5CCS2PLD–Programming Language Design Paradigms

7–Functional Programming: Types and Semantics

7.1–Introduction to Types

Dr. Odinaldo Rodrigues and Prof. Maribel Fernández

odinaldo.rodrigues@kcl.ac.uk
Room BH(N) 7.08, +44 (0)20 7848 2087 (not in use)
Department of Informatics
King's College London

Outline

1. Typing
2. Polymorphism
3. Type inference

Typing

Introduction

Types help to ensure that a program behaves in a precise, pre-determined way.

Values are divided into classes, called *types*, and each type is associated with a set of operations.

Types may be given explicitly by the programmer or inferred automatically by the type system.

The following bit of code illustrates how types are explicitly given in a typical C++ program:

```
void wait (string message) {
  cout « message « endl;
  cout « "Press ENTER to continue..." « flush;
  cin.ignore (100, '\n');
}
```

Static vs. Dynamic Typing

Checking that types are being used in the correct way is sometimes referred to as *typing*.

Typing may be checked before execution (static typing) or during execution (dynamic typing)

- Statically typed languages include C, C++, Java and Haskell
- Dynamically typed languages include Python, JavaScript and PHP

Static typing

Every valid expression must have a type. Expressions that cannot be “typed” are considered erroneous and are rejected by the compiler without evaluation.

Advantages of statically typed languages:

- Types help detecting errors at an early stage.
- Types help in the design of software since they are a simple form of specification.

A program that passes the type controls is not guaranteed to be correct, but it is free of type errors at runtime.

Types in Haskell

The set of predefined types includes:

- Basic data types: Booleans, characters, numbers.
  For example, `Bool, Char, Int, Integer, Float, Double`
- Constructed types: tuples, lists, strings.
  For example, `[Integer]` is the type of lists of (arbitrary precision) integers, `(Int, Bool)` is a pair (binary tuple) of an integer and a Boolean
- Function types.
  For example, `Char` $\rightarrow$ `Bool`, `(Int` $\rightarrow$ `Int)` $\rightarrow$ `Int`
  (Note that the arrow type constructor associates to the right!)

The programmer can also define new types.

POLYMORPHISM

Polymorphism

Type systems can be:

- *monomorphic* where every expression has at most one type, or
- *polymorphic* where some expressions have more than one type.

EXAMPLE

Functional composition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

where `a`, `b` and `c` are *type variables*.

Polymorphic Types

Formally, the language of polymorphic types is defined as a set of *terms* built out of *type variables* (`a`, `b`, `c`, ...), and *type constructors* which are either atomic (`Integer`, `Float`, ...) or take arguments (e.g. $T_1 \to T_2$, $[T]$).

$$
\begin{array}{lcl}
V & ::= & \texttt{a}, \texttt{b}, \texttt{c} \ldots \\
C & ::= & \texttt{Int}, \texttt{Bool}, \texttt{Char}, \to, \texttt{[]}, \texttt{()} \ldots \\
T & ::= & V \mid C\,(T_1, T_2, \ldots T_n)
\end{array}
$$

A polymorphic type represents the set of its *instances*, obtained by substituting type variables by types.

Type variables

Type variables already exist in languages such as C++ to specify polymorphic types.

Here is an example:

```
template<typename A, typename B>
pair<B,A> flip_pair(const pair<A,B> &p) {
  return pair<B,A>(p.second, p.first);
};
```

Examples of polymorphism in Haskell

Type variables can be instantiated to different types:

EXAMPLE

From `square :: Integer → Integer` and `sqrt :: Integer → Float`, we can infer that `quad = square · square` is of type `Integer → Integer`, using the signature of `(.) :: (b -> c) -> (a -> b) -> (a -> c)`:

```
(·) :: (Integer → Integer) →
    (Integer → Integer) → (Integer → Integer)
```

But `sqrt · square` is of type `Integer → Float`:

```
(·) :: (Integer → Float) →
    (Integer → Integer) → (Integer → Float)
```

Examples of polymorphism

The `error` function is also polymorphic

```
error :: String → a
```

```
fact :: Integer -> Integer
fact n
    | n > 0   = n * (fact (n - 1))
    | n == 0  = 1
    | n < 0   = error "negative argument"
```

Here the function `error` is used with type `String → Integer`

Overloading

Overloading is a related notion (also called *ad-hoc polymorphism*) where several functions, with different types, share the same name.

This is useful when we want to use the same operation for distinct types (which may even "look" similar but may be represented differently internally).

How to define an operation for multiple types

In Haskell, we may want to specify a general operation such as addition for both integers or reals.

However, a polymorphic type such as

`(+) ::` $a \rightarrow a \rightarrow a$

is too general: because it would allow addition to be used with *any* type, e.g., characters.

