Data Types

- Why do we need data types?
  - Data resides in memory
  - Memory has no specific structure by itself
  - It’s up to interpretation to make sense out of it
  - Sometimes: value ranges

- Type systems help to apply and check for a syntactically consistent interpretation
  - Rule out programs that do not make any sense
  - Allow certain operations while disallowing others
  - Enforce syntactic discipline
  - Induce invariants

Data Types in C

- boolean
  - true or false, represented as 0 or (1 or -1)
  - only in C99: bool

- char
  - smallest addressable type
  - can represent a single character

- int
  - integer type, at least 16 bits

- long
  - integer type, at least 32 bits

- long long
  - integer type, at least 64 bits

Data Types

- Which data types does a computer understand?
  - Bits?
  - Bytes?
  - Words?

- Which data types do we know from programming languages?
  - boolean, scalar types, structs, …
Signed vs. Unsigned
- all the before mentioned types are signed by default in C
  - One bit is used to store the sign, the rest for the actual value
  - Two’s complement: \((-2^{n-1}) \cdot (2^{n-1} - 1)\)
- unsigned types have same representation in memory, different interpretation
  - \(0 \leq 2^n\)
- Data types have limited range, overflow may occur

Type Systems
- Representation Independence
- Types as a safety measure in programming languages
  - Robin Milner: Well-typed programs don’t go wrong.
- Invariants and Equational properties

Example: Ariane 5
- Example: The Explosion of Ariane 5
  - From the Official Report:
    - On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.
    - The failure of the Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was due to specification and design errors in the software of the inertial reference system.
    - The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer.

Example: Ariane 5
- The Inquiry Board recommends:
  - R5 Review all flight software (including embedded software), and in particular:
    - Identify all implicit assumptions made by the code and its justification documents on the values of quantities provided by the equipment...
  - R9 Include external (to the project) participants when reviewing specifications, code and justification documents. Make sure that these reviews consider the substance of arguments, rather than check that verifications have been made.
Type Systems

- Preventing execution errors by capturing them at compile time
  - Preventing a specific class of execution errors
  - Idea: Static analysis is easier than observing runtime behavior

Safety

- But... many languages are compiled to low-level unsafe code
- Poses the challenge of compiler correctness

Typed Arithmetic Expressions

- Terms can evaluate to:

  - t ::= terms
  - true
  - false
  - if t then t else t
  - 0
  - succ t
  - pred t
  - iszero t

  - constant true
  - constant false
  - conditional
  - constant zero
  - successor
  - predecessor
  - zero test

Typed Arithmetic Expressions

- Terms can evaluate to:

  - v ::= values
  - true
  - false
  - nv

  - false value
  - numeric value

- nv ::= numeric values

  - 0
  - succ nv

  - zero value
  - successor value

Or...

Evaluation Rules

\[
\begin{align*}
&E_{\text{IfTrue}} \quad \text{if } \text{true then } t_2 \text{ else } t_3 \rightarrow t_2 \\
&E_{\text{IfFalse}} \quad \text{if } \text{false then } t_2 \text{ else } t_3 \rightarrow t_3 \\
&E_{\text{If}} \quad \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow t_1 \rightarrow t_2' \rightarrow t_3' \quad \text{(E-If)}
\end{align*}
\]
**Evaluation Rules**

- $t_1 \rightarrow t'_1$
- $\text{succ } t_1 \rightarrow \text{succ } t'_1$
- $\text{pred } 0 \rightarrow 0$
- $\text{pred } (\text{succ } (n v)) \rightarrow n v$
- $t_1 \rightarrow t'_1$
- $\text{pred } t_1 \rightarrow \text{pred } t'_1$
- $\text{iszero } 0 \rightarrow \text{true}$
- $\text{iszero } \text{succ } (n v) \rightarrow \text{false}$
- $t_1 \rightarrow t'_1$
- $\text{iszero } t_1 \rightarrow \text{iszero } t'_1$

**Example**

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

**Stuck Term**

- No further evaluation rule applies that would bring the term into a normal form
- Erroneous program

- A Type System can tell us without actual evaluation that a term does not get stuck

**Typing Rules**

- true : Bool
- false : Bool
- if $t_1 : T$ then $t_2 : T$ else $t_3 : T$

- We can now say “a term has type $T$” when it obviously evaluates to a value of the appropriate form
- Obviously = statically, without performing evaluation
- if true then false else true has type Bool
- pred(succ(succ(0))) has type Nat
Typing Rules

- $0 : \text{Nat}$ (T-Zero)
- $t_1 : \text{Nat} \quad \text{suc} \quad t_1 : \text{Nat}$ (T-Succ)
- $t_1 : \text{Nat} \quad \text{pre} \quad t_1 : \text{Nat}$ (T-Pred)
- $t_1 : \text{Nat} \quad \text{iszero} \quad t_1 : \text{Bool}$ (T-IsZero)

Typing Relation

- Def: The **Typing Relation** for arithmetic expressions is the smallest binary relation between terms and types satisfying all instances of the typing rules.
- Def: A term $t$ is **typeable** or well-typed if there is some $T$ such that $t : T$

  Idea: Answers of the sort: If the term has any type at all, then it must be type Bool.

