CDM

Loop Programs

Klaus Sutner

Carnegie Mellon University

Program Equivalence

Primitive recursive functions are easily computable, at least as a matter of principle. But obviously they are quite far removed from anything resembling practically computable functions – the time and space requirements are too high to allow for execution on any physically realistic machine.

Another issue is that it is very difficult to verify properties of primitive recursive functions.

For example, it is undecidable whether two primitive recursive programs P_1 and P_2 (of the same arity) determine the same function:

$$[\![P_1]\!] = [\![P_2]\!]$$

Why should one care?

Program Transformations

It is a standard technique to optimize programs by applying certain (presumably admissible) transformations to the source code. For example, index manipulations for array variables within loops are a good candidate for optimization.

If the optimizations are done by hand by a programmer it would be most desirable to be able to verify their correctness: the resulting program should have the same input/output behavior/

Of course, it would also be nice to be able to show some kind of optimality, but we'll ignore this issue.

So the question is: are there any interesting classes of computable functions for which the Equivalence Problem is decidable?

We need to scale back significantly from primitive recursive functions.

Rudimentary Functions

Here is another clone of simple number-theoretic functions $\mathbb{N}^n \to \mathbb{N}$. Compared to primitive recursive functions, we allow more basic functions but require no admissible operations other than just composition.

Definition

A function is rudimentary if it is a member of the least clone containing constants 0, 1, addition, predecessor, division and remainder with fixed modulus, and an if-then function.

Predecessor means

$$\pi(x) = \begin{cases} x - 1 & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

The Clone

By an if-then function we mean the following:

$$W(x, y) = \begin{cases} y & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

This is similar to the classical question-mark operator in the C programming language (the third argument is fixed to 0 in this case): `x ? y : 0`.

In other words, the rudimentary functions are the clone

$$\text{clone}(0, 1, +, \pi, .\operatorname{div} m, .\bmod m, W;)$$

Rudimentary Programs

The last definition is in terms of a class of functions.

Alternatively, we can introduce a small programming language that allows one to write programs for precisely the rudimentary functions. Informally, here are the building-blocks for this language. We assume all variables are ranging over $\mathbb{N}$, and we assume constants 0 and 1.

addition	$x + y$
predecessor	$x \dot{-} 1$
remainder	$x \bmod c$
division	$x \operatorname{div} c$
which	if $x > 0$ then y else 0
assignments	$x =$ expression
sequential composition	$P; Q$

Rudimentary Programs, contd.

Nota bene:

- The two-variable modulus operation $x \bmod y$ is not allowed; rather, the second argument must be a constant.
- There are no loop constructs, nor is there recursion of any kind.

How about semantics? We need a "meaning" $[\![P]\!]$ for each rudimentary program.

Since rudimentary programs involve variables this is slightly more painful.

Let $V = \{x_1, x_2, \ldots, x_k\}$ be the collection of all variables in a program. Define an environment to be a binding $E : V \to \mathbb{N}$ of numbers to all variables.

Semantics for Terms

For each term t in the language and any environment E we can define the value $[\![t]\!]_E$ by induction on the structure of t:

$$
\begin{aligned}
[\![0/1]\!]_E &= 0/1 \\
[\![s + t]\!]_E &= [\![s]\!]_E + [\![t]\!]_E \\
[\![s \stackrel{\bullet}{-} t]\!]_E &= [\![s]\!]_E \stackrel{\bullet}{-} [\![t]\!]_E \\
[\![s \bmod c]\!]_E &= [\![s]\!]_E \bmod [\![c]\!]_E \\
[\![s \operatorname{div} c]\!]_E &= [\![s]\!]_E \operatorname{div} [\![c]\!]_E \\
[\![W(s,t))]\!]_E &= [\![t]\!]_E \text{ if } [\![s]\!] > 0\text{, } 0 \text{ otherwise.}
\end{aligned}
$$

Semantics for Programs

For a rudimentary program we define its meaning to be a map from environments to environments as follows:

$$[\![x = t]\!](E) = E[[\![t]\!]_E/x]$$

$$[\![P;Q]\!](E) = [\![Q]\!]([\![P]\!](E))$$

To associate a rudimentary program with an arithmetic function $\mathbb{N}^n \to \mathbb{N}$ we need to designate some variables for input and output. One also should adopt some conventions about initialization to avoid partial functions (say, all variables are initially 0). The details are slightly tedious but quite straightforward.

