Utrecht University

Concepts of Programming Language Design
MinHS

Gabriele Keller
Tom Smeding

Overview

- We're going to look into two simple, Turing-complete programming languages:
    - MinHs (functional), TinyC (procedural/imperative)
- We look at what functional programming is about
- We will make our abstract machines more realistic

MinHs: The essence of functional programming

- MinHs, a stripped down, purely functional language
  - purely functional: no side effects, functions are first class citizens

MinHs: The essence of functional programming

- What does it mean if a language has no side effects?
  - the value of a function only depends on the values of its arguments, not on some implicit state
  - evaluating a function does not alter an observable state

global counter

no argument, value depends on state of counter

same call, different results

```
int count = 0;

int inc () {
  count = count + 1;
  return count;
}

int y = inc();
int z = inc();
```

global initial counter, value can't change

current counter value needs to be passed as argument

current counter value needs to be passed as argument

```
count :: Int
count = 0;

inc :: Int -> Int
inc curr = curr + 1

y = inc count
z = inc y
```

Utrecht University

MinHs: The essence of functional programming

- Pure functions result in referential transparency
  - we can replace an expression with its value anywhere in the program
  - equational reasoning is possible

```
inc() + inc()  =  2 * inc()
```

this equality only valid in a language with referential transparency

MinHs: The essence of functional programming

```
count :: Int
count = 0

inc :: Int -> Int
inc curr = curr + 1

y = inc count
z = inc y
```

```
   z
= {def of z}
   inc y
= {def of y}
   inc (inc count)
= {def of inc}
   inc (count + 1)
= {def of inc}
   (count + 1) + 1
```

Utrecht University

MinHs: The essence of functional programming

- The absence of implicit state makes it easier to write concurrent and parallel programs

```
foo x = y + z
  where
    y = bar x
    z = baz x
```

both can be evaluated in parallel, no need to worry about altering a shared state

MinHs: The essence of functional programming

- Note that the absence of state makes loops (while/for) impossible

```
while condition
  do
    something
```

- Instead, repetition has to be modelled via recursion

```
f x = if (condition x)
        then result
        else f (update x)
```

MinHs: The essence of functional programming

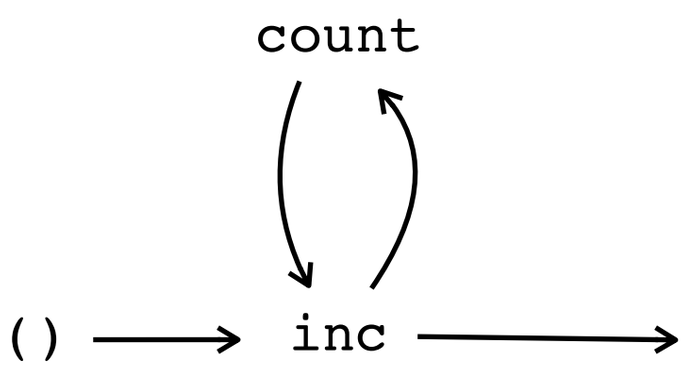
- Some data structures are all about destructive updates
  - e.g., arrays, hash tables
  - In functional languages, other data structures are often used instead
  - lists instead of arrays
  - sophisticated tree structures for lookup tables

MinHs: The essence of functional programming

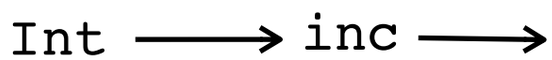
- But we can’t do without state!
  - programs have to be able to do I/O, otherwise they are useless
  - destructive updates of data structures if often important for efficiency reasons (e.g., updating one element in an array)

- State is not a problem for referential transparency, implicit state is!

```
int inc () {
  count = count + 1;
  return count;
}
```




```
inc :: Int -> Int
inc curr = curr + 1
```

