Diophantine Equations and the DPRM Theorem

Jonas Bayer[*] Marco David[*] Benedikt Stock[*]

Abhik Pal Yuri Matiyasevich[†] Dierk Schleicher

March 17, 2025

Abstract

We present a formalization of Matiyasevich's proof of the DPRM theorem, which states that every recursively enumerable set of natural numbers is Diophantine. This result from 1970 yields a negative solution to Hilbert's 10th problem over the integers. To represent recursively enumerable sets in equations, we implement and arithmetize register machines. We formalize a general theory of Diophantine sets and relations to reason about them abstractly. Using several number-theoretic lemmas, we prove that exponentiation has a Diophantine representation.

Contents

[*]Equal contribution.

[†]Contributed by supplying a detailed proof and an initial introduction to Isabelle.

Overview A previous short paper [2] gives an overview of the formalization. In particular, the challenges of implementing the notion of diophantine predicates is discussed and a formal definition of register machines is described. Another meta-publication [1] recounts our learning experience throughout this project.

The present formalisation is based on Yuri Matiyasevich's monograph [5] which contains a full proof of the DPRM theorem. This result or parts of its proof have also been formalized in other interactive theorem provers, notably in Coq [4], Lean [3] and Mizar [7, 6].

Acknowledgements We want to thank everyone who participated in the formalization during the early stages of this project: Deepak Aryal, Bogdan Ciurezu, Yiping Deng, Prabhat Devkota, Simon Dubischar, Malte HaSSler, Yufei Liu and Maria Antonia Oprea. Moreover, we would like to express our sincere gratitude to the entire welcoming and supportive Isabelle community. In particular, we are indebted to Christoph Benzmüller for his expertise, his advice and for connecting us with relevant experts in the field. Among those, we specially want to thank Mathias Fleury for all his help with Isabelle. Finally, we would like to thank the DFG for supporting our attendance at several events and conferences, allowing us to present the project to a broad audience.

1 Diophantine Equations

```
theory Parametric-Polynomials
  imports Main
  abbrevs ++ = + and
        -- = − and
        ** = ∗ and
        00 = 0 and
        11 = 1
begin
```

1.1 Parametric Polynomials

This section defines parametric polynomials and builds up the infrastructure to later prove that a given predicate or relation is Diophantine. The formalization follows [5].

```
type-synonym assignment = nat ⇒ nat
```

Definition of parametric polynomials with natural number coefficients and their evaluation function

```
datatype ppolynomial =
    Const nat |
    Param nat |
    Var   nat |
    Sum  ppolynomial ppolynomial (infixl ‹+› 65) |
    NatDiff ppolynomial ppolynomial (infixl ‹−› 65) |
    Prod ppolynomial ppolynomial (infixl ‹∗› 70)

fun ppeval :: ppolynomial ⇒ assignment ⇒ assignment ⇒ nat  where
    ppeval (Const c) p v = c |
    ppeval (Param x) p v = p x |
    ppeval (Var x) p v = v x |
    ppeval (D1 + D2) p v = (ppeval D1 p v) + (ppeval D2 p v) |

    ppeval (D1 − D2) p v = (ppeval D1 p v) − (ppeval D2 p v) |
    ppeval (D1 ∗ D2) p v = (ppeval D1 p v) ∗ (ppeval D2 p v)

definition Sq-pp (‹- ^2› [99] 75) where Sq-pp P = P ∗ P

definition is-dioph-set :: nat set ⇒ bool where
    is-dioph-set A = (∃ P1 P2::ppolynomial. ∀ a. (a ∈ A)
                            ⟷ (∃ v. ppeval P1 (λx. a) v = ppeval P2 (λx.
a) v))

datatype polynomial =
    Const nat |
    Param nat |
    Sum  polynomial polynomial (infixl ‹[+]› 65) |
```

NatDiff polynomial polynomial (**infixl** ‹[−]› *65*) |
Prod polynomial polynomial (**infixl** ‹[∗]› *70*)

fun *peval* :: *polynomial* ⇒ *assignment* ⇒ *nat* **where**
peval (*Const c*) *p* = *c* |
peval (*Param x*) *p* = *p x* |
peval (*Sum D1 D2*) *p* = (*peval D1 p*) + (*peval D2 p*) |

peval (*NatDiff D1 D2*) *p* = (*peval D1 p*) − (*peval D2 p*) |
peval (*Prod D1 D2*) *p* = (*peval D1 p*) ∗ (*peval D2 p*)

definition *sq-p* :: *polynomial* ⇒ *polynomial* (‹- [^2]› [*99*] *75*) **where** *sq-p P* = *P* [∗] *P*

definition *zero-p* :: *polynomial* (‹**0**›) **where** *zero-p* = *Const 0*
definition *one-p* :: *polynomial* (‹**1**›) **where** *one-p* = *Const 1*

lemma *sq-p-eval*: *peval* (*P*[^2]) *p* = (*peval P p*)^*2*
⟨*proof*⟩

fun *convert* :: *polynomial* ⇒ *ppolynomial* **where**
convert (*Const c*) = (*ppolynomial.Const c*) |
convert (*Param x*) = (*ppolynomial.Param x*) |
convert (*D1* [+] *D2*) = (*convert D1*) **+** (*convert D2*) |
convert (*D1* [−] *D2*) = (*convert D1*) **−** (*convert D2*) |
convert (*D1* [∗] *D2*) = (*convert D1*) **∗** (*convert D2*)

lemma *convert-eval*: *peval P a* = *ppeval* (*convert P*) *a v*
⟨*proof*⟩

definition *list-eval* :: *polynomial list* ⇒ *assignment* ⇒ (*nat* ⇒ *nat*) **where**
list-eval PL a = *nth* (*map* (λ*x*. *peval x a*) *PL*)

end

1.2 Variable Assignments

The following theory defines manipulations of variable assignments and proves elementary facts about these. Such preliminary results will later be necesary to e.g. prove that conjunction is diophantine.

theory *Assignments*
imports *Parametric-Polynomials*
begin

definition *shift* :: *nat list* ⇒ *nat* ⇒ *assignment* **where**
shift l a ≡ λ*i*. *l* ! (*i* + *a*)

definition *push* :: *assignment* ⇒ *nat* ⇒ *assignment* **where**
push a n i = (*if i* = *0 then n else a* (*i*−*1*))

definition *push-list* :: *assignment* ⇒ *nat list* ⇒ *nat* ⇒ *nat* **where**
push-list a ns i = (*if i* < *length ns then* (*ns*!*i*) *else a* (*i* − *length ns*))

lemma *push0*: *push a n 0* = *n*
⟨*proof*⟩

lemma *push-list-empty*: *push-list a* [] = *a*
⟨*proof*⟩

lemma *push-list-singleton*: *push-list a* [*n*] = *push a n*
⟨*proof*⟩

lemma *push-list-eval*: *i* < *length ns* ⟹ *push-list a ns i* = *ns*!*i*
⟨*proof*⟩

lemma *push-list1*: *push* (*push-list a ns*) *n* = *push-list a* (*n* # *ns*)
⟨*proof*⟩

lemma *push-list2-aux*: (*push-list* (*push a n*) *ns*) *i* = *push-list a* (*ns* @ [*n*]) *i*
⟨*proof*⟩

lemma *push-list2*: (*push-list* (*push a n*) *ns*) = *push-list a* (*ns* @ [*n*])
⟨*proof*⟩

fun *pull-param* :: *ppolynomial* ⇒ *ppolynomial* ⇒ *ppolynomial* **where**
pull-param (*ppolynomial.Param 0*) *repl* = *repl* |
pull-param (*ppolynomial.Param* (*Suc n*)) - = (*ppolynomial.Param n*) |
pull-param (*D1* + *D2*) *repl* = (*pull-param D1 repl*) + (*pull-param D2 repl*) |
pull-param (*D1* − *D2*) *repl* = (*pull-param D1 repl*) − (*pull-param D2 repl*) |
pull-param (*D1* ∗ *D2*) *repl* = (*pull-param D1 repl*) ∗ (*pull-param D2 repl*) |
pull-param P repl = *P*

fun *var-set* :: *ppolynomial* ⇒ *nat set* **where**
var-set (*ppolynomial.Const c*) = {} |
var-set (*ppolynomial.Param x*) = {} |
var-set (*ppolynomial.Var x*) = {*x*} |
var-set (*D1* + *D2*) = *var-set D1* ∪ *var-set D2* |
var-set (*D1* − *D2*) = *var-set D1* ∪ *var-set D2* |
var-set (*D1* ∗ *D2*) = *var-set D1* ∪ *var-set D2*

definition *disjoint-var* :: *ppolynomial* ⇒ *ppolynomial* ⇒ *bool* **where**
disjoint-var P Q = (*var-set P* ∩ *var-set Q* = {})

named-theorems *disjoint-vars*

lemma *disjoint-var-sym*: *disjoint-var P Q* = *disjoint-var Q P*

⟨*proof*⟩

lemma *disjoint-var-sum*[*disjoint-vars*]: *disjoint-var* (*P1* + *P2*) *Q* = (*disjoint-var* *P1* *Q* ∧ *disjoint-var* *P2* *Q*)
⟨*proof*⟩

lemma *disjoint-var-diff*[*disjoint-vars*]: *disjoint-var* (*P1* − *P2*) *Q* = (*disjoint-var* *P1* *Q* ∧ *disjoint-var* *P2* *Q*)
⟨*proof*⟩

lemma *disjoint-var-prod*[*disjoint-vars*]: *disjoint-var* (*P1* ∗ *P2*) *Q* = (*disjoint-var* *P1* *Q* ∧ *disjoint-var* *P2* *Q*)
⟨*proof*⟩

lemma *aux-var-set*:
assumes ∀ *i* ∈ *var-set P*. *x i* = *y i*
shows *ppeval P a x* = *ppeval P a y*
⟨*proof*⟩

First prove that disjoint variable sets allow the unification into one variable assignment

definition *zip-assignments* :: *ppolynomial* ⇒ *ppolynomial* ⇒ *assignment* ⇒ *assignment* ⇒ *assignment*
where *zip-assignments P Q v w i* = (*if i* ∈ *var-set P then v i else w i*)

lemma *help-eval-zip-assignments1*:
shows *ppeval P1 a* (λ*i*. *if i* ∈ *var-set P1* ∪ *var-set P2 then v i else w i*)
= *ppeval P1 a* (λ*i*. *if i* ∈ *var-set P1 then v i else w i*)
⟨*proof*⟩

lemma *help-eval-zip-assignments2*:
shows *ppeval P2 a* (λ*i*. *if i* ∈ *var-set P1* ∪ *var-set P2 then v i else w i*)
= *ppeval P2 a* (λ*i*. *if i* ∈ *var-set P2 then v i else w i*)
⟨*proof*⟩

lemma *eval-zip-assignments1*:
fixes *v w*
assumes *disjoint-var P Q*
defines *x* ≡ *zip-assignments P Q v w*
shows *ppeval P a v* = *ppeval P a x*
⟨*proof*⟩

lemma *eval-zip-assignments2*:
fixes *v w*
assumes *disjoint-var P Q*
defines *x* ≡ *zip-assignments P Q v w*
shows *ppeval Q a w* = *ppeval Q a x*
⟨*proof*⟩

lemma *zip-assignments-correct*:
assumes *ppeval P1 a v = ppeval P2 a v* **and** *ppeval Q1 a w = ppeval Q2 a w*
and *disjoint-var (P1 + P2) (Q1 + Q2)*
defines *x ≡ zip-assignments (P1 + P2) (Q1 + Q2) v w*
shows *ppeval P1 a x = ppeval P2 a x* **and** *ppeval Q1 a x = ppeval Q2 a x*
⟨*proof*⟩

lemma *disjoint-var-unifies*:
assumes *∃ v1. ppeval P1 a v1 = ppeval P2 a v1* **and** *∃ v2. ppeval Q1 a v2 = ppeval Q2 a v2*
and *disjoint-var (P1 + P2) (Q1 + Q2)*
shows *∃ v. ppeval P1 a v = ppeval P2 a v ∧ ppeval Q1 a v = ppeval Q2 a v*
⟨*proof*⟩

A function to manipulate variables in ppolynomials

fun *push-var* :: *ppolynomial ⇒ nat ⇒ ppolynomial* **where**
push-var (ppolynomial.Var x) n = ppolynomial.Var (x + n) |
push-var (D1 + D2) n = push-var D1 n + push-var D2 n |
push-var (D1 − D2) n = push-var D1 n − push-var D2 n |
push-var (D1 ∗ D2) n = push-var D1 n ∗ push-var D2 n |
push-var D n = D

lemma *push-var-bound*: *x ∈ var-set (push-var P (Suc n)) ⟹ x > n*
⟨*proof*⟩

definition *pull-assignment* :: *assignment ⇒ nat ⇒ assignment* **where**
pull-assignment v n = (λx. v (x+n))

lemma *push-var-pull-assignment*:
shows *ppeval (push-var P n) a v = ppeval P a (pull-assignment v n)*
⟨*proof*⟩

lemma *max-set*: *finite A ⟹ ∀ x∈A. x ≤ Max A*
⟨*proof*⟩

fun *push-param* :: *polynomial ⇒ nat ⇒ polynomial* **where**
push-param (Const c) n = Const c |
push-param (Param x) n = Param (x + n) |
push-param (Sum D1 D2) n = Sum (push-param D1 n) (push-param D2 n) |
push-param (NatDiff D1 D2) n = NatDiff (push-param D1 n) (push-param D2 n) |
push-param (Prod D1 D2) n = Prod (push-param D1 n) (push-param D2 n)

definition *push-param-list* :: *polynomial list ⇒ nat ⇒ polynomial list* **where**
push-param-list s k ≡ map (λx. push-param x k) s

lemma *push-param0*: *push-param P 0* = *P*
⟨*proof*⟩

lemma *push-push-aux*: *peval* (*push-param P* (*Suc m*)) (*push a n*) = *peval* (*push-param P m*) *a*
⟨*proof*⟩

lemma *push-push*:
shows *length ns* = *n* ⟹ *peval* (*push-param P n*) (*push-list a ns*) = *peval P a*
⟨*proof*⟩

lemma *push-push-simp*:
shows *peval* (*push-param P* (*length ns*)) (*push-list a ns*) = *peval P a*
⟨*proof*⟩

lemma *push-push1*: *peval* (*push-param P 1*) (*push a k*) = *peval P a*
⟨*proof*⟩

lemma *push-push-map*: *length ns* = *n* ⟹
list-eval (*map* (λ*x*. *push-param x n*) *ls*) (*push-list a ns*) = *list-eval ls a*
⟨*proof*⟩

lemma *push-push-map-i*: *length ns* = *n* ⟹ *i* < *length ls* ⟹
peval (*map* (λ*x*. *push-param x n*) *ls* ! *i*) (*push-list a ns*) = *list-eval ls a i*
⟨*proof*⟩

lemma *push-push-map1*: *i* < *length ls* ⟹
peval (*map* (λ*x*. *push-param x 1*) *ls* ! *i*) (*push a n*) = *list-eval ls a i*
⟨*proof*⟩

end

1.3 Diophantine Relations and Predicates

theory *Diophantine-Relations*
imports *Assignments*
begin

datatype *relation* =
NARY nat list ⇒ *bool polynomial list*
| *AND relation relation* (**infixl** ‹[∧]› *35*)
| *OR relation relation* (**infixl** ‹[∨]› *30*)
| *EXIST-LIST nat relation* (‹[∃-] -› *10*)

fun *eval* :: *relation* ⇒ *assignment* ⇒ *bool* **where**
eval (*NARY R PL*) *a* = *R* (*map* (λ*P*. *peval P a*) *PL*)
| *eval* (*AND D1 D2*) *a* = (*eval D1 a* ∧ *eval D2 a*)

| *eval* (*OR D1 D2*) *a* = (*eval D1 a* ∨ *eval D2 a*)
| *eval* ([∃ *n*] *D*) *a* = (∃ *ks*::*nat list*. *n* = *length ks* ∧ *eval D* (*push-list a ks*))

definition *is-dioph-rel* :: *relation* ⇒ *bool* **where**
is-dioph-rel DR = (∃ P_1 P_2::*ppolynomial*. ∀ *a*. (*eval DR a*) ⟷ (∃ *v*. *ppeval* P_1 *a* *v* = *ppeval* P_2 *a v*))

definition *UNARY* :: (*nat* ⇒ *bool*) ⇒ *polynomial* ⇒ *relation* **where**
UNARY R P = *NARY* (λ*l*. *R* (*l*!*0*)) [*P*]

lemma *unary-eval*: *eval* (*UNARY R P*) *a* = *R* (*peval P a*)
⟨*proof*⟩

definition *BINARY* :: (*nat* ⇒ *nat* ⇒ *bool*) ⇒ *polynomial* ⇒ *polynomial* ⇒ *relation* **where**
BINARY R P_1 P_2 = *NARY* (λ*l*. *R* (*l*!*0*) (*l*!*1*)) [P_1, P_2]

lemma *binary-eval*: *eval* (*BINARY R* P_1 P_2) *a* = *R* (*peval* P_1 *a*) (*peval* P_2 *a*)
⟨*proof*⟩

definition *TERNARY* :: (*nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*)
⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial* ⇒ *relation* **where**
TERNARY R P_1 P_2 P_3 = *NARY* (λ*l*. *R* (*l*!*0*) (*l*!*1*) (*l*!*2*)) [P_1, P_2, P_3]

lemma *ternary-eval*: *eval* (*TERNARY R* P_1 P_2 P_3) *a* = *R* (*peval* P_1 *a*) (*peval* P_2 *a*) (*peval* P_3 *a*)
⟨*proof*⟩

definition *QUATERNARY* :: (*nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*)
⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial* ⇒ *relation* **where**
QUATERNARY R P_1 P_2 P_3 P_4 = *NARY* (λ*l*. *R* (*l*!*0*) (*l*!*1*) (*l*!*2*) (*l*!*3*)) [P_1, P_2, P_3, P_4]

definition *EXIST* :: *relation* ⇒ *relation* (‹[∃] -› *10*) **where**
([∃] *D*) = ([∃ *1*] *D*)

definition *TRUE* **where** *TRUE* = *UNARY* ((=) *0*) (*Const 0*)