At any rate, a function f is rudimentary iff there is a rudimentary program P such that $[\![P]\!] = f$.

Exercise

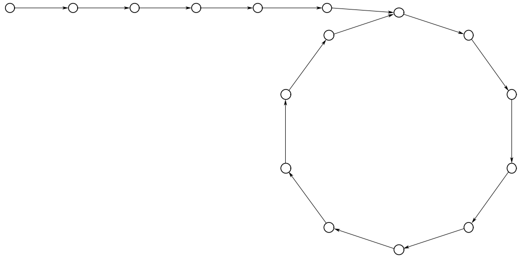
Give a formal definition of the semantics of a rudimentary program.

Generalized Mods

As an example, consider the generalized mod function

$$x \bmod (t,p) = \begin{cases} x & \text{if } x < t, \\ t + (x-t) \bmod p & \text{otherwise.} \end{cases}$$

This describes the position of a particle moving along an ultimately periodic orbit of transient length $t \geq 0$ and period $p > 0$.



A Rudimentary Program

Here is a program that computes the generalized mod; x is the input variable and z the output variable.

```
s0 = x - (t-1);
s1 = 1 - s0;
xx = s0 - 1;
z0 = s0 ? t + xx mod p : 0;
z1 = s1 ? x : 0;
z  = z0 + z1;
```

Note that the first line is just an abbreviation; we should really set `s0 = x;` followed by $t-1$ decrement operations `s0 = s0 - 1;`

Exercises

Exercise

Prove that the generalized mod program above works as claimed.

Exercise

Show that if-then-else is an admissible construct in rudimentary programs (which could be used to slightly simplify the generalized mod program). Find some other admissible constructs.

Floyd's Trick

Recall Floyd's cycle finding algorithm: given a function f on a finite set A and a point $a \in A$ one can find a point on the limit cycle of the orbit of a by calculating the least $r > 0$ such that

$$f^r(a) = f^{2r}(a)$$

The computation is memoryless (at least if we assume that the objects in A have constant size).

Writing t for the transient length of the orbit of a and p for the period length we are looking for the least $r > 0$ such that

$$r = 2r \mod (t, p)$$

How About Multiplication?

Surely, rudimentary functions are easily computable – but it would seem they are a bit too feeble to cover everything that is needed in a real-world application.

For example, multiplication is conspicuously absent from the list of basic rudimentary functions. Could there be a rudimentary program that computes multiplication?

Intuitively, the answer should be “No”: we have no loop construct, so there seems to be no way to repeatedly apply addition to simulate multiplication.

Of course, that’s a plausibility argument, not a proof.

Affine Bounds

We can produce linear (actually, affine) functions such as

$$(x, y) \mapsto 2x + 3y + 5.$$

Using mods can generate periodic behavior and the if-then construct can be used to produce arbitrary values at finitely many places. Still, it appears that we cannot get more than eventually piecewise linear functions. So multiplication should not be rudimentary.

Lemma

For every rudimentary function $f : \mathbb{N}^n \to \mathbb{N}$ *there are constants* $c_0, \ldots, c_n$ *such that*

$$f(x_1, \ldots, x_n) \leq c_0 + \sum_i c_i x_i.$$

Proof

Proof is by structural induction on f.

The claim is easily verified for the basic functions.

If f is obtained by composition, the only interesting case is $f = g + h$. But then the affine bounds for g and h can be combined to yield an affine bound for f.

$\square$

Multiplication is not Rudimentary

Corollary

Multiplication is not rudimentary.

Elementary Functions

Clearly, multiplication must be included in any practical collection of easily computable number-theoretic functions.