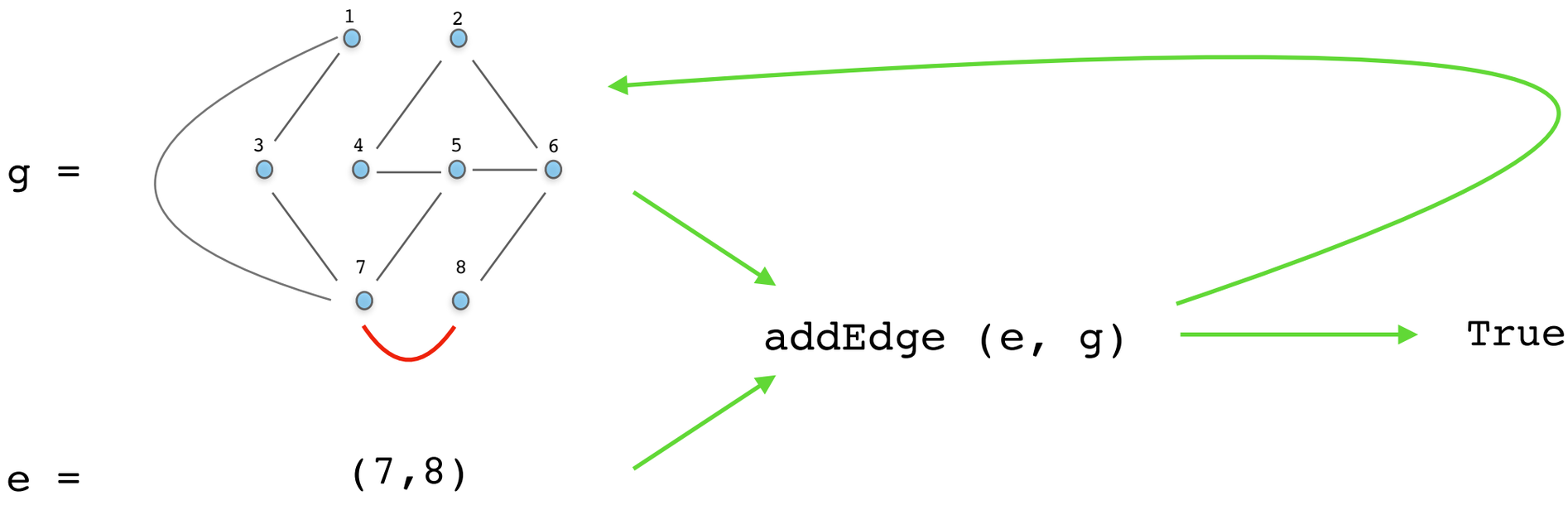


MinHs: The essence of functional programming

- Example:
  - graph computations: manipulating graphs, annotating graph nodes or edges
  - add an edge to a graph, return true if edge is new, false otherwise

- With destructive updates:

```
Bool addEdge (Edge edge, Graph g)
```

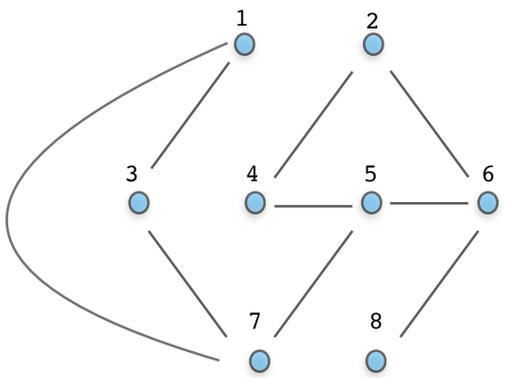


MinHs: The essence of functional programming

- In a purely functional world:

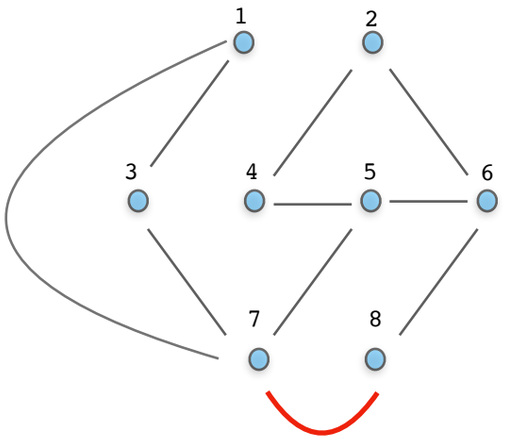
```
addEdge :: Edge -> Graph -> (Bool, Graph)
```

g =



e = (7,8)

addEdge e g



True

MinHs: The essence of functional programming

- Observation:
  - we usually don't need the old graph/state after the update, so it's unnecessary to keep it around
  - how can we ensure that the old graph is indeed not accessed again in the program?

- Idea:
  - thread the state (in this case, the graph) through the computations
  - only allow limited access to the state:
    - no copying
    - no 'losing' the state

MinHs: The essence of functional programming

- Graph computation:

```
type GraphComp a = (Graph -> (a, Graph))

addEdge             :: Edge -> GraphComp Bool
neighbours          :: Node -> GraphComp [Node]
```

- Functions over graphs are composed together, without making the graph directly accessible:

```
(|>) :: GraphComp a -> (a -> GraphComp b) -> GraphComp b

(|>) f1 f2 = f1f2
  where
     f1f2 graph =
       let (a, newGraph) = f1 graph
       in  f2 a newGraph

(|>>) :: GraphComp a -> GraphComp b -> GraphComp b

(|>>) f1 f2 = f1f2
  where
     f1f2 graph =
       let (a, newGraph) = f1 graph
       in  f2 newGraph
```

```
addEdge (1,2) |>> addEdge (1,4) |>> neighbours 1
```

Utrecht University

MinHs: The essence of functional programming

- This is how IO computations are handled in Haskell
- An IO computation gets the state of the current world, returns a value and a new state of the world:

```
type IO a = World -> (a, World)

putStrLn :: String -> IO ()
readLn   :: IO String
return   :: a -> IO a
```

representation simplified

- do-notation is a convenient way to compose such functions

```
main :: IO ()
main = do
  { putStrLn "What's your name?"
  ; name <- readLn
  ; let newStr = ("Hi " ++ name)
  ; putStrLn newStr
  }
```

- preserves referential transparency, even though we have side effects 'behind the scenes'
- no safe function of type `IO a -> a`

MinHs: The essence of functional programming

- Observation:

  - any function which calls an IO function has IO type

  - functions which have an observable effect on the world, or depend on the state of the world, have type IO

