Software System Design and Implementation

Machine-checked Properties

Gabriele Keller
Manuel Chakravarty

The University of New South Wales
School of Computer Science and Engineering
Sydney, Australia
Checking properties

• Logical properties are key to specifying the intended meaning of programs

  ‣ Types,

  ‣ QuickCheck properties,

  ‣ Hoare triples,

  ‣ and so on

• How can computers help to check these properties?
Checking properties dynamically

- Property-based testing (QuickCheck)
  - Properties as program fragments
  - Randomised test case generation
Checking properties dynamically

- Assertions
  - Assertions in *design by contract* (Eiffel, D, Ada)
    - specify pre- & postconditions of methods
  - invariants of objects
    - assertions can be extracted as documentation
  - Evaluating properties during program execution (testing & debugging)
Checking properties statically

• Proof checkers (theorem provers)
  ▸ In general, (at least parts of) proofs need to be supplied manually

• Static analysis
  ▸ Abstract interpretation, flow analysis, and so on

• Type checking
  ▸ Types are properties
  ▸ Type checking is a form of theorem proving (decidable logic)
Hybrid approaches

- Contracts
  - May be checked statically or dynamically
  - Possibly static checking delaying checking of residual properties until runtime

- Gradual typing
  - Statically checks for type errors in some parts of a program
  - Leaves other parts to be checked dynamically
Compiler integration provides extra leverage

- Compiler-checked properties are automatically checked on every compiler run
  - Cannot diverge from source code
  - Provide checked documentation

- Types
  - Are used and understood by every developer
  - Are tightly integrated with the language

Let's look at some particularly expressive types
Generalised Algebraic Data Types
GADTs

• Also called indexed data types

• Use of a type argument to specify a property of the data type
  
  ‣ E.g., a datatype of expression terms with the type of the expression as a type argument

• Simultaneously restricts the values of the GADT
  
  ‣ E.g., a list type indexed by the length of the list
Motivating example: a type-safe evaluator

```haskell
data Expr = BConst Bool
    | IConst Int
    | Times Expr Expr  -- arguments must be of type Int
    | If   Expr Expr Expr  -- 1st argument must be a Bool,
                     -- 2nd & 3rd of same type

data Value = IVal  Int
    | BVal Bool
```

Evaluation result

Informal type constraints

Expression evaluation

Evaluation result

```
eval :: Expr -> ???
```

Result

Expression evaluation
data Expr  = BConst Bool  
            | IConst Int  
            | Times Expr Expr  
            | Less Expr Expr  
            | And  Expr Expr  
            | If   Expr Expr Expr  

data Result = IVal Int  
            | BVal Bool  

eval :: Expr -> Result  
eval (IConst n)  
  = IVal n  
eval (BConst b)  
  = BVal b  
eval (Times ex1 ex2)  
  = case (eval ex1, eval ex2) of  
      (IVal n1, IVal n2) -> IVal (n1 + n2)  
      (_, _)             -> error "illegal expr"  
...
• The evaluated expressions are dynamically typed (like, say, Python programs)

• During evaluation, we check that operators (e.g., addition) receive operands of compatible type

• If the types are not compatible, we yield a runtime error (or exception)
Strongly typed languages:

```
Expr \rightarrow \text{eval} \rightarrow \text{Result}
```

**precondition:**
- input is an Expr

**postcondition:**
- input is a Result

**invariant:**
- any value of type Expr
- indeed valid object of that type.
- E.g., BConst 5
- statically excluded

Type checker ensures
- **precondition** observed whenever function is called
- **invariants** hold at any time during program execution
- **postcondition** holds after every call of the function (no guarantee for non-termination, or in case of run-time error)
Strongly typed languages:

eval

Expr → Result

precondition: input is an Expr
postcondition: input is a Result

invariant:
any value of type Expr
indeed valid object of that type.
E.g., BConst 5
statically excluded

To get the most out of the type checker, we need to

• make functions total, if feasible
• make types as precise as possible

If (IConst 5) (IConst True) (IConst 2)
Key idea

- Parametrise expressions by the type of value they evaluate to
  - Expressions have unique types (don't change during evaluation)
  - Expression of type $\tau$ evaluates to a value of type $\tau$
- In Haskell: `Expr t` is an expression of type $t$

