M_DERIVE(T_ALG.E_BH) :=
UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M_LSB_BH) * G_M_INFO_DERIVE(T_ALG.E_BH)));
```

A tighter type system, where implicit conversions of this kind are prohibited, would have prevented this terrible accident from occurring.

1.2.1 Main Uses of Types

There are several important uses for type systems:

Error Detection The most obvious benefit of static type checking is that it enables early detection of errors. In practice, static typechecking exposes a surprisingly broad range of errors. The strength of this effect depends on the expressiveness of the type system, as well as on the programming task in question: Programs manipulating a variety of data structures (e.g., a compiler) offer more purchase for the typechecker than programs involving just a few simple types (e.g., scientific applications, though even here refined type systems support dimension analysis).

Abstraction Type systems further enforce disciplined programming. In particular, for large-scale software composition, type systems form the backbone of the *module language* used to package and tie together the components of large systems. Types show up in the interfaces of modules. Indeed, an interface can be viewed as "the type of a module". Structuring large systems in terms of modules with clear interfaces leads to a more abstract style of design.

Documentation Types are useful when *reading* programs. The type declarations in procedure headers and module interfaces constitute a form of documentation giving useful hints about behavior. Importantly, unlike descriptions embedded in comments, this form of documentation cannot become outdated.

Maintenance For certain kinds of programs, a typechecker can be of great value as a *maintenance* tool. For example, a programmer who wishes to change the definition of a complex data structure need not search by hand to find all places within a large program where code involving this structure should be modified. Once the definition of the datatype changes, the developer can simply run the compiler and examine the locations where typechecking has failed.

A more advanced way of utilizing the type system is by building specialized productivity tools on top of it. An example of this is the Jungloid [6] tool, which answers queries such as "What code takes an `IEditorPart` as input and returns an `IDocumentationProvider` object that represents the data contained in the editor?" (formally phrased as (`IEditorPart`; `IDocumentationProvider`)). The solution is a "path" connecting between the types in the query where path edges are due to field accesses, method invocations (both static and instance methods), conversions, etc.

Language Safety Informally, "safe" languages are languages that make it impossible for the developer to shoot herself in the foot while programming. A safe language *protects its own abstractions.* That is, the language ensures the integrity of abstractions it provides of machine services, as well as of higher-order abstractions introduced by the programmer based on the definitional facilities of the language. For example, a language may provide arrays—with access and update operations—as an abstraction of the underlying memory. A programmer using the language then expects than an array can be changed only by using the update operation, and not, for example, by writing past the end of some other data structure. Similarly, it should hold that lexically scoped variables can only be accessed from within their scope. In a safe language, such abstractions can be used *abstractly*, i.e., without keeping in mind all sorts of low-level details such as the layout of data structures in memory.

Note that language safety is not the same as static type safety: It can be achieved via static typechecking, but also by run-time checks that trap nonsensical operations and raise an exception. For example, Scheme is a safe language, though it has no static type system. The idea behind dynamic typing is to associate types with *values* but not *variables*. Thus, a variable can refer to a value of any type, but safety can still be enforced at run-time.

Here are some examples of statically versus dynamically checked languages:

Statically Checked	Dynamically Checked
ML, Haskell, Java	Lisp, Scheme, Perl, Postscript

Following are some code examples demonstrating the two categories above, as well as the case of unsafe languages (e.g., C and C++):

Category	Code	Consequence
Static checking (Java)	`String one = 1;`	Compile-time error

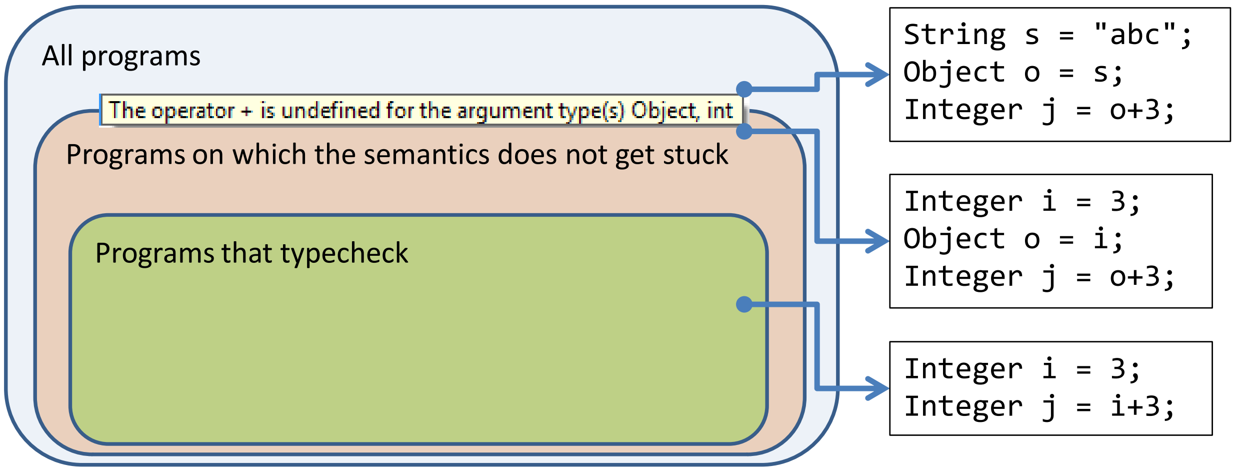
Dynamic checking (Python)	`>>> foo = ‘‘x’’`
	`>>> foo = foo+2`
		Run-time error

Unsafe (C)	`printf(‘‘%s’’, 12)`	Undefined behavior

Static and dynamic typechecking each have their advantages and disadvantages, and there is no clear answer which alternative is better. Static typechecking enables finding bugs at compile time, thereby increasing the reliability of the program. Static checking also allows the compiler to prove properties of the given program, which may enable compile-time optimizations. Finally, typechecked code is usually amenable to more efficient execution because there is less need for run-time checks. On the other hand, static type systems are restrictive, and sometimes reject a correct program. It is also more burdensome to prototype in a statically-typed language. Last, just-in-time compilation favours dynamic typing, because this facilitates the compiler's work, thereby enabling faster compilation.