Bounded constant all quantifier (i.e. recursive conjunction)

fun *ALLC-LIST* :: *nat list* ⇒ (*nat* ⇒ *relation*) ⇒ *relation* (‹[∀ *in* -] -›) **where**
[∀ *in* []] *DF* = *TRUE* |
[∀ *in* (*l* # *ls*)] *DF* = (*DF l* [∧] [∀ *in ls*] *DF*)

lemma *ALLC-LIST-eval-list-all*: *eval* ([∀ *in L*] *DF*) *a* = *list-all* (λ*l*. *eval* (*DF l*) *a*) *L*
⟨*proof*⟩

lemma *ALLC-LIST-eval*: *eval* ([∀ *in* *L*] *DF*) *a* = (∀ *k*<*length* *L*. *eval* (*DF* (*L*!*k*)) *a*)
⟨*proof*⟩

definition *ALLC* :: *nat* ⇒ (*nat* ⇒ *relation*) ⇒ *relation* (‹[∀<-] -›) **where**
[∀<*n*] *D* ≡ [∀ *in* [*0*..<*n*]] *D*

lemma *ALLC-eval*: *eval* ([∀<*n*] *DF*) *a* = (∀ *k*<*n*. *eval* (*DF* *k*) *a*)
⟨*proof*⟩

fun *concat* :: ′*a list list* ⇒ ′*a list* **where**
concat [] = [] |
concat (*l* # *ls*) = *l* @ *concat ls*

fun *splits* :: ′*a list* ⇒ *nat list* ⇒ ′*a list list* **where**
splits L [] = [] |
splits L (*n* # *ns*) = (*take n L*) # (*splits* (*drop n L*) *ns*)

lemma *split-concat*:
splits (*map f* (*concat pls*)) (*map length pls*) = *map* (*map f*) *pls*
⟨*proof*⟩

definition *LARY* :: (*nat list list* ⇒ *bool*) ⇒ (*polynomial list list*) ⇒ *relation* **where**
LARY R PLL = *NARY* (λ*l*. *R* (*splits l* (*map length PLL*))) (*concat PLL*)

lemma *LARY-eval*:
fixes *PLL* :: *polynomial list list*
shows *eval* (*LARY R PLL*) *a* = *R* (*map* (*map* (λ*P*. *peval P a*)) *PLL*)
⟨*proof*⟩

lemma *or-dioph*:
assumes *is-dioph-rel A* **and** *is-dioph-rel B*
shows *is-dioph-rel* (*A* [∨] *B*)
⟨*proof*⟩

lemma *exists-disjoint-vars*:
fixes *Q1 Q2* :: *ppolynomial*
fixes *A* :: *relation*
assumes *is-dioph-rel A*
shows ∃ *P1 P2*. *disjoint-var* (*P1* + *P2*) (*Q1* + *Q2*)
∧ (∀ *a*. *eval A a* ⟷ (∃ *v*. *ppeval P1 a v* = *ppeval P2 a v*))
⟨*proof*⟩

lemma *and-dioph*:
assumes *is-dioph-rel A* **and** *is-dioph-rel B*
shows *is-dioph-rel* (*A* [∧] *B*)
⟨*proof*⟩

definition *eq* (**infix** ‹[=]› *50*) **where** *eq* Q R ≡ *BINARY* (=) Q R
definition *lt* (**infix** ‹[<]› *50*) **where** *lt* Q R ≡ *BINARY* (<) Q R
definition *le* (**infix** ‹[≤]› *50*) **where** *le* Q R ≡ Q [<] R [∨] Q [=] R
definition *gt* (**infix** ‹[>]› *50*) **where** *gt* Q R ≡ R [<] Q
definition *ge* (**infix** ‹[≥]› *50*) **where** *ge* Q R ≡ Q [>] R [∨] Q [=] R

named-theorems *defs*
lemmas [*defs*] = *zero-p-def one-p-def eq-def lt-def le-def gt-def ge-def LARY-eval UNARY-def BINARY-def TERNARY-def QUATERNARY-def ALLC-LIST-eval ALLC-eval*

named-theorems *dioph*
lemmas [*dioph*] = *or-dioph and-dioph*

lemma *true-dioph*[*dioph*]: *is-dioph-rel TRUE*
⟨*proof*⟩

lemma *eq-dioph*[*dioph*]: *is-dioph-rel* (Q [=] R)
⟨*proof*⟩

lemma *lt-dioph*[*dioph*]: *is-dioph-rel* (Q [<] R)
⟨*proof*⟩

definition *zero* (‹[*0*=] -› [*60*] *60*) **where**[*defs*]: *zero* Q ≡ **0** [=] Q
lemma *zero-dioph*[*dioph*]: *is-dioph-rel* ([*0*=] Q)
⟨*proof*⟩

lemma *gt-dioph*[*dioph*]: *is-dioph-rel* (Q [>] R)
⟨*proof*⟩

lemma *le-dioph*[*dioph*]: *is-dioph-rel* (Q [≤] R)
⟨*proof*⟩

lemma *ge-dioph*[*dioph*]: *is-dioph-rel* (Q [≥] R)
⟨*proof*⟩

Bounded Constant All Quantifier, dioph rules

lemma *ALLC-LIST-dioph*[*dioph*]: *list-all* (*is-dioph-rel* ∘ *DF*) L ⟹ *is-dioph-rel* ([∀ *in* L] *DF*)
⟨*proof*⟩

lemma *ALLC-dioph*[*dioph*]: ∀ i<n. *is-dioph-rel* (*DF* i) ⟹ *is-dioph-rel* ([∀<n] *DF*)
⟨*proof*⟩

end

1.4 Existential quantification is Diophantine

theory *Existential-Quantifier*
  imports *Diophantine-Relations*
begin

lemma *exist-list-dioph*[*dioph*]:
  fixes *D*
  assumes *is-dioph-rel D*
  shows *is-dioph-rel* ([∃ *n*] *D*)
⟨*proof*⟩

lemma *exist-dioph*[*dioph*]:
  fixes *D*
  assumes *is-dioph-rel D*
  shows *is-dioph-rel* ([∃] *D*)
  ⟨*proof*⟩

lemma *exist-eval*[*defs*]:
  shows *eval* ([∃] *D*) *a* = (∃ *k*. *eval D* (*push a k*))
  ⟨*proof*⟩

end

1.5 Mod is Diophantine

theory *Modulo-Divisibility*
  imports *Existential-Quantifier*
begin

Divisibility is diophantine

definition *dvd* (‹*DVD* - -› *1000*) **where** *DVD Q R* ≡ (*BINARY* (*dvd*) *Q R*)

lemma *dvd-repr*:
  fixes *a b* :: *nat*
  shows *a dvd b* ⟷ (∃ *x*. *x* ∗ *a* = *b*)
  ⟨*proof*⟩

lemma *dvd-dioph*[*dioph*]: *is-dioph-rel* (*DVD Q R*)
⟨*proof*⟩

declare *dvd-def*[*defs*]

definition *mod* (‹*MOD* - - -› *1000*)
  where *MOD A B C* ≡ (*TERNARY* (λ*a b c*. *a mod b* = *c mod b*) *A B C*)
declare *mod-def*[*defs*]

lemma *mod-repr*:
  fixes *a b c* :: *nat*

shows *a mod b* = *c mod b* ⟷ (∃ *x y*. *c* + *x*∗*b* = *a* + *y*∗*b*)
⟨*proof*⟩

lemma *mod-dioph*[*dioph*]:
fixes *A B C*
defines *D* ≡ (*MOD A B C*)
shows *is-dioph-rel D*
⟨*proof*⟩

declare *mod-def*[*defs*]

end

2 Exponentiation is Diophaninte

2.1 Expressing Exponentiation in terms of the alpha function

theory *Exponentiation*
imports *Complex-Main*
begin

locale *Exp-Matrices*
begin

2.1.1 2x2 matrices and operations

datatype *mat2* = *mat* (*mat-11* : *int*) (*mat-12* : *int*) (*mat-21* : *int*) (*mat-22* : *int*)
datatype *vec2* = *vec* (*vec-1*: *int*) (*vec-2*: *int*)

fun *mat-plus*:: *mat2* ⇒ *mat2* ⇒ *mat2* **where**
mat-plus A B = *mat* (*mat-11 A* + *mat-11 B*) (*mat-12 A* + *mat-12 B*)
(*mat-21 A* + *mat-21 B*) (*mat-22 A* + *mat-22 B*)

fun *mat-mul*:: *mat2* ⇒ *mat2* ⇒ *mat2* **where**
mat-mul A B = *mat* (*mat-11 A* ∗ *mat-11 B* + *mat-12 A* ∗ *mat-21 B*)
(*mat-11 A* ∗ *mat-12 B* + *mat-12 A* ∗ *mat-22 B*)
(*mat-21 A* ∗ *mat-11 B* + *mat-22 A* ∗ *mat-21 B*)
(*mat-21 A* ∗ *mat-12 B* + *mat-22 A* ∗ *mat-22 B*)

fun *mat-pow*:: *nat* ⇒ *mat2* ⇒ *mat2* **where**
mat-pow 0 - = *mat 1 0 0 1* |
mat-pow n A = *mat-mul A* (*mat-pow* (*n* − *1*) *A*)

lemma *mat-pow-2*[*simp*]: *mat-pow 2 A* = *mat-mul A A*
⟨*proof*⟩

fun *mat-det*::*mat2* ⇒ *int* **where**
mat-det M = *mat-11 M* ∗ *mat-22 M* − *mat-12 M* ∗ *mat-21 M*

fun *mat-scalar-mult*::*int* ⇒ *mat2* ⇒ *mat2* **where**
  mat-scalar-mult a M = *mat* (*a* ∗ *mat-11 M*) (*a* ∗ *mat-12 M*) (*a* ∗ *mat-21 M*) (*a* ∗ *mat-22 M*)

fun *mat-minus*:: *mat2* ⇒ *mat2* ⇒ *mat2* **where**
  mat-minus A B = *mat* (*mat-11 A* − *mat-11 B*) (*mat-12 A* − *mat-12 B*)
    (*mat-21 A* − *mat-21 B*) (*mat-22 A* − *mat-22 B*)

fun *mat-vec-mult*:: *mat2* ⇒ *vec2* ⇒ *vec2* **where**
  mat-vec-mult M v = *vec* (*mat-11 M* ∗ *vec-1 v* + *mat-12 M* ∗ *vec-2 v*)
    (*mat-21 M* ∗ *vec-1 v* + *mat-21 M* ∗ *vec-2 v*)

definition *ID* :: *mat2* **where** *ID* = *mat 1 0 0 1*
declare *mat-det.simps*[*simp del*]

2.1.2 Properties of 2x2 matrices

lemma *mat-neutral-element*: *mat-mul ID N* = *N* ⟨*proof*⟩

lemma *mat-associativity*: *mat-mul* (*mat-mul D B*) *C* = *mat-mul D* (*mat-mul B C*)
  ⟨*proof*⟩

lemma *mat-exp-law*: *mat-mul* (*mat-pow n M*) (*mat-pow m M*) = *mat-pow* (*n*+*m*) *M*
  ⟨*proof*⟩

lemma *mat-exp-law-mult*: *mat-pow* (*n*∗*m*) *M* = *mat-pow n* (*mat-pow m M*) (**is** *?P n*)
  ⟨*proof*⟩

lemma *det-mult*: *mat-det* (*mat-mul M1 M2*) = (*mat-det M1*) ∗ (*mat-det M2*)
  ⟨*proof*⟩

2.1.3 Special second-order recurrent sequences

Equation 3.2

fun *α*:: *nat* ⇒ *nat* ⇒ *int* **where**
    α b 0 = *0* |
    α b (*Suc 0*) = *1* |
  alpha-n: *α b* (*Suc* (*Suc n*)) = (*int b*) ∗ (*α b* (*Suc n*)) − (*α b n*)

Equation 3.3

lemma *alpha-strictly-increasing*:
  shows *int b* ≥ *2* ⟹ *α b n* < *α b* (*Suc n*) ∧ *0* < *α b* (*Suc n*)
⟨*proof*⟩

lemma *alpha-strictly-increasing-general*:

fixes b n m::*nat*
assumes $b > 2 \wedge m > n$
shows $\alpha\ b\ m > \alpha\ b\ n$
⟨*proof*⟩

Equation 3.4

lemma *alpha-superlinear*: $b > 2 \Longrightarrow int\ n \leq \alpha\ b\ n$
⟨*proof*⟩

A simple consequence that's often useful; could also be generalized to alpha using alpha linear

lemma *alpha-nonnegative*:
shows $b > 2 \Longrightarrow \alpha\ b\ n \geq 0$
⟨*proof*⟩

Equation 3.5

lemma *alpha-linear*: $\alpha\ 2\ n = n$
⟨*proof*⟩

Equation 3.6 (modified)

lemma *alpha-exponential-1*: $b > 0 \Longrightarrow int\ b\ \hat{}\ n \leq \alpha\ (b + 1)\ (n+1)$
⟨*proof*⟩

lemma *alpha-exponential-2*: $int\ b{>}2 \Longrightarrow \alpha\ b\ (n+1) \leq (int\ b)\hat{}(n)$
⟨*proof*⟩

2.1.4 First order relation

Equation 3.7 - Definition of A

fun A :: $nat \Rightarrow nat \Rightarrow mat2$ **where**
$A\ b\ 0 = mat\ 1\ 0\ 0\ 1\ |$
A-n: $A\ b\ n = mat\ (\alpha\ b\ (n + 1))\ (-(\alpha\ b\ n))\ (\alpha\ b\ n)\ (-(\alpha\ b\ (n - 1)))$

Equation 3.9 - Definition of B

fun B :: $nat \Rightarrow mat2$ **where**
$B\ b = mat\ (int\ b)\ (-1)\ 1\ 0$

declare *A.simps*[*simp del*]
declare *B.simps*[*simp del*]

Equation 3.8

lemma *A-rec*: $b{>}2 \Longrightarrow A\ b\ (Suc\ n) = mat\text{-}mul\ (A\ b\ n)\ (B\ b)$
⟨*proof*⟩

Equation 3.10

lemma *A-pow*: $b{>}2 \Longrightarrow A\ b\ n = mat\text{-}pow\ n\ (B\ b)$
⟨*proof*⟩

2.1.5 Characteristic equation

Equation 3.11

lemma *A-det*: $b>2 \Longrightarrow$ *mat-det* $(A\ b\ n) = 1$
⟨*proof*⟩

Equation 3.12

lemma *alpha-det1*:
assumes $b>2$
shows $(\alpha\ b\ (Suc\ n))\hat{}2 - (int\ b) * \alpha\ b\ (Suc\ n) * \alpha\ b\ n + (\alpha\ b\ n)\hat{}2 = 1$
⟨*proof*⟩

Equation 3.12

lemma *alpha-det2*:
assumes $b>2$ $n>0$
shows $(\alpha\ b\ (n-1))\hat{}2 - (int\ b) * (\alpha\ b\ (n-1) * (\alpha\ b\ n)) + (\alpha\ b\ n)\hat{}2 = 1$
⟨*proof*⟩

Equations 3.14 to 3.17

lemma *alpha-char-eq*:
fixes $x\ y\ b$:: *nat*
shows $(y < x \wedge x * x + y * y = 1 + b * x * y) \Longrightarrow (\exists\, m.\ int\ y = \alpha\ b\ m \wedge int\ x = \alpha\ b\ (Suc\ m))$
⟨*proof*⟩

lemma *alpha-char-eq2*:
assumes $(x*x + y*y = 1 + b * x * y)$ $b>2$
shows $(\exists\, n.\ int\ x = \alpha\ b\ n)$
⟨*proof*⟩

2.1.6 Divisibility properties

The following lemmas are needed in the proof of equation 3.25

lemma *representation*:
fixes $k\ m$ :: *nat*
assumes $k > 0$ $n = m\ mod\ k$ $l = (m-n) div\ k$
shows $m = n+k*l \wedge 0 \leq n \wedge n \leq k-1$ ⟨*proof*⟩

lemma *div-3251*:
fixes $b\ k\ m$:: *nat*
assumes $b>2$ **and** $k>0$
defines $n \equiv m\ mod\ k$
defines $l \equiv (m-n)\ div\ k$
shows $A\ b\ m =$ *mat-mul* $(A\ b\ n)$ (*mat-pow* l $(A\ b\ k))$
⟨*proof*⟩

lemma *div-3252*:
fixes $a\ b\ c\ d\ m$ :: *int* **and** l :: *nat*

defines *M* ≡ *mat a b c d*
assumes *mat-21 M mod m* = *0*
shows (*mat-21* (*mat-pow l M*)) *mod m* = *0* (**is** *?P l*)
⟨*proof*⟩

lemma *div-3253*:
fixes *a b c d m*:: *int* **and** *l* :: *nat*
defines *M* ≡ *mat a b c d*
assumes *mat-21 M mod m* = *0*
shows ((*mat-11* (*mat-pow l M*)) − *a*^*l*) *mod m* = *0* (**is** *?P l*)
⟨*proof*⟩

Equation 3.25

lemma *divisibility-lemma1*:
fixes *b k m*:: *nat*
assumes *b*>*2* **and** *k*>*0*
defines *n* ≡ *m mod k*
defines *l* ≡ (*m*−*n*) *div k*
shows *α b m mod α b k* = *α b n* ∗ (*α b* (*k*+*1*)) ^ *l mod α b k*
⟨*proof*⟩

Prerequisite lemma for 3.27

lemma *div-coprime*:
assumes *b*>*2 n* ≥ *0*
shows *coprime* (*α b k*) (*α b* (*k*+*1*)) (**is** *?P*)
⟨*proof*⟩

Equation 3.27

lemma *divisibility-lemma2*:
fixes *b k m*:: *nat*
assumes *b*>*2* **and** *k*>*0*
defines *n* ≡ *m mod k*
defines *l* ≡ (*m*−*n*) *div k*
assumes *α b k dvd α b m*
shows *α b k dvd α b n*
⟨*proof*⟩

Equation 3.23 - main result of this section

theorem *divisibility-alpha*:
assumes *b*>*2* **and** *k*>*0*
shows *α b k dvd α b m* ⟷ *k dvd m* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

2.1.7 Divisibility properties (continued)

Equation 3.28 - main result of this section

lemma *divisibility-equations*:
assumes *0*: *m* = *k*∗*l* **and** *b*>*2 m*>*0*

shows $A\ b\ m = mat\text{-}pow\ l\ (mat\text{-}minus\ (mat\text{-}scalar\text{-}mult\ (\alpha\ b\ k)\ (B\ b))$
$(mat\text{-}scalar\text{-}mult\ (\alpha\ b\ (k-1))\ ID))$
⟨*proof*⟩

lemma *divisibility-cong*:
fixes $e\ f$ :: *int*
fixes l :: *nat*
fixes M :: *mat2*
assumes $mat\text{-}22\ M = 0\ mat\text{-}21\ M = 1$
shows $(mat\text{-}21\ (mat\text{-}pow\ l\ (mat\text{-}minus\ (mat\text{-}scalar\text{-}mult\ e\ M)\ (mat\text{-}scalar\text{-}mult\ f\ ID))))\ mod\ e\hat{\ }2 = (-1)\hat{\ }(l-1)*l*e*f\hat{\ }(l-1)*(mat\text{-}21\ M)\ mod\ e\hat{\ }2$
$\wedge\ mat\text{-}22\ (mat\text{-}pow\ l\ (mat\text{-}minus\ (mat\text{-}scalar\text{-}mult\ e\ M)\ (mat\text{-}scalar\text{-}mult\ f\ ID)))\ mod\ e\hat{\ }2 = (-1)\hat{\ }l\ *f\hat{\ }l\ mod\ e\hat{\ }2$
(**is** $?P\ l \wedge ?Q\ l$)
⟨*proof*⟩

lemma *divisibility-congruence*:
assumes $m = k*l$ **and** $b>2\ m>0$
shows $\alpha\ b\ m\ mod\ (\alpha\ b\ k)\hat{\ }2 = ((-1)\hat{\ }(l-1)*l*(\alpha\ b\ k)*(\alpha\ b\ (k-1))\hat{\ }(l-1))\ mod\ (\alpha\ b\ k)\hat{\ }2$
⟨*proof*⟩

Main result section 3.5

theorem *divisibility-alpha2*:
assumes $b>2\ m>0$
shows $(\alpha\ b\ k)\hat{\ }2\ dvd\ (\alpha\ b\ m) \longleftrightarrow k*(\alpha\ b\ k)\ dvd\ m$ (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

2.1.8 Congruence properties

In this section we will need the inverse matrices of A and B

fun *A-inv* :: $nat \Rightarrow nat \Rightarrow mat2$ **where**
$A\text{-}inv\ b\ n = mat\ (-\alpha\ b\ (n-1))\ (\alpha\ b\ n)\ (-\alpha\ b\ n)\ (\alpha\ b\ (n+1))$

fun *B-inv* :: $nat \Rightarrow mat2$ **where**
$B\text{-}inv\ b = mat\ 0\ 1\ (-1)\ b$

lemma *A-inv-aux*: $b>2 \Longrightarrow n>0 \Longrightarrow \alpha\ b\ n * \alpha\ b\ n - \alpha\ b\ (Suc\ n) * \alpha\ b\ (n - Suc\ 0) = 1$
⟨*proof*⟩

lemma *A-inverse*[*simp*]: $b>2 \Longrightarrow n>0 \Longrightarrow mat\text{-}mul\ (A\text{-}inv\ b\ n)\ (A\ b\ n) = ID$
⟨*proof*⟩

lemma *B-inverse*[*simp*]: $mat\text{-}mul\ (B\ b)\ (B\text{-}inv\ b) = ID$ ⟨*proof*⟩

declare *A-inv.simps B-inv.simps*[*simp del*]