Define type expressions as a data type & adapt the evaluator
Type indices

- The type argument $t$ in $\text{Expr } t$ is a type index

  - Type indices constraint the formation of values
  
    - The type checker rejects malformed terms; e.g.,
      
      \[
      \text{If (Const 1) (Const 2) (Const 3)} \quad -- \text{type error!}
      \]

  - Our expressions can only have the types $\text{Expr } \text{Int}$ and $\text{Expr } \text{Bool}$

\[
\text{If } :: \quad \text{Expr } \text{Bool} \rightarrow \text{Expr } s \rightarrow \text{Expr } s \rightarrow \text{Expr } s
\]
Calculating with types
Singleton types

- Indexed type, where the type index uniquely identifies the value
  - As types are sets of values, singleton types are one-element sets

- Let's look at an example: singleton Booleans

```haskell
data Bool where
  False :: Bool
  True :: Bool
```

```haskell
data SBool (b :: Bool) where
  SFalse :: SBool False
  STrue :: SBool True
```
• If a function returns a value of type `SBool False`, we know the return value without executing the function (modulo non-termination)

• Singleton types enable us to reflect values to the type level

• Why is this useful?
  - Stronger types characterise the behaviour of a program more precisely
  - Haskell type checker as proof checker
Values, Types, and Kinds in plain Haskell

- Values (including functions and data constructors) have types
- Types and type constructors have kinds

Kinds

- *
- * → *
- * → * → *

Types

- Int :: *
- Bool :: *
- Maybe :: * → *
- Maybe Bool :: *
- a → a → Bool :: *

Values

- 5 :: Int
- True :: Bool
- Just :: a → Maybe a
- Just True :: Maybe Bool
- (==) :: a → a → Bool
Singleton natural numbers

• We can characterise the set of natural numbers inductively as follows

  ‣ Zero (0) is a natural number

  ‣ If \( n \) is a natural number, the successor of \( n \) (\( n + 1 \)) is a natural number

• This characterisation is based on the Peano axioms of natural numbers

```haskell
data Nat where
  Z :: Nat
  S :: Nat -> Nat

data Nat
  = Z
  | S Nat

data SNat (n :: Nat) where
  Zero :: SNat Z
  Succ :: SNat m -> SNat (S m)
```
With data kinds:

- Now, kinds and types overlap (e.g., Nat can be used both as a type and a kind)

**Kinds & Data Kinds**

- \* \n- \text{Nat} \rightarrow \* \rightarrow \ast \rightarrow \ast \rightarrow \ast
- \text{Nat}

**Dependent Types**

- \text{SNat} Z :: \ast
- Z :: \text{Nat}
- \text{SNat} :: \text{Nat} \rightarrow \ast
- \text{Nat} :: \ast

**Values**

- 5 :: \text{Int}
- \text{Z} :: \text{Nat}
- S(S Z) :: \text{Nat}
- S :: \text{Nat} \rightarrow \text{Nat}
- S Z :: \text{Nat}
Let's take a step back — types versus values

• Types are static; values are dynamic

• **Type erasure property:** types don't impact a program's semantics

• Types characterise part of a program's behaviour:
  
  ‣ Each value has a **unique type**, but usually a type stands for **many values**
  
  ‣ In contrast: a singleton types has a **unique value**

• Singleton types lift data from the value to the type level

• How about computations (functions) on the type level?
Type families

• There are two forms of type families in Haskell
  
  ‣ Type synonym families: effectively provide functions on types
  
  ‣ Data type families: essentially are a form of open (or, extensible) GADTs
  
• We will focus on type synonym families, which differ from value functions:
  
  ‣ They need to be terminating — how do we know (halting problem)?
  
  ‣ Limited syntax and obviously no side effects
  
  ‣ They are extensible (like type classes)
Computing with types

• With type families, we can define arithmetic operations on type-level numerals

• We can also tie type-level to value-level computations

\[
\text{type family } (+) \ (n :: \text{Nat}) \ (m :: \text{Nat}) :: \text{Nat}
\]