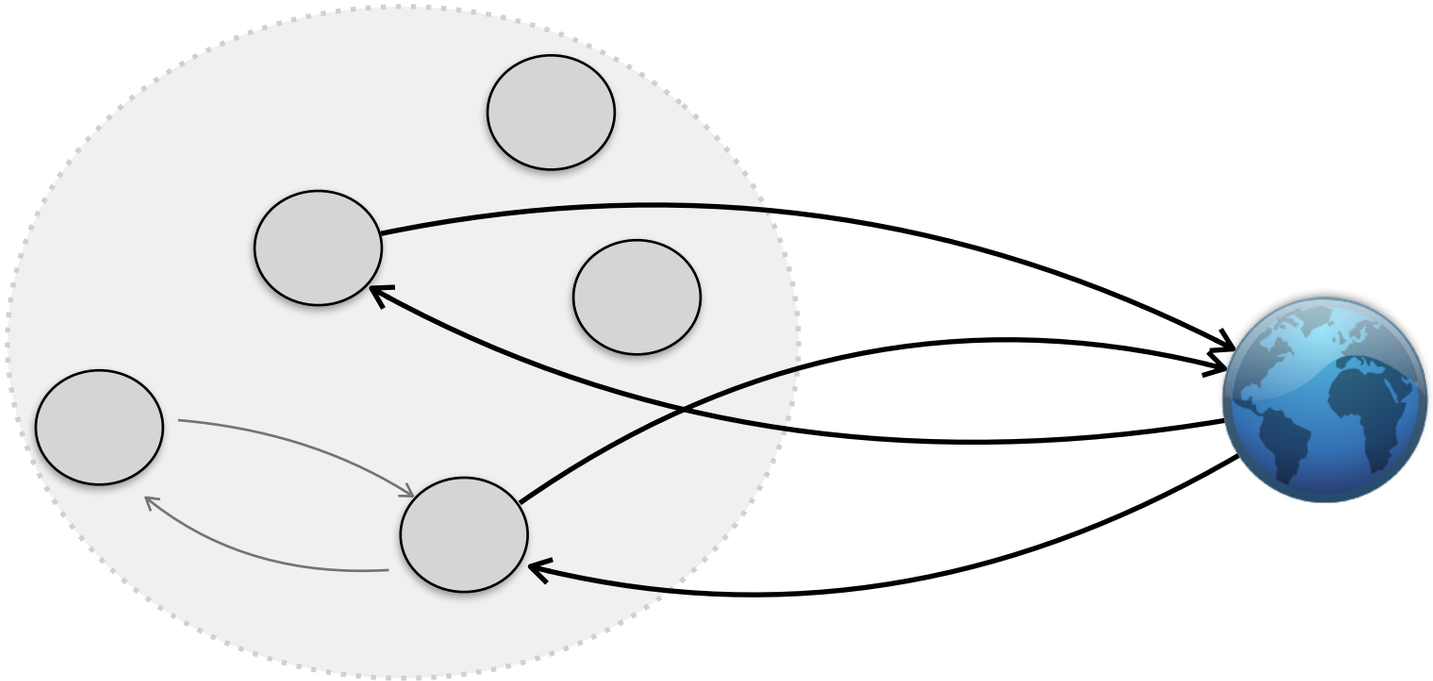
  - IO functions can call side-effect free functions, but not the other way around

  - this has a profound impact on the way purely functional programs need to be designed!

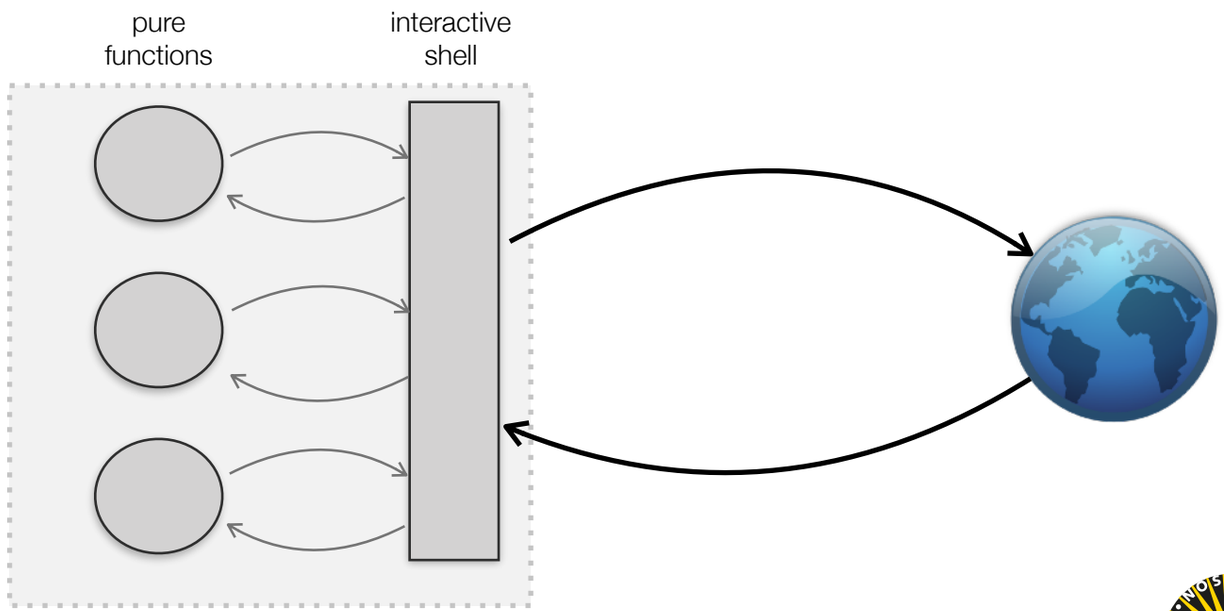
Utrecht University

MinHs: The essence of functional programming

- Design in a stateful language:



- Design in pure language



Utrecht University

MinHs: The essence of functional programming

- Functions as first class citizens
  - functions can accept other functions as arguments, return functions as result
  - no fundamental distinction between functions and value
- This also has an effect on how programs are structured and designed
  - functions as combination of other functions
  - recursive data structures (lists, trees) manipulated using higher-order functions instead of explicit recursion

```
sum1 []     = 0
sum1 (x:xs) = x + sum1 xs
```

Utrecht University

MinHs: The essence of functional programming

- The design of an application written in a (higher-order) functional language is often in terms of these ‘patterns’.

- Example: the pattern we observed for threading state

```
type GraphComp a = (Graph -> (Graph, a))
type IO a        = (World -> (World, a))
```

```
(>>=)  :: m a -> (a -> m b) -> m b
return :: a -> m a
```

Patterns

```
(>>=)  :: m a -> (a -> m b) -> m b
return :: a -> m a
```

- Type constructors for which we can define those two functions in a way that they fulfil the three properties below are called a Monad, and we can use the do-notation to compose them
  - Left identity `return a >>= h ≡ h a`
  - Right identity `f >>= return ≡ f`
  - Associativity `(f >>= g) >>= h ≡ f >>= (\x -> g x >>= h)`
- Convenient to use for stateful computations, but also other type constructors (like lists) are in this class

Functional Programming

- Based on the lambda-calculus
- Lisp is one of the oldest high-level languages (Fortran is older), 1958
  - Clojure, CommonLisp, Scheme, Racket in this family of languages
- ML (Meta Language), strict & statically typed
  - OCaml, F#, Standard ML
- Haskell
  - lazy, strongly typed
  - Agda, Bluespec
  - Clean (older than Haskell, same family, with uniqueness type system)
- Erlang
  - concurrent, communication based on message passing
- Many other languages have features from functional languages
  - Python, C#, Rust, Go, C++, Java, PurseScript, Java, …

MinHs: The essence of functional programming

- MinHs, a stripped down, purely functional language

  - purely functional - no side effects, functions are first class citizens

  - strict evaluation

    – Haskell is lazy, Standard ML, OCaml strict

  - statically typed

    – not all functional languages are statically typed (Haskell, Standard ML family of languages are, Lisp-like languages are dynamically typed)

  - types have to be provided by the programmer - no type inference, only type checking

Utrecht University

Concrete Syntax

- The BNF is ambiguous, but the usual precedence and associativity rules apply:

Variables	***Ident***	::=	...
Integer values	***Int***	::=	...
Boolean values	***Bool***	::=	`True` | `False`
Types	***Type***	::=	`Bool` | `Int` | ***Type*** $\rightarrow$ ***Type*** | `(` ***Type*** `)`
Infix Operators $\otimes$		::=	`+` | `*` | `-` | `=` | `<` | `>` | `>=` | `<=`
Expressions	***Expr***	::=	***Ident*** | ***Int*** | ***Bool*** | `(` ***Expr*** `)` | ***Expr*** $\otimes$ ***Expr*** | ***Expr*** ***Expr***
			| `if` ***Expr*** `then` ***Expr*** `else` ***Expr***
			| `recfun` ***Ident*** `::` `(` ***Type*** $\rightarrow$ ***Type***`)` ***Ident*** `=` ***Expr***

- The function type constructor is right associative

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \quad = \quad \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3) \neq \quad (\tau_1 \rightarrow \tau_2) \rightarrow \tau_3$$