From a formal standpoint, the meaning of type safety is that if a program typechecks, then the semantics will not get stuck when executing it. Figure 1.1 illustrates this using three examples in Java syntax. The upper code snippet shows an “illegal” program. The middle snippet shows a legal program that does not typecheck. In both cases, the compiler issues the error enclosed in a yellow rectangle in Figure 1.1. Finally, the bottom snippet shows a variant that typechecks.

The below table characterizes the type safety of several common programming languages:


All programs
The operator + is undefined for the argument type(s) Object, int
Programs on which the semantics does not get stuck
Programs that typecheck

```
String s = "abc";
Object o = s;
Integer j = o+3;
```

```
Integer i = 3;
Object o = i;
Integer j = o+3;
```

```
Integer i = 3;
Integer j = i+3;
```

Figure 1.1: Illustration of the Formal Meaning of Type Safety (in Java)

Safety	**Example Languages**	**Explanation**
Not safe	C and C++	Type casts, pointer arithmetic, , explicit deallocation, dangling pointers, unions
Almost safe	Pascal	Explicit deallocation, dangling pointers, variant records
Safe	Lisp, ML, Smalltalk, Java	Complete type checking

Pascal is considered "mostly safe", because dangling pointers (where a pointer may be deallocated by the programmer, thus pointing to a location not allocated to the program) and variant records (a data structure used to hold a value that could take on several different yet fixed types) are the only two violations of type safety.

C is considered "unsafe": It inherited the safety violations of Pascal, but added more of its own, such as pointer arithmetic, type casts and unions instead of variant records. Unions in C are less safe than variant records in that a variant record is tagged by a "tagfield ", which explicitly indicates the type of the value currently stored in the record. This enables, to some extent, compile-time checking that the record is manipulated correctly. C does not support the "tagfield" construct, and thus the compiler cannot apply any type checking to code using unions. An example of a type-safety violation in C unions is the following code:

```
int main(void) {
 union {
   int i;
   double d;
 } z;

 z.d = 42.7;
 printf(``%d\n'', z.i); /* Oops */
 return 0;
}
```

Optimization Type information in programs can be used for many kinds of optimizations. One example is finding components of C structs. (The component-finding problem also arises in object-oriented languages.) A struct consists of a set of entries of different types. Upon encountering an expression $s.e$ where a entry e of struct instance s is accessed, the compiler must generate machine code that, given the location of s in memory, finds the location of field e. If the compiler can compute the type of the struct at compile time, then the type information can be used to generate efficient code.

Another example is boxing in languages supporting polymorphism. For a polymorphic function to work properly, all its inputs must fit a common format; for this, boxing—i.e., heap allocating and handling through a pointer—is sometimes necessary. However, this extra work can be quite expensive from a performance standpoint. One way of addressing this problem, which relies on type information, is *monomorphization*: duplication of polymorphic functions once for each instantiation type to obtain a monomorphic version of the function. This solution increases code size, though experimental evidence suggests that this increase is tolerable [5].

1.2.2 Untyped Arithmetic Expressions

To explore the concepts of type systems and type inference to a reasonable depth, we use a "toy" language of numbers and booleans, which contains just a handful of syntactic forms: the boolean constants `true` and `false`, conditional expressions, the numeric constant 0, the arithmetic operators `succ` and `pred` (successor and predecessor, respectively), and a testing `iszero` operation that returns `true` iff applied to 0. These forms are expressed in the grammar listed in Figure 1.2.