Definition (Kalmár 1943)

A function is elementary if it is an element of the least clone containing constants 0, 1, addition, proper subtraction, bounded sums and bounded products.

So we are dealing with

$$\mathsf{clone}(0, 1, +, \dot{-}; \sum_{\mathrm{bnd}}, \prod_{\mathrm{bnd}})$$

Bounded Sums/Products

The bounded sum/product operators are defined like so:

$$f(x, \boldsymbol{y}) = \sum_{z<x} g(z, \boldsymbol{y})$$
$$f(x, \boldsymbol{y}) = \prod_{z<x} g(z, \boldsymbol{y})$$

Elementary functions turn out to be powerful enough to deal with most “practical” problems, in particular they are much more powerful than rudimentary functions. Still, they are far weaker than primitive recursive functions in general.

Some Elementary Functions

While multiplication is absent from the definition, it is easily implemented as an elementary function:

$$x \cdot y = \sum_{z<x} y.$$

Likewise, exponentiation is

$$y^x = \prod_{z<x} y,$$

so that exponential polynomials are also elementary. Note, though, that it is not so clear that one can continue on to super-exponentiation, super-super-exponentiation, and so on.

Exercises

Exercise

Explain what definition by cases means for elementary functions and show that this operation is admissible.

Exercise

Show that quotients and remainders are elementary (as two-variable functions).

Bounding Elementary Functions

So how would we go about showing that some function fails to be elementary? Following the method that worked for rudimentary functions it is tempting to find a nice bound. We can simplify the n-dimensional input a bit by considering the largest component:

$$f(\boldsymbol{x}) \leq \ldots \max \boldsymbol{x} \ldots$$

where $\max \boldsymbol{x} = \max\big(x_i \mid i = 1, \ldots, n \big)$.

This time it is a bit more difficult to come up with the right bound.

Super-Exponentiation

Define super-exponentiation as follows:

$$2{\uparrow}(0, z) = z$$

$$2{\uparrow}(k + 1, z) = 2^{2{\uparrow}(k,z)}$$

Lemma

For every elementary function f there is some k such that

$$f(\boldsymbol{x}) \leq 2{\uparrow}(k, \max \boldsymbol{x})$$

Corollary

The function $2{\uparrow}k = 2{\uparrow}(k, 0)$ is not elementary.

A General Framework: LOOP Programs

We have seen three classes of “easily computable” number-theoretic functions: rudimentary, elementary and primitive recursive. There is a nice uniform description for all of these using a rather natural, loop-based programming language.

Again variables range over $\mathbb{N}$, and we have a single constant 0. The programs are described informally as follows:

reset	$x = 0$
increment	$x = x + 1$
assignments	$x = y$
sequential composition	$P; Q$
control	do x : P od

Note that there are no arithmetic operations, no conditionals, no subroutines, . . .

The Loops

The semantics are clear except for the loop construct:

```
          do x :  P  od
```

This is intended to mean:

Execute P exactly n times where n is the value of x before the loop is entered.

In other words, if P changes the value of x the number of executions will still be the same. We could get the same effect by not allowing x to appear in P. It follows that all loop programs terminate, regardless of the input.

Note that this would not be true for while-loops. As we will see, while-loops are significantly more powerful than our do-loops.

Here are some typical examples for the use of loops.

LOOP Addition

Addition and multiplication are not a primitive operation, but are easy to implement addition in a LOOP program. We indicate input and output variables in a comment line:

```
//  add : x, y --> z
    z = x;
    do y :
          z = z+1;
    od
```

```
//  mult : x, y -> z
    z = 0;
    do x :
          do y :
                z = z+1;
          od
    od
```

LOOP Predecessor

How about the predecessor function? In some frameworks this is the first real challenge.

$$\text{pred}(x) = x \mathbin{\dot{-}} 1 = \begin{cases} 0 & \text{if } x = 0, \\ x - 1 & \text{otherwise.} \end{cases}$$

