Total Functions: Why and How

Code as Relation Can Be Non-Deterministic

If R is meaning of a piece of code, it can generally be non-deterministic.

+ it is easy to define approximate relation and construct fixedpoint: replace recursive call at some depth with $\emptyset$
+ if we cannot represent some part of code, we model it as full relation (havoc everything)
- if the code is deterministic, we do not automatically have this information

Example: hash table with one static array (linear probing for simplicity).

```
var code = hash(key)
var i = 0
while(table(code + i).key != key && table(code + i) != None && i < max) {
  i = i + 1
}
if (table(code + i).key == key)
  Some(table(code + i).value)
else
  None()
```

`https://github.com/epfl-lara/bolts/tree/master/data-structures/maps`

Hash Code Must be Non-Deterministic

Why are hash tables efficient?

- Lookup only starts from hash(key)
- Do not check all array entries

If we do not find it looking starting from a given place, we conclude it is not there.

Relies on the following invariant: for every key that is in the table, it can be found by searching from the unique value of hash(key).

If we can have, for $code_1 \neq code_2$,

$$(key, code_1) \in hash, \quad (key, code_2) \in hash$$

then this invariant cannot be maintained meaningfully.

Total Functions

What if instead of relations we use *total functions* as the primary model for code?

- even our interpreter for non-deterministic concurrent program example was described using total functions, so this model can represent mathematical reasoning about non-deterministic programs thanks to extra arguments (such as schedule)
- total functions are the meaning of function symbols in first-order logic
- they are also the choice in ACL2 and Isabelle/HOL proof frameworks

Fixedpoints and Total Functions?

```
def f =
  if (x > 0) {
    x = x − 1
    f
  }
```

$$E(r) = \Big(\Delta_{x \tilde{>} 0} \circ \big(\rho(x = x-1) \circ r \big) \cup \Delta_{x \tilde{\leq} 0}$$

No guarantee that this is a total function. Indeed, when program does not terminate, the relation does not correspond to a total function.

Domain theory: Add "bottom" elements to the domains to turn partial functions into total functions extra elements: we have a function, but if it the type is Int, we model it mathematically as the set $\mathbb{Z} \cup \{\bot\}$.
Bottom represents non-termination. Then we can define partial order and fixedpoints.
Downside: we lose many properties of integers. Everything is possibly $\bot$. There is mathematical equality that compares also $\bot$ and the computable equality, which returns an element of $\{true, false, \bot\}$. We choose not to go there (see the original Edinburgh LCF prover).

Function Definitions as Axioms

```
def f(x: BigInt) =
  if (x == 0) 0
  else f(x − 1)
```

$$\forall x. f(x) = \mathit{if}\ (x = 0)\ \mathit{then}\ 0\ \mathit{else}\ f(x-1)$$

Very appealing: we have function symbol f that represents total function and it is always assumed to be integer.

What sort of total functions satisfy the above axiom?

Function Definitions as Axioms

```
def f(x: BigInt) =
  if (x == 0) 0
  else f(x − 1)
```

$$\forall x. f(x) = \mathit{if}\ (x = 0)\ \mathit{then}\ 0\ \mathit{else}\ f(x - 1)$$

Very appealing: we have function symbol f that represents total function and it is always assumed to be integer.

What sort of total functions satisfy the above axiom?

The above axiom constrains f to be equal to zero on non-negative integers and to be equal to some other constants on negative ones (it does not constraint the constant).

Function Definitions as Axioms

```
def f(x: BigInt) =
  if (x == 0) 0
  else f(x - 1)
```

$$\forall x. f(x) = \textit{if } (x = 0) \textit{ then } 0 \textit{ else } f(x-1)$$

Very appealing: we have function symbol f that represents total function and it is always assumed to be integer.

What sort of total functions satisfy the above axiom?

The above axiom constrains f to be equal to zero on non-negative integers and to be equal to some other constants on negative ones (it does not constraint the constant).

What would be the axiom for def g(x:BigInt):BigInt = 1 + g(x)

Function Definitions as Axioms

```
def f(x: BigInt) =
  if (x == 0) 0
  else f(x − 1)
```

$$\forall x. f(x) = \mathit{if}\ (x = 0)\ \mathit{then}\ 0\ \mathit{else}\ f(x - 1)$$

Very appealing: we have function symbol f that represents total function and it is always assumed to be integer.

What sort of total functions satisfy the above axiom?

The above axiom constrains f to be equal to zero on non-negative integers and to be equal to some other constants on negative ones (it does not constraint the constant).

What would be the axiom for def g(x:BigInt):BigInt = 1 + g(x)
From $g(0) = 1 + g(0)$, we get $0 = 1$ and we can prove anything.

Termination

Most expressive proof frameworks (ACL2, Isabelle/HOL, Coq, Stainless):

- Recognize that it is extremely helpful to have total functions
- Have as one of the basic concepts a total recursive function and for which we can use function definition as an axiom.
- Such recursive functions must be proven terminating!