A program in this language is simply a term built from the forms given by the of Figure 1.2. Here are some examples of programs, along with the result of evaluating them:

- `if false then 0 else succ 0;`

t ::=

`true`	*constant true*
`false`	*constant false*
`if t then t else t`	*conditional*
`0`	*constant zero*
`succ t`	*successor*
`pred t`	*predecessor*
`iszero t`	*zero test*

Figure 1.2: Grammar of the Language of Arithmetic Expressions

⇒ succ 0

- `iszero (pred (succ 0));`

  ⇒ `true`

- `if (iszero (pred (pred (pred (pred 0))))) then succ 0 else pred 0;`

  ⇒ `succ 0`

- `succ (if iszero (pred 0) then succ 0 else succ (succ 0));`

  ⇒ `succ 0`

- `iszero (if iszero (succ 0) then succ 0 else pred 0)`

  ⇒ `true`

- `if (pred 0) then succ 0 else succ (succ 0)`

  ⇒ $\perp$

- `succ (if (iszero (pred 0)) then iszero (succ 0) else false)`

  ⇒ $\perp$

- `succ (if (iszero (pred 0)) then iszero (0) else succ 0)`

  ⇒ $\perp$

1.2.3 Structural Operational Semantics

Intuitively, programs such as

```
succ true
```

or

```
if 0 then 0 else 0
```

are “illegal” in that they have type errors. To capture this more formally, and more generally, assign “meaning” to programs in our language, we make use of the Structural Operational Semantics (SOS) [2]. SOS provides a framework for giving an operational semantics to programming languages. A distinguishing feature of SOS is that it emphasizes the individual steps comprising the computation, which is useful in capturing the behavior of programs with pointers and/or multithreading.

In the labelled transition system generated by SOS, transition $\gamma \xrightarrow{\mathtt{s}} \gamma'$ is interpreted as saying that the *first step* of executing statement `s` in state γ yields state γ'. If the result for configuration $\langle \gamma, \mathtt{s} \rangle$ is not available, then this configuration is considered *stuck*.

SOS enables inductive proofs of program properties: The evaluation relation ($\rightarrow$) is defined inductively by providing the ground axioms, as well as the inference rules. The meaning of a program is then a set of derivation trees. Note that this already suffices to synthesize an actual interpreter, though an interpreter built in this manner would probably suffer from poor performance.

For our purposes, we shall use the axioms and transition rules in Figure 1.3, where the axioms are E-IFTRUE, E-IFFALSE, E-PREDZERO, E-PREDSUCC, E-ISZEROZERO and E-ISZERONZERO (all the one-line statements).

Our SOS has the following properties:

- *Determinism* If we can derive both t_1 and t_2 from t, then t_1 and t_2 are necessarily equal. Formally, this as stated as follows:

  $$t \rightarrow t_1 \wedge t \rightarrow t_2 \Rightarrow t_1 = t_2.$$

  Proof sketch: The proof is by induction on a derivation of $t \rightarrow t'$. At each step of the induction, we assume the desired result for all smaller derivations, and proceed by a case analysis of the evaluation rule used at the root of the derivation.[1] Below are some of the cases:

  – If the last rule used in derivation $t \rightarrow t'$ is E-IFTRUE, then t is known to have the form `if` t_1 `then` t_2 `else` t_3, where t_1=**true**. This rules out the possibility that the last rule in $t \rightarrow t''$ is E-FALSE. Otherwise, the implication is that t_1 is equal both to **true** and to **false** at the same time. The last rule in $t \rightarrow t''$ cannot be E-IF either, since the permise of this rule demands that $t \rightarrow t'_1$ for some t'_1, but **true** does not evaluate to anything. This implies that the last rule in the second derivation is must be E-IFTRUE, and thus $t' = t''$.

[1] Notice that the induction here is not on the length of an evaluation sequence: We are looking just at a single step of evaluation. Instead, we are performing induction on the structure of t, since the structure of an evaluation derivation directly follows the structure of the term being reduced.

  - Analogously, if the last rule used in the derivation of $t \to t'$ is E-IFFALSE, then the last rule in the derivation of $t \to t''$ is necessarily the same and the result is immediate.
  - ...

Note that in general, determinism is not an inherent property of the SOS. In fact, one advantage of the SOS over more abstract semantics (such as the natural operational semantics) is that it is well suited to express parallelism. Here is an example of a useful pair of rules for indicating that all possible interleavings are possible during the evaluation of two statements, which leads to nondeterminism:

$$\frac{\texttt{t}_1 \to \texttt{t}_1'}{\texttt{t}_1 \texttt{ par } \texttt{t}_2 \to \texttt{t}_1' \texttt{ par } \texttt{t}_2} \quad (\text{PAR}^1)$$

$$\frac{\texttt{t}_2 \to \texttt{t}_2'}{\texttt{t}_1 \texttt{ par } \texttt{t}_2 \to \texttt{t}_1 \texttt{ par } \texttt{t}_2'} \quad (\text{PAR}^2)$$

Nondeterminism implies that the $\to$ relation is no longer *functional*.