This requires a little trick, which is not totally obvious. We use an extra variable that lags behind.

```
// pred : x -> z
    z = 0;
    v = 0;
    do x :
            z = v;
            v = v+1;
    od
```

LOOP Sign and Conditionals

The sign function sign$(x) = \min(x, 1)$ can also be implemented by abusing the loop construct as a Boolean test.

```
// sign : x -> z
    z = 0;
    v = 0;
    v = v+1;
    do x :
            z = v;
    od
```

Now we can build conditionals. This is: `if( x > 0 ) P;`

```
    z = sign(x);        // sloppy but hey ...
    do z :
            P;
    od
```

Semantics

More precisely, suppose a loop program P involves variables $x_1, x_2, \ldots, x_k$. Given numerical values to all this variables, a single execution of P will produce new values $x'_1, x'_2, \ldots, x'_k$. Hence we can associate a map

$$[\![P]\!] : \mathbb{N}^k \to \mathbb{N}^k$$

with P. The precise definition of $[\![P]\!]$ is by induction on the buildup of P, see below.

To define the usual input/output behavior we compose $[\![P]\!]$ with injections $\iota : \mathbb{N}^n \to \mathbb{N}^k$ and projections $\pi : \mathbb{N}^k \to \mathbb{N}$.

Application: Semantics of LOOP Programs

Here is a more detailed definition of the semantics of a LOOP program, using iteration.

Define an environment to be a map that assigns a value to all variables.

Write $\mathcal{E}$ for all environments and $\mathcal{P}$ for all loop programs (including variables).

Let $E[t/x]$ be the environment that is the same as E except that the value at x is t.

Definition

Definition

The semantics of LOOP programs is a function

$$\|.\| : \mathcal{P} \to (\mathcal{E} \to \mathcal{E})$$

So $\|P\|$ is a map from environments to environments that explains precisely how execution of a program changes the contents of the variables (think of them as registers in some machine executing P).

Semantics Defined

$\|.\|$ is defined be induction on the build-up of the program.

$$\begin{aligned}\|x = 0\|(E) &= E[0/x] \\ \|x = y\|(E) &= E[y/x] \\ \|x{+}{+}\|(E) &= E[E(x)+1/x] \\ \|P;Q\|(E) &= \|Q\|(\|P\|(E)) \\ \|\texttt{do}\ x : P\ \texttt{od}\|(E) &= \|P\|^{E(x)}(E)\end{aligned}$$

Here $E[v/x]$ means: keep environment E except that the new value of variable x is not v.

So we only use one-point overwrites, composition and iteration; nothing else is needed for LOOP programs.

Examples

Let P be the program

```
do x:  x++ od
```

Then $\|P\|(a) = 2a$.

How about the program Q given by

```
do x:
   do x: x++ od
od
```

We can compute the semantics of Q using the previous example:

$$\|Q\|(a) = \|P\|^a(a) = a\,2^a$$

Exercises

Exercise

Determine the semantics of

```
    u = 0; v = 0;
    do x :
          u++; t = u; u = v; v = t;
    od
```

Loop Computability

Next we introduce the class of all functions computable by loop programs (under some reasonable input/output convention).

Definition

A function $f : \mathbb{N}^n \to \mathbb{N}$ is loop-computable if there is a LOOP program P, an injection ι and a projection π such that $\pi \circ [\![P]\!] \circ \iota = f$.
A relation $A \subseteq \mathbb{N}^n$ is loop-decidable if its characteristic function is loop-computable.

Of course, the program P may have more than just n variables.

Note that unlike with rudimentary and elementary functions we have started with a syntactic characterization, a class of expressions or programs.

The question arises whether there is a way to describe the loop-computable functions directly via basic functions and closure operations. For example, it is clear that loop-computable function are closed under composition.

Exercises

Exercise

Show how loop-computable functions are closed under definition by cases.

Exercise

Show that exponentiation 2^x and super-exponentiation $2 \uparrow x$ are loop-computable.

Exercise

Show that the next-prime function $n \mapsto \min\big(x > n \mid x \text{ prime} \big)$ is loop-computable.