Whereas `1 + 2` and `1.05 + 2.55` make sense, ‘`a`‘`+ ‘b‘` does not!

More on overloading

There are several solutions to this problem:

1. Some languages use different symbols/names for operations on specific types.
   - `+` and `+.` in Caml
   - `div 2 3` and `2 / 3` in Haskell
2. Enrich the language of types:
   For example, `(+) ::` $(\texttt{Integer} \to \texttt{Integer} \to \texttt{Integer}) \wedge (\texttt{Float} \to \texttt{Float} \to \texttt{Float})$
3. Define the notion of a *type class*, as in e.g., Haskell:
   `(+) ::` $\texttt{Num a} \Rightarrow \texttt{a} \to \texttt{a} \to \texttt{a}$,
   that is, `(+)` has type $\texttt{a} \to \texttt{a} \to \texttt{a}$ where `a` is in the class `Num`.

TYPE INFERENCE

Type inference

Most modern functional languages do not require that the programmer provides the type for the expressions used. The compiler is able to *infer* a type, if one exists.

- The type of a complex expression is inferred using the operations used in the expression and the type of its atomic constituents, assigning the most general type possible.
- The way that the different components are put together to form the expression indicates the constraints that the type variables must satisfy.
- If the constraints cannot be solved, the inference fails and the expression is deemed untypeable.

The type of an expression only depends on its components.

Examples of type inference

EXAMPLE

Consider the definition `square x = x * x`.

What is the type of the function `square`?

- It is a function, so the most general type is $a \to b$.
- Using `square`'s definition, `x :: a` and `x * x :: b`.
- We know that `(*) :: Int -> Int -> Int` is a primitive function.
- Therefore, if `x :: Int`, then `x * x :: Int`, and then both `a` and `b` must be of type `Int`.
- Therefore, `square :: Int -> Int`.

Examples of type inference

EXAMPLE

Can we find a more general type for `square x = x * x`?

We know that the most general type of `(*)` is `Num c => c -> c -> c`, so we also know that if `x :: Num c => c`, then `x * x :: Num c => c`.

We now have the constraints

`a=(Num c => c)` and `b=(Num c => c)` on the domain and co-domain of square, respectively, which we can solve by instantiating

`a` with `Num c => c` and `b` with `Num c => c`.

Therefore, `square :: Num c => c -> c`.

Examples of type inference

EXAMPLE

Given that `square :: Int -> Int`, what is the type of the expression

`square square 3`?

Remember that application associates to the left, so the expression reads `(square square) 3`.

So the first `square` expects an argument of type `Int`, but it is given an argument of type `(Int -> Int)` and the expression cannot be typed.

This is because the constraint `Int = (Int -> Int)` cannot be solved.

5CCS2PLD–Programming Language Design Paradigms

7–Functional Programming: Types and Semantics

7.2–Lists and User-defined types

Dr. Odinaldo Rodrigues and Prof. Maribel Fernández

odinaldo.rodrigues@kcl.ac.uk
Room BH(N) 7.08, +44 (0)20 7848 2087 (not in use)
Department of Informatics
King's College London

Outline

Lists

Lists

Linked lists are an important primitive data type in most functional languages.

- `[] :: [a]` — the empty list
- `(:) :: a -> [a] -> [a]` — add an element to a list (“cons”)
- `head :: [a] -> a` — return the first element of a non-empty list
- `tail :: [a] -> [a]` — return the remainder of a non-empty list

Lists are so common they have their own special syntax:

e.g. `(1 : 2 : 3 : [])` would be abbreviated to `[1, 2, 3]`.

Examples with lists

```
[] :: [a]
```

```
(1 : 2 : []) :: [Int]
```

or

```
[1, 2] :: [Int]
```

```
['c', 'a', 't'] :: [Char]
```

```
[3.0, 3.1, 3.14, 3.141, 3.1415] :: [Double]
```

```
[[True, False], [True]] :: [[Bool]]
```

Notice that strings are simply lists of characters! The expressions

```
"cat"==['c','a','t']
"cat"=='c':['a','t']
```

evaluate to `True`.

Functions on lists

Functions on lists and other constructed types are usually defined using pattern matching.

EXAMPLE

```
size :: [a] -> Int
size [] = 0
size (x : xs) = 1 + size xs


size [] → 0
size [1, 6, 1, 8, 0] → 1 + size [6, 1, 8, 0] →* 5
```

Exercise

Write the definition for a function `elem' x l` that checks whether an element `x` exists in a list `l`.

What is the type of `elem'`?

Exercise

Write the definition for a function `take' n l` that returns the first `n` elements of a list `l`.

What is the type of `take'`?

Sample Solution

```
take' n l :: Int -> [a] -> [a]

take' 0 l = []

take' n [] = []

take' n (x : xs) = x : (take' (n - 1) xs)
```

User-defined Types

User-defined types

We can define our own new types by declaring the *data* and *type* constructors of that type.