- *Semantic Meaning* For every term t, there is some normal form t', such that $t \to^* t'$, where term u is in normal form if no evaluation rule applies to it. Further, t is in normal form iff t is a value (i.e., either a boolean or an integer). Thus, the following holds:

  $\forall t \in Terms.\iota\, t' \in NormalForm.t \to^* t'$, where $Terms$ is the language of the grammar in Figure 1.2, and $NormalForm$ is the set of all terms in normal form.

  Proof sketch: The proof follows from the observation that each evaluation step reduces the size of the term, and that size is a termination measure because the usual order on the natural numbers is well founded.

Note that the definition of the SOS is inductive, and thus we wish to find the smallest relation $\to$ that satisfies the rules in Figure 1.3.

It is an interesting exercise to check which properties the evaluation relation ($\to$) necessarily satisfies. It turns out that it is not *transitive*, as the following example demonstrates:

- (`iszero (succ (pred 0))`,`iszero (succ 0)`) $\in \to$ (E-ISZERO using E-PREDZERO)
- (`iszero (succ 0)`,**false**) $\in \to$ (E-ISZERO using E-ISZERONZERO)

- However, `iszero (succ (pred 0))` $\not\to$ **false**

It is also not *reflexive*: In fact, in all the rules appearing in Figure 1.3, the left-hand term differs from the right-hand term. Finally, the evaluation relation is not symmetric. For example, `iszero (succ 0)` $\to$ **false** but **false** $\not\to$ `iszero (succ 0)`.

Here are some (positive as well as negative) examples of how SOS is used to compute the (mathematical) meaning of a program:

Claim. `iszero (pred (succ 0))` $\to^*$ `true`

Proof.

$$\begin{array}{llr}
 & & \texttt{iszero (pred (succ 0))} \\
\Rightarrow & \text{(E-PREDSUCC)} & \texttt{iszero 0} \\
\Rightarrow & \text{(E-ISZEROZERO)} & \texttt{true}
\end{array}$$

Claim. `if (iszero (pred (pred 0))) then succ 0 else pred 0` $\to^*$ `succ 0`

Proof.

$$\begin{array}{clr}
 & & \texttt{if (iszero (pred (pred 0))) then succ 0 else pred 0} \\
\overset{E-PREDZERO}{\Rightarrow} & \text{(E-IF)} & \texttt{if (iszero (pred 0)) then succ 0 else pred 0} \\
\overset{E-PREDZERO}{\Rightarrow} & \text{(E-IF)} & \texttt{if (iszero 0) then succ 0 else pred 0} \\
\overset{E-ISZEROZERO}{\Rightarrow} & \text{(E-IF)} & \texttt{if true then succ 0 else pred 0} \\
\Rightarrow & \text{(E-IFTRUE)} & \texttt{succ 0}
\end{array}$$

Claim. Evaluating `succ (if (iszero (pred 0)) then iszero (succ 0) else false)` results in a stuck state.

Proof.

$$\begin{array}{clr}
 & & \texttt{succ (if (iszero (pred 0)) then iszero (succ 0) else false)} \\
\overset{E-PREDZERO}{\Rightarrow} & \text{(E-SUCC)} & \texttt{succ (if (iszero (0)) then iszero (succ 0) else false)} \\
\overset{E-ISZEROZERO}{\Rightarrow} & \text{(E-SUCC)} & \texttt{succ (if (true) then iszero (succ 0) else false)} \\
\overset{E-IFTRUE}{\Rightarrow} & \text{(E-SUCC)} & \texttt{succ (iszero (succ 0))} \\
\overset{E-ISZERONZERO}{\Rightarrow} & \text{(E-SUCC)} & \texttt{succ (false)} \\
\overset{Dom(succ)=Nat}{\Rightarrow} & & \perp
\end{array}$$

1.2.4 Typed Arithmetic Expressions

We now augment our language with static types. Recall that stuck terms correspond to erroneous or meaningless programs. We would therefore like to be able to tell, without actually evaluating a term, that its evaluation will definitely not get stuck. To do this, we need to distinguish between terms whose result is a numeric value and terms whose result is a boolean. We thus introduce two *types*, `Nat` and `Bool`, for classifying terms in this way.