Proving Termination

Each function execution yields a trace as a map from natural numbers to expressions that need to be evaluated (given by call by value operational semantics of a language with recursive functions):

$$e_1 \rightsquigarrow e_2 \rightsquigarrow \ldots e_3 \rightsquigarrow$$

Termination: for every input x, sequence where e_1 is $f(x)$ is finite.

One proof method: ensure "inner" recursive call to $f(x_1)$ always uses arguments x_2 such that $x_1 > x_2$ for some well-founded $>$.

Proving Termination

Each function execution yields a trace as a map from natural numbers to expressions that need to be evaluated (given by call by value operational semantics of a language with recursive functions):

$$e_1 \leadsto e_2 \leadsto \ldots e_3 \leadsto$$

Termination: for every input x, sequence where e_1 is $f(x)$ is finite.

One proof method: ensure "inner" recursive call to $f(x_1)$ always uses arguments x_2 such that $x_1 > x_2$ for some well-founded $>$. What does "inner" mean?

- not every call is evaluated, depends on conditionals
- recursive calls can be nested as in $f(t_1(f(t_2)))$; to know the argument of outer call we need to know the value computed for $f(t_2)$.

Lee, Jones, Ben-Amram. The size-change principle for program termination, POPL'01

- identify well-founded relation $>$ on the domain of arguments
- determine for each call if arguments must be smaller or are possibly equal
- analyze the resulting graph and show every cycle must $>$-decrease the argument

Well Founded Relations

Given an arbitrary set D, relation $(D,>)$ is well founded if there is no infinite sequence (map from natural numbers i to elements x_i) such that $x_i > x_{i+1}$ for every i.

Such infinite sequence is called descending chain (descending = taking a $>$ step).

Observe: well-founded relation is irreflexive. It has no finite cycles. Its transitive closure is also irreflexive.

Absence of finite cycles is not sufficient for a relation to be well founded: we can have relation $0, 2, 4, 6, \ldots$.

Trivial but useful fact: given any total function $m : A \rightarrow D$ where $(D,>)$ is well founded, then relation $a_1 \succ a_2$ defined by $m(a_1) > m(a_2)$ is a well founded relation on A. We call m a *measure*.

For us of interest are well-founded relations on countable sets. There are many!

Examples of Well-Founded Relations

Set of non-negative natural numbers with usual > relation. $1000 > 20 > 3 > 1 > 0$

Examples of Well-Founded Relations

Set of non-negative natural numbers with usual > relation. $1000 > 20 > 3 > 1 > 0$

Set of pairs (p,q) for $p,q \geq 0$ positive integers, with lexicographical order relation: $(p,q) > (p',q')$ iff

$$p > p' \vee (p = p' \wedge q > q')$$

Example: $(100,2) > (100,1) > (99,123456) > (99,123455) > \dots$

Examples of Well-Founded Relations

Set of non-negative natural numbers with usual > relation. $1000 > 20 > 3 > 1 > 0$

Set of pairs (p,q) for $p,q \geq 0$ positive integers, with lexicographical order relation:
$(p,q) > (p',q')$ iff

$$p > p' \vee (p = p' \wedge q > q')$$

Example: $(100,2) > (100,1) > (99,123456) > (99,123455) > \ldots$
Isomorphic copy into rationals: map (p,q) into $p-1/q$.
Example becomes: $100-1/2 > 100-1/1 > 99-1/123456 > 99-1/123455 > \ldots$

Examples of Well-Founded Relations

Set of non-negative natural numbers with usual > relation. $1000 > 20 > 3 > 1 > 0$

Set of pairs (p,q) for $p,q \geq 0$ positive integers, with lexicographical order relation: $(p,q) > (p',q')$ iff

$$p > p' \vee (p = p' \wedge q > q')$$

Example: $(100,2) > (100,1) > (99,123456) > (99,123455) > \dots$
Isomorphic copy into rationals: map (p,q) into $p-1/q$.
Example becomes: $100-1/2 > 100-1/1 > 99-1/123456 > 99-1/123455 > \dots$

Take an arbitrary set S of rational numbers of the form $p-1/q$. Does this set contain the least element a such that $\forall x \in S.(x > a \vee x = a)$?

Well-Ordered Set

Assume an irreflexive *total* order $(D,>)$ (every two elements comparable).
Define $x < y$ by $y > x$. Define $x \leq y$ as $(x < y \vee x = y)$.