Equation 3.33

lemma *congruence*:
  assumes *b1 mod q* = *b2 mod q*
  shows α *b1 n mod q* = α *b2 n mod q*
⟨*proof*⟩

Equation 3.34

lemma *congruence2*:
  fixes *b1* :: *nat*
  assumes *b>=2*
  shows (α *b n*) *mod* (*b* − *2*) = *n mod* (*b* − *2*)
⟨*proof*⟩

lemma *congruence-jpos*:
  fixes *b m j l* :: *nat*
  assumes *b>2* **and** *2∗l∗m+j>0*
  defines *n* ≡ *2∗l∗m+j*
  shows *A b n* = *mat-mul* (*mat-pow l* (*mat-pow 2* (*A b m*))) (*A b j*)
⟨*proof*⟩

lemma *congruence-inverse*: *b>2* ⟹ *mat-pow* (*n+1*) (*B-inv b*) = *A-inv b* (*n+1*)
  ⟨*proof*⟩

lemma *congruence-inverse2*:
  fixes *n b* :: *nat*
  assumes *b>2*
  shows *mat-mul* (*mat-pow n* (*B b*)) (*mat-pow n* (*B-inv b*)) = *mat 1 0 0 1*
⟨*proof*⟩

lemma *congruence-mult*:
  fixes *m* :: *nat*
  assumes *b>2*
  shows *n>m ==> mat-pow* (*nat*(*int n*− *int m*)) (*B b*) = *mat-mul* (*mat-pow n* (*B b*)) (*mat-pow m* (*B-inv b*))
⟨*proof*⟩

lemma *congruence-jneg*:
  fixes *b m j l* :: *nat*
  assumes *b>2* **and** *2∗l∗m* > *j* **and** *j>=1*
  defines *n* ≡ *nat*(*int 2∗l∗m*− *int j*)
  shows *A b n* = *mat-mul* (*mat-pow l* (*mat-pow 2* (*A b m*))) (*A-inv b j*)
⟨*proof*⟩

lemma *matrix-congruence*:
  fixes *Y Z* :: *mat2*
  fixes *b m j l* :: *nat*
  assumes *b>2*
  defines *X* ≡ *mat-mul Y Z*
  defines *a* ≡ *mat-11 Y* **and** *b0*≡ *mat-12 Y* **and** *c* ≡ *mat-21 Y* **and** *d* ≡ *mat-22*

Y
defines $e \equiv$ *mat-11 Z* **and** $f \equiv$ *mat-12 Z* **and** $g \equiv$ *mat-21 Z* **and** $h \equiv$ *mat-22 Z*
defines $v \equiv \alpha\ b\ (m+1) - \alpha\ b\ (m-1)$
assumes *a mod v = a1 mod v* **and** *b0 mod v = b1 mod v* **and** *c mod v = c1 mod v* **and** *d mod v = d1 mod v*
shows *mat-21 X mod v* = $(c1*e+d1*g)$ *mod v* $\wedge$ *mat-22 X mod v* = $(c1*f+d1*h)$ *mod v* (**is** *?P* $\wedge$ *?Q*)
⟨*proof*⟩

3.38

lemma *congruence-Abm*:
fixes *b m n* :: *nat*
assumes $b>2$
defines $v \equiv \alpha\ b\ (m+1) - \alpha\ b\ (m-1)$
shows (*mat-21* (*mat-pow n* (*mat-pow 2* (*A b m*))) *mod v* = *0 mod v*)
$\wedge$ (*mat-22* (*mat-pow n* (*mat-pow 2* (*A b m*))) *mod v* = $((-1)\hat{}n)$ *mod v*) (**is** *?P n* $\wedge$ *?Q n*)
⟨*proof*⟩

3.36 requires two lemmas 361 and 362

lemma *361*:
fixes *b m j l* :: *nat*
assumes $b>2$
defines $n \equiv 2*l*m + j$
defines $v \equiv \alpha\ b\ (m+1) - \alpha\ b\ (m-1)$
shows $(\alpha\ b\ n)$ *mod v* = $((-1)\hat{}l * \alpha\ b\ j)$ *mod v*
⟨*proof*⟩

lemma *362*:
fixes *b m j l* :: *nat*
assumes $b>2$ **and** $2*l*m > j$ **and** $j>=1$
defines $n \equiv 2*l*m - j$
defines $v \equiv \alpha\ b\ (m+1) - \alpha\ b\ (m-1)$
shows $(\alpha\ b\ n)$ *mod v* = $-((-1)\hat{}l * \alpha\ b\ j)$ *mod v*
⟨*proof*⟩

Equation 3.36

lemma *36*:
fixes *b m j l* :: *nat*
assumes $b>2$
assumes $(n = 2 * l * m + j \vee (n = 2 * l * m - j \wedge 2 * l * m > j \wedge j \geq 1))$
defines $v \equiv \alpha\ b\ (m+1) - \alpha\ b\ (m-1)$
shows $(\alpha\ b\ n)$ *mod v* = $\alpha\ b\ j$ *mod v* $\vee$ $(\alpha\ b\ n)$ *mod v* = $-\alpha\ b\ j$ *mod v* ⟨*proof*⟩

2.1.9 Diophantine definition of a sequence alpha

definition *alpha-equations* :: *nat* ⇒ *nat* ⇒ *nat*
⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**

alpha-equations a b c r s t u v w x y = (
— 3.41 $b > 3$ ∧
— 3.42 $u\hat{}2 + t\hat{}2 = 1 + b * u * t$ ∧
— 3.43 $s\hat{}2 + r\hat{}2 = 1 + b * s * r$ ∧
— 3.44 $r < s$ ∧
— 3.45 $u\hat{}2$ *dvd* s ∧
— 3.46 $v + 2 * r = (b) * s$ ∧
— 3.47 w *mod* $v = b$ *mod* v ∧
— 3.48 w *mod* $u = 2$ *mod* u ∧
— 3.49 $2 < w$ ∧
— 3.50 $x\hat{}2 + y\hat{}2 = 1 + w * x * y$ ∧
— 3.51 $2 * a < u$ ∧
— 3.52 $2 * a < v$ ∧
— 3.53 a *mod* $v = x$ *mod* v ∧
— 3.54 $2 * c < u$ ∧
— 3.55 c *mod* $u = x$ *mod* u)

The sufficiency

lemma *alpha-equiv-suff*:
fixes *a b c::nat*
assumes ∃ *r s t u v w x y. alpha-equations a b c r s t u v w x y*
shows $3 < b \wedge$ *int* $a = (\alpha\ b\ c)$
⟨*proof*⟩

3.7.2 The necessity

lemma *add-mod*:
fixes *p q* :: *int*
assumes p *mod* $2 = 0$ q *mod* $2 = 0$
shows $(p+q)$ *mod* $2 = 0 \wedge (p-q)$ *mod* $2 = 0$
⟨*proof*⟩

lemma *one-odd*:
fixes *b n* :: *nat*
assumes $b>2$
shows $(\alpha\ b\ n)$ *mod* $2 = 1 \vee (\alpha\ b\ (n+1))$ *mod* $2 = 1$
⟨*proof*⟩

lemma *oneodd*:
fixes *b n* :: *nat*
assumes $b>2$
shows *odd* $(\alpha\ b\ n) =$ *True* ∨ *odd* $(\alpha\ b\ (n+1)) =$ *True*
⟨*proof*⟩

lemma *cong-solve-nat*: $a \neq 0 \Longrightarrow \exists x.\ (a*x)$ *mod* $n = ($*gcd* $a\ n)$ *mod* n
for *a n* :: *nat*
⟨*proof*⟩

lemma *cong-solve-coprime-nat*: *coprime* (*a::nat*) (*n::nat*) $\Longrightarrow \exists x.\ (a*x)$ *mod* $n =$ 1 *mod* n

⟨*proof*⟩

lemma *chinese-remainder-aux-nat*:
fixes *m1 m2* :: *nat*
assumes *a*:*coprime m1 m2*
shows $\exists$ *b1 b2*. *b1 mod m1* = *1 mod m1* $\wedge$ *b1 mod m2* = *0 mod m2* $\wedge$ *b2 mod m1* = *0 mod m1* $\wedge$ *b2 mod m2* = *1 mod m2*
⟨*proof*⟩

lemma *cong-scalar2-nat*: *a mod m* = *b mod m* $\Longrightarrow$ (*k*∗*a*) *mod m* = (*k*∗*b*) *mod m*
for *a b k* :: *nat*
⟨*proof*⟩

lemma *chinese-remainder-nat*:
fixes *m1 m2* :: *nat*
assumes *a*: *coprime m1 m2*
shows $\exists$ *x*. *x mod m1* = *u1 mod m1* $\wedge$ *x mod m2* = *u2 mod m2*
⟨*proof*⟩

lemma *nat-int1*: $\forall$ (*w*::*nat*) (*u*::*int*).*u*>*0* $\Longrightarrow$ (*w mod nat u* = *2 mod nat u* $\Longrightarrow$ *int w mod u* = *2 mod u*)
⟨*proof*⟩

lemma *nat-int2*: $\forall$ (*w*::*nat*) (*b*::*nat*) (*v*::*int*).*u*>*0* $\Longrightarrow$ (*w mod nat v* = *b mod nat v* $\Longrightarrow$ *int w mod v* = *int b mod v*)
⟨*proof*⟩

lemma *lem*:
fixes *u t*::*int* **and** *b*::*nat*
assumes *u*^*2*−*int b*∗*u*∗*t*+*t*^*2*=*1 u*≥*0 t*≥*0*
shows (*nat u*)^*2*+(*nat t*)^*2*=*1*+*b*∗(*nat u*)∗(*nat t*)
⟨*proof*⟩

The necessity

lemma *alpha-equiv-nec*:
b > *3* $\wedge$ *a* = α *b c* $\Longrightarrow$ $\exists$ *r s t u v w x y*. *alpha-equations a b c r s t u v w x y*
⟨*proof*⟩

2.1.10 Exponentiation is Diophantine

Equations 3.80-3.83

lemma *86*:
fixes *b r* **and** *q*::*int*
defines *m* ≡ *b* ∗ *q* − *q* ∗ *q* − *1*
shows (*q* ∗ α *b* (*r* + *1*) − α *b r*) *mod m* = (*q* ^ (*r* + *1*)) *mod m*
⟨*proof*⟩

This is a more convenient version of (86)

lemma *860*:

fixes *b r* **and** *q::int*
defines $m \equiv b * q - q * q - 1$
shows $(q * \alpha\ b\ r - (int\ b * \alpha\ b\ r - \ \alpha\ b\ (Suc\ r)))\ mod\ m = (q \mathbin{\hat{}} r)\ mod\ m$
⟨*proof*⟩

We modify the equivalence (88) in a similar manner

lemma *88*:
fixes *b r p q:: nat*
defines $m \equiv int\ b * int\ q - int\ q * int\ q - 1$
assumes $int\ q \mathbin{\hat{}} r < m$ **and** $q > 0$
shows $int\ p = int\ q \mathbin{\hat{}} r \longleftrightarrow int\ p < m\ \wedge\ (q * \alpha\ b\ r - (int\ b * \alpha\ b\ r - \ \alpha\ b$ $(Suc\ r)))\ mod\ m = int\ p\ mod\ m$
⟨*proof*⟩

lemma *89*:
fixes *r p q :: nat*
assumes $q > 0$
defines $b \equiv nat\ (\alpha\ (q + 4)\ (r + 1)) + q * q + 2$
defines $m \equiv int\ b * int\ q - int\ q * int\ q - 1$
shows $int\ q \mathbin{\hat{}} r < m$
⟨*proof*⟩
end

The final equivalence

theorem *exp-alpha*:
fixes *p q r :: nat*
shows $p = q \mathbin{\hat{}} r \longleftrightarrow ((q = 0 \wedge r = 0 \wedge p = 1) \vee$
$(q = 0 \wedge 0 < r \wedge p = 0) \vee$
$(q > 0 \wedge (\exists\, b\ m.$
$b = \ Exp\text{-}Matrices.\alpha\ (q + 4)\ (r + 1) + q * q + 2 \wedge$
$m = b * q - q * q - 1 \wedge$
$p < m \wedge$
$p\ mod\ m = ((q * Exp\text{-}Matrices.\alpha\ b\ r) - (int\ b * Exp\text{-}Matrices.\alpha$ $b\ r\ - Exp\text{-}Matrices.\alpha\ b\ (r + 1)))\ mod\ m)))$
⟨*proof*⟩

lemma *alpha-equivalence*:
fixes *a b c*
shows $3 < b \wedge int\ a = Exp\text{-}Matrices.\alpha\ b\ c \longleftrightarrow (\exists\, r\ s\ t\ u\ v\ w\ x\ y.\ Exp\text{-}Matrices.alpha\text{-}equations$ $a\ b\ c\ r\ s\ t\ u\ v\ w\ x\ y)$
⟨*proof*⟩

end

2.2 Diophantine description of alpha function

theory *Alpha-Sequence*
imports *Modulo-Divisibility Exponentiation*
begin

The alpha function is diophantine

definition *alpha* (‹[- = α - -]› *1000*)
  where [X = α B N] ≡ (*TERNARY* (λb n x. b > 3 ∧ x = *Exp-Matrices*.α b n) B N X)

lemma *alpha-dioph*[*dioph*]:
  fixes B N X
  defines D ≡ [X = α B N]
  shows *is-dioph-rel* D
⟨*proof*⟩

declare *alpha-def*[*defs*]

end

2.3 Exponentiation is a Diophantine Relation

theory *Exponential-Relation*
  imports *Alpha-Sequence Exponentiation*
begin

definition *exp-equations* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
  exp-equations p q r b m = (b = *Exp-Matrices*.α (q + 4) (r + 1) + q * q + 2 ∧
    m + q^2 + 1 = b * q ∧
    p < m ∧
    (p + b * *Exp-Matrices*.α b r) *mod* m = (q * *Exp-Matrices*.α b r +
    Exp-Matrices.α b (r + 1)) *mod* m)

lemma *exp-repr*:
  fixes p q r :: *nat*
  shows p = q^r ⟷ ((q = 0 ∧ r = 0 ∧ p = 1) ∨
    (q = 0 ∧ 0 < r ∧ p = 0) ∨
    (q > 0 ∧ (∃ b m :: *nat*. *exp-equations* p q r b m))) (**is** ?P ⟷ ?Q)
⟨*proof*⟩

definition *exp* (‹[- = - ^ -]› *1000*)
  where [Q = R ^ S] ≡ (*TERNARY* (λa b c. a = b ^ c) Q R S)

lemma *exp-dioph*[*dioph*]:
  fixes P Q R :: *polynomial*
  defines D ≡ [P = Q ^ R]
  shows *is-dioph-rel* D
⟨*proof*⟩

declare *exp-def*[*defs*]

end

2.4 Digit function is Diophantine

theory *Digit-Function*
  imports *Exponential-Relation Digit-Expansions.Bits-Digits*
begin

definition *digit* (‹[- = *Digit* - - -]› [*999*] *1000*)
  where [*D* = *Digit AA K BASE*] ≡ (*QUATERNARY* (λ*d a k b*. *b* > *1*
                                  ∧ *d* = *nth-digit a k b*) *D AA K BASE*)
lemma *mod-dioph2*[*dioph*]:
  fixes *A B C*
  defines *D* ≡ (*MOD A B C*)
  shows *is-dioph-rel D*
⟨*proof*⟩

lemma *digit-dioph*[*dioph*]:
  fixes *D A B K* :: *polynomial*
  defines *DR* ≡ [*D* = *Digit A K B*]
  shows *is-dioph-rel DR*
⟨*proof*⟩

declare *digit-def*[*defs*]

end

2.5 Binomial Coefficient is Diophantine

theory *Binomial-Coefficient*
  imports *Digit-Function*
begin

lemma *bin-coeff-diophantine*:
  shows *c* = *a choose b* ⟷ (∃ *u*.(*u* = *2*^(*Suc a*) ∧ *c* = *nth-digit* ((*u*+*1*)^*a*) *b u*))
⟨*proof*⟩

definition *binomial-coefficient* (‹[- = - *choose* -]› *1000*)
  where [*A* = *B choose C*] ≡ (*TERNARY* (λ*a b c*. *a* = *b choose c*) *A B C*)

lemma *binomial-coefficient-dioph*[*dioph*]:
  fixes *A B C* :: *polynomial*
  defines *DR* ≡ [*C* = *A choose B*]
  shows *is-dioph-rel DR*
⟨*proof*⟩

declare *binomial-coefficient-def*[*defs*] odd function is diophantine

lemma *odd-dioph-repr*:
  fixes *a* :: *nat*
  shows *odd a* ⟷ (∃ *x*::*nat*. *a* = *2*∗*x* + *1*)
  ⟨*proof*⟩

definition *odd-lift* (‹*ODD* -› [*999*] *1000*)
  where *ODD A* ≡ (*UNARY* (*odd*) *A*)

lemma *odd-dioph*[*dioph*]:
  fixes *A*
  defines *DR* ≡ (*ODD A*)
  shows *is-dioph-rel DR*
⟨*proof*⟩

declare *odd-lift-def*[*defs*]

end

2.6 Binary orthogonality is Diophantine

theory *Binary-Orthogonal*
 imports *Binomial-Coefficient Digit-Expansions.Binary-Operations Lucas-Theorem.Lucas-Theorem*
begin

lemma *equiv-with-lucas*: *nth-digit* = *Lucas-Theorem.nth-digit-general*
  ⟨*proof*⟩

lemma *lm0241-ortho-binom-equiv*:(*a* ⊥ *b*) ⟷ *odd* ((*a* + *b*) *choose b*) (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

definition *orthogonal* (**infix** ‹[⊥]› *50*)
  where *P* [⊥] *Q* ≡ (*BINARY* (λ*a b*. *a* ⊥ *b*) *P Q*)

lemma *orthogonal-dioph*[*dioph*]:
  fixes *P Q*
  defines *DR* ≡ (*P* [⊥] *Q*)
  shows *is-dioph-rel DR*
⟨*proof*⟩

declare *orthogonal-def*[*defs*]

end

2.7 Binary masking is Diophantine

theory *Binary-Masking*
  imports *Binary-Orthogonal*

begin

lemma *lm0243-masks-binom-equiv*: ($b \preceq c$) $\longleftrightarrow$ *odd* (*c choose b*) (**is** *?P* $\longleftrightarrow$ *?Q*)
⟨*proof*⟩

definition *masking* (‹- [$\preceq$] -› *60*)
  where P [$\preceq$] $Q \equiv$ (*BINARY* ($\lambda a\ b.\ a \preceq b$) $P\ Q$)

lemma *masking-dioph*[*dioph*]:
  fixes $P\ Q$
  defines $DR \equiv$ (P [$\preceq$] Q)
  shows *is-dioph-rel DR*
⟨*proof*⟩

declare *masking-def*[*defs*]

end

2.8 Binary and is Diophantine

theory *Binary-And*
  imports *Binary-Masking Binary-Orthogonal*
begin

lemma *lm0244*: (a && b) $\preceq a$
⟨*proof*⟩

lemma *lm0245*: (a && b) $\preceq b$
  ⟨*proof*⟩

lemma *bitAND-lt-left*: m && $n \leq m$
  ⟨*proof*⟩
lemma *bitAND-lt-right*: m && $n \leq n$
  ⟨*proof*⟩