Utrecht University

MinHs

- Restrictions of the language
  - MinHs doesn’t have a let-bindings
  - a function name is only visible in its own body
  - functions have only one argument at a time
- We could extend the language to change this, but it doesn’t add any interesting problems
- These restrictions do not affect express expressiveness of the language

MinHs

- Example

```
recfun divBy5 :: (Int ⟶ Int) x =
  if x < 5
    then 0
    else 1 + divBy5 (x - 5)
```

```
recfun div :: (Int ⟶ (Int ⟶ Int)) x =
  recfun div' :: (Int ⟶ Int) y =
    if x < y
      then 0
      else 1 + div (x - y) y
```

- Function application is left associative, so the two expressions are equivalent:

```
div 15 5

(div 15) 5
```

Utrecht University

Abstract Syntax

- First-order abstract syntax:
  - Terms of the form $(Op\ t_1\ \ldots\ t_n)$
    - we need to store the type information, so types are also terms, but for convenience, we leave them in concrete syntax form, that is, we write

      `(Int → Int)` not `(FunType Int Int)`

  - We don't formalise the translation rules. Informally:
    - replace all infix by prefix operators:
      - e_1 `+` e_2 becomes `(Plus` e_1 e_2`)`
      - `if` e_1 `then` e_2 `else` e_3 becomes `(If` e_1 e_2 e_3`)`
    - application becomes explicit
      - $e_1\ e_2$ becomes `(Apply` e_1 e_2`)`
    - function definitions
      - `recfun (`f `::` $\tau_1 \rightarrow \tau_2$`)` x `=` e becomes `(Recfun` τ_1 τ_2 f x e`)`

Abstract Syntax

- Higher-order abstract syntax:
  - only the representation of the functions changes with respect to first order
  - variable name x and function name f are bound in e
    - `(Recfun` τ_1 τ_2 `(`$f.x.e$`))`
  - the scope of f and x is e

Static Semantics of MinHs

- We have to check that
  - all variables are defined
  - all expressions are well typed
- What about the environment?
  - the environment has to contain type information
    - $\Gamma = \{x_1 : \texttt{Int}, x_2 : \texttt{Bool}, f : \texttt{Int} \rightarrow \texttt{Bool}, \ldots\}$
  - for the moment, we assume that all function and variable names are unique

Utrecht University

Type and Scope Checking

- Proceeds by using typing rules over the structure of the abstract syntax of MinHs
- Essentially, an extension of the scoping rules
- We need typing rules for
  - constant values, variables
  - operators
  - function definitions
  - application

- We define a typing judgement of the form

$$\Gamma \vdash t : \tau$$

stating that term t *is a legal higher order syntax term of the language and has type* τ *under the environment* Γ

Typing Rules for MinHs

$$\boxed{\Gamma \vdash t : \tau}$$

$$\frac{}{\Gamma \vdash \texttt{(Num } i \texttt{) : Int}}$$

$$\frac{b \in \texttt{\{True, False \}}}{\Gamma \vdash \texttt{(Const } b\texttt{): Bool}}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Int} \qquad \Gamma \vdash t_2 : \texttt{Int}}{\Gamma \vdash \texttt{(Plus } t_1\ t_2\texttt{) : Int}}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash \texttt{(If } t_1\ t_2\ t_3\texttt{) : } \tau}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Typing Rules for MinHs

- Functions and applications:

$$\frac{\Gamma \vdash\ t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash\ t_2 : \tau_1}{\Gamma \vdash\ (\texttt{Apply}\ \ t_1\ \ t_2) : \tau_2}$$

$$\frac{\Gamma \cup \{ f : \tau_1 \to \tau_2\ ,\ x :\ \tau_1 \} \vdash\ t : \tau_2}{\Gamma \vdash\ (\texttt{Recfun}\ \ \tau_1\ \ \tau_2\ \ (f.x.t)) : \tau_1 \to \tau_2}$$