LOOP is PR

Theorem

The loop-computable functions are precisely the primitive recursive functions.

Proof.

The only difficult part in the argument is to show that

- loop-computable functions are closed under primitive recursion, and
- application of a loop operation preserves primitive recursiveness.

For simplicity assume that there is only one parameter y. Suppose $P : y \to z$ and $Q : x, u, y \to z$ are two loop programs computing $g : \mathbb{N} \to \mathbb{N}$ and $h : \mathbb{N}^3 \to \mathbb{N}$.

We show that $f = \text{Prec}[g, h] : \mathbb{N}^2 \to \mathbb{N}$ is loop-computable by constructing a loop program for f directly.

Proof, contd.

Here is a program Pf for f. Variable s is new.

```
//  Pf:   x, y --> z
    s = x;
    x = 0;
    P;                    // y --> z
    do s:
         u = z;
         Q;               // x,u,y --> z
         x = x+1;
    od
```

It is a good exercise to determine the semantics $[\![\mathtt{Pf}]\!]$ in detail.

Proof, contd.

For the opposite direction consider a loop program P with variables $x_1, x_2, \dots, x_k$ and semantics $f = [\![P]\!] : \mathbb{N}^k \to \mathbb{N}^k$.

We may assume by induction that $f = (f_1, \dots, f_k)$ is primitive recursive in the sense that each of the f_i is so p.r. Now consider program `Q`:

```
do x: P od
```

For simplicity, assume $x = x_1$. Define the vector valued function

$$F(0, \boldsymbol{x}) = \boldsymbol{x}$$
$$F(n+1, \boldsymbol{x}) = [\![P]\!](F(n, \boldsymbol{x}))$$

Using sequence numbers we can show that all the component functions of F are primitive recursive. But then $[\![Q]\!](\boldsymbol{x}) = F(x_1, \boldsymbol{x})$.

$\square$

A Hierarchy

There is a natural way to distinguish between loop programs of different complexity: we can count the nesting depth of the loops in the program.

Definition

Let Loop(k) be the class of loop programs that have nesting depth at most k.

Level 0 is not very interesting: it is easy to see that any scalar function that is Loop(0)-computable is of the form

$$f(\boldsymbol{x}) = c \cdot x_i + d$$

where c and d are both constant, $c \in \{0, 1\}$.

Exercise

Give a detailed proof of the last claim.

Level 1

But Loop(1) turns out to be somewhat more complicated.

We have seen that addition and predecessor are Loop(1)-computable.

It is easy to modify the argument for conditionals above to show that if-then applied to Loop(1)-computable functions produces another Loop(1)-computable function.

Since Loop(1) is trivially closed under composition we are missing only remainders and integer division with fixed modulus to show that all the basic rudimentary functions are Loop(1)-computable.

Taking Mods

The following program computes the remainder for modulus 2.

```
// mod2 : x -> u
    u = 0;
    v = 1;
    do x :
            t = u;    // swap u and v
            u = v;
            v = t;
    od
```

Exercise

Show that $x \bmod m$ *is* Loop(1)*-computable for ever fixed modulus* m*.*

Taking Divs

The following program computes the integer quotient $x/2$.

```
// div2 : x -> u
   u = 0;
   v = 0;
   do x :
         u++;
         t = u;    // swap u and v
         u = v;
         v = t;
   od
```

Exercise

Show that $x \operatorname{div} m$ *is* Loop(1)*-computable for ever fixed modulus* m.

Rudimentary is Loop 1

Theorem

A function is Loop(1)*-computable if, and only if, it is rudimentary.*

Proof.

We have just verified that rudimentary programs are Loop(1)-computable.

The other direction is more difficult: every Loop(1) program is already rudimentary. For the proof, consider a single loop `Q`:

```
        do x: P od
```

Suppose the variables are $\boldsymbol{x}$ where $x = x_1$, as before. The loop body `P` is Loop(0), so the effect of executing it `P` is a linear function on each of the variables:

$$x'_i = c_i \cdot x_{p(i)} + d_i$$

where $c_i \in \{0, 1\}$ and $d_i \geq 0$ constant.