lemmas *bitAND-lt* = *bitAND-lt-right bitAND-lt-left*

lemma *auxm3-lm0246*:
  shows *bin-carry a b k* = *bin-carry a b k mod 2*
  ⟨*proof*⟩

lemma *auxm2-lm0246*:
  assumes ($\forall r < n$.(*nth-bit a r* + *nth-bit b r* $\leq$ *1*))
  shows (*nth-bit* (a+b) n) = (*nth-bit a n* + *nth-bit b n*) *mod 2*
  ⟨*proof*⟩

lemma *auxm1-lm0246*: $a \preceq (a+b) \Longrightarrow$ ($\forall n$. *nth-bit a n* + *nth-bit b n* $\leq$ *1*) (**is** *?P*
$\Longrightarrow$ *?Q*)
⟨*proof*⟩

lemma *aux0-lm0246*:$a \preceq (a+b) \longrightarrow (a+b)¡\ n = a\ ¡\ n + b\ ¡\ n$
⟨*proof*⟩

lemma *aux1-lm0246*:$a \preceq b \longrightarrow (\forall n.\ nth\text{-}bit\ (b-a)\ n = nth\text{-}bit\ b\ n - nth\text{-}bit\ a\ n)$
⟨*proof*⟩

lemma *lm0246*:$(a - (a\ \&\&\ b)) \perp (b - (a\ \&\&\ b))$
⟨*proof*⟩

lemma *aux0-lm0247*:$(nth\text{-}bit\ a\ k) * (nth\text{-}bit\ b\ k) \leq 1$
⟨*proof*⟩

lemma *lm0247-masking-equiv*:
fixes *a b c* :: *nat*
shows $(c = a\ \&\&\ b) \longleftrightarrow (c \preceq a \wedge c \preceq b \wedge (a - c) \perp (b - c))$ (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

definition *binary-and* (‹[- = - && -]› *1000*)
where [*A* = *B* && *C*] ≡ (*TERNARY* (λ*a b c*. *a* = *b* && *c*) *A B C*)

lemma *binary-and-dioph*[*dioph*]:
fixes *A B C* :: *polynomial*
defines *DR* ≡ [*A* = *B* && *C*]
shows *is-dioph-rel DR*
⟨*proof*⟩

declare *binary-and-def*[*defs*]

definition *binary-and-attempt* :: *polynomial* ⇒ *polynomial* ⇒ *polynomial* (‹- &? -›) **where**
A &? *B* ≡ *Const 0*

end

3 Register Machines

3.1 Register Machine Specification

theory *RegisterMachineSpecification*
imports *Main*
begin

3.1.1 Basic Datatype Definitions

The following specification of register machines is inspired by [8] (see [9] for the corresponding AFP article).

```
type-synonym register = nat
type-synonym tape = register list


type-synonym state = nat
datatype instruction =
  isadd: Add (modifies : register) (goes-to : state) |
  issub: Sub (modifies : register) (goes-to : state) (goes-to-alt : state) |
  ishalt: Halt
where
  modifies Halt = 0 |
  goes-to-alt (Add - next) = next


type-synonym program = instruction list


type-synonym configuration = (state ∗ tape)
```

3.1.2 Essential Functions to operate the Register Machine

```
definition read :: tape ⇒ program ⇒ state ⇒ nat
  where read t p s = t ! (modifies (p!s))

definition fetch :: state ⇒ program ⇒ nat ⇒ state where
  fetch s p v = (if issub (p!s) ∧ v = 0 then goes-to-alt (p!s)
                    else if ishalt (p!s) then s
                    else goes-to (p!s))

definition update :: tape ⇒ instruction ⇒ tape where
  update t i = (if ishalt i then t
                 else if isadd i then list-update t (modifies i) (t!(modifies i) + 1)
                    else list-update t (modifies i) (if t!(modifies i) = 0 then 0 else
(t!(modifies i)) − 1) )

definition step :: configuration ⇒ program ⇒ configuration
  where
    (step ic p) = (let nexts = fetch (fst ic) p (read (snd ic) p (fst ic));
                      nextt = update (snd ic) (p!(fst ic))
                      in (nexts, nextt))

fun steps :: configuration ⇒ program ⇒ nat ⇒ configuration
  where
    steps-zero: (steps c p 0) = c
  | steps-suc:  (steps c p (Suc n)) = (step (steps c p n) p)
```

3.1.3 Validity Checks and Assumptions

```
fun instruction-state-check :: nat ⇒ instruction ⇒ bool
  where isc-halt: instruction-state-check - Halt = True
```

| *isc-add*: *instruction-state-check m* (*Add - ns*) = (*ns* < *m*)
| *isc-sub*: *instruction-state-check m* (*Sub - ns1 ns2*) = ((*ns1* < *m*) & (*ns2* < *m*))

fun *instruction-register-check* :: *nat* ⇒ *instruction* ⇒ *bool*
where *instruction-register-check - Halt* = *True*
| *instruction-register-check n* (*Add reg -*) = (*reg* < *n*)
| *instruction-register-check n* (*Sub reg - -*) = (*reg* < *n*)

fun *program-state-check* :: *program* ⇒ *bool*
where *program-state-check p* = *list-all* (*instruction-state-check* (*length p*)) *p*

fun *program-register-check* :: *program* ⇒ *nat* ⇒ *bool*
where *program-register-check p n* = *list-all* (*instruction-register-check n*) *p*

fun *tape-check-initial* :: *tape* ⇒ *nat* ⇒ *bool*
where *tape-check-initial t a* = (*t* ≠ [] ∧ *t*!*0* = *a* ∧ (∀ *l*>*0*. *t* ! *l* = *0*))

fun *program-includes-halt* :: *program* ⇒ *bool*
where *program-includes-halt p* = (*length p* > *1* ∧ *ishalt* (*p* ! (*length p* −*1*)) ∧ (∀ *k*<*length p*−*1*. ¬ *ishalt* (*p*!*k*)))

Is Valid and Terminates

definition *is-valid*
where *is-valid c p* = (*program-includes-halt p* ∧ *program-state-check p*
∧ (*program-register-check p* (*length* (*snd c*))))

definition *is-valid-initial*
where *is-valid-initial c p a* = ((*is-valid c p*)
∧ (*tape-check-initial* (*snd c*) *a*)
∧ (*fst c* = *0*))

definition *correct-halt*
where *correct-halt c p q* = (*ishalt* (*p* ! (*fst* (*steps c p q*))) — halting
∧ (∀ *l*<(*length* (*snd c*)). *snd* (*steps c p q*) ! *l* = *0*))

definition *terminates* :: *configuration* ⇒ *program* ⇒ *nat* ⇒ *bool*
where *terminates c p q* = ((*q*>*0*)
∧ (*correct-halt c p q*)
∧ (∀ *x*<*q*. ¬ *ishalt* (*p* ! (*fst* (*steps c p x*)))))

definition *initial-config* :: *nat* ⇒ *nat* ⇒ *configuration* **where**
initial-config n a = (*0*, (*a* # *replicate n 0*))

end

3.2 Simple Properties of Register Machines

theory *RegisterMachineProperties*
  imports *RegisterMachineSpecification*
begin

lemma *step-commutative*: *steps* (*step c p*) *p t* = *step* (*steps c p t*) *p*
  ⟨*proof*⟩

lemma *step-fetch-correct*:
  fixes *t* :: *nat*
    and *c* :: *configuration*
    and *p* :: *program*
  assumes *is-valid c p*
  defines *ct* ≡ (*steps c p t*)
  shows *fst* (*steps* (*step c p*) *p t*) = *fetch* (*fst ct*) *p* (*read* (*snd ct*) *p* (*fst ct*))
  ⟨*proof*⟩

3.2.1 From Configurations to a Protocol

Register Values

definition *R* :: *configuration* ⇒ *program* ⇒ *nat* ⇒ *nat* ⇒ *nat*
  where *R c p n t* = (*snd* (*steps c p t*)) ! *n*

fun *RL* :: *configuration* ⇒ *program* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
  RL c p b 0 l = ((*snd c*) ! *l*) |
  RL c p b (*Suc t*) *l* = ((*snd c*) ! *l*) + *b* ∗ (*RL* (*step c p*) *p b t l*)

lemma *RL-simp-aux*:
  ‹*snd c* ! *l* + *b* ∗ *RL* (*step c p*) *p b t l* =
    RL c p b t l + *b* ∗ (*b* ^ *t* ∗ *snd* (*step* (*steps c p t*) *p*) ! *l*)›
  ⟨*proof*⟩

declare *RL.simps*[*simp del*]
lemma *RL-simp*:
  RL c p b (*Suc t*) *l* = (*snd* (*steps c p* (*Suc t*)) ! *l*) ∗ *b* ^(*Suc t*) + (*RL c p b t l*)
⟨*proof*⟩

State Values

definition *S* :: *configuration* ⇒ *program* ⇒ *nat* ⇒ *nat* ⇒ *nat*
  where *S c p k t* = (*if* (*fst* (*steps c p t*) = *k*) *then* (*Suc 0*) *else 0*)

definition *S2* :: *configuration* ⇒ *nat* ⇒ *nat*
  where *S2 c k* = (*if* (*fst c*) = *k then 1 else 0*)

fun *SK* :: *configuration* ⇒ *program* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
  where *SK c p b 0 k* = (*S2 c k*) |
   SK c p b (*Suc t*) *k* = (*S2 c k*) + *b* ∗ (*SK* (*step c p*) *p b t k*)

lemma *SK-simp-aux*:
‹*SK c p b* (*Suc* (*Suc t*)) *k* =
S2 (*steps c p* (*Suc* (*Suc t*))) *k* ∗ *b* ^ *Suc* (*Suc t*) + *SK c p b* (*Suc t*) *k*›
⟨*proof*⟩

declare *SK.simps*[*simp del*]
lemma *SK-simp*:
SK c p b (*Suc t*) *k* = (*S2* (*steps c p* (*Suc t*)) *k*) ∗ *b* ^(*Suc t*) + (*SK c p b t k*)
⟨*proof*⟩

Zero-Indicator Values

definition *Z* :: *configuration* ⇒ *program* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
Z c p n t = (*if* (*R c p n t* > *0*) *then 1 else 0*)

definition *Z2* :: *configuration* ⇒ *nat* ⇒ *nat* **where**
Z2 c n = (*if* (*snd c*) ! *n* > *0 then 1 else 0*)

fun *ZL* :: *configuration* ⇒ *program* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where *ZL c p b 0 l* = (*Z2 c l*) |
ZL c p b (*Suc t*) *l* = (*Z2 c l*) + *b* ∗ (*ZL* (*step c p*) *p b t l*)

lemma *ZL-simp-aux*:
Z2 c l + *b* ∗ *ZL* (*step c p*) *p b t l* =
ZL c p b t l + *b* ∗ (*b* ^ *t* ∗ *Z2* (*step* (*steps c p t*) *p*) *l*)
⟨*proof*⟩

declare *ZL.simps*[*simp del*]
lemma *ZL-simp*:
ZL c p b (*Suc t*) *l* = (*Z2* (*steps c p* (*Suc t*)) *l*) ∗ *b* ^(*Suc t*) + (*ZL c p b t l*)
⟨*proof*⟩

3.2.2 Protocol Properties

lemma *Z-bounded*: *Z c p l t* ≤ *1*
⟨*proof*⟩

lemma *S-bounded*: *S c p k t* ≤ *1*
⟨*proof*⟩

lemma *S-unique*: ∀ *k*≤*length p*. (*k* ≠ *fst* (*steps c p t*) ⟶ *S c p k t* = *0*)
⟨*proof*⟩

fun *cells-bounded* :: *configuration* ⇒ *program* ⇒ *nat* ⇒ *bool* **where**
cells-bounded conf p c = ((∀ *l*<(*length* (*snd conf*)). ∀ *t*. *2*^*c* > *R conf p l t*)
∧ (∀ *k t*. *2*^*c* > *S conf p k t*)
∧ (∀ *l t*. *2*^*c* > *Z conf p l t*))

lemma *steps-tape-length-invar*: *length* (*snd* (*steps c p t*)) = *length* (*snd c*)
⟨*proof*⟩

lemma *step-is-valid-invar*: *is-valid c p* ⟹ *is-valid* (*step c p*) *p*
⟨*proof*⟩

fun *fetch-old*
where
(*fetch-old p s* (*Add r next*) -) = *next*
| (*fetch-old p s* (*Sub r next nextalt*) *val*) = (*if val* = *0 then nextalt else next*)
| (*fetch-old p s Halt* -) = *s*

lemma *fetch-equiv*:
assumes *i* = *p*!*s*
shows *fetch s p v* = *fetch-old p s i v*
⟨*proof*⟩

lemma *p-contains*: *is-valid-initial ic p a* ⟹ (*fst* (*steps ic p t*)) < *length p*
⟨*proof*⟩

lemma *steps-is-valid-invar*: *is-valid c p* ⟹ *is-valid* (*steps c p t*) *p*
⟨*proof*⟩

lemma *terminates-halt-state*: *terminates ic p q* ⟹ *is-valid-initial ic p a*
⟹ *ishalt* (*p* ! (*fst* (*steps ic p q*)))
⟨*proof*⟩

lemma *R-termination*:
fixes *l* :: *register* **and** *ic* :: *configuration*
assumes *is-val*: *is-valid ic p* **and** *terminate*: *terminates ic p q* **and** *l*: *l* < *length* (*snd ic*)
shows ∀ *t*≥*q*. *R ic p l t* = *0*
⟨*proof*⟩

lemma *terminate-c-exists*: *is-valid ic p* ⟹ *terminates ic p q* ⟹ ∃ *c*>*1*. *cells-bounded ic p c*
⟨*proof*⟩

end

3.3 Simulation of a Register Machine

theory *RegisterMachineSimulation*
imports *RegisterMachineProperties Digit-Expansions.Binary-Operations*
begin

definition *B* :: *nat* ⇒ *nat* **where**
(*B c*) = *2*^(*Suc c*)

definition *RLe c p b q l* = (∑ *t* = *0..q. b^t* ∗ *R c p l t*)
definition *SKe c p b q k* = (∑ *t* = *0..q. b^t* ∗ *S c p k t*)
definition *ZLe c p b q l* = (∑ *t* = *0..q. b^t* ∗ *Z c p l t*)

fun *sum-radd* :: *program* ⇒ *register* ⇒ (*nat* ⇒ *nat*) ⇒ *nat*
where *sum-radd p l f* = (∑ *k* = *0..length p−1. if isadd* (*p*!*k*) ∧ *l* = *modifies* (*p*!*k*) *then f k else 0*)

abbreviation *sum-radd-abbrev* (‹∑ *R*+ - - -› [*999*, *999*, *999*] *1000*)
where (∑ *R*+ *p l f*) ≡ (*sum-radd p l f*)

fun *sum-rsub* :: *program* ⇒ *register* ⇒ (*nat* ⇒ *nat*) ⇒ *nat*
where *sum-rsub p l f* = (∑ *k* = *0..length p−1. if issub* (*p*!*k*) ∧ *l* = *modifies* (*p*!*k*) *then f k else 0*)

abbreviation *sum-rsub-abbrev* (‹∑ *R*− - - - › [*999*, *999*, *999*] *1000*)
where (∑ *R*− *p l f*) ≡ (*sum-rsub p l f*)

fun *sum-sadd* :: *program* ⇒ *state* ⇒ (*nat* ⇒ *nat*) ⇒ *nat*
where *sum-sadd p d f* = (∑ *k* = *0..length p−1. if isadd* (*p*!*k*) ∧ *d* = *goes-to* (*p*!*k*) *then f k else 0*)

abbreviation *sum-sadd-abbrev* (‹∑ *S*+ - - - › [*999*, *999*, *999*] *1000*)
where (∑ *S*+ *p d f*) ≡ (*sum-sadd p d f*)

fun *sum-ssub-nzero* :: *program* ⇒ *state* ⇒ (*nat* ⇒ *nat*) ⇒ *nat*
where *sum-ssub-nzero p d f* = (∑ *k* = *0..length p−1. if issub* (*p*!*k*) ∧ *d* = *goes-to* (*p*!*k*) *then f k else 0*)

abbreviation *sum-ssub-nzero-abbrev* (‹∑ *S*− - - - › [*999*, *999*, *999*] *1000*)
where (∑ *S*− *p d f*) ≡ (*sum-ssub-nzero p d f*)

fun *sum-ssub-zero* :: *program* ⇒ *state* ⇒ (*nat* ⇒ *nat*) ⇒ *nat*
where *sum-ssub-zero p d f* = (∑ *k* = *0..length p−1. if issub* (*p*!*k*) ∧ *d* = *goes-to-alt* (*p*!*k*) *then f k else 0*)

abbreviation *sum-ssub-zero-abbrev* (‹∑ *S0* - - - › [*999*, *999*, *999*] *1000*)
where (∑ *S0 p d f*) ≡ (*sum-ssub-zero p d f*)

declare *sum-radd.simps*[*simp del*]
declare *sum-rsub.simps*[*simp del*]
declare *sum-sadd.simps*[*simp del*]
declare *sum-ssub-nzero.simps*[*simp del*]
declare *sum-ssub-zero.simps*[*simp del*]

Special sum cong lemmas

lemma *sum-sadd-cong*:
assumes $\forall k.\ k \leq length\ p{-}1 \wedge isadd\ (p!k) \wedge l = goes\text{-}to\ (p!k) \longrightarrow f\ k = g\ k$
shows $\sum S{+}\ p\ l\ f = \sum S{+}\ p\ l\ g$
⟨*proof*⟩

lemma *sum-ssub-nzero-cong*:
assumes $\forall k.\ k \leq length\ p - 1 \wedge issub\ (p!k) \wedge l = goes\text{-}to\ (p!k) \longrightarrow f\ k = g\ k$
shows $\sum S{-}\ p\ l\ f = \sum S{-}\ p\ l\ g$
⟨*proof*⟩

lemma *sum-ssub-zero-cong*:
assumes $\forall k.\ k \leq length\ p - 1 \wedge issub\ (p!k) \wedge l = goes\text{-}to\text{-}alt\ (p!k) \longrightarrow f\ k = g$
k
shows $\sum S0\ p\ l\ f = \sum S0\ p\ l\ g$
⟨*proof*⟩

lemma *sum-radd-cong*:
assumes $\forall k.\ k \leq length\ p - 1 \wedge isadd\ (p!k) \wedge l = modifies\ (p!k) \longrightarrow f\ k = g\ k$
shows $\sum R{+}\ p\ l\ f = \sum R{+}\ p\ l\ g$
⟨*proof*⟩

lemma *sum-rsub-cong*:
assumes $\forall k.\ k \leq length\ p - 1 \wedge issub\ (p!k) \wedge l = modifies\ (p!k) \longrightarrow f\ k = g\ k$
shows $\sum R{-}\ p\ l\ f = \sum R{-}\ p\ l\ g$
⟨*proof*⟩