Typing Derivation

- if iszero $0$ then $0$ else succ $0 : \text{Nat}$

  if iszero $0 : \text{Nat}$
  $\quad$ iszero $0 : \text{Bool}$ (T-IsZero)
  $\quad$ succ $0 : \text{Nat}$ (T-Zero)
  $\quad$ if iszero $0$ then $0$ else succ $0 : \text{Nat}$ (T-If)

  Every pair $(t, T)$ in the typing relation can be justified by such a derivation tree.

Inversion Lemma

- If $\text{true} : R$, then $R = \text{Bool}$.
- If $\text{false} : R$, then $R = \text{Bool}$.
- If $\text{if} \quad t_1 \quad \text{then} \quad t_2 \quad \text{else} \quad t_3 : R$, then $t_1 : \text{Bool}, t_2 : R$, and $t_3 : R$.
- If $0 : R$, then $R = \text{Nat}$.
- If $\text{suc} \quad t_3 : R$, then $R = \text{Nat}$ and $t_3 : \text{Nat}$.
- If $\text{pred} \quad t_3 : R$, then $R = \text{Nat}$ and $t_3 : \text{Nat}$.
- If iszero $t_1 : R$, then $R = \text{Bool}$ and $t_1 : \text{Nat}$.

  Practically describes an algorithm to construct a typing
Limitations of Typing

- What is the type of the following term:
  \[ \text{if false then true else 0} \]

- Well-formed versus well-typed.

- Type systems are often conservative. They do not predict the outcome of every possible program but only cover cases in which a safe approximation can be made.

Safety: Soundness of Type Systems

- Progress
  - A well-typed term \( t \) is not stuck
    - If \( t : T \) then either \( t \) is a value or \( t \rightarrow t' \) for some \( t' \).

- Preservation
  - Types are preserved by one-step evaluation
    - If \( t : T \) and \( t \rightarrow t' \) then \( t' : T \)

Canonical Forms

- Lemma:
  - If \( v \) is a value of type \( \text{Bool} \), then \( v \) is either \( \text{true} \) or \( \text{false} \).
  - If \( v \) is a value of type \( \text{Nat} \), then \( v \) is a numeric value.

Progress

- Progress: If \( t : T \) then either \( t \) is a value or \( t \rightarrow t' \) for some \( t' \).

- Proof by induction on a derivation of \( t : T \):
  - Case T-True: \( t = \text{true} \Rightarrow t \) is a value
  - Case T-False: \( t = \text{false} \Rightarrow t \) is a value
  - Case T-Zero: \( t = 0 \Rightarrow t \) is a value
Progress

- Progress: If \( t : T \) then either \( t \) is a value or \( t \rightarrow t' \) for some \( t' \).

- Case T-If: Inversion lemma tells us that for \( t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \),
  \[ t_3 : \text{Bool}, t_2 : T, t_3 : T \]

  - The following evaluation rules apply: E-IfTrue, E-IfFalse, and E-If
  - Assume that the theorem holds for \( t_3 \).
    - If \( t_1 \) is a value, then \( t_1 : \text{Bool} \) and \( t_1 \) must be either true or false. In this case, the reduction can continue through E-IfTrue and E-IfFalse.
    - There exists a \( t' \) such that \( t_1 \rightarrow t' \) (since we know \( t_1 : \text{Bool} \)) and the reduction of \( t \) can be accomplished via E-If.

Preservation

- Preservation: If \( t : T \) and \( t \rightarrow t' \) then \( t' : T \)

  - Case T-True: \( t = \text{true} \), \( T = \text{Bool} \), \( t \rightarrow t' \) is not possible.
  - Case T-False: \( t = \text{false} \), \( T = \text{Bool} \), \( t \rightarrow t' \) is not possible.
  - Case T-Zero: \( t = 0 \), \( T = \text{Nat} \), \( t \rightarrow t' \) is not possible.
  - Case T-Succ: \( t = \text{succ } t_3 \), \( T = \text{Nat} \), \( t_3 : \text{Nat} \)
    - Only E-Succ can derive \( a \rightarrow t' \), since \( t_3 : \text{Nat} \) also \( t'_3 : \text{Nat} \) by induction hypothesis

Progress

- Progress: If \( t : T \) then either \( t \) is a value or \( t \rightarrow t' \) for some \( t' \).

  - Case T-Succ: \( t = \text{succ } t_1 \Rightarrow t_1 \) is a value or E-Succ applies
  - Case T-Pred: \( t = \text{pred } t_1 \Rightarrow \text{E-PredZero } (t_1 \) is a value), E-PredSucc (\( t = \text{pred } (\text{succ } (t_2)) \) \( t_2 \) is a value), or E-Pred applies.
  - Case T-IsZero: \( t = \text{iszero } t_1 \Rightarrow \text{E-IsZeroZero, E-IsZeroSucc, E-IsZero apply.} \)