$$\texttt{if}\ \textbf{true}\ \texttt{then}\ \texttt{t}_1\ \texttt{else}\ \texttt{t}_2 \rightarrow \texttt{t}_1 \qquad \text{(E-IFTRUE)}$$

$$\texttt{if}\ \textbf{false}\ \texttt{then}\ \texttt{t}_1\ \texttt{else}\ \texttt{t}_2 \rightarrow \texttt{t}_2 \qquad \text{(E-IFFALSE)}$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 \rightarrow \texttt{t}_1'\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3} \qquad \text{(E-IF)}$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{succ}\ \texttt{t}_1 \rightarrow \texttt{succ}\ \texttt{t}_1'} \qquad \text{(E-SUCC)}$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{pred}\ \texttt{t}_1 \rightarrow \texttt{pred}\ \texttt{t}_1'} \qquad \text{(E-PRED)}$$

$$\texttt{pred 0} \rightarrow \texttt{0} \qquad \text{(E-PREDZERO)}$$

$$\texttt{pred (succ t)} \rightarrow \texttt{t} \qquad \text{(E-PREDSUCC)}$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{iszero}\ \texttt{t}_1 \rightarrow \texttt{iszero}\ \texttt{t}_1'} \qquad \text{(E-ISZERO)}$$

$$\texttt{iszero 0} \rightarrow \textbf{true} \qquad \text{(E-ISZEROZERO)}$$

$$\texttt{iszero (succ t)} \rightarrow \textbf{false} \qquad \text{(E-ISZERONZERO)}$$

Figure 1.3: SOS Rules for Untyped Arithmetic Expressions

For example, the term `if true then false else true` has type `Bool`, while `pred (succ (pred (succ 0)))` has type `Nat`.

Recall that C is not a safe language. Here is an example demonstrating this:

```
int* x = malloc();
int* y = x;
free (x);
int* z = malloc();
if (y == z) printf(``Strange''); // L
```

In this example, `x` and `y` are aliased. However, if the compiler does not perform alias analysis, and—as an optimization—reuses heap locations to save allocation time, the result is that `y` and `z` would point to the same heap location at label `L`, and therefore ``` ``Strange'' ``` would be printed.

The above program typechecks. However, according to the following (fragment of) SOS for C, where we denote the free (allocated) heap locations in state s using the $s.free$ ($s.alloc$) notation, this program is not safe:

$$\frac{(\texttt{x}, s) \mapsto v}{(\texttt{free}(\texttt{x});, s) \to s[\texttt{x} \mapsto \mathbf{null}, s.free = s.free \cup \{v\}, s.alloc = s.alloc \setminus \{v\}]} \quad \text{(E-FREE)}$$

$$\frac{s.free \neq \emptyset}{(\texttt{malloc}();, s) \to s'.\ (\iota v \in s.free.\ (s'.free = s.free \setminus \{v\}, s'.alloc = s.alloc \cup \{v\}))} \quad \text{(E-MALLOC)}$$

$$\frac{(\texttt{E}, s) \to v}{(\texttt{L} = \texttt{E};, s) \to s \cup \{\texttt{L} \mapsto v\}} \quad \text{(E-ASSIGN)}$$

The typing relation for arithmetic expressions, written “`t : T`”, is defined by a set of inference rules assigning types to terms. These are summarized in Figure 1.4. The rules T-TRUE and T-FALSE assign the type `Bool` to the boolean constants true and false. Rule T-IF assigns a type to a conditional expression based on the types of its subexpressions: the guard $\texttt{t}_1$ must evaluate to a boolean, while $\texttt{t}_2$ and $\texttt{t}_3$ must both evaluate to values of the same type. The two uses of the single metavariable `T` express the constraint that the result of the `if` is the type of the then- and else- branches, and that this may be any type (either `Nat` or `Bool`).

The rules for numbers have a similar form. T-ZERO gives the type `Nat` to the constant 0. T-SUCC gives a term of the form `succ` $\texttt{t}_1$ the type `Nat`, as long as $\texttt{t}_1$ has type `Nat`.

$$\textbf{true} : \texttt{Bool} \qquad \text{(T-TRUE)}$$

$$\textbf{false} : \texttt{Bool} \qquad \text{(T-FALSE)}$$

$$\textbf{0} : \texttt{Nat} \qquad \text{(T-ZERO)}$$

$$\frac{\texttt{t}_1 : \texttt{Bool},\ \texttt{t}_2 : \texttt{T},\ \texttt{t}_3 : \texttt{T}}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{T}} \qquad \text{(T-IF)}$$

$$\frac{\texttt{t} : \texttt{Nat}}{\texttt{iszero t} : \texttt{Bool}} \qquad \text{(T-ISZERO)}$$

$$\frac{\texttt{t} : \texttt{Nat}}{\texttt{pred t} : \texttt{Nat}} \qquad \text{(T-PRED)}$$

$$\frac{\texttt{t} : \texttt{Nat}}{\texttt{succ t} : \texttt{Nat}} \qquad \text{(T-SUCC)}$$

Figure 1.4: Type Rules for Booleans (Bool) and Natural Numbers (Nat)

Likewise, T-PRED and T-ISZERO say that `pred` yields a `Nat` when its argument has type `Nat` and `iszero` yields a `Bool` when its argument has type `Nat`.