We say that $(D,>)$ structure is a *well-order* iff for every non-empty set $S \subseteq D$, there exists an element $a \in S$ such that $\forall x \in S.\ a \leq x$.
(a is the least element of S)

Lemma: $(D,>)$ is well founded iff it is a well order.
Proof sketch (using axiom of choice): negate both side of the equivalence.
If $(D,>)$ is not well founded, it has an infinite descending chain $x_1 > x_2 > \ldots$. The elements in this chain form a set, and this set does not have the least element (given any $a = x_i$, then a is not the least because $x_i > x_{i+1}$).
Conversely, let S be a non-empty set that does not have the least element. Take any element $x_1 \in S$. Since x_1 in particular is not the least element, there exists x_2 so that $\neg(x_1 \leq x_2)$; thus $x_1 > x_2$. Repeating this process we construct an infinite descending chain.

Termination Methodology

Given a recursive function *def* $f(x)$, to show it is terminating, show that that the recursive calls, $f(t)$ we have satisfy $x > t$ according to well-founded relation $>$.

Two parts of the problem:

- having a library of well-founded relations or methods to show relations are well founded
- for a given recursive definition, finding a mapping (measure) into a well founded relation that proves its termination

Decreases clause in stainless: one approach to finding mapping (to lexicographic orders)

Primitive Recursion

If a function is on non-negative natural numbers and decreases one of its arguments, it is terminating.

If we use such function later to define subsequent functions that decrease one of the arguments, we again get terminating functions.

In this way, we obtain essentially primitive recursive functions. Example: $2^{2^{2^n}}$

Functions defined from common arithmetic primitives but that produce very large results are often not primitive recursive and are more difficult to prove terminating (they are "approaching" $f(x) = 1 + f(x)$).

Demo: Some Primitive Recursive Functions

```
def times(m: BigInt, n: BigInt): BigInt = {
  require(m >= 0 && n >= 0)
  decreases(n)

  if n == 0 then BigInt(0)
  else m + times(m, n − 1)
} ensuring(res => res >= 0 && res == m*n)

def power(m: BigInt, n: BigInt): BigInt = {
  require(m >= 0 && n >= 0)
  decreases(n)

  if n == 0 then BigInt(1)
  else times(m, power(m, n − 1))
} ensuring(_ >= 0)
```

Demo: Some Primitive Recursive Functions

```
def power(m: BigInt, n: BigInt): BigInt = {
  require(m >= 0 && n >= 0)
  decreases(n)

  if n == 0 then BigInt(1)
  else times(m, power(m, n - 1))
} ensuring(_ >= 0)

def tower(m: BigInt, n: BigInt): BigInt = {
  require(m >= 0 && n >= 0)
  decreases(n)

  if n == 0 then BigInt(1)
  else power(m, tower(m, n - 1))
} ensuring(_ >= 0)
```

Ackermann Function

```
def a(m: BigInt, n: BigInt, op: BigInt): BigInt = {
  require(m >= 1 && n >= 1 && op >= 1)
  decreases(op, n)

  if op == 1 then m + n
  else if n == 1 then m
  else a(m, a(m, n − 1, op), op−1)
} ensuring(res =>
    res >= 1
```

Ackermann Function

```
def a(m: BigInt, n: BigInt, op: BigInt): BigInt = {
  require(m >= 1 && n >= 1 && op >= 1)
  decreases(op, n)

  if op == 1 then m + n
  else if n == 1 then m
  else a(m, a(m, n - 1, op), op-1)
} ensuring(res =>
    res >= 1
    && ((op != 1) || res == m + n)
    && ((op != 2) || res == times(m, n))
    && ((op != 3) || res == power(m, n))
    && ((op != 4) || res == tower(m, n))
  )
```

Function $f(n) = a(n, n, n)$ grows faster than any primitive recursive function. Provably terminating with lexicographic order. Can use as a measure for other functions.

How to Model Possibly Non-terminating Functions?

```
def f(x: BigInt): BigInt = {
  1 + f(x)
} ensuring(res => post(res))
```

How to Model Possibly Non-terminating Functions?

```
def f(x: BigInt): BigInt = {
  1 + f(x)
} ensuring(res => post(res))
```

```
Warning ] Nonterm.scala:3:7: Function f loops given inputs:
[Warning ] x: BigInt −> x
                def f(x: BigInt): BigInt = {
```

Counter for “Time Credits”

```
sealed case class Credit(var n: BigInt) {
  require(n >= 0)

  def use: Unit =
    require(n > 0)
    n = n - 1
}
```

Proceed Only If Credits Left

```
def f(x: BigInt): BigInt = {
  1 + f(x)
} ensuring(res => post(res))
```

$\Downarrow$ transform automatically

```
def f(x: BigInt)(c: Credit): Option[BigInt] = {
  decreases(c.n)
  val op1 = BigInt(1)
  if c.n <= 0 then return None[BigInt]()
  c.use
  f(x)(c) match
    case None() => None[BigInt]()
    case Some(op2) =>
      val tmp1 = op1 + op2
      if c.n <= 0 then return None[BigInt]() else c.use
      Some(op1 + op2)
} ensuring(res => res == None[BigInt]() || post(res.get))
```