EXAMPLE

```
data Nat = Zero | Succ Nat
```

Here, `Nat` is a new type and `Zero` and `Succ` are its *data constructors*.

EXAMPLE

```
data Seq a = Empty | Cons a (Seq a)
```

Here `Seq a` is a new polymorphic, recursive type, whose elements are built using one of the data constructors `Empty` or `Cons`.

Type constructors define new polymorphic types. `Seq` is a type constructor.

More on user-defined types

Data constructors are used to build terms of a type.

EXAMPLE

- `Zero :: Nat`
- `(Succ (Succ Zero)) :: Nat`
- `Empty :: Seq a`
- `(Cons (Succ Zero) Empty) :: Seq Nat`

Data constructors can be used when giving definitions by pattern matching.

EXAMPLE

```
isempty Empty = True
isempty (Cons x y) = False
```

Exercise

- Write the definition for a data type `Tree a` representing binary trees with data stored in the leaves. Use data constructors `Leaf` and `Branch`.
- Using your newly created type, define a tree with two branches. The left branch should contain a tree composed by the single leaf with value 1 and the right branch should be composed by a tree with two leaves storing the values 2 and 3.
- Write a definition for a function `height t` which computes the height of a tree `t`, where the height of a leaf is 1.
  (Hint: you will need a local definition of a function `max x y`)
- What is the type of the function `height`?

Sample Solution

- ```
  data Tree a = Leaf a | Branch (Tree a) (Tree a)
  ```
- ```
  height :: Tree a -> Int
  height (Leaf _) = 1
  height (Branch l r) = 1 + max (height l) (height r)
     where max x y = if x > y then x else y
  ```

Sample use:

`height (Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)))` $\rightarrow^*$ `3`

Structural Induction

Structural Induction

To reason with recursive types we can use the *Principle of Structural Induction*.

EXAMPLE

In the case of `Nat`:

To prove a property P for all elements of `Nat` we have to

1. Prove $P(\texttt{Zero})$ — the *base case* of the induction.
2. Prove that if $P(n)$ holds — the *induction hypothesis*, then $P(\texttt{Succ}\ n)$ holds.

This is the familiar *Principle of Mathematical Induction:*

$$(P(0) \wedge \forall n.(P(n) \Rightarrow P(n+1))) \Leftrightarrow \forall n.P(n)$$

Example of induction

We can define addition on Nat by pattern matching:

```
add Zero x = x
add (Succ x) y = Succ (add x y)
```

and prove by induction that `Zero` is a neutral element, that is, for all Nat numbers *n*:

```
add n Zero = n
```

1. *Base case:* To prove `add Zero Zero = Zero` we use the definition of `add`.
2. *Induction:* By definition of `add`:
   `add (Succ n) Zero = Succ (add n Zero)`
   which is equal to `Succ n`, by the induction hypothesis.

Exercise

Consider the Haskell function `sumNat` introduced in a previous exercise:

```
sumNat x
  | x == 0 = 0
  | x > 0 = x + sumNat (x-1)
  | otherwise = error "Negative argument!"
```

Prove by induction that `sumNat n` $= \frac{n(n+1)}{2}$, for all natural numbers `n`.

Sample proof

To prove by induction, we need to establish that the property holds for the *base* case (i.e., the first valid argument 0), and that given that the property holds for some arbitrary value n, then it also holds for n's successor (i.e., for $n + 1$).

Base: `sumNat 0` $= \frac{0 \times 1}{2} =$ `0`.

Indeed, `0` is the value returned by `sumNat` for the argument 0.

Induction Hypothesis (IH): Assume that for a given argument `n`:

$$\texttt{sumNat n} = \frac{\texttt{n(n+1)}}{2}.$$

5CCS2PLD–Programming Language Design Paradigms

7–Functional Programming: Types and Semantics

7.3–Introduction to functional semantics

Dr. Odinaldo Rodrigues and Prof. Maribel Fernández

odinaldo.rodrigues@kcl.ac.uk
Room BH(N) 7.08, +44 (0)20 7848 2087 (not in use)
Department of Informatics
King's College London

Outline

INTRODUCTION

Introduction

We will give a precise, formal description of the behaviour of functional programs using *transition systems*.

The language that we consider is a small, first-order, functional programming language working on integers and Booleans, called SFUN.

We will define the syntax of SFUN, then give its semantics using the structural operational semantics approach.

Evaluation strategies

We can classify functional languages according to the evaluation strategy they implement.

- Call-by-value: argument expressions are evaluated before applying function definitions
- Call-by-name: function definitions are applied before evaluating argument expressions

First we will give a semantics of SFUN which corresponds to a call-by-value evaluation strategy, then we will show how the semantics has to be modified to model a call-by-name strategy.