Formally, the typing relation for arithmetic expressions is the smallest binary relation between terms and types satisfying all instances of the rules in Figure 1.4. A term t is *typable* (or *well typed*) if there is some `T` such that `t : T`.

Inversion of Typing The "inversion of the typing" lemma—also known as the "generation" lemma—gives us a compendium of basic statements of the form "If a term of the form `succ` $\texttt{t}_1$ has any type at all, then it has type `Nat`.". These appear in Figure 1.6. Each of these statements follows immediately from the shape of the corresponding typing rule.

The inversion lemma leads directly to a recursive algorithm for calculating the types of terms, since it tells us, for a term of each syntactic form, how to calculate its type (if it has one) from the types of its subterms.

In Section 1.2.3, we introduced the concept of evaluation derivation, where we illustrated the derivation tree from a term to its value. Similarly a *typing derivation* is a tree of instances of the typing rules, from a term `t` to its type `T` to conclude that `t : T`. Figure 1.7 illustrates the derivation tree for the typing statement `if iszero 0 then 0 else pred 0 : Nat`.

Type Inference Contrary to type checking, where the input is both a term `t` and a type `T`, and the algorithm is to decide whether `t : T`, the input to a type-inference problem is just the term `t`, and the output is a type `T` such that `t : T` (if possible). One way of implementing type inference is by traversing the AST bottom up, and assign types to nodes according to the typing rules, where the typing axioms handle the leaf nodes.

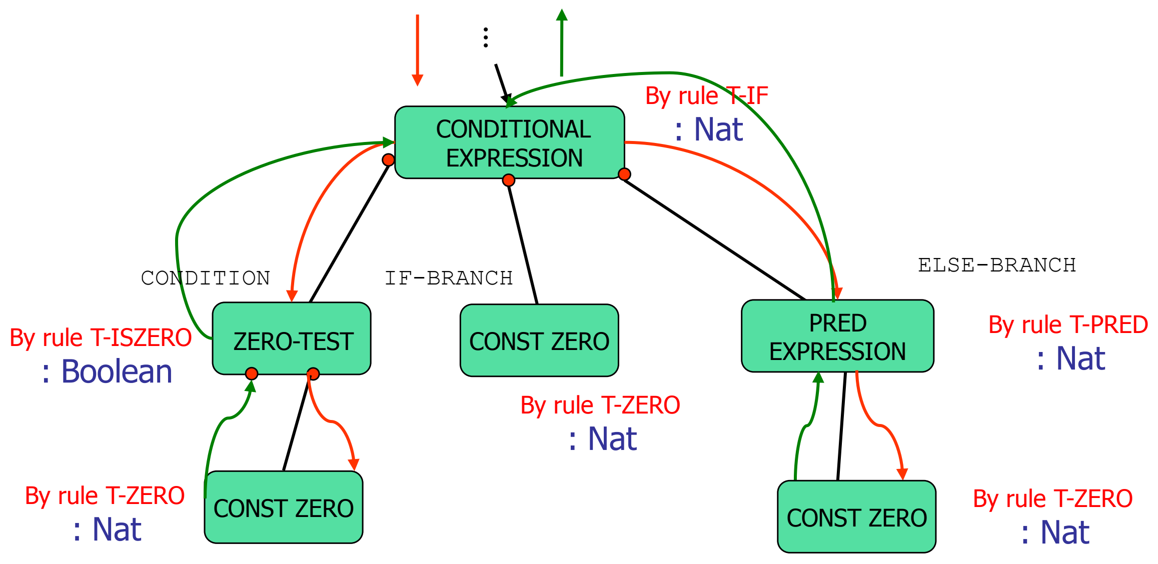
We illustrate the above algorithm using the example program in Figure 1.5. The algorithm scans the AST in bottom-up, prefix order. IT starts from the AST root, and descends to the leftmost "CONST ZERO" node, which it assigns type `Nat` according to the T-ZERO axiom. The next step is to assign the `iszero 0` expression type `Bool`, which is possible by applying the T-ISZERO rule. Next, the true branch of the `if` condition is visited, where—again—the T-ZERO axiom is applied. Finally, the false branch is visited, where the "CONST ZERO" node gets type `Nat` according to T-ZERO and then the `pred 0` expression gets type `Nat` according to T-PRED.

Uniqueness of Types An important theorem, dubbed the "Uniqueness of Types" theorem, states that each term `t` has at most one type. That is, if `t` is typable, then its type is unique. Moreover, there is just one derivation of this typing built from the inference rules in Figure 1.4. The proof is by structural induction on `t` using the appropriate clause of the inversion lemma (plus the induction hypothesis) for each case.