Inversion

- Observation
  - there is only one rule for each type of expression
  - the typing is syntax directed
    - the form of the syntax uniquely defines the typing rule
  - as a result, ***the inversion principle*** is applicable
- Example: typing rule for if-expressions:

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (\texttt{If}\ t_1\ t_2\ t_3) : \tau}$$

  - The rule states that

    if $\Gamma \vdash t_1 : \texttt{Bool}$ and $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_1 : \tau$ are all derivable,

    then $\Gamma \vdash (\texttt{If}\ t_1\ t_2\ t_3) : \tau$ is derivable

  - Inversion (since there is only one rule for `If`):

    if $\Gamma \vdash (\texttt{If}\ t_1\ t_2\ t_3) : \tau$ is derivable

    then, $\Gamma \vdash t_1 : \texttt{Bool}$, $\Gamma \vdash t_2 : \tau$ and $\Gamma \vdash t_3 : \tau$ are derivable

    because the above rule must have been used

Utrecht University

Inversion

- Hence, we can conclude inverse rules

$$\frac{\Gamma \vdash (\texttt{If}\ t_1\ t_2\ t_3) : \tau}{\Gamma \vdash t_1 : \texttt{Bool}}$$

$$\frac{\Gamma \vdash (\texttt{If}\ t_1\ t_2\ t_3) : \tau}{\Gamma \vdash t_1 : \tau}$$

$$\frac{\Gamma \vdash (\texttt{If}\ t_1\ t_2\ t_3) : \tau}{\Gamma \vdash t_2 : \tau}$$

- Inversion hold for all other typing rules in MinHs as well
- Formally, it can very easily (really!) be proven using rule induction

Dynamic Semantics of MinHs

- Structured operational semantics (SOS)
  - Initial states:
    - all well typed expression
  - Final states:
    - boolean and integer constants
    - and functions!

  .....

- Evaluation of built-in operations: just like for the arithmetic expression language

Structural Operational Semantics of MinHs

- Evaluation of `if`-expressions

$$\frac{e_1 \mapsto e_1'}{\texttt{(If } e_1\ e_2\ e_3\texttt{)} \mapsto \texttt{(If } e_1'\ e_2\ e_3\texttt{)}}$$

$$\frac{}{\texttt{(If (Const True) } e_2\ e_3\texttt{)} \mapsto e_2}$$

$$\frac{}{\texttt{(If (Const False) } e_2\ e_3\texttt{)} \mapsto e_3}$$

Structural Operational Semantics of MinHs

- How about functions?

$$\texttt{(Recfun } \tau_1 \ \ \tau_2 \ \ \texttt{(}f.x.t\texttt{))} \mapsto ?$$

- Function application

```
(recfun f :: Int -> Int x = x * (x + 1)) 5
```

evaluates to

```
5 * (5 + 1)
```

- There is a similarity to let-bindings in the arithmetic expression language
- We replace the variable (function parameter) by the function argument (after it has been evaluated)

Utrecht University

Structural Operational Semantics of MinHs

- How about recursion?

```
(recfun f :: Int -> Int x =
   if (x < 1)
     then 1
     else x * f(x - 1)) 3
```

to

```
if (3 < 1)
      then 1
      else 3 * f(3 - 1))
```

something is wrong here - f occurs now free in the expression!

Structural Operational Semantics of MinHs

- How about recursion?

```
(recfun f :: Int -> Int x =
   if (x < 1)
     then 1
     else x * f(x-1)) 3
```

to

```
if (3 < 1) then 1
          else 3 * (recfun f :: Int -> Int x =
                          if (x < 1)
                            then 1
                            else x * f(x-1))
                    (3-1))
```