Syntax of SFUN

Let $\mathcal{V}$ be a set of variables $\{x, y, z, \ldots\}$ and let $\mathcal{F}$ be a set of function names $\{f_1, f_2, \ldots f_i, \ldots\}$, each with a fixed arity $\langle f_i \rangle$.

The *terms* of the language SFUN are defined by the grammar:

$$op ::= + \mid - \mid * \mid / \qquad bop ::= > \mid < \mid = \mid \leq \mid \geq$$
$$t ::= n \mid b \mid x \mid t_1\ op\ t_2 \mid t_1\ bop\ t_2 \mid \neg t_1 \mid t_1 \wedge t_2$$
$$\mid \texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid f(t_1, \ldots, t_{\langle f \rangle})$$

- where n represents an integer value, $n \in \mathbb{Z}$
- and b represents a Boolean value, $b \in \{True, False\}$

Variables

The notation $vars(t)$ is used to denote the variables that occur in the term t.

EXAMPLE

- $vars(x) = \{x\}$
- $vars(f(y, z)) = \{y, z\}$.

A *closed term* is a term such that $vars(t) = \varnothing$, that is a term that contains no variables.

Programs in SFUN

A program in SFUN is a set of recursive equations:

$$\begin{aligned} f_1(x_1, \ldots, x_{\langle f_1 \rangle}) &= d_1 \\ &\vdots \\ f_k(x_1, \ldots, x_{\langle f_k \rangle}) &= d_k \end{aligned}$$

such that

- $d_1, \ldots, d_k$ are terms,
- $vars(d_i) \subseteq \{x_1, \ldots, x_{\langle f_i \rangle}\}, \forall i.\, 1 \leq i \leq k$,
- there is only one equation for each function name f_i.

Equations may be recursive, thus the terms d_i may contain occurrences of $f_1, \ldots, f_k$.

Examples of programs in SFUN

EXAMPLE

- $max(x, y) = \texttt{if}\ x \geq y\ \texttt{then}\ x\ \texttt{else}\ y$
  $fact(x) = \texttt{if}\ x \leq 0\ \texttt{then}\ 1\ \texttt{else}\ x * fact(x-1)$

- $square(x) = x * x$
  $quadratic(x, a, b, c) = a * square(x) + b * x + c$

- $mod(x, y) = \texttt{if}\ x - y < 0\ \texttt{then}\ x\ \texttt{else}\ mod(x-y, y)$
  $even(x) = mod(x, 2) = 0$
  $collatz(x) = \texttt{if}\ x = 1\ \texttt{then}\ 1\ \texttt{else if}\ even(x)\ \texttt{then}\ x/2\ \texttt{else}\ 3 * x + 1$

Operational Semantics

Operational Semantics of SFUN

We will now give the semantics of SFUN in the structural operational semantics style. We assume that programs are well-typed.

- We will define the *evaluation relation* (a big-step semantics) for terms in the context of the program P using a *transition system* where *configurations* are just terms.
- The *values* of the system are integer and Boolean values.
- The evaluation relation relates closed SFUN terms to values in the context of P, and is denoted by $\Downarrow_P$.
- The evaluation strategy that we will model first is a *call-by-value* strategy.

Call-by-value evaluation of SFUN

The evaluation relation $\Downarrow_P$ is defined inductively as follows:

$$\frac{}{n \Downarrow_P n}\ (\mathrm{n}) \qquad \frac{}{b \Downarrow_P b}\ (\mathrm{b})$$

$$\frac{t_1 \Downarrow_P n_1 \quad t_2 \Downarrow_P n_2}{t_1\ op\ t_2 \Downarrow_P n}\ (\mathrm{op})\ \mathit{if}\ n_1\ op\ n_2 = n \qquad \frac{t_1 \Downarrow_P n_1 \quad t_2 \Downarrow_P n_2}{t_1\ bop\ t_2 \Downarrow_P b}\ (\mathrm{bop})\ \mathit{if}\ n_1\ bop\ n_2 = b$$

$$\frac{t_1 \Downarrow_P b_1 \quad t_2 \Downarrow_P b_2}{t_1\ \wedge\ t_2 \Downarrow_P b}\ (\mathrm{and})\ \mathit{if}\ b_1 \wedge b_2 = b \qquad \frac{t \Downarrow_P b_1}{\neg t \Downarrow_P b}\ (\mathrm{not})\ \mathit{if}\ b = \neg b_1$$

$$\frac{t_0 \Downarrow_P \mathit{True} \quad t_1 \Downarrow_P v_1}{\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \Downarrow_P v_1}\ (\mathrm{if_t}) \qquad \frac{t_0 \Downarrow_P \mathit{False} \quad t_2 \Downarrow_P v_2}{\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \Downarrow_P v_2}\ (\mathrm{if_f})$$

$$\frac{t_1 \Downarrow_P v_1 \quad \ldots \quad t_{\langle f_i \rangle} \Downarrow_P v_{\langle f_i \rangle} \quad d_i\{x_1 \mapsto v_1, \ldots, x_{\langle f_i \rangle} \mapsto v_{\langle f_i \rangle}\} \Downarrow_P v}{f_i(t_1, \ldots, t_{\langle f_i \rangle}) \Downarrow_P v}\ (\mathrm{fn_{val}})$$