Note that while the simple type system we are dealing with satisfies that every term has a single type (if it has any type at all), and there is always just one derivation tree witnessing this fact, type systems with subtyping require that both of these properties be relaxed: A single term may have many types, and there may in general be many ways of deriving the statement that a given term has a given type.

Safety = Progress + Preservation As discussed above, the most basic property of a type system is safety (also called soundness): Well-typed terms do not "go wrong". Formally, a term goes wrong if its evaluation reaches a "stuck state" that is not designated as a final value, but where the evaluation rules do not tell us what to do next. We would like to assert that well-typed terms do not get stuck. This is done in two steps, commonly known as the progress and preservation theorems:

- *Progress* A well-typed term is not stuck. Either it is a value or it can take a step according to the evaluation rules.
- *Preservation* If a well-typed term takes a step of evaluation, then the resulting term is also well typed.



`if iszero 0 then 0 else pred 0`

Figure 1.5: Example Program and Its Corresponding AST Annotated with Types

`true : R`	$\Rightarrow$	`R : Bool`
`false : R`	$\Rightarrow$	`R : Bool`
`if` $\mathtt{t}_1$ `then` $\mathtt{t}_2$ `else` $\mathtt{t}_3$ `: R`	$\Rightarrow$	$\mathtt{t}_1$ `: Bool,` $\mathtt{t}_2$ `: T,` $\mathtt{t}_3$ `: T`
`0 : R`	$\Rightarrow$	`R : Nat`
`succ t : R`	$\Rightarrow$	`R : Nat, t : Nat`
`pred t : R`	$\Rightarrow$	`R : Nat, t : Nat`
`iszero t : R`	$\Rightarrow$	`R : Bool, t : Nat`

Figure 1.6: Rules of the Generation Lemma

$$\dfrac{\dfrac{\dfrac{}{0 : \mathtt{Nat}}\,\text{T-Zero}}{\mathtt{iszero}\ 0 : \mathtt{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{0 : \mathtt{Nat}}\,\text{T-Zero} \qquad \dfrac{\dfrac{}{0 : \mathtt{Nat}}\,\text{T-Zero}}{\mathtt{pred}\ 0 : \mathtt{Nat}}\,\text{T-Pred}}{\mathtt{if}\ \mathtt{iszero}\ 0\ \mathtt{then}\ 0\ \mathtt{else}\ \mathtt{pred}\ 0 : \mathtt{Nat}}\,\text{T-If}$$

Figure 1.7: Derivation Tree for `if iszero 0 then 0 else pred 0 :  Nat`

Together, these two properties—whose proof, taken from [8], appears below—tell us that a well-typed term can never reach a stuck state during evaluation.

For the proof of the progress theorem, it is convenient to record a couple of facts about the possible shapes of the canonical forms of types `Bool` and `Nat` (i.e., the well-typed values of these types):

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.

2. If `v` is a value of type `Nat`, then `v` is a numeric value according to the grammar in Figure 1.2.

Proof sketch: For the first part, according to the grammar in Figure 1.2, values in our language are in one of the following forms: **true**, **false**, `0` and `succ nv`, where `nv` $\in$ `Nat`. The first two cases immediately yield the desired result. As for the remaining two cases, these cannot occur: `v` is assumed to be of type `Bool`, and the inversion lemma tells us that `0` and `succ nv` can have only type `Nat`, and not `Bool`. The second part is proved in a similar fashion.

Now we can prove the progress theorem. The lemma on canonical forms facilitates the proof in establishing certain useful facts. For example, if we know that term $\texttt{t}_1$ in expression $\texttt{t=pred t}_1$ is a value, then the lemma guarantees that this value is either `0` or `succ nv` for some `nv`, and thus one of the rules E-PREDZERO or E-PREDSUCC applies to `t`.

Proof sketch: The proof of the progress theorem is done by induction on a derivation of `t : T`. The T-TRUE, T-FALSE and T-ZERO cases are immediate, since `t` in these cases is a value. Next, consider case case of T-IF, where

$$\texttt{t = if t}_1\texttt{ then t}_2\texttt{ else t}_3 \text{ with } \texttt{t}_1 : \texttt{Bool} \quad \texttt{t}_2 : \texttt{T} \quad \texttt{t}_3 : \texttt{T}.$$

By the induction hypothesis, either $\texttt{t}_1$ is a value, or else there is some $\texttt{t}'_1$ such that that $\texttt{t}_1 \to \texttt{t}'_1$. If $\texttt{t}_1$ is a value, then the canonical-forms lemma assures us that it must be either `true` or `false`, in which case either E-IFTRUE or E-IFFALSE applies to `t`. On the other hand, if $\texttt{t}_1 \to \texttt{t}'_1$, then by T-IF, $\texttt{t = if t}'_1\texttt{ then t}_2\texttt{ else t}_3$.