Dependencies

Here $p : [n] \to [n]$ is an arbitrary dependency map that indicates how values are passed from one variable to another in an assignment `xi = xj;`

To deal with assignments of the form `xi = 0;` it is convenient to introduce a phantom variable x_0 whose value is fixed at 0. Then we can rewrite the new values as

$$x'_i = x_{p(i)} + d_i$$

Now consider the dependency graph $G = \langle V, E \rangle$ where

- $V = \{x_0, x_1, \ldots, x_n\}$
- $E = \{\, (x_j, x_i) \mid j = p(i) \,\}$

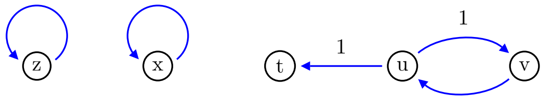
Thus, there is a path in G from x_j to x_i if the value of x_j propagates to x_i, provided the loop is executed sufficiently often.

Example: Division by 2

Variables are $\{z, x, u, v, t\}$ where z is clamped at 0.

```
u = z; v = z;
do x :
        u = u + 1;
        t = u; u = v; v = t;
od
```

The dependency graph looks like so (the edge labels indicate the increment factor d_i).



Dependency is Unicyclic

Note that the indegree of every node in G is at most 1. Hence the strongly connected components of G are just cycles, and all these components are isolated from each other.

But then the values of the registers on a cycle at time t (after t executions of the loop) are of the form

$$x = \begin{cases} a_0 & \text{if } t = 0, \\ a \cdot (t \operatorname{div} l) + \sum_{i < t \bmod l} a_i & \text{otherwise.} \end{cases}$$

where l is the length of the cycle, and the a_i are the labels, a being the sum of all these labels.

Hence the final value of a variable after execution of `P` is a rudimentary function of the initial values. Since rudimentary functions are closed under composition, the whole Loop(1) program can only compute a rudimentary function.

$\square$

Equivalence of Programs

We define equivalence of programs in terms of the function computed by the respective programs, a purely extensional characterization.

Definition

Two programs P and Q are equivalent if they compute the same function $\mathbb{N}^n \to \mathbb{N}$.

Note that this is a weaker notion than having the same semantics, $[\![P]\!] = [\![Q]\!]$. For example, there is no reason why equivalent programs should use the same internal variables.

More importantly, the two programs can be based on two different ways of computing the function in question. Clearly, this type of equivalence is very hard to detect.

Entscheidungsproblem

For each class $\mathcal{C}$ of programs such as rudimentary, elementary, Loop(k), primitive recursive we now have a decision problem:

Problem:	**Program Equivalence for** $\mathcal{C}$
Instance:	Two programs P and Q in class $\mathcal{C}$.
Question:	Are P and Q equivalent?

This is the problem one would like to solve when one verifies the correctness of program transformations: Q is obtained from P by applying a transformation, and we want to make sure that the meaning of P has not changed.

Inequivalence

Alas, for technical reasons it is often better to consider Inequivalence: Is there some input on which the two programs yield different output?

One reason for this surprising twist is that Inequivalence of programs is semi-decidable, even if we have no constraints on the class $\mathcal{C}$ (actually, we have to insist on total functions rather than partial ones as in the case of arbitrary computable functions): For Inequivalence one can conduct a brute-force search over all possible inputs to find an $\boldsymbol{x}$ such that

$$P(\boldsymbol{x}) \neq Q(\boldsymbol{x}).$$

This may appear rather non-sensical, but in lower complexity classes the distinction between Equivalence and Inequivalence becomes quite important. Of course, as far as decidability is concerned, there is no difference between Equivalence and Inequivalence.

Loop 0

Proposition

Inequivalence for Loop(0) *programs is decidable in polynomial time.*

Proof.