Examples of call-by-value evaluation

Let P be the program:

$$\mathit{infinity} = \mathit{infinity} + 1$$
$$\mathit{fortytwo}(x) = 42$$
$$\mathit{square}(x) = x * x$$

The term $\mathit{fortytwo}(0)$ has the value 42, that is, $\mathit{fortytwo}(0) \Downarrow_P 42$.

Proof.

$$\dfrac{\dfrac{}{0 \Downarrow_P 0}\,(\mathrm{n}) \qquad \dfrac{}{42\{x \mapsto 0\} \Downarrow_P 42}\,(\mathrm{n})}{\mathit{fortytwo}(0) \Downarrow_P 42}\,(\mathrm{fn_{val}})$$

□

More examples of call-by-value evaluation

- The term *fortytwo*(*infinity*) does not have a value, because the evaluation of the argument *infinity* gives no value. A derivation for *infinity* cannot be constructed because it recurses infinitely on the rule (fn_{val}).
- The term *square*$(2+1)$ has the value 9, *square*$(2+1) \Downarrow_P 9$.

Proof.

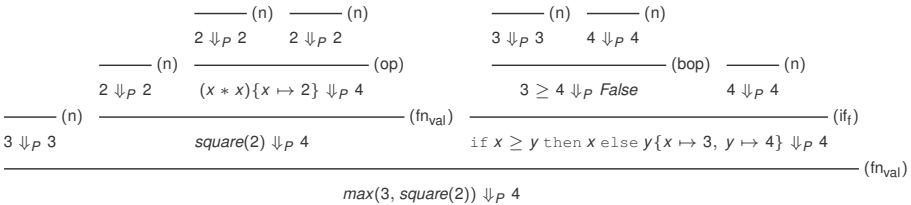
$$\dfrac{\dfrac{\dfrac{}{2 \Downarrow_P 2}\,(\text{n}) \quad \dfrac{}{1 \Downarrow_P 1}\,(\text{n})}{2+1 \Downarrow_P 3}\,(\text{op}) \quad \dfrac{\dfrac{}{3 \Downarrow_P 3}\,(\text{n}) \quad \dfrac{}{3 \Downarrow_P 3}\,(\text{n})}{(x*x)\{x \mapsto 3\} \Downarrow_P 9}\,(\text{op})}{\textit{square}(2+1) \Downarrow_P 9}\,(\text{fn}_{\text{val}})$$

□

Exercise

Give the call-by-value derivation for the evaluation of the term $max(3, square(2))$.

Solution


(n)
3 ⇓P 3
(n)
2 ⇓P 2
(n)
2 ⇓P 2
(op)
(x ∗ x){x ↦ 2} ⇓P 4
(fnval)
square(2) ⇓P 4
(n)
3 ⇓P 3
(n)
4 ⇓P 4
(bop)
3 ≥ 4 ⇓P False
(n)
4 ⇓P 4
(ift)
if x ≥ y then x else y{x ↦ 3, y ↦ 4} ⇓P 4
(fnval)
max(3, square(2)) ⇓P 4

Unicity of Normal Forms

Determinism: unicity of normal forms

The evaluation strategy defined by the axioms and rules is deterministic, that is:

Theorem

For any closed term t, $t \Downarrow_P v_1$ and $t \Downarrow_P v_2$ implies $v_1 = v_2$.

Proof.

By rule induction. □

Details of proof of determinism

We distinguish cases according to the rule that applies to t. For any term, there is only one rule that can be applied.

Base cases (axioms):

- If t is a integer n then $n \Downarrow_P n$ using the axiom (n)
- If t is a Boolean b then $b \Downarrow_P b$ using the axiom (b)

Therefore there is only one value in both cases.

There are no more base cases, because t is closed and thus cannot contain variables.

More details of proof of determinism

Inductive cases (rules):

- Assume t is the term $f(t_1, \ldots, t_{\langle f \rangle})$.
- Then, using the rule (fn_{val}), $f_i(t_1, \ldots, t_{\langle f_i \rangle}) \Downarrow_P v$ if and only if $t_1 \Downarrow_P v_1, \ldots, t_{\langle f_i \rangle} \Downarrow_P v_{\langle f_i \rangle}$, and $d_i\{x_1 \mapsto v_1, \ldots, x_{\langle f_i \rangle} \mapsto v_{\langle f_i \rangle}\} \Downarrow_P v$
- By the induction hypothesis, there is at most one value for each term $t_1, \ldots, t_n$ and $d_i\{x_1 \mapsto v_1, \ldots, x_{\langle f_i \rangle} \mapsto v_{\langle f_i \rangle}\}$.
- Therefore v is uniquely determined.