Utrecht University

Structural Operational Semantics of MinHs

- Evaluation rules for function application (strict):

$$\frac{}{\texttt{(Apply (Recfun } \tau_1 \ \ \tau_2 \ \ (f.x.t)) \ v) \mapsto}$$

$$\frac{}{\texttt{(Apply } e_1 \ e_2) \mapsto}$$

$$\frac{}{\texttt{(Apply(Recfun ...) } e) \mapsto}$$

Utrecht University

Static and Dynamic Semantics

- MinHs is a type-safe (or strongly typed) language
- What exactly do we mean by this?
  - these terms are used by different authors to mean different things
  - in general, it refers to guarantees about the run-time behaviour derived from static properties of the program
    - Robin Milner: "*Well typed programs never go wrong*"
      - do not exhibit undefined behaviour
    - we define type safety to be the following two properties:
      - preservation
      - progress
    - we look at both preservation and progress in turn

Preservation and Progress

- Preservation:
  - Idea: evaluation does not change the type of an expression
  - Formally: If $\vdash e : \tau$ and $e \mapsto e'$, then $\vdash e' : \tau$

- Progress:
  - Idea: a well-defined program can not get stuck
  - Formally: If e is well typed, then either e is a final state, or there exists e', with $e \mapsto e'$

$$e : \tau \mapsto e_1 : \tau \mapsto e_2 : \tau \mapsto e_3 : \tau \mapsto e_4 : \tau \quad \ldots$$

- Together is means that a program will either evaluate to a value of the promised type, or run forever

Type Safety

- For any language to be type safe, the progress and preservation properties need to hold!
- Strictly speaking, the term type safety only makes sense in the context of a formal static and dynamic semantics
- This is one reason why formal methods in programming languages are essential
- The more expressive a type system is, the more information and assertions the type checker can derive at compile type
  - type systems usually should be decidable
  - but there are exceptions
- MinHs is type safe
  - we can show that progress and preservation hold!
  - but what if the language contains partial operations, like division?

Run-time Errors and Safety

- Stuck states: in a type safe language language, stuck states correspond to ill-defined programs, e.g.,
  - use (+) on values of function type, for example
  - treat an integer value as a pointer
  - use an integer as function

```
let x = 1 in x 5
```

- Unsafe languages/operations do not get stuck
  - something happens, but its not predictable and/or portable:

```
void boom () {
  void (f*)(void) = 0xdeadbeef;
  f ();
}
```

Run-time Errors and Safety

- How can we deal with **partial functions**, for example division by zero?

$$\frac{\Gamma \vdash t_1 : \texttt{Int} \qquad \Gamma \vdash t_2 : \texttt{Int}}{\Gamma \vdash (\texttt{Div}\ t_1\ t_2) : \texttt{Int}}$$

Problem: the expression `5/0` is well-typed, but does not evaluate to a value.

- Since this a mismatch between the static and dynamic semantics, there are two ways to fix this:

(1) Change static semantics: can we enhance the type system to check for division by zero?

- in general, such a type system would not be decidable
- there exist systems that approximate this

(2) Change dynamic semantics: can we modify the semantics such that the devision by zero does not lead to a stuck state

- this approach is widely used for type safe languages

Utrecht University

Run-time Errors and Safety

- Application of a partial function can yield `Error`

$$\frac{}{\texttt{Div}\ v\ (\texttt{Num}\ 0) \mapsto \texttt{Error}}$$

  - An `Error` interrupts any computation

$$\frac{}{\texttt{Plus Error}\ e \mapsto \texttt{Error}}$$

$$\frac{}{\texttt{Plus}\ e\ \texttt{Error} \mapsto \texttt{Error}}$$

$$\frac{}{\texttt{If Error}\ e_1\ e_2 \mapsto \texttt{Error}}$$

and so on.....

Run-time errors and Safety

- Typing the `Error` value:
  - a run-time error can have any type

$$\frac{}{\Gamma \vdash \texttt{Error} : \tau}$$

- What type of situations lead to checked run-time errors in Haskell?
  - could they be avoided?

Undefined behaviour

- Many languages have some undefined behaviour
  - indexing out of range
  - overflow/underflow
  - accessing uninitialised variables
  - order of evaluation of (stateful) subexpressions in an expression
- Why?
- In general, undefined behaviour is a correctness/safety/security problem