Given the program P, we can easily compute an index i and constants $c \in \{0, 1\}$, $d \geq 0$ such that P computes

$$\boldsymbol{x} \mapsto c \cdot x_i + d$$

But then equivalence and hence inequivalence are trivial to check: index and constants have to be the same for both programs.

$\square$

Exercise

Devise a fast algorithm to test Equivalence of Loop(0) *programs.*

Loop 2 and Up

Lemma

Inequivalence for Loop(2) *programs is undecidable.*

Proof. Given a multivariate integer polynomial $p(\boldsymbol{x})$ one can easily build a program P that computes

$$\mathsf{signbar}(p(\boldsymbol{x})) \in \{0, 1\}$$

P is naturally level 2 since all the arithmetic can be handled there.

Let Q be the trivial program that computes the constant 0 function. Then Inequivalence for P and Q comes down solving a Diophantine equation, which problem is undecidable by Matiyasevic's theorem.

$\square$

Loop 1

That leaves Inequivalence for level 1 open: can we check if two rudimentary functions disagree on some input?

One might suspect that Inequivalence of rudimentary functions is indeed decidable since these functions are in some sense periodic or piecewise affine.

But the details bear some careful explanation: level 2 is not far away, and there Inequivalence is already undecidable.

As it turns out, Inequivalence for Loop1 program is decidable, but is already $\mathbb{NP}$-hard, so there is likely no fast algorithm for checking equivalence of such programs.

Partitioning the Domain

Definition

Define an equivalence relation $\equiv_{\beta,\mu}$ on $\mathbb{N}^n$ as follows: $\boldsymbol{x}$ and $\boldsymbol{y}$ are equivalent if

- $x_i < \beta \vee y_i < \beta$ implies $x_i = y_i$, and
- $x_i \geq \beta \wedge y_i \geq \beta$ implies $x_i = y_i \pmod{\mu}$.

Each equivalence class of $\equiv_{\beta,\mu}$ is either a singleton or infinite.

The number of equivalence classes is $(\beta + \mu)^n$: each component of the vector is either completely fixed (if it is less than β) or fixed modulo μ.

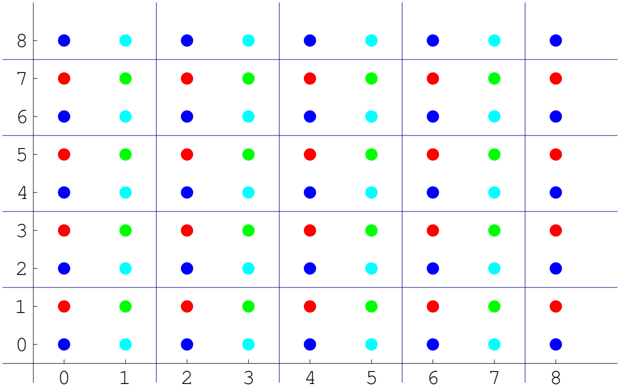
A simple case arises when $\beta = 0$: then we are simply subdividing $\mathbb{N}^n$ into hypercubes of size μ^n.

Example

Consider the function

$$f(x, y) = x + x \bmod 2 + y \operatorname{div} 2 + 1.$$

For $\beta = 0$, $\mu = 2$ we get the following classes:



Example, contd.

$$f(x, y) = x + x \bmod 2 + y \operatorname{div} 2 + 1$$

The corresponding four component functions for the equivalence classes are

$$\begin{array}{cc} x + \frac{1}{2}y + \frac{1}{2} & x + \frac{1}{2}y + \frac{3}{2} \\ x + \frac{1}{2}y + 1 & x + \frac{1}{2}y + 2 \end{array}$$

In essence, the mod terms affect the additive constant and the div terms produce the fractional coefficients.

Describing Rudimentary Functions

Theorem

For each rudimentary function $f : \mathbb{N}^n \to \mathbb{N}$ there are constants β and μ such the restriction of f to the equivalence classes of $\equiv_{\beta,\mu}$ is an affine function:

$$f(\boldsymbol{x}) = \sum_i c_i \cdot x_i + c.$$

Proof.