The cases corresponding to the other rules are similar.

5CCS2PLD–Programming Language Design Paradigms

7–Functional Programming: Types and Semantics

7.4–Call-by-name, types and typing

Dr. Odinaldo Rodrigues and Prof. Maribel Fernández

odinaldo.rodrigues@kcl.ac.uk
Room BH(N) 7.08, +44 (0)20 7848 2087 (not in use)
Department of Informatics
King's College London

Outline

1. Call-by-name evaluation
2. Types
3. Typing

CALL-BY-NAME EVALUATION

Recalling the call-by-*value* evaluation rule

We have seen that terms involving a function application were evaluated using the call-by-value rule (fn_{val}) below:

$$\frac{t_1 \Downarrow_P v_1 \quad \ldots \quad t_{\langle f_i \rangle} \Downarrow_P v_{\langle f_i \rangle} \quad d_i\{x_1 \mapsto v_1, \ldots, x_{\langle f_i \rangle} \mapsto v_{\langle f_i \rangle}\} \Downarrow_P v}{f_i(t_1, \ldots, t_{\langle f_i \rangle}) \Downarrow_P v} \ (\text{fn}_{\text{val}})$$

This rules requires that the arguments given to the function are evaluated before the function is applied. Consider the functions in the program P:

$$\mathit{infinity} = \mathit{infinity} + 1$$
$$\mathit{fortytwo}(x) = 42$$

The use of fn_{val} can become problematic or inefficient in the evaluation of terms such as $\mathit{fortytwo}(\mathit{infinity})$ or $\mathit{fortytwo}(2^{3542})$.

Call-by-name evaluation of SFUN

In order to model the call-by-name strategy we need to change the rule that defines the behaviour of application.

We *replace it* with the following rule:

$$\frac{d_i\{x_1 \mapsto t_1, \ldots, x_{\langle f_i \rangle} \mapsto t_{\langle f_i \rangle}\} \Downarrow_P v}{f_i(t_1, \ldots, t_{\langle f_i \rangle}) \Downarrow_P v}\ (\mathrm{fn}_{\mathrm{name}})$$

The system remains deterministic:

Theorem

For any closed term t, if $t \Downarrow_P v_1$ and $t \Downarrow_P v_2$ then $v_1 = v_2$.

Proof.

By rule induction. □

Examples of call-by-name evaluation

Recall the program P:

$$\mathit{infinity} = \mathit{infinity} + 1$$
$$\mathit{fortytwo}(x) = 42$$
$$\mathit{square}(x) = x * x$$

Using the call-by-name semantics the term $\mathit{fortytwo}(0)$ also has the value 42, $\mathit{fortytwo}(0) \Downarrow_P 42$.

However in contrast to call-by-value, $\mathit{fortytwo}(\mathit{infinity}) \Downarrow_P 42$, because the argument *infinity* is discarded without being evaluated.

$$\dfrac{\dfrac{}{42\{x \mapsto \mathit{infinity}\} \Downarrow_P 42}\,(\mathrm{n})}{\mathit{fortytwo}(\mathit{infinity}) \Downarrow_P 42}\,(\mathrm{fn}_{\mathrm{name}})$$

More examples of call-by-name evaluation

EXAMPLE

The term $square(2+1)$ also has the value 9, $square(2+1) \Downarrow_P 9$, but note the different derivation and repeated computations in comparison to the call-by-value semantics.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{2 \Downarrow_P 2}\,(\mathrm{n}) \quad \dfrac{}{1 \Downarrow_P 1}\,(\mathrm{n})}{2+1 \Downarrow_P 3}\,(\mathrm{op}) \quad \dfrac{\dfrac{}{2 \Downarrow_P 2}\,(\mathrm{n}) \quad \dfrac{}{1 \Downarrow_P 1}\,(\mathrm{n})}{2+1 \Downarrow_P 3}\,(\mathrm{op})}{(x * x)\{x \mapsto 2+1\} \Downarrow_P 9}\,(\mathrm{op})}{square(2+1) \Downarrow_P 9}\,(\mathrm{fn}_{\mathrm{name}})$$

Exercise

Give the call-by-name derivation for the evaluation of the term $max(3, square(2))$.