For T-SUCC, where

$$\texttt{t = succ t}_1 \text{ with } \texttt{t}_1 : \texttt{Nat},$$

the induction hypothesis tells us that either $\texttt{t}_1$ is a value or there is some $\texttt{t}'_1$ such that $\texttt{t}_1 \to \texttt{t}'_1$. If $\texttt{t}_1$ is a value, then by the canonical-forms lemma, it is a numeric value, and thus `t` is also a numeric value. Alternatively, if $\texttt{t}_1 \to \texttt{t}'_1$, then by E-SUCC, $\texttt{succ t}_1 \to \texttt{succ t}'_1$.

The remaining cases—those of T-PRED and T-ISZERO—are proved similarly.

The proof that types are preserved is also by induction on a derivation of `t`. At each step of the induction, we assume that the desired property holds for all subderivations (i.e., that if `s : S` and `s` $\rightarrow$ `s'`, then `s' : S`, whenever `s : S` is proved by a subderivation of the present one) and proceed by case analysis on the final rule in the derivation.

Proof sketch: We show only a subset of the cases (the others are similar):

- T-TRUE `t = true` `T = Bool`: If the last rule in the derivation is T-TRUE, then we know from the form of this rule that `t` must be the constant `true` and `T` must be `Bool`. However, then `t` is a value, so it cannot be the case that `t` $\rightarrow$ `t'` for any `t'`, and the requirements of the theorem are vacuously satisfied.

- T-SUCC `t = succ` $\mathtt{t}_1$ `T = Nat` $\mathtt{t}_1$ `: Nat`: By inspecting the evaluation rules in Figure 1.3, we see that there is just one rule, E-SUCC, that can be used to derive `t` $\rightarrow$ `t'`. The form of this rule tells us that $\mathtt{t}_1 \rightarrow \mathtt{t}'_1$. Since we also know $\mathtt{t}_1$ `: Nat`, we can apply the induction hypothesis to obtain $\mathtt{t}'_1$ `: Nat`, from which we obtain `succ` $\mathtt{t}'_1$ `: Nat`, i.e., `t' : T`, by applying rule T-SUCC.

- ...

Some refer to the preservation theorem as *subject reduction* or *subject evaluation*. The reason is that a typing statement `t : T` can be thought of as a sentence, "`t` has type `T`.". The term `t` is the subject of the sentence, and the subject-reduction property then says that the truth of the sentence is preserved under reduction of the subject.

To appreciate the importance of demanding both progress and preservation, consider the effect of removing the `true : Bool` typing rule. If we then consider program

```
    if (iszero 0) then 0 else succ 0,
```

then this expression is well typed with type `Nat`. This is because `iszero 0` has type `Bool` according to the T-ISZERO rule, and `0` and `succ 0` both have type `Nat` according to the T-ZERO and T-SUCC rules, respectively. However, if we now perform the `iszero 0` $\rightarrow$ **true** derivation step, then we arrive at the expression

```
    if true then 0 else succ 0,
```

which is no longer typable.

Thus, we have lost preservation. We started with a well-typed expression, but after taking a step according to the semantics, we arrived at an expression that is not typable. Notice that progress is still retained: Well-typed terms are not stuck. This was true before, and necessarily remains true if we preserve the original SOS rules and reduce the set of well-typed terms (which is what we did). The consequence is that we cannot prove our original program safe using our type system having given up on preservation.

A final comment is that unlike uniqueness of types, which does not hold in all type systems, progress and preservation are considered basic requirements. Still, there *are* languages where these properties do not hold, but which can nevertheless be considered type safe. A notable example is Java. Formalizing the operational semantics of Java in a small-step style does not yield type preservation in the form given here (cf. [8], chapter 19). However, this turns out to be an artifact of the formalization, rather than a defect in the Java language. since it disappears, for example, in a big-step presentation of the semantics.

1.3 Summary

Type systems provide a useful mechanism for enforcing certain safety properties in a conservative manner. The type system can be combined with the run-time system and/or static program analysis. An important property of type systems is that they enable interaction with the developer, and thus form a kind of specification. An alternative to type systems is static program analysis, where abstractions of concrete values are inferred at every program point.

Bibliography

[1] `http://langpop.com/`.

[2] `www.cs.vu.nl/~wanf/pubs/sos.pdf`.

[3] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2011.

[4] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. In *PLDI*, pages 1–12, 2001.

[5] Xavier Leroy. The effectiveness of type-based unboxing. Technical report, Boston College, Computer Science Department, 1997.

[6] D. Mandelin, L. Xu R., and Bodík D. Kimelman. Jungloid mining: helping to navigate the api jungle. pages 48–61, 2005.

[7] J. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2004.

[8] B.C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.