Use induction on the buildup of f. Here are the important cases.

$f_1 + f_2$	$\beta = \max(\beta_1, \beta_2)$	$\mu = \mu_1 \mu_2$
$f_1 \dot{-} 1$	$\beta = \beta_1 + \mu_1$	$\mu = \mu_1$
$W(f_1, f_2)$	$\beta = \max(\beta_1, \beta_2) + \mu_2$	$\mu = \mu_1 \mu_2$
$f_1 \operatorname{div} c$	$\beta = \beta_1$	$\mu = c\mu_1$
$f_1 \bmod c$	$\beta = \beta_1$	$\mu = c\mu_1$

Simple Example

Suppose $g(x)$ is affine on the classes of $\equiv_{0,\mu}$.

So there is a family of μ many affine functions G_i such that

$$g(x) = G_{x \bmod \mu}(x)$$

Set $f(x) = g(x) \bmod c$. Then

$$f(x) = G_{x \bmod \mu}(x) \bmod c = G_{x \bmod \mu}(x \bmod c) \bmod c$$

and we have to distinguish at most $c\mu$ classes for f.

A Basis Set

One can push things a bit further and show that if a rudimentary function f is piecewise affine with respect to $\equiv_{\beta,\mu}$ then f is completely determined by its values on the basis set

$$S = \{\, \boldsymbol{x} \mid x_i \leq \beta + 2\mu \,\}.$$

In other words, if g is another rudimentary function with parameters β and μ and we have

$$\forall\, \boldsymbol{x} \in S\,(f(\boldsymbol{x}) = g(\boldsymbol{x}))$$

then the two functions already agree everywhere.

Putting Things Together

Note that $\equiv_{\beta',\mu'}$ refines $\equiv_{\beta,\mu}$ whenever $\beta' \geq \beta$ and $\mu' = c\mu$.

Hence we can always choose the same parameters for any two functions.

Theorem

Let f_1 and f_2 be two rudimentary functions with parameters β_1, μ_1 and β_2, μ_2, respectively. Then the two functions are equivalent if the agree on

$$\{\, \boldsymbol{x} \mid x_i \leq \max(\beta_1, \beta_2) + 2\mu_1\mu_2 \,\}.$$

Inequivalence

So how hard is it to check Inequivalence for Loop(1)?

We can easily compute the parameters β and μ from the programs, and as we have seen we may assume they are the same.

What is left is a simple search over the test set

$$S = \{\, \boldsymbol{x} \mid x_i \leq \beta + 2\mu \,\}.$$

There is a slight problem, though: there are exponentially many values to check. So a deterministic algorithm would be exponential, but if we allow nondeterminism we can get away with polynomial time: guess the difference and verify it, all in polynomial time.

While Programs

Loop-computable is the same a primitive recursive. We know that in order to move from p.r. to full computability it suffices to add unbounded search.

What does one have to add to loop programs to obtain full computability?

It turns out, we only need to replace the fixed-number-of-steps loop construct by a while-loop that has no a priori bounds on the number of executions.

```
        while x:  P;
```

The point is that program P may contain variable x and change its value. The loop terminates if x attains values 0.

Theorem

While programs correspond exactly to general recursive functions.

Proofsketch

It is not hard to see that while-computable functions are all general recursive.

Suppose we have a program with a single while-loop. All parts of the program other than the actual while-loop are easily seen to be primitive recursive.

The while-loop itself can be expressed by iterating the loop body until the condition fails to hold. This corresponds to performing an unbounded search.

With a little more effort we can also handle nested while-loops: one can always get away with just a single while-loop.

Proof, contd.

It suffices to show (see Kleene's Normal Form theorem) that unbounded search applied to a primitive recursive predicate is while-computable.

We already know that all p.r. relations are loop-computable and hence also while-computable. But search (say, for a root of f) is easy with a while-loop:

```
r = 1;
s = 0;
while r:
        r = f(s);
        s = s + 1;
```

$\square$