Solution

$$\dfrac{\dfrac{}{3 \Downarrow_P 3}\,(\mathrm{n}) \qquad \dfrac{\dfrac{\dfrac{}{2 \Downarrow_P 2}\,(\mathrm{n}) \quad \dfrac{}{2 \Downarrow_P 2}\,(\mathrm{n})}{(x * x)\{x \mapsto 2\} \Downarrow_P 4}}{square(2) \Downarrow_P 4}}{3 \geq (square(2)) \Downarrow_P False}\,(\mathrm{bop}) \qquad \dfrac{\dfrac{\dfrac{}{2 \Downarrow_P 2}\,(\mathrm{n}) \quad \dfrac{}{2 \Downarrow_P 2}\,(\mathrm{n})}{(x * x)\{x \mapsto 2\} \Downarrow_P 4}\,(\mathrm{op})}{square(2) \Downarrow_P 4}\,(\mathrm{fn_{name}})$$

$$\dfrac{}{\texttt{if } x \geq y \texttt{ then } x \texttt{ else } y\{x \mapsto 3,\ y \mapsto square(2)\} \Downarrow_P 4}\,(\mathrm{if_f})$$

$$\dfrac{}{max(3, square(2)) \Downarrow_P 4}\,(\mathrm{fn_{name}})$$

Types

Types for SFUN

The grammar defining the syntax of SFUN allows us to build terms such as $1 + \mathit{True}$ which do not make sense.

In the definition of the semantics of SFUN we only considered *well-typed* terms.

The *base types*, β, and the *types* τ of SFUN are defined as follows:

$$\beta ::= \mathsf{int} \mid \mathsf{bool}$$
$$\tau ::= (\beta_1, \ldots, \beta_n) \rightarrow \beta$$

In the case of a type τ such that $n = 0$ (the type of a function with no arguments), τ may be written simply as β.

Well-typed terms in SFUN

The set of well-typed terms can be defined using a relation $\Gamma \vdash_\varepsilon t : \tau$.

- Γ is a *variable environment*, or simply *environment*, given as a finite partial function from variables to base types
- ε is a *function environment* assigning to each function name a type respecting its arity: that is, if $\langle f \rangle = 0$ then $\varepsilon(f) = \beta$ and if $\langle f \rangle = n$ where $n \geq 1$, then $\varepsilon(f) = (\beta_1, \ldots, \beta_n) \to \beta$
- t is a SFUN term
- τ is a type

The relation $\Gamma \vdash_\varepsilon t : \tau$ can be read:

"If the variable x has type $\Gamma(x)$ for each $x \in dom(\Gamma)$ and the functions $f_1, \ldots, f_k$ have types $\varepsilon(f_1), \ldots, \varepsilon(f_k)$ then the term t has type τ."

Typing

Typing SFUN terms

The well-typedness relation is inductively defined by the following system of axioms and rules. Note that in the case of the application of a function f, if $\langle f \rangle = 0$ then the rule reduces to an axiom, because there are no arguments to check.

$$\frac{}{\Gamma \vdash_\varepsilon b : \text{bool}} \qquad \frac{}{\Gamma \vdash_\varepsilon n : \text{int}} \qquad \frac{}{\Gamma \vdash_\varepsilon x : \beta} \; \textit{if } \Gamma(x) = \beta$$

$$\frac{\Gamma \vdash_\varepsilon t_1 : \text{int} \quad \Gamma \vdash_\varepsilon t_2 : \text{int}}{\Gamma \vdash_\varepsilon t_1 \; op \; t_2 : \text{int}} \qquad \frac{\Gamma \vdash_\varepsilon t_1 : \text{int} \quad \Gamma \vdash_\varepsilon t_2 : \text{int}}{\Gamma \vdash_\varepsilon t_1 \; bop \; t_2 : \text{bool}} \qquad \frac{\Gamma \vdash_\varepsilon t : \text{bool}}{\Gamma \vdash_\varepsilon \neg t : \text{bool}}$$

$$\frac{\Gamma \vdash_\varepsilon t_1 : \text{bool} \quad \Gamma \vdash_\varepsilon t_2 : \text{bool}}{\Gamma \vdash_\varepsilon t_1 \wedge t_2 : \text{bool}} \qquad \frac{\Gamma \vdash_\varepsilon t_0 : \text{bool} \quad \Gamma \vdash_\varepsilon t_1 : \tau \quad \Gamma \vdash_\varepsilon t_2 : \tau}{\Gamma \vdash_\varepsilon \texttt{if } t_0 \texttt{ then } t_1 \texttt{ else } t_2 : \tau}$$

$$\frac{\Gamma \vdash_\varepsilon t_1 : \beta_1 \quad \ldots \quad \Gamma \vdash_\varepsilon t_{\langle f \rangle} : \beta_{\langle f \rangle}}{\Gamma \vdash_\varepsilon f(t_1, \ldots, t_{\langle f \rangle}) : \beta} \; \textit{if } \varepsilon(f) = (\beta_1, \ldots, \beta_{\langle f \rangle}) \to \beta$$