Properties and simple lemmas

lemma *RLe-equivalent*: $RL\ c\ p\ b\ q\ l = RLe\ c\ p\ b\ q\ l$
⟨*proof*⟩

lemma *SKe-equivalent*: $SK\ c\ p\ b\ q\ k = SKe\ c\ p\ b\ q\ k$
⟨*proof*⟩

lemma *ZLe-equivalent*: $ZL\ c\ p\ b\ q\ l = ZLe\ c\ p\ b\ q\ l$
⟨*proof*⟩

lemma *sum-radd-distrib*: $a * (\sum R{+}\ p\ l\ f) = (\sum R{+}\ p\ l\ (\lambda k.\ a * f\ k))$
⟨*proof*⟩

lemma *sum-rsub-distrib*: $a * (\sum R{-}\ p\ l\ f) = (\sum R{-}\ p\ l\ (\lambda k.\ a * f\ k))$
⟨*proof*⟩

lemma *sum-sadd-distrib*: $a * (\sum S{+}\ p\ d\ f) = (\sum S{+}\ p\ d\ (\lambda k.\ a * f\ k))$ **for** a
⟨*proof*⟩

lemma *sum-ssub-nzero-distrib*: $a * (\sum S{-}\ p\ d\ f) = (\sum S{-}\ p\ d\ (\lambda k.\ a * f\ k))$ **for**
a
⟨*proof*⟩

lemma *sum-ssub-zero-distrib*: $a * (\sum S0\ p\ d\ f) = (\sum S0\ p\ d\ (\lambda k.\ a * f\ k))$ **for** a
⟨*proof*⟩

lemma *sum-distrib*:
fixes SX :: *program* ⇒ *nat* ⇒ (*nat* ⇒ *nat*) ⇒ *nat*
and p :: *program*

assumes *SX-simps*: $\bigwedge h.\ SX\ p\ x\ h = (\sum k = 0..length\ p{-}1.\ if\ g\ x\ k\ then\ h\ k\ else\ 0)$

shows $SX\ p\ x\ h1 + SX\ p\ x\ h2 = SX\ p\ x\ (\lambda k.\ h1\ k + h2\ k)$
⟨*proof*⟩

lemma *sum-commutative*:
fixes SX :: *program* ⇒ *nat* ⇒ (*nat* ⇒ *nat*) ⇒ *nat*
and p :: *program*

assumes *SX-simps*: $\bigwedge h.\ SX\ p\ x\ h = (\sum k = 0..length\ p{-}1.\ if\ g\ x\ k\ then\ h\ k\ else\ 0)$

shows $(\sum t{=}0..q{::}nat.\ SX\ p\ x\ (\lambda k.\ f\ k\ t))$
$= (SX\ p\ x\ (\lambda k.\ \sum t{=}0..q.\ f\ k\ t))$
⟨*proof*⟩

lemma *sum-radd-commutative*: $(\sum t{=}0..(q{::}nat).\ \sum R{+}\ p\ l\ (\lambda k.\ f\ k\ t)) = (\sum R{+}\ p\ l\ (\lambda k.\ \sum t{=}0..q.\ f\ k\ t))$
⟨*proof*⟩
lemma *sum-rsub-commutative*: $(\sum t{=}0..(q{::}nat).\ \sum R{-}\ p\ l\ (\lambda k.\ f\ k\ t)) = (\sum R{-}\ p\ l\ (\lambda k.\ \sum t{=}0..q.\ f\ k\ t))$
⟨*proof*⟩
lemma *sum-sadd-commutative*: $(\sum t{=}0..(q{::}nat).\ \sum S{+}\ p\ l\ (\lambda k.\ f\ k\ t)) = (\sum S{+}\ p\ l\ (\lambda k.\ \sum t{=}0..q.\ f\ k\ t))$
⟨*proof*⟩
lemma *sum-ssub-nzero-commutative*: $(\sum t{=}0..(q{::}nat).\ \sum S{-}\ p\ l\ (\lambda k.\ f\ k\ t)) = (\sum S{-}\ p\ l\ (\lambda k.\ \sum t{=}0..q.\ f\ k\ t))$
⟨*proof*⟩
lemma *sum-ssub-zero-commutative*: $(\sum t{=}0..(q{::}nat).\ \sum S0\ p\ l\ (\lambda k.\ f\ k\ t)) = (\sum S0\ p\ l\ (\lambda k.\ \sum t{=}0..q.\ f\ k\ t))$
⟨*proof*⟩

lemma *sum-int*: $c \le a + b \Longrightarrow int(a + b - c) = int(a) + int(b) - int(c)$
⟨*proof*⟩

lemma *ZLe-bounded*: $b > 2 \Longrightarrow ZLe\ c\ p\ b\ q\ l < b\ \hat{}\ (Suc\ q)$
⟨*proof*⟩

lemma *SKe-bounded*: $b > 2 \Longrightarrow SKe\ c\ p\ b\ q\ k < b\ \hat{}\ (Suc\ q)$
⟨*proof*⟩

lemma *mult-to-bitAND*:
  assumes *cells-bounded*: *cells-bounded ic p c*
and $c > 1$
and $b = B\ c$

shows $(\sum t{=}0..q.\ b\hat{\ }t * (Z\ ic\ p\ l\ t * S\ ic\ p\ k\ t))$
$= ZLe\ ic\ p\ b\ q\ l\ \&\&\ SKe\ ic\ p\ b\ q\ k$
⟨*proof*⟩

lemma *sum-bt*:
  fixes *b q* :: *nat*
  assumes $b > 2$
  shows $(\sum t = 0..q.\ b\hat{\ }t) < b\ \hat{\ }(Suc\ q)$
  ⟨*proof*⟩

lemma *mult-to-bitAND-state*:
  assumes *cells-bounded*: *cells-bounded ic p c*
and *c*: $c > 1$
and *b*: $b = B\ c$

shows $(\sum t{=}0..q.\ b\hat{\ }t * ((1 - Z\ ic\ p\ l\ t) * S\ ic\ p\ k\ t))$
$= ((\sum t = 0..q.\ b\hat{\ }t) - ZLe\ ic\ p\ b\ q\ l)\ \&\&\ SKe\ ic\ p\ b\ q\ k$
⟨*proof*⟩

end

3.4 Single step relations

3.4.1 Registers

theory *SingleStepRegister*
  imports *RegisterMachineSimulation*
begin

lemma *single-step-add*:
  fixes *c* :: *configuration*
    and *p* :: *program*
    and *l* :: *register*
    and *t a* :: *nat*

defines $cs \equiv fst\ (steps\ c\ p\ t)$

assumes *is-val*: *is-valid-initial c p a*
    and *l*: $l < length\ tape$

shows $(\sum R{+}\ p\ l\ (\lambda k.\ S\ c\ p\ k\ t))$
$= (if\ isadd\ (p!cs) \wedge l = modifies\ (p!cs)\ then\ 1\ else\ 0)$
⟨*proof*⟩

lemma *single-step-sub*:
fixes c :: *configuration*
and p :: *program*
and l :: *register*
and t a :: *nat*

defines $cs \equiv fst\ (steps\ c\ p\ t)$

assumes *is-val*: *is-valid-initial* $c\ p\ a$

shows $(\sum R-\ p\ l\ (\lambda k.\ Z\ c\ p\ l\ t * S\ c\ p\ k\ t))$
$= (if\ issub\ (p!cs) \wedge l = modifies\ (p!cs)\ then\ Z\ c\ p\ l\ t\ else\ 0)$
⟨*proof*⟩

lemma *lm04-06-one-step-relation-register-old*:
fixes l::*register*
and ic::*configuration*
and p::*program*

defines $s \equiv fst\ ic$
and $tape \equiv snd\ ic$

defines $m \equiv length\ p$
and $tape' \equiv snd\ (step\ ic\ p)$

assumes *is-val*: *is-valid* $ic\ p$
and l: ‹$l < length\ tape$›

shows $(tape'!l) = (tape!l) + (if\ isadd\ (p!s) \wedge l = modifies\ (p!s)\ then\ 1\ else\ 0)$
$- Z\ ic\ p\ l\ 0 * (if\ issub\ (p!s) \wedge l = modifies\ (p!s)\ then\ 1$
$else\ 0)$
⟨*proof*⟩

lemma *lm04-06-one-step-relation-register*:
fixes l :: *register*
and c :: *configuration*
and p :: *program*
and t :: *nat*
and a :: *nat*

defines $r \equiv R\ c\ p$
defines $s \equiv S\ c\ p$

assumes *is-val*: *is-valid-initial* $c\ p\ a$
and l: $l < length\ (snd\ c)$

shows $r\ l\ (Suc\ t) = r\ l\ t + (\sum R+\ p\ l\ (\lambda k.\ s\ k\ t))$
$- (\sum R-\ p\ l\ (\lambda k.\ (Z\ c\ p\ l\ t) * s\ k\ t))$

⟨*proof*⟩

end

3.4.2 States

theory *SingleStepState*
 imports *RegisterMachineSimulation*
begin

lemma *lm04-07-one-step-relation-state*:
 fixes d :: *state*
  and c :: *configuration*
  and p :: *program*
  and t :: *nat*
  and a :: *nat*

defines $r \equiv R\ c\ p$
defines $s \equiv S\ c\ p$
defines $z \equiv Z\ c\ p$
defines $cs \equiv fst\ (steps\ c\ p\ t)$

assumes *is-val*: *is-valid-initial* $c\ p\ a$
  and $d < length\ p$

shows $s\ d\ (Suc\ t) = (\sum S{+}\ p\ d\ (\lambda k.\ s\ k\ t))$
$+ (\sum S{-}\ p\ d\ (\lambda k.\ z\ (modifies\ (p!k))\ t * s\ k\ t))$
$+ (\sum S0\ p\ d\ (\lambda k.\ (1 - z\ (modifies\ (p!k))\ t) * s\ k\ t))$
$+ (if\ ishalt\ (p!cs) \wedge d = cs\ then\ Suc\ 0\ else\ 0)$
⟨*proof*⟩

end

3.5 Multiple step relations

3.5.1 Registers

theory *MultipleStepRegister*
 imports *SingleStepRegister*
begin

lemma *lm04-22-multiple-register*:
 fixes c :: *nat*
  and l :: *register*
  and ic :: *configuration*
  and p :: *program*
  and q :: *nat*
  and a :: *nat*

 defines $b == B\ c$

and *m* == *length p*
and *n* == *length* (*snd ic*)

assumes *is-val*: *is-valid-initial ic p a*
assumes *c-gt-cells*: *cells-bounded ic p c*
assumes *l*: $l < n$
and $0 < l$
and *q*: $q > 0$

assumes *terminate*: *terminates ic p q*

assumes *c*: $c > 1$

defines *r* == *RLe ic p b q*
and *z* == *ZLe ic p b q*
and *s* == *SKe ic p b q*

shows $r\ l = b * r\ l$
$+ b * (\sum R+\ p\ l\ s)$
$- b * (\sum R-\ p\ l\ (\lambda k.\ z\ l\ \&\&\ s\ k))$
⟨*proof*⟩

lemma *lm04-23-multiple-register1*:
fixes *c* :: *nat*
and *l* :: *register*
and *ic* :: *configuration*
and *p* :: *program*
and *q* :: *nat*
and *a* :: *nat*

defines *b* == *B c*
and *m* == *length p*
and *n* == *length* (*snd ic*)

assumes *is-val*: *is-valid-initial ic p a*
assumes *c-gt-cells*: *cells-bounded ic p c*
assumes *l*: $l = 0$
and *q*: $q > 0$

assumes *c*: $c > 1$

assumes *terminate*: *terminates ic p q*

defines *r* == *RLe ic p b q*
and *z* == *ZLe ic p b q*
and *s* == *SKe ic p b q*

shows $r\ l = a + b * r\ l$
$+ b * (\sum R+\ p\ l\ s)$

$-\ b * (\sum R-\ p\ l\ (\lambda k.\ z\ l\ \&\&\ s\ k))$
⟨*proof*⟩

end

3.5.2 States

theory *MultipleStepState*
imports *SingleStepState*
begin

lemma *lm04-24-multiple-step-states*:
fixes *c* :: *nat*
and *l* :: *register*
and *ic* :: *configuration*
and *p* :: *program*
and *q* :: *nat*
and *a* :: *nat*

defines $b == B\ c$
and $m == length\ p$

assumes *is-val*: *is-valid-initial ic p a*
assumes *c-gt-cells*: *cells-bounded ic p c*
assumes *d*: $d \leq m-1$ **and** $0 < d$
and *q*: $q > 0$

assumes *terminate*: *terminates ic p q*

assumes *c*: $c > 1$

defines $r \equiv RLe\ ic\ p\ b\ q$
and $z \equiv ZLe\ ic\ p\ b\ q$
and $s \equiv SKe\ ic\ p\ b\ q$
and $e \equiv \sum t = 0..q.\ b\hat{}t$

shows $s\ d = b * (\sum S+\ p\ d\ s)$
$+\ b * (\sum S-\ p\ d\ (\lambda k.\ z\ (modifies\ (p!k))\ \&\&\ s\ k))$
$+\ b * (\sum S0\ p\ d\ (\lambda k.\ (e - z\ (modifies\ (p!k)))\ \&\&\ s\ k))$
⟨*proof*⟩

lemma *lm04-25-multiple-step-state1*:
fixes *c* :: *nat*
and *l* :: *register*
and *ic* :: *configuration*
and *p* :: *program*
and *q* :: *nat*
and *a* :: *nat*

defines *b* == *B c*
and *m* == *length p*

assumes *is-val*: *is-valid-initial ic p a*
assumes *c-gt-cells*: *cells-bounded ic p c*
assumes *d*: *d=0*
and *q*: *q* > *0*

assumes *terminate*: *terminates ic p q*

assumes *c*: *c* > *1*

defines *r* ≡ *RLe ic p b q*
and *z* ≡ *ZLe ic p b q*
and *s* ≡ *SKe ic p b q*
and *e* ≡ $\sum$ *t* = *0..q*. *b*^*t*

shows *s d* = *1* + *b* ∗ ($\sum$ *S*+ *p d s*)
+ *b* ∗ ($\sum$ *S*− *p d* (λ*k*. *z* (*modifies* (*p*!*k*)) && *s k*))
+ *b* ∗ ($\sum$ *S0 p d* (λ*k*. (*e* − *z* (*modifies* (*p*!*k*))) && *s k*))
⟨*proof*⟩

lemma *halting-condition-04-27*:
fixes *c* :: *nat*
and *l* :: *register*
and *ic* :: *configuration*
and *p* :: *program*
and *q* :: *nat*
and *a* :: *nat*

defines *b* == *B c*
and *m* == *length p* − *1*

assumes *is-val*: *is-valid-initial ic p a*
and *q*: *q* > *0*

assumes *terminate*: *terminates ic p q*

shows *SKe ic p b q m* = *b* ^ *q*
⟨*proof*⟩

lemma *state-q-bound*:
fixes *c* :: *nat*
and *l* :: *register*
and *ic* :: *configuration*
and *p* :: *program*
and *q* :: *nat*
and *a* :: *nat*

defines *b* == *B c*
and *m* == *length p* − *1*

assumes *is-val*: *is-valid-initial ic p a*
and *q*: *q* > *0*
and *terminate*: *terminates ic p q*
and *c*: *c* > *0*

assumes *k*<*m*

shows *SKe ic p b q k* < *b* ^ *q*
⟨*proof*⟩

end

3.6 Masking properties

theory *MachineMasking*
imports *RegisterMachineSimulation* ../*Diophantine*/*Binary-And*
begin

definition *E* :: *nat* ⇒ *nat* ⇒ *nat* **where**
(*E q b*) = (∑ *t* = *0*..*q*. *b*^*t*)

lemma *e-geom-series*:
assumes *b* ≥ *2*
shows (*E q b* = *e*) ⟷ ((*b*−*1*) ∗ *e* = *b*^(*Suc q*) − *1*) (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

definition *D* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
(*D q c b*) = (∑ *t* = *0*..*q*. (*2*^*c* − *1*) ∗ *b*^*t*)

lemma *d-geom-series*:
assumes *b* = *2*^(*Suc c*)
shows (*D q c b* = *d*) ⟷ ((*b*−*1*) ∗ *d* = (*2*^*c* − *1*) ∗ (*b*^(*Suc q*) − *1*)) (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

definition *F* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
(*F q c b*) = (∑ *t* = *0*..*q*. *2*^*c* ∗ *b*^*t*)

lemma *f-geom-series*:
assumes *b* = *2*^(*Suc c*)
shows (*F q c b* = *f*) ⟷ ((*b*−*1*) ∗ *f* = *2*^*c* ∗ (*b*^(*Suc q*) − *1*))
⟨*proof*⟩

lemma *aux-lt-implies-mask*:
assumes $a < 2\hat{\ }k$
shows $\forall r \geq k.\ a$ ¡ $r = 0$
⟨*proof*⟩

lemma *lt-implies-mask*:
fixes $a\ b :: nat$
assumes $\exists k.\ a < 2\hat{\ }k \wedge (\forall r<k.\ nth\text{-}bit\ b\ r = 1)$
shows $a \preceq b$
⟨*proof*⟩

lemma *mask-conversed-shift*:
fixes $a\ b\ k :: nat$
assumes *asm*: $a \preceq b$
shows $a * 2\hat{\ }k \preceq b * 2\hat{\ }k$
⟨*proof*⟩

lemma *base-summation-bound*:
fixes $c\ q :: nat$
and $f :: (nat \Rightarrow nat)$

defines *b*: $b \equiv B\ c$
assumes *bound*: $\forall t.\ f\ t < 2\ \hat{\ }\ Suc\ c - (1::nat)$

shows $(\sum t = 0..q.\ f\ t * b\hat{\ }t) < b\hat{\ }(Suc\ q)$
⟨*proof*⟩

lemma *mask-conserved-sum*:
fixes $y\ c\ q :: nat$
and $x :: (nat \Rightarrow nat)$

defines *b*: $b \equiv B\ c$
assumes *mask*: $\forall t.\ x\ t \preceq y$
assumes *xlt*: $\forall t.\ x\ t \leq 2\ \hat{\ }\ c - Suc\ 0$
assumes *ylt*: $y \leq 2\ \hat{\ }\ c - Suc\ 0$

shows $(\sum t = 0..q.\ x\ t * b\hat{\ }t) \preceq (\sum t = 0..q.\ y * b\hat{\ }t)$
⟨*proof*⟩

lemma *aux-powertwo-digits*:
fixes $k\ c :: nat$
assumes $k < c$
shows $nth\text{-}bit\ (2\hat{\ }c)\ k = 0$
⟨*proof*⟩

lemma *obtain-digit-rep*:
fixes $x\ c :: nat$

shows x && $2\hat{\ }c = (\sum t{<}Suc\ c.\ 2\hat{\ }t * (nth\text{-}bit\ x\ t) * (nth\text{-}bit\ (2\hat{\ }c)\ t))$
⟨*proof*⟩

lemma *nth-digit-bitAND-equiv*:
fixes $x\ c :: nat$
shows $2\hat{\ }c * nth\text{-}bit\ x\ c = (x$ && $2\hat{\ }c)$
⟨*proof*⟩

lemma *bitAND-single-digit*:
fixes $x\ c :: nat$
assumes $2 \hat{\ } c \leq x$
assumes $x < 2 \hat{\ } Suc\ c$

shows $nth\text{-}bit\ x\ c = 1$
⟨*proof*⟩

lemma *aux-bitAND-distrib*: $2 * (a$ && $b) = (2 * a)$ && $(2 * b)$
⟨*proof*⟩

lemma *bitAND-distrib*: $2\hat{\ }c * (a$ && $b) = (2\hat{\ }c * a)$ && $(2\hat{\ }c * b)$
⟨*proof*⟩

lemma *bitAND-linear-sum*:
fixes $x\ y :: nat \Rightarrow nat$
and $c :: nat$
and $q :: nat$

defines b: $b == 2 \hat{\ } Suc\ c$

assumes xb: $\forall t.\ x\ t < 2 \hat{\ } Suc\ c - 1$
assumes yb: $\forall t.\ y\ t < 2 \hat{\ } Suc\ c - 1$

shows $(\sum t = 0..q.\ (x\ t$ && $y\ t) * b\hat{\ }t) =$
$(\sum t = 0..q.\ x\ t * b\hat{\ }t)$ && $(\sum t = 0..q.\ y\ t * b\hat{\ }t)$
⟨*proof*⟩

lemma *dmask-aux0*:
fixes $x :: nat$
assumes $x > 0$
shows $(2 \hat{\ } x - Suc\ 0)\ div\ 2 = 2 \hat{\ } (x - 1) - Suc\ 0$
⟨*proof*⟩

lemma *dmask-aux*:
fixes $c :: nat$
shows $d < c \Longrightarrow (2\hat{\ }c - Suc\ 0)\ div\ 2\hat{\ }d = 2 \hat{\ } (c - d) - Suc\ 0$
⟨*proof*⟩

lemma *register-cells-masked*:
  fixes *l* :: *register*
    and *t* :: *nat*
    and *ic* :: *configuration*
    and *p* :: *program*

assumes *cells-bounded*: *cells-bounded ic p c*
assumes *l*: *l* < *length* (*snd ic*)

shows *R ic p l t* ⪯ *2^c* − *1*
⟨*proof*⟩

lemma *lm04-15-register-masking*:
  fixes *c* :: *nat*
    and *l* :: *register*
    and *ic* :: *configuration*
    and *p* :: *program*
    and *q* :: *nat*

defines *b* == *B c*
defines *d* == *D q c b*

assumes *cells-bounded*: *cells-bounded ic p c*
assumes *l*: *l* < *length* (*snd ic*)

defines *r* == *RLe ic p b q*

shows *r l* ⪯ *d*
⟨*proof*⟩

lemma *zero-cells-masked*:
  fixes *l* :: *register*
    and *t* :: *nat*
    and *ic* :: *configuration*
    and *p* :: *program*

assumes *l*: *l* < *length* (*snd ic*)

shows *Z ic p l t* ⪯ *1*
⟨*proof*⟩

lemma *lm04-15-zero-masking*:
  fixes *c* :: *nat*
    and *l* :: *register*
    and *ic* :: *configuration*
    and *p* :: *program*
    and *q* :: *nat*

defines *b* == *B c*
defines *e* == *E q b*

assumes *cells-bounded*: *cells-bounded ic p c*
assumes *l*: *l* < *length* (*snd ic*)
assumes *c*: *c* > *0*

defines *z* == *ZLe ic p b q*

shows *z l* ⪯ *e*
⟨*proof*⟩

lemma *lm04-19-zero-register-relations*:
  fixes *c* :: *nat*
    and *l* :: *register*
    and *t* :: *nat*
    and *ic* :: *configuration*
    and *p* :: *program*

assumes *cells-bounded*: *cells-bounded ic p c*
assumes *l*: *l* < *length* (*snd ic*)

defines *z* == *Z ic p*
defines *r* == *R ic p*

shows *2*^*c* ∗ *z l t* = (*r l t* + *2*^*c* − *1*) && *2*^*c*
⟨*proof*⟩

lemma *lm04-20-zero-definition*:
  fixes *c* :: *nat*
    and *l* :: *register*
    and *ic* :: *configuration*
    and *p* :: *program*
    and *q* :: *nat*

defines *b* == *B c*
defines *f* == *F q c b*
defines *d* == *D q c b*

assumes *cells-bounded*: *cells-bounded ic p c*
assumes *l*: *l* < *length* (*snd ic*)

assumes *c*: *c* > *0*

defines *z* == *ZLe ic p b q*
defines *r* == *RLe ic p b q*

shows *2*^*c* ∗ *z l* = (*r l* + *d*) && *f*

⟨*proof*⟩

lemma *state-mask*:
fixes *c* :: *nat*
and *l* :: *register*
and *ic* :: *configuration*
and *p* :: *program*
and *q* :: *nat*
and *a* :: *nat*

defines $b \equiv B\ c$
and $m \equiv$ *length p* $-$ *1*

defines $e \equiv E\ q\ b$

assumes *is-val*: *is-valid-initial ic p a*
and *q*: $q > 0$
and $c > 0$

assumes *terminate*: *terminates ic p q*
shows *SKe ic p b q k* $\preceq$ *e*
⟨*proof*⟩

lemma *state-sum-mask*:
fixes *c* :: *nat*
and *l* :: *register*
and *ic* :: *configuration*
and *p* :: *program*
and *q* :: *nat*
and *a* :: *nat*

defines $b \equiv B\ c$
and $m \equiv$ *length p* $-$ *1*

defines $e \equiv E\ q\ b$

assumes *is-val*: *is-valid-initial ic p a*
and *q*: $q > 0$
and $c > 0$
and $b > 1$

assumes $M \leq m$

assumes *terminate*: *terminates ic p q*
shows ($\sum k \leq M$. *SKe ic p b q k*) $\preceq$ *e*
⟨*proof*⟩

end

4 Arithmetization of Register Machines

4.1 A first definition of the arithmetizing equations

theory *MachineEquations*
  imports *MultipleStepRegister MultipleStepState MachineMasking*
begin

definition *mask-equations* :: *nat* ⇒ (*register* ⇒ *nat*) ⇒ (*register* ⇒ *nat*)
  ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*
  where (*mask-equations n r z c d e f*) = ((∀ *l*<*n*. (*r l*) ⪯ *d*)
    ∧ (∀ *l*<*n*. (*z l*) ⪯ *e*)
    ∧ (∀ *l*<*n*. *2*^*c* ∗ (*z l*) = (*r l* + *d*) && *f*))

definition *reg-equations* :: *program* ⇒ (*register* ⇒ *nat*) ⇒ (*register* ⇒ *nat*) ⇒ (*state* ⇒ *nat*)
  ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
  (*reg-equations p r z s b a n q*) = (
   — 4.22 (∀ *l*>*0*. *l* < *n* ⟶ *r l* = *b*∗*r l* + *b*∗∑ *R*+ *p l* (λ*k*. *s k*) − *b*∗∑ *R*− *p l* (λ*k*. *s k* && *z l*))
  ∧ — 4.23 (*r 0* = *a* + *b*∗*r 0* + *b*∗∑ *R*+ *p 0* (λ*k*. *s k*) − *b*∗∑ *R*− *p 0* (λ*k*. *s k* && *z 0*))
  ∧ (∀ *l*<*n*. *r l* < *b* ^ *q*)) — Extra equation not in Matiyasevich's book. Needed to show that all registers are empty at time q

definition *state-equations* :: *program* ⇒ (*state* ⇒ *nat*) ⇒ (*register* ⇒ *nat*) ⇒
  nat ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
  state-equations p s z b e q m = (
— 4.24 (∀ *d*>*0*. *d*≤*m* ⟶ *s d* = *b*∗∑ *S*+ *p d* (λ*k*. *s k*) + *b*∗∑ *S*− *p d* (λ*k*. *s k* && *z* (*modifies* (*p*!*k*)))
  + *b*∗∑ *S0 p d* (λ*k*. *s k* && (*e* − *z* (*modifies* (*p*!*k*)))))
  ∧ — 4.25 (*s 0* = *1* + *b*∗∑ *S*+ *p 0* (λ*k*. *s k*) + *b*∗∑ *S*− *p 0* (λ*k*. *s k* && *z* (*modifies* (*p*!*k*)))
  + *b*∗∑ *S0 p 0* (λ*k*. *s k* && (*e* − *z* (*modifies* (*p*!*k*)))))
  ∧ — 4.27 (*s m* = *b*^*q*)
  ∧ (∀ *k*≤*m*. *s k* ⪯ *e*) ∧ (∀ *k*<*m*. *s k* < *b* ^ *q*) — these equations are not from the book
  ∧ (∀ *M*≤*m*. (∑ *k*≤*M*. *s k*) ⪯ *e*) — this equation is added, too)

definition *state-unique-equations* :: *program* ⇒ (*state*⇒*nat*) ⇒ *nat* ⇒ *nat* ⇒*bool* **where**
  state-unique-equations p s m e = ((∑ *k*=*0*..*m*. *s k*) ⪯ *e* ∧ (∀ *k*≤*m*. *s k* ⪯ *e*))

definition *rm-constants* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where
rm-constants q c b d e f a = (
— 4.14 ($b = B\ c$)
∧ — 4.16 ($d = D\ q\ c\ b$)
∧ — 4.18 ($e = E\ q\ b$) — 4.19 left out (compare book)
∧ — 4.21 ($f = F\ q\ c\ b$)
∧ — extra equation not in the book $c > 0$
∧ — 4.26 ($a < 2\hat{}c$) ∧ ($q>0$))

definition *rm-equations-old* :: *program* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
rm-equations-old p q a n = (
∃ *b c d e f* :: *nat*.
∃ *r z* :: *register* ⇒ *nat*.
∃ *s* :: *state* ⇒ *nat*.
mask-equations n r z c d e f
∧ *reg-equations p r z s b a n q*
∧ *state-equations p s z b e q* (*length p* − *1*)
∧ *rm-constants q c b d e f a*)

end

4.2 Preliminary commutation relations

theory *CommutationRelations*
imports *RegisterMachineSimulation MachineEquations*
begin

lemma *aux-commute-bitAND-sum*:
fixes *N C* :: *nat*
and *fxt* :: *nat* ⇒ *nat*
shows $\forall i \le N.\ \forall j \le N.\ i \neq j \longrightarrow (\forall k.\ (fct\ i)$ ¡ $k * (fct\ j)$ ¡ $k = 0)$
$\Longrightarrow (\sum k \le N.\ fct\ k$ && $C) = (\sum k \le N.\ fct\ k)$ && C
⟨*proof*⟩

lemma *aux-commute-bitAND-sum-if*:
fixes *N const* :: *nat*
assumes *nocarry*: $\forall i \le N.\ \forall j \le N.\ i \neq j \longrightarrow (\forall k.\ (fct\ i)$ ¡ $k * (fct\ j)$ ¡ $k = 0)$
shows $(\sum k \le N.$ *if cond k then fct k* && *const else 0*)
= $(\sum k \le N.$ *if cond k then fct k else 0*) && *const*
⟨*proof*⟩

lemma *mod-mod*:
fixes *x a b* :: *nat*
shows *x mod 2^a mod 2^b = x mod 2^(min a b)*
⟨*proof*⟩

lemma *carry-gen-pow2-reduct*:

assumes *c>0*
defines *b*: $b \equiv 2$ ^ (*Suc c*)
assumes *nth-digit x* (*t−1*) (*2^Suc c*) ¡ *c* = *0*
and *nth-digit y* (*t−1*) (*2^Suc c*) ¡ *c* = *0*
shows *k≤c* ⟹ *bin-carry* (*nth-digit x t b*) (*nth-digit y t b*) *k*
= *bin-carry x y* (*Suc c* ∗ *t* + *k*)
⟨*proof*⟩

lemma *nth-digit-bound*:
fixes *c* **defines** $b \equiv 2$ ^ (*Suc c*)
shows *nth-digit x t b* < *2* ^ (*Suc c*)
⟨*proof*⟩

lemma *digit-wise-block-additivity*:
fixes *c*
defines $b \equiv 2$ ^ *Suc c*
assumes *nth-digit x* (*t−1*) (*2^Suc c*) ¡ *c* = *0*
and *nth-digit y* (*t−1*) (*2^Suc c*) ¡ *c* = *0*
and *k≤c*
and *c>0*
shows *nth-digit* (*x+y*) *t b* ¡ *k* = (*nth-digit x t b* + *nth-digit y t b*) ¡ *k*
⟨*proof*⟩

lemma *block-additivity*:
assumes *c* > *0*
defines $b \equiv 2$ ^ *Suc c*
assumes *nth-digit x* (*t−1*) *b* ¡ *c* = *0*
and *nth-digit y* (*t−1*) *b* ¡ *c* = *0*
and *nth-digit x t b* ¡ *c* = *0*
and *nth-digit y t b* ¡ *c* = *0*

shows *nth-digit* (*x+y*) *t b* = *nth-digit x t b* + *nth-digit y t b*
⟨*proof*⟩

lemma *block-to-sum*:
assumes *c>0*
defines *b*: $b \equiv 2$ ^ (*Suc c*)
assumes *yltx-digits*: ∀ *t'*. *nth-digit y t' b* ≤ *nth-digit x t' b*
shows *y mod b^t* ≤ *x mod b^t*
⟨*proof*⟩

lemma *narry-gen-pow2-reduct*:
assumes *c>0*
defines *b*: $b \equiv 2$ ^ (*Suc c*)
assumes *yltx-digits*: ∀ *t'*. *nth-digit y t' b* ≤ *nth-digit x t' b*
shows *k≤c* ⟹ *bin-narry* (*nth-digit x t b*) (*nth-digit y t b*) *k*
= *bin-narry x y* (*Suc c* ∗ *t* + *k*)
⟨*proof*⟩

lemma *digit-wise-block-subtractivity*:
fixes *c*
defines $b \equiv 2 \ \hat{}\ Suc\ c$
assumes *yltx-digits*: $\forall t'.\ nth\text{-}digit\ y\ t'\ b \leq nth\text{-}digit\ x\ t'\ b$
and $k \leq c$
and $c > 0$
shows $nth\text{-}digit\ (x - y)\ t\ b$ ¡ $k = (nth\text{-}digit\ x\ t\ b - nth\text{-}digit\ y\ t\ b)$ ¡ k
⟨*proof*⟩

lemma *block-subtractivity*:
assumes $c > 0$
defines $b \equiv 2 \ \hat{}\ Suc\ c$
assumes *block-wise-lt*: $\forall t'.\ nth\text{-}digit\ y\ t'\ b \leq nth\text{-}digit\ x\ t'\ b$
shows $nth\text{-}digit\ (x - y)\ t\ b = nth\text{-}digit\ x\ t\ b - nth\text{-}digit\ y\ t\ b$
⟨*proof*⟩

lemma *bitAND-nth-digit-commute*:
assumes *b-def*: $b = 2\hat{}(Suc\ c)$
shows $nth\text{-}digit\ (x\ \&\&\ y)\ t\ b = nth\text{-}digit\ x\ t\ b\ \&\&\ nth\text{-}digit\ y\ t\ b$
⟨*proof*⟩

lemma *bx-aux*:
shows $b > 1 \Longrightarrow nth\text{-}digit\ (b\hat{}x)\ t'\ b = (if\ x = t'\ then\ 1\ else\ 0)$
⟨*proof*⟩

context
fixes $c\ b :: nat$
assumes *b-def*: $b \equiv 2\hat{}(Suc\ c)$
assumes *c-gt0*: $c > 0$
begin

lemma *b-gt1*: $b > 1$ ⟨*proof*⟩

Commutation relations with sums

lemma *finite-sum-nth-digit-commute*:
fixes $M :: nat$
shows $\forall t.\ \forall k \leq M.\ nth\text{-}digit\ (fct\ k)\ t\ b < 2\hat{}c \Longrightarrow$
$\forall t.\ (\sum i = 0..M.\ nth\text{-}digit\ (fct\ i)\ t\ b) < 2\hat{}c \Longrightarrow$
$nth\text{-}digit\ (\sum i = 0..M.\ fct\ i)\ t\ b = (\sum i = 0..M.\ (nth\text{-}digit\ (fct\ i)\ t\ b))$
⟨*proof*⟩

lemma *sum-nth-digit-commute-aux*:
fixes *g*
defines *SX-def*: $SX \equiv \lambda l\ m\ (fct :: nat \Rightarrow nat).\ (\sum k = 0..m.\ if\ g\ l\ k\ then\ fct\ k$ $else\ 0)$
assumes *nocarry*: $\forall t.\ \forall k \leq M.\ nth\text{-}digit\ (fct\ k)\ t\ b < 2\hat{}c$
and *nocarry-sum*: $\forall t.\ (SX\ l\ M\ (\lambda k.\ nth\text{-}digit\ (fct\ k)\ t\ b)) < 2\hat{}c$
shows $nth\text{-}digit\ (SX\ l\ M\ fct)\ t\ b = SX\ l\ M\ (\lambda k.\ nth\text{-}digit\ (fct\ k)\ t\ b)$

⟨*proof*⟩

lemma *sum-nth-digit-commute*:
  fixes *g*
  defines *SX-def*: $SX \equiv \lambda p\ l\ (fct :: nat \Rightarrow nat).\ (\sum k = 0..length\ p - 1.\ if\ g\ l\ k$ $then\ fct\ k\ else\ 0)$
    assumes *nocarry*: $\forall t.\ \forall k \le length\ p - 1.\ nth\text{-}digit\ (fct\ k)\ t\ b < 2\hat{}c$
    and *nocarry-sum*: $\forall t.\ (SX\ p\ l\ (\lambda k.\ nth\text{-}digit\ (fct\ k)\ t\ b)) < 2\hat{}c$
  shows $nth\text{-}digit\ (SX\ p\ l\ fct)\ t\ b = SX\ p\ l\ (\lambda k.\ nth\text{-}digit\ (fct\ k)\ t\ b)$
⟨*proof*⟩

Commute inside, need assumption for all partial sums

lemma *finite-sum-nth-digit-commute2*:
  fixes $M :: nat$
  shows $\forall t.\ \forall k \le M.\ nth\text{-}digit\ (fct\ k)\ t\ b < 2\hat{}c \Longrightarrow$
    $\forall t.\ \forall m \le M.\ nth\text{-}digit\ (\sum i=0..m.\ fct\ i)\ t\ b < 2\hat{}c \Longrightarrow$
    $nth\text{-}digit\ (\sum i=0..M.\ fct\ i)\ t\ b = (\sum i=0..M.\ (nth\text{-}digit\ (fct\ i)\ t\ b))$
⟨*proof*⟩

lemma *sum-nth-digit-commute-aux2*:
  fixes *g*
  defines *SX-def*: $SX \equiv \lambda l\ m\ (fct :: nat \Rightarrow nat).\ (\sum k = 0..m.\ if\ g\ l\ k\ then\ fct\ k$ $else\ 0)$
    assumes *nocarry*: $\forall t.\ \forall k \le M.\ nth\text{-}digit\ (fct\ k)\ t\ b < 2\hat{}c$
    and *nocarry-sum*: $\forall t.\ \forall m \le M.\ nth\text{-}digit\ (SX\ l\ m\ fct)\ t\ b < 2\hat{}c$
  shows $nth\text{-}digit\ (SX\ l\ M\ fct)\ t\ b = SX\ l\ M\ (\lambda k.\ nth\text{-}digit\ (fct\ k)\ t\ b)$
⟨*proof*⟩

lemma *sum-nth-digit-commute2*:
  fixes *g p*
  defines *SX-def*: $SX \equiv \lambda p\ l\ (fct :: nat \Rightarrow nat).\ (\sum k = 0..length\ p - 1.\ if\ g\ l\ k$ $then\ fct\ k\ else\ 0)$
    assumes *nocarry*: $\forall t.\ \forall k \le length\ p - 1.\ nth\text{-}digit\ (fct\ k)\ t\ b < 2\hat{}c$
    and *nocarry-sum*: $\forall t.\ \forall m \le length\ p - 1.\ nth\text{-}digit\ (SX\ (take\ (Suc\ m)\ p)\ l\ fct)$ $t\ b < 2\hat{}c$
  shows $nth\text{-}digit\ (SX\ p\ l\ fct)\ t\ b = SX\ p\ l\ (\lambda k.\ nth\text{-}digit\ (fct\ k)\ t\ b)$
⟨*proof*⟩

end

end

4.3 From multiple to single step relations

theory *MultipleToSingleSteps*
  imports *MachineEquations CommutationRelations ../Diophantine/Binary-And*
begin

This file contains lemmas that are needed to prove the <– direction of conclusion4.5 in the file MachineEquationEquivalence. In particular, it is shown that single step equations follow from the multiple step relations. The key idea of Matiyasevich's proof is to code all register and state values over the time into one large number. A further central statement in this file shows that the decoding of these numbers back to the single cell contents is indeed correct.

context
  fixes *a* :: *nat*
    and *ic*:: *configuration*
    and *p* :: *program*
    and *q* :: *nat*
    and *r z* :: *register* ⇒ *nat*
    and *s* :: *state* ⇒ *nat*
    and *b c d e f* :: *nat*
    and *m n* :: *nat*
    and *Req Seq Zeq*

 assumes *m-def*: *m* ≡ *length p* − *1*
    and *n-def*: *n* ≡ *length* (*snd ic*)

 assumes *is-val*: *is-valid-initial ic p a*

 assumes *m-eq*: *mask-equations n r z c d e f*
    and *r-eq*: *reg-equations p r z s b a n q*
    and *s-eq*: *state-equations p s z b e q m*
    and *c-eq*: *rm-constants q c b d e f a*

 assumes *Seq-def*: *Seq* = (λ*k t*. *nth-digit* (*s k*) *t b*)
    and *Req-def*: *Req* = (λ*l t*. *nth-digit* (*r l*) *t b*)
    and *Zeq-def*: *Zeq* = (λ*l t*. *nth-digit* (*z l*) *t b*)

begin

Basic properties

lemma *n-gt0*: *n*>*0*
 ⟨*proof*⟩

lemma *f-def*: *f* = (∑ *t* = *0*..*q*. *2*^*c* ∗ *b*^*t*)
 ⟨*proof*⟩
lemma *e-def*: *e* = (∑ *t* = *0*..*q*. *b*^*t*)
 ⟨*proof*⟩
lemma *d-def*: *d* = (∑ *t* = *0*..*q*. (*2*^*c* − *1*) ∗ *b*^*t*)
 ⟨*proof*⟩
lemma *b-def*: *b* = *2*^(*Suc c*)
 ⟨*proof*⟩

lemma *b-gt1*: *b* > *1* ⟨*proof*⟩

lemma *c-gt0*: $c > 0$ ⟨*proof*⟩
lemma *h0*: $1 < (2::nat)\hat{}c$
⟨*proof*⟩

lemma *rl-fst-digit-zero*:
assumes $l < n$
shows *nth-digit* $(r\ l)\ t\ b$ ¡ $c = 0$
⟨*proof*⟩

lemma *e-mask-bound*:
assumes $x \preceq e$
shows *nth-digit* $x\ t\ b \leq 1$
⟨*proof*⟩

lemma *sk-bound*:
shows $\forall\, t\ k.\ k \leq length\ p - 1 \longrightarrow nth\text{-}digit\ (s\ k)\ t\ b \leq 1$
⟨*proof*⟩

lemma *sk-bitAND-bound*:
shows $\forall\, t\ k.\ k \leq length\ p - 1 \longrightarrow nth\text{-}digit\ (s\ k\ \&\&\ x\ k)\ t\ b \leq 1$
⟨*proof*⟩

lemma *s-bound*:
shows $\forall\, j<m.\ s\ j < b\ \hat{}\ q$
⟨*proof*⟩

lemma *sk-sum-masked*:
shows $\forall\, M \leq m.\ (\sum k \leq M.\ s\ k) \preceq e$
⟨*proof*⟩

lemma *sk-sum-bound*:
shows $\forall\, t\ M.\ M \leq length\ p - 1 \longrightarrow nth\text{-}digit\ (\sum k \leq M.\ s\ k)\ t\ b \leq 1$
⟨*proof*⟩

lemma *sum-sk-bound*:
shows $(\sum k \leq length\ p - 1.\ nth\text{-}digit\ (s\ k)\ t\ b) \leq 1$
⟨*proof*⟩

lemma *bitAND-sum-lt*: $(\sum k \leq length\ p - 1.\ nth\text{-}digit\ (s\ k\ \&\&\ x\ k)\ t\ b)$
$\leq (\sum k \leq length\ p - 1.\ Seq\ k\ t)$
⟨*proof*⟩

lemma *states-unique-RAW*:
$\forall\, k \leq m.\ Seq\ k\ t = 1 \longrightarrow (\forall\, j \leq m.\ j \neq k \longrightarrow Seq\ j\ t = 0)$
⟨*proof*⟩

lemma *block-sum-radd-bound*:
shows $\forall t.\ (\sum R+\ p\ l\ (\lambda k.\ nth\text{-}digit\ (s\ k)\ t\ b)) \leq 1$
⟨*proof*⟩

lemma *block-sum-rsub-bound*:
shows $\forall t.\ (\sum R-\ p\ l\ (\lambda k.\ nth\text{-}digit\ (s\ k\ \&\&\ z\ l)\ t\ b)) \leq 1$
⟨*proof*⟩

lemma *block-sum-rsub-special-bound*:
shows $\forall t.\ (\sum R-\ p\ l\ (\lambda k.\ nth\text{-}digit\ (s\ k)\ t\ b)) \leq 1$
⟨*proof*⟩

lemma *block-sum-sadd-bound*:
shows $\forall t.\ (\sum S+\ p\ j\ (\lambda k.\ nth\text{-}digit\ (s\ k)\ t\ b)) \leq 1$
⟨*proof*⟩

lemma *block-sum-ssub-bound*:
shows $\forall t.\ (\sum S-\ p\ j\ (\lambda k.\ nth\text{-}digit\ (s\ k\ \&\&\ z\ (l\ k))\ t\ b)) \leq 1$
⟨*proof*⟩

lemma *block-sum-szero-bound*:
shows $\forall t.\ (\sum S0\ p\ j\ (\lambda k.\ nth\text{-}digit\ (s\ k\ \&\&\ (e - z\ (l\ k)))\ t\ b)) \leq 1$
⟨*proof*⟩

lemma *sum-radd-nth-digit-commute*:
shows $nth\text{-}digit\ (\sum R+\ p\ l\ (\lambda k.\ s\ k))\ t\ b = \sum R+\ p\ l\ (\lambda k.\ nth\text{-}digit\ (s\ k)\ t\ b)$
⟨*proof*⟩

lemma *sum-rsub-nth-digit-commute*:
shows $nth\text{-}digit\ (\sum R-\ p\ l\ (\lambda k.\ s\ k\ \&\&\ z\ l))\ t\ b$
$= \sum R-\ p\ l\ (\lambda k.\ nth\text{-}digit\ (s\ k\ \&\&\ z\ l)\ t\ b)$
⟨*proof*⟩

lemma *sum-sadd-nth-digit-commute*:
shows $nth\text{-}digit\ (\sum S+\ p\ j\ (\lambda k.\ s\ k))\ t\ b = \sum S+\ p\ j\ (\lambda k.\ nth\text{-}digit\ (s\ k)\ t\ b)$
⟨*proof*⟩

lemma *sum-ssub-nth-digit-commute*:
shows $nth\text{-}digit\ (\sum S-\ p\ j\ (\lambda k.\ s\ k\ \&\&\ z\ (l\ k)))\ t\ b$
$= \sum S-\ p\ j\ (\lambda k.\ nth\text{-}digit\ (s\ k\ \&\&\ z\ (l\ k))\ t\ b)$
⟨*proof*⟩

lemma *sum-szero-nth-digit-commute*:
shows $nth\text{-}digit\ (\sum S0\ p\ j\ (\lambda k.\ s\ k\ \&\&\ (e - z\ (l\ k))))\ t\ b$
$= \sum S0\ p\ j\ (\lambda k.\ nth\text{-}digit\ (s\ k\ \&\&\ (e - z\ (l\ k)))\ t\ b)$
⟨*proof*⟩

lemma *block-bound-impl-fst-digit-zero*:
assumes $nth\text{-}digit\ x\ t\ b \leq 1$

shows (*nth-digit x t b*) ¡ *c* = *0*
⟨*proof*⟩

lemma *sum-radd-block-bound*:
shows *nth-digit* (∑ *R+ p l* (λ*k. s k*)) *t b* ≤ *1*
⟨*proof*⟩
lemma *sum-radd-fst-digit-zero*:
shows (*nth-digit* (∑ *R+ p l s*) *t b*) ¡ *c* = *0*
⟨*proof*⟩

lemma *sum-sadd-block-bound*:
shows *nth-digit* (∑ *S+ p j* (λ*k. s k*)) *t b* ≤ *1*
⟨*proof*⟩
lemma *sum-sadd-fst-digit-zero*:
shows (*nth-digit* (∑ *S+ p j s*) *t b*) ¡ *c* = *0*
⟨*proof*⟩

lemma *sum-ssub-block-bound*:
shows *nth-digit* (∑ *S− p j* (λ*k. s k* && *z* (*l k*))) *t b* ≤ *1*
⟨*proof*⟩
lemma *sum-ssub-fst-digit-zero*:
shows (*nth-digit* (∑ *S− p j* (λ*k. s k* && *z* (*l k*))) *t b*) ¡ *c* = *0*
⟨*proof*⟩

lemma *sum-szero-block-bound*:
shows *nth-digit* (∑ *S0 p j* (λ*k. s k* && (*e* − *z* (*l k*)))) *t b* ≤ *1*
⟨*proof*⟩
lemma *sum-szero-fst-digit-zero*:
shows (*nth-digit* (∑ *S0 p j* (λ*k. s k* && (*e* − *z* (*l k*)))) *t b*) ¡ *c* = *0*
⟨*proof*⟩

lemma *sum-rsub-special-block-bound*:
shows *nth-digit* (∑ *R− p l* (λ*k. s k*)) *t b* ≤ *1*
⟨*proof*⟩

lemma *sum-state-special-block-bound*:
shows *nth-digit* (∑ *S+ p j* (λ*k. s k*)
+ ∑ *S0 p j* (λ*k. s k* && (*e* − *z* (*l k*)))) *t b* ≤ *1*
⟨*proof*⟩
lemma *sum-state-special-fst-digit-zero*:
shows (*nth-digit* (∑ *S+ p j* (λ*k. s k*)
+ ∑ *S0 p j* (λ*k. s k* && (*e* − *z* (*modifies* (*p*!*k*))))) *t b*) ¡ *c*
= *0*
⟨*proof*⟩

Main three redution lemmas: Zero Indicators, Registers, States

lemma *Z*:
assumes *l*<*n*
shows *Zeq l t* = (*if Req l t* > *0 then Suc 0 else 0*)

⟨*proof*⟩

lemma *zl-le-rl*: $l<n \Longrightarrow z\ l \leq r\ l$ **for** l
⟨*proof*⟩

lemma *modifies-valid*: $\forall k\leq m.\ modifies\ (p!k) < n$
⟨*proof*⟩

lemma *seq-bound*: $k \leq length\ p - 1 \Longrightarrow Seq\ k\ t \leq 1$
⟨*proof*⟩

lemma *skzl-bitAND-to-mult*:
assumes $k \leq length\ p - 1$
assumes $l < n$
shows $nth\text{-}digit\ (z\ l)\ t\ b\ \&\&\ nth\text{-}digit\ (s\ k)\ t\ b = (Zeq\ l\ t) * Seq\ k\ t$
⟨*proof*⟩

lemma *skzl-bitAND-to-mult2*:
assumes $k \leq length\ p - 1$
assumes $\forall k \leq length\ p - 1.\ l\ k < n$
shows $(1 - nth\text{-}digit\ (z\ (l\ k))\ t\ b)\ \&\&\ nth\text{-}digit\ (s\ k)\ t\ b$
$= (1 - Zeq\ (l\ k)\ t) * Seq\ k\ t$
⟨*proof*⟩

lemma *state-equations-digit-commute*:
assumes $t < q$ **and** $j \leq m$
defines $l \equiv \lambda k.\ modifies\ (p!k)$
shows $nth\text{-}digit\ (s\ j)\ (Suc\ t)\ b =$
$(\sum S{+}\ p\ j\ (\lambda k.\ Seq\ k\ t))$
$+ (\sum S{-}\ p\ j\ (\lambda k.\ Zeq\ (l\ k)\ t * Seq\ k\ t))$
$+ (\sum S0\ p\ j\ (\lambda k.\ (1 - Zeq\ (l\ k)\ t) * Seq\ k\ t))$
⟨*proof*⟩

lemma *aux-nocarry-sk*:
assumes $t\leq q$
shows $i \neq j \longrightarrow i\leq m \longrightarrow j\leq m \longrightarrow nth\text{-}digit\ (s\ i)\ t\ b * nth\text{-}digit\ (s\ j)\ t\ b = 0$
⟨*proof*⟩

lemma *nocarry-sk*:
assumes $i \neq j$ **and** $i\leq m$ **and** $j\leq m$
shows $(s\ i)$ ¡ $k * (s\ j)$ ¡ $k = 0$
⟨*proof*⟩

lemma *commute-sum-rsub-bitAND*: $\sum R{-}\ p\ l\ (\lambda k.\ s\ k\ \&\&\ z\ l) = \sum R{-}\ p\ l\ (\lambda k.\ s\ k)\ \&\&\ z\ l$
⟨*proof*⟩

lemma *sum-rsub-bound*: $l<n \Longrightarrow \sum R-\ p\ l\ (\lambda k.\ s\ k\ \&\&\ z\ l) \leq r\ l + \sum R+\ p\ l\ s$
⟨*proof*⟩

Obtaining single step register relations from multiple step register relations

lemma *mult-to-single-reg*:
$c>0 \Longrightarrow l<n \Longrightarrow Req\ l\ (Suc\ t) = Req\ l\ t + (\sum R+\ p\ l\ (\lambda k.\ Seq\ k\ t))$
$- (\sum R-\ p\ l\ (\lambda k.\ (Zeq\ l\ t) * Seq\ k\ t))$ **for** $l\ t$
⟨*proof*⟩

Obtaining single step state relations from multiple step state relations

lemma *mult-to-single-state*:
fixes $t\ j :: nat$
defines $l \equiv \lambda k.\ modifies\ (p!k)$
shows $j\leq m \Longrightarrow t<q \Longrightarrow Seq\ j\ (Suc\ t) = (\sum S+\ p\ j\ (\lambda k.\ Seq\ k\ t))$
$+ (\sum S-\ p\ j\ (\lambda k.\ Zeq\ (l\ k)\ t * Seq\ k\ t))$
$+ (\sum S0\ p\ j\ (\lambda k.\ (1 - Zeq\ (l\ k)\ t) * Seq\ k\ t))$
⟨*proof*⟩

Conclusion: The central equivalence showing that the cell entries obtained from r s z indeed coincide with the correct cell values when executing the register machine. This statement is proven by induction using the single step relations for Req and Seq as well as the statement for Zeq.

lemma *rzs-eq*:
$l<n \Longrightarrow j\leq m \Longrightarrow t\leq q \Longrightarrow R\ ic\ p\ l\ t = Req\ l\ t \land Z\ ic\ p\ l\ t = Zeq\ l\ t \land S\ ic\ p\ j\ t = Seq\ j\ t$
⟨*proof*⟩

end

end

4.4 Arithmetizing equations are Diophantine

theory *Equation-Setup* **imports** *../Register-Machine/RegisterMachineSpecification*
../Diophantine/Diophantine-Relations

begin

locale *register-machine* =
fixes $p :: program$
and $n :: nat$
assumes *p-nonempty*: $length\ p > 0$
and *valid-program*: $program\text{-}register\text{-}check\ p\ n$
assumes *n-gt-0*: $n > 0$

begin

definition $m :: nat$ **where**

m ≡ *length p* − *1*

lemma *modifies-yields-valid-register*:
assumes *k* < *length p*
shows *modifies* (*p*!*k*) < *n*
⟨*proof*⟩

end

locale *rm-eq-fixes* = *register-machine* +
fixes *a b c d e f* :: *nat*
and *q* :: *nat*
and *r z* :: *register* ⇒ *nat*
and *s* :: *state* ⇒ *nat*

end

4.4.1 Preliminary: Register machine sums are Diophantine

theory *Register-Machine-Sums* **imports** *Diophantine-Relations*
../Register-Machine/RegisterMachineSimulation

begin

fun *sum-polynomial* :: (*nat* ⇒ *polynomial*) ⇒ *nat list* ⇒ *polynomial* **where**
sum-polynomial f [] = *Const 0* |
sum-polynomial f (*i*#*idxs*) = *f i* [+] *sum-polynomial f idxs*

lemma *sum-polynomial-eval*:
peval (*sum-polynomial f idxs*) *a* = (∑ *k*=*0*..<*length idxs*. *peval* (*f* (*idxs*!*k*)) *a*)
⟨*proof*⟩

definition *sum-program* :: *program* ⇒ (*nat* ⇒ *polynomial*) ⇒ *polynomial*
(‹[∑ -] -› [*100*, *100*] *100*) **where**
[∑ *p*] *f* ≡ *sum-polynomial f* [*0*..<*length p*]

lemma *sum-program-push*: *m* = *length ns* ⟹ *length l* = *length p* ⟹
peval ([∑ *p*] (λ*k*. *if g k then map* (λ*x*. *push-param x m*) *l* ! *k else h k*)) (*push-list a ns*)
= *peval* ([∑ *p*] (λ*k*. *if g k then l* ! *k else h k*)) *a*
⟨*proof*⟩

definition *sum-radd-polynomial* :: *program* ⇒ *register* ⇒ (*nat* ⇒ *polynomial*) ⇒ *polynomial*
(‹[∑ *R*+] - - -›) **where**
[∑ *R*+] *p l f* ≡ [∑ *p*] (λ*k*. *if isadd* (*p*!*k*) ∧ *l* = *modifies* (*p*!*k*) *then f k else Const 0*)

lemma *sum-radd-polynomial-eval*[*defs*]:
  assumes *length p* > *0*
  shows *peval* ([$\sum$ *R*+] *p l f*) *a* = ($\sum$ *R*+ *p l* (λ*x*. *peval* (*f x*) *a*))
⟨*proof*⟩

definition *sum-rsub-polynomial* :: *program* ⇒ *register* ⇒ (*nat* ⇒ *polynomial*) ⇒ *polynomial*
  (‹[$\sum$ *R*−] - - -›) **where**
  [$\sum$ *R*−] *p l f* ≡ [$\sum$ *p*] (λ*k*. *if issub* (*p*!*k*) ∧ *l* = *modifies* (*p*!*k*) *then f k else Const 0*)

lemma *sum-rsub-polynomial-eval*[*defs*]:
  assumes *length p* > *0*
  shows *peval* ([$\sum$ *R*−] *p l f*) *a* = ($\sum$ *R*− *p l* (λ*x*. *peval* (*f x*) *a*))
⟨*proof*⟩

definition *sum-sadd-polynomial* :: *program* ⇒ *state* ⇒ (*nat* ⇒ *polynomial*) ⇒ *polynomial*
  (‹[$\sum$ *S*+] - - -›) **where**
  [$\sum$ *S*+] *p d f* ≡ [$\sum$ *p*] (λ*k*. *if isadd* (*p*!*k*) ∧ *d* = *goes-to* (*p*!*k*) *then f k else Const 0*)

lemma *sum-sadd-polynomial-eval*[*defs*]:
  assumes *length p* > *0*
  shows *peval* ([$\sum$ *S*+] *p d f*) *a* = ($\sum$ *S*+ *p d* (λ*x*. *peval* (*f x*) *a*))
⟨*proof*⟩

definition *sum-ssub-nzero-polynomial* :: *program* ⇒ *state* ⇒ (*nat* ⇒ *polynomial*) ⇒ *polynomial*
  (‹[$\sum$ *S*−] - - -›) **where**
  [$\sum$ *S*−] *p d f* ≡ [$\sum$ *p*] (λ*k*. *if issub* (*p*!*k*) ∧ *d* = *goes-to* (*p*!*k*) *then f k else Const 0*)

lemma *sum-ssub-nzero-polynomial-eval*[*defs*]:
  assumes *length p* > *0*
  shows *peval* ([$\sum$ *S*−] *p d f*) *a* = ($\sum$ *S*− *p d* (λ*x*. *peval* (*f x*) *a*))
⟨*proof*⟩

definition *sum-ssub-zero-polynomial* :: *program* ⇒ *state* ⇒ (*nat* ⇒ *polynomial*) ⇒ *polynomial*
  (‹[$\sum$ *S0*] - - -›) **where**
  [$\sum$ *S0*] *p d f* ≡ [$\sum$ *p*] (λ*k*. *if issub* (*p*!*k*) ∧ *d* = *goes-to-alt* (*p*!*k*) *then f k else Const 0*)

lemma *sum-ssub-zero-polynomial-eval*[*defs*]:
  assumes *length p* > *0*
  shows *peval* ([$\sum$ *S0*] *p d f*) *a* = ($\sum$ *S0 p d* (λ*x*. *peval* (*f x*) *a*))
⟨*proof*⟩

end
theory *RM-Sums-Diophantine* **imports** *Equation-Setup ../Diophantine/Register-Machine-Sums ../Diophantine/Binary-And*

begin

context *register-machine*
begin

definition *sum-ssub-nzero-of-bit-and* :: *polynomial* ⇒ *nat* ⇒ *polynomial list* ⇒ *polynomial list*
⇒ *relation*
(‹[- = ∑ S− - '(- && -')]›) **where**
[*x* = ∑ *S*− *d* (*s* && *z*)] ≡ *let x'* = *push-param x* (*length p*);
s' = *push-param-list s* (*length p*);
z' = *push-param-list z* (*length p*)
in [∃ *length p*] [∀ <*length p*] (λ*i*. [*Param i* = *s'*!*i* && *z'*!*i*])
[∧] *x'* [=] ([∑ *S*−] *p d Param*)

lemma *sum-ssub-nzero-of-bit-and-dioph*[*dioph*]:
fixes *s z* :: *polynomial list* **and** *d* :: *nat* **and** *x*
shows *is-dioph-rel* [*x* = ∑ *S*− *d* (*s* && *z*)]
⟨*proof*⟩

lemma *sum-rsub-nzero-of-bit-and-eval*:
fixes *z s* :: *polynomial list* **and** *d* :: *nat* **and** *x* :: *polynomial*
assumes *length s* = *Suc m length z* = *Suc m length p* > *0*
shows *eval* [*x* = ∑ *S*− *d* (*s* && *z*)] *a*
⟷ *peval x a* = ∑ *S*− *p d* (λ*k*. *peval* (*s*!*k*) *a* && *peval* (*z*!*k*) *a*) (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

definition *sum-ssub-zero-of-bit-and* :: *polynomial* ⇒ *nat* ⇒ *polynomial list* ⇒ *polynomial list*
⇒ *relation*
(‹[- = ∑ *S0* - '(- && -')]›) **where**
[*x* = ∑ *S0 d* (*s* && *z*)] ≡ *let x'* = *push-param x* (*length p*);
s' = *push-param-list s* (*length p*);
z' = *push-param-list z* (*length p*)
in [∃ *length p*] [∀ <*length p*] (λ*i*. [*Param i* = *s'*!*i* && *z'*!*i*])
[∧] *x'* [=] [∑ *S0*] *p d Param*

lemma *sum-ssub-zero-of-bit-and-dioph*[*dioph*]:
fixes *s z* :: *polynomial list* **and** *d* :: *nat* **and** *x*
shows *is-dioph-rel* [*x* = ∑ *S0 d* (*s* && *z*)]
⟨*proof*⟩

lemma *sum-rsub-zero-of-bit-and-eval*:

fixes *z s* :: *polynomial list* **and** *d* :: *nat* **and** *x* :: *polynomial*
assumes *length s* = *Suc m length z* = *Suc m length p* > *0*
shows *eval* [$x = \sum S0\ d\ (s\ \&\&\ z)$] *a*
$\longleftrightarrow$ *peval x a* = $\sum S0\ p\ d\ (\lambda k.\ peval\ (s!k)\ a\ \&\&\ peval\ (z!k)\ a)$ (**is** *?P* $\longleftrightarrow$ *?Q*)
⟨*proof*⟩

end

end

4.4.2 Register Equations

theory *Register-Equations* **imports** *../Register-Machine/MultipleStepRegister Equation-Setup ../Diophantine/Register-Machine-Sums ../Diophantine/Binary-And HOL−Library.Rewrite*

begin

context *rm-eq-fixes*
begin

Equation 4.22

definition *register-0* :: *bool* **where**
register-0 ≡ $r\ 0 = a + b*r\ 0 + b*\sum R+\ p\ 0\ s - b*\sum R-\ p\ 0\ (\lambda k.\ s\ k\ \&\&\ z\ 0)$

Equation 4.23

definition *register-l* :: *bool* **where**
register-l ≡ $\forall l>0.\ l < n \longrightarrow r\ l = b*r\ l + b*\sum R+\ p\ l\ s - b*\sum R-\ p\ l\ (\lambda k.\ s\ k\ \&\&\ z\ l)$

Extra equation not in Matiyasevich's book

definition *register-bound* :: *bool* **where**
register-bound ≡ $\forall l < n.\ r\ l < b \,\hat{}\, q$

definition *register-equations* :: *bool* **where**
register-equations ≡ *register-0* ∧ *register-l* ∧ *register-bound*

end

context *register-machine*
begin

definition *sum-rsub-of-bit-and* :: *polynomial* ⇒ *nat* ⇒ *polynomial list* ⇒ *polynomial* ⇒ *relation*
(‹[- = $\sum R-$ - ′(- && -′)]›) **where**
[$x = \sum R-\ d\ (s\ \&\&\ zl)$] ≡ *let* x' = *push-param x* (*length p*);

$s' = push\text{-}param\text{-}list\ s\ (length\ p);$
$zl' = push\text{-}param\ zl\ (length\ p)$
$in\ [\exists\ length\ p]\ [\forall <length\ p]\ (\lambda i.\ [Param\ i = s'!i\ \&\&\ zl'])$
$[\wedge]\ x'\ [=]\ [\sum R-]\ p\ d\ Param$

lemma *sum-rsub-of-bit-and-dioph*[*dioph*]:
fixes *s* :: *polynomial list* **and** *d* :: *nat* **and** *x zl* :: *polynomial*
shows *is-dioph-rel* $[x = \sum R-\ d\ (s\ \&\&\ zl)]$
⟨*proof*⟩

lemma *sum-rsub-of-bit-and-eval*:
fixes *z s* :: *polynomial list* **and** *d* :: *nat* **and** *x* :: *polynomial*
assumes $length\ s = Suc\ m\ length\ p > 0$
shows $eval\ [x = \sum R-\ d\ (s\ \&\&\ zl)]\ a$
$\longleftrightarrow peval\ x\ a = \sum R-\ p\ d\ (\lambda k.\ peval\ (s!k)\ a\ \&\&\ peval\ zl\ a)$ (**is** *?P* $\longleftrightarrow$ *?Q*)
⟨*proof*⟩

lemma *register-0-dioph*[*dioph*]:
fixes *A b* :: *polynomial*
fixes *r z s* :: *polynomial list*
assumes $length\ r = n\ length\ z = n\ length\ s = Suc\ m$
defines $DR \equiv LARY\ (\lambda ll.\ rm\text{-}eq\text{-}fixes.register\text{-}0\ p\ (ll!0!0)\ (ll!0!1)$
$(nth\ (ll!1))\ (nth\ (ll!2))\ (nth\ (ll!3)))\ [[A,\ b],\ r,\ z,\ s]$
shows *is-dioph-rel DR*
⟨*proof*⟩

lemma *register-l-dioph*[*dioph*]:
fixes *b* :: *polynomial*
fixes *r z s* :: *polynomial list*
assumes $length\ r = n\ length\ z = n\ length\ s = Suc\ m$
defines $DR \equiv LARY\ (\lambda ll.\ rm\text{-}eq\text{-}fixes.register\text{-}l\ p\ n\ (ll!0!0)$
$(nth\ (ll!1))\ (nth\ (ll!2))\ (nth\ (ll!3)))\ [[b],\ r,\ z,\ s]$
shows *is-dioph-rel DR*
⟨*proof*⟩

lemma *register-bound-dioph*:
fixes *b q* :: *polynomial*
fixes *r* :: *polynomial list*
assumes $length\ r = n$
defines $DR \equiv LARY\ (\lambda ll.\ rm\text{-}eq\text{-}fixes.register\text{-}bound\ n\ (ll!0!0)\ (ll!0!1)\ (nth\ (ll!1)))$
$[[b,\ q],\ r]$
shows *is-dioph-rel DR*
⟨*proof*⟩

definition *register-equations-relation* :: *polynomial* ⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial list* ⇒ *polynomial list* ⇒ *polynomial list* ⇒ *relation* (‹[*REG*] - - - - - -›) **where**
[*REG*] *a b q r z s* ≡ *LARY* (λ*ll*. *rm-eq-fixes.register-equations p n* (*ll*!*0*!*0*) (*ll*!*0*!*1*) (*ll*!*0*!*2*)
(*nth* (*ll*!*1*)) (*nth* (*ll*!*2*)) (*nth* (*ll*!*3*))) [[*a*, *b*, *q*], *r*, *z*, *s*]

lemma *reg-dioph*:
fixes *A b q r z s*
assumes *length r* = *n length z* = *n length s* = *Suc m*
defines *DR* ≡ [*REG*] *A b q r z s*
shows *is-dioph-rel DR*
⟨*proof*⟩

end

end

4.4.3 State 0 equation

theory *State-0-Equation* **imports** ../*Register-Machine*/*MultipleStepState*
RM-Sums-Diophantine ../*Diophantine*/*Binary-And*

begin

context *rm-eq-fixes*
begin

Equation 4.24

definition *state-0* :: *bool* **where**
state-0 ≡ *s 0* = *1* + *b*∗∑ *S*+ *p 0 s* + *b*∗∑ *S*− *p 0* (λ*k*. *s k* && *z* (*modifies* (*p*!*k*)))
+ *b*∗∑ *S0 p 0* (λ*k*. *s k* && (*e* − *z* (*modifies* (*p*!*k*))))

end

context *register-machine*
begin

no-notation *ppolynomial.Sum* (**infixl** ‹+› *65*)
no-notation *ppolynomial.NatDiff* (**infixl** ‹−› *65*)
no-notation *ppolynomial.Prod* (**infixl** ‹∗› *70*)

lemma *state-0-dioph*:
fixes *b e* :: *polynomial*
fixes *z s* :: *polynomial list*
assumes *length z* = *n length s* = *Suc m*

defines *DR* ≡ *LARY* (λ*ll*. *rm-eq-fixes.state-0 p* (*ll*!*0*!*0*) (*ll*!*0*!*1*)
(*nth* (*ll*!*1*)) (*nth* (*ll*!*2*))) [[*b*, *e*], *z*, *s*]
shows *is-dioph-rel DR*
⟨*proof*⟩

end

end

4.4.4 State d equation

theory *State-d-Equation* **imports** *State-0-Equation*

begin

context *rm-eq-fixes*
begin

Equation 4.25

definition *state-d* :: *bool* **where**
state-d ≡ ∀ *d*>*0*. *d*≤*m* ⟶ *s d* = *b*∗∑ *S*+ *p d s* + *b*∗∑ *S*− *p d* (λ*k*. *s k* && *z* (*modifies* (*p*!*k*)))
+ *b*∗∑ *S0 p d* (λ*k*. *s k* && (*e* − *z* (*modifies* (*p*!*k*))))

Combining the two

definition *state-relations-from-recursion* :: *bool* **where**
state-relations-from-recursion ≡ *state-0* ∧ *state-d*

end

context *register-machine*
begin

lemma *state-d-dioph*:
fixes *b e* :: *polynomial*
fixes *z s* :: *polynomial list*
assumes *length z* = *n length s* = *Suc m*
defines *DR* ≡ *LARY* (λ*ll*. *rm-eq-fixes.state-d p* (*ll*!*0*!*0*) (*ll*!*0*!*1*)
(*nth* (*ll*!*1*)) (*nth* (*ll*!*2*)))
[[*b*, *e*], *z*, *s*]
shows *is-dioph-rel DR*
⟨*proof*⟩

lemma *state-relations-from-recursion-dioph*:
fixes *b e* :: *polynomial*
fixes *z s* :: *polynomial list*
assumes *length z* = *n length s* = *Suc m*

defines *DR* ≡ *LARY* (λ*ll*. *rm-eq-fixes.state-relations-from-recursion p* (*ll*!*0*!*0*) (*ll*!*0*!*1*)
(*nth* (*ll*!*1*)) (*nth* (*ll*!*2*)))
[[*b*, *e*], *z*, *s*]
shows *is-dioph-rel DR*
⟨*proof*⟩

end

end

4.4.5 State unique equations

theory *State-Unique-Equations* **imports** *../Register-Machine/MultipleStepState Equation-Setup ../Diophantine/Register-Machine-Sums*

../Diophantine/Binary-And

begin

context *rm-eq-fixes*
begin

Equations not in the book:

definition *state-mask* :: *bool* **where**
state-mask ≡ ∀ *k*≤*m*. *s k* ⪯ *e*

definition *state-bound* :: *bool* **where**
state-bound ≡ ∀ *k*<*m*. *s k* < *b* ^ *q*

definition *state-unique-equations* :: *bool* **where**
state-unique-equations ≡ *state-mask* ∧ *state-bound*

end

context *register-machine*
begin

lemma *state-mask-dioph*:
fixes *e* :: *polynomial*
fixes *s* :: *polynomial list*
assumes *length s* = *Suc m*
defines *DR* ≡ *LARY* (λ*ll*. *rm-eq-fixes.state-mask p* (*ll*!*0*!*0*) (*nth* (*ll*!*1*))) [[*e*], *s*]
shows *is-dioph-rel DR*
⟨*proof*⟩

lemma *state-bound-dioph*:

fixes *b q* :: *polynomial*
fixes *s* :: *polynomial list*
assumes *length s* = *Suc m*
defines *DR* ≡ *LARY* (λ*ll*. *rm-eq-fixes.state-bound p* (*ll*!*0*!*0*) (*ll*!*0*!*1*) (*nth* (*ll*!*1*)))
[[*b*, *q*], *s*]
shows *is-dioph-rel DR*
⟨*proof*⟩

lemma *state-unique-equations-dioph*:
fixes *b q e* :: *polynomial*
fixes *s* :: *polynomial list*
assumes *length s* = *Suc m*
defines *DR* ≡ *LARY*
(λ*ll*. *rm-eq-fixes.state-unique-equations p* (*ll*!*0*!*0*) (*ll*!*0*!*1*) (*ll*!*0*!*2*) (*nth*
(*ll*!*1*)))
[[*b*, *e*, *q*], *s*]
shows *is-dioph-rel DR*
⟨*proof*⟩

end

end

4.4.6 Wrap-up: Combining all state equations

theory *All-State-Equations* **imports** *State-Unique-Equations State-d-Equation*

begin

The remaining equations:

context *rm-eq-fixes*
begin

Equation 4.27

definition *state-m* :: *bool* **where**
state-m ≡ *s m* = *b* ^ *q*

Equation not in the book

definition *state-partial-sum-mask* :: *bool* **where**
state-partial-sum-mask ≡ ∀ *M*≤*m*. (∑ *k*≤*M*. *s k*) ⪯ *e*

Wrapping it all up

definition *state-equations* :: *bool* **where**
state-equations ≡ *state-relations-from-recursion* ∧ *state-unique-equations*
∧ *state-partial-sum-mask* ∧ *state-m*

end

context *register-machine*

begin

lemma *state-m-dioph*:
fixes *b q* :: *polynomial*
fixes *s* :: *polynomial list*
assumes *length s* = *Suc m*
defines *DR* ≡ *LARY* (*λll. rm-eq-fixes.state-m p* (*ll*!*0*!*0*) (*ll*!*0*!*1*) (*nth* (*ll*!*1*)))
[[*b*, *q*], *s*]
shows *is-dioph-rel DR*
⟨*proof*⟩

lemma *state-partial-sum-mask-dioph*:
fixes *e* :: *polynomial*
fixes *s* :: *polynomial list*
assumes *length s* = *Suc m*
defines *DR* ≡ *LARY* (*λll. rm-eq-fixes.state-partial-sum-mask p* (*ll*!*0*!*0*) (*nth* (*ll*!*1*))) [[*e*], *s*]
shows *is-dioph-rel DR*
⟨*proof*⟩

definition *state-equations-relation* :: *polynomial* ⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial list*
⇒ *polynomial list* ⇒ *relation* (‹[*STATE*] - - - - -›)**where**
[*STATE*] *b e q z s* ≡ *LARY* (*λll. rm-eq-fixes.state-equations p* (*ll*!*0*!*0*) (*ll*!*0*!*1*) (*ll*!*0*!*2*)
(*nth* (*ll*!*1*)) (*nth* (*ll*!*2*)))
[[*b*, *e*, *q*], *z*, *s*]

lemma *state-equations-dioph*:
fixes *b e q* :: *polynomial*
fixes *s z* :: *polynomial list*
assumes *length s* = *Suc m length z* = *n*
defines *DR* ≡ [*STATE*] *b e q z s*
shows *is-dioph-rel DR*
⟨*proof*⟩

end

end

4.4.7 Equations for masking relations

theory *Mask-Equations*
imports ../*Register-Machine*/*MachineMasking Equation-Setup* ../*Diophantine*/*Binary-And*

abbrevs *mb* = ⪯

begin

context *rm-eq-fixes*
begin

Equation 4.15

definition *register-mask* :: *bool* **where**
register-mask ≡ ∀ *l* < *n*. *r l* ⪯ *d*

Equation 4.17

definition *zero-indicator-mask* :: *bool* **where**
zero-indicator-mask ≡ ∀ *l* < *n*. *z l* ⪯ *e*

Equation 4.20

definition *zero-indicator-0-or-1* :: *bool* **where**
zero-indicator-0-or-1 ≡ ∀ *l*<*n*. *2*^*c* ∗ *z l* = (*r l* + *d*) && *f*

definition *mask-equations* :: *bool* **where**
mask-equations ≡ *register-mask* ∧ *zero-indicator-mask* ∧ *zero-indicator-0-or-1*

end

context *register-machine*
begin

lemma *register-mask-dioph*:
fixes *d r*
assumes *n* = *length r*
defines *DR* ≡ (*NARY* (λ*l*. *rm-eq-fixes.register-mask n* (*l*!*0*) (*shift l 1*)) ([*d*] @ *r*))
shows *is-dioph-rel DR*
⟨*proof*⟩

lemma *zero-indicator-mask-dioph*:
fixes *e z*
assumes *n* = *length z*
defines *DR* ≡ (*NARY* (λ*l*. *rm-eq-fixes.zero-indicator-mask n* (*l*!*0*) (*shift l 1*)) ([*e*] @ *z*))
shows *is-dioph-rel DR*
⟨*proof*⟩

lemma *zero-indicator-0-or-1-dioph*:
fixes *c d f r z*
assumes *n* = *length r* **and** *n* = *length z*
defines *DR* ≡ *LARY* (λ*ll*. *rm-eq-fixes.zero-indicator-0-or-1 n* (*ll*!*0*!*0*) (*ll*!*0*!*1*) (*ll*!*0*!*2*)
(*nth* (*ll*!*1*)) (*nth* (*ll*!*2*))) [[*c*, *d*, *f*], *r*, *z*]
shows *is-dioph-rel DR*
⟨*proof*⟩

definition *mask-equations-relation* (‹[*MASK*] - - - - - -›) **where**
[*MASK*] *c d e f r z* ≡ *LARY* (λ*ll*. *rm-eq-fixes.mask-equations n*
(*ll*!*0*!*0*) (*ll*!*0*!*1*) (*ll*!*0*!*2*) (*ll*!*0*!*3*) (*nth* (*ll*!*1*)) (*nth* (*ll*!*2*)))
[[*c*, *d*, *e*, *f*], *r*, *z*]

lemma *mask-equations-relation-dioph*:
fixes *c d e f r z*
assumes *n* = *length r* **and** *n* = *length z*
defines *DR* ≡ [*MASK*] *c d e f r z*
shows *is-dioph-rel DR*
⟨*proof*⟩

end

end

4.4.8 Equations for arithmetization constants

theory *Constants-Equations* **imports** *Equation-Setup ../Register-Machine/MachineMasking*
../