Example of typing SFUN terms

EXAMPLE

Given that $\Gamma(x) = \text{int}$ and $\varepsilon(\mathit{fact}) = (\text{int}) \to \text{int}$ give the derivation for the typing of term $\texttt{if}\ x \leq 0\ \texttt{then}\ 1\ \texttt{else}\ x * \mathit{fact}(x-1)$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_\varepsilon x : \text{int}} \quad \dfrac{}{\Gamma \vdash_\varepsilon 0 : \text{int}}}{\Gamma \vdash_\varepsilon x \leq 0 : \text{bool}} \quad \dfrac{}{\Gamma \vdash_\varepsilon 1 : \text{int}} \quad \dfrac{\dfrac{}{\Gamma \vdash_\varepsilon x : \text{int}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash_\varepsilon x : \text{int}} \quad \dfrac{}{\Gamma \vdash_\varepsilon 1 : \text{int}}}{\Gamma \vdash_\varepsilon x - 1 : \text{int}}}{\Gamma \vdash_\varepsilon \mathit{fact}(x-1) : \text{int}}}{\Gamma \vdash_\varepsilon x * \mathit{fact}(x-1) : \text{int}}}{\Gamma \vdash_\varepsilon \texttt{if}\ x \leq 0\ \texttt{then}\ 1\ \texttt{else}\ x * \mathit{fact}(x-1) : \text{int}}$$

Exercise

Given that $\varepsilon(max) = (\text{int}, \text{int}) \rightarrow \text{int}$ and $\varepsilon(square) = (\text{int}) \rightarrow \text{int}$, give the derivation for the typing of term $max(3, square(2))$.

Solution

Given that

$$\varepsilon(\mathit{max}) = (\text{int}, \text{int}) \to \text{int} \text{ and } \varepsilon(\mathit{square}) = (\text{int}) \to \text{int}.$$

We have that:

$$\dfrac{\dfrac{}{\Gamma \vdash_\varepsilon 3 : \text{int}} \quad \dfrac{\dfrac{}{\Gamma \vdash_\varepsilon 2 : \text{int}}}{\Gamma \vdash_\varepsilon \mathit{square}(2) : \text{int}}}{\Gamma \vdash_\varepsilon \mathit{max}(3, \mathit{square}(2)) : \text{int}}$$

Typing SFUN programs

Given a program P in SFUN

$$\begin{aligned} f_1(x_1, \ldots, x_{\langle f_1 \rangle}) &= t_1 \\ &\vdots \\ f_k(x_1, \ldots, x_{\langle f_k \rangle}) &= t_k \end{aligned}$$

and a function environment ε, P is typeable, if for each equation $f_i(x_1, \ldots, x_{\langle f_i \rangle}) = t_i$, there exists an environment Γ_i and a type τ_i, such that

- $\Gamma_i \vdash_\varepsilon f_i(x_1, \ldots, x_{\langle f_i \rangle}) : \tau_i$
- $\Gamma_i \vdash_\varepsilon t_i : \tau_i$

Example of typing SFUN programs

Recall again the program P of the previous examples:

$$\begin{aligned} \mathit{infinity} &= \mathit{infinity} + 1 \\ \mathit{fortytwo}(x) &= 42 \\ \mathit{square}(x) &= x * x \end{aligned}$$

EXAMPLE

Given a function environment ε where:

- $\varepsilon(\mathit{infinity}) = \text{int}$
- $\varepsilon(\mathit{fortytwo}) = (\text{int}) \rightarrow \text{int}$
- $\varepsilon(\mathit{square}) = (\text{int}) \rightarrow \text{int}$

P is a typeable program. All the equations are typeable with type int, the first in an empty environment and the latter two using an environment Γ, such that $\Gamma(x) = \text{int}$.

Proving properties of SFUN programs

We can use proof by induction on SFUN programs directly.

Given that $fact(x) = \texttt{if}\ x \leq 0\ \texttt{then}\ 1\ \texttt{else}\ x * fact(x-1)$, prove that fact(n) = n! for all natural numbers n.

Proof by induction:

Base case ($n = 0$): We have to show that $fact(0) = 0! = 1$.

By the definition of $fact(x)$, $0 \leq 0$ evaluates to *True*, and then by the left-hand side of the conditional, $fact(0) = 1$, as expected.

Proving properties of SFUN programs (cont.)

Induction hypothesis: Assume $fact(n) = n!$

Induction step: Show that $fact(n+1) = (n+1)!$

$n+1 \leq 0$ will evaluate to *False*. By the right-hand side of the conditional, $fact(n+1) = (n+1) * fact((n+1)-1)$ which is equal to $(n+1) * fact(n)$. By the inductive hypothesis, this is equal to $(n+1) * n!$, which is simply $(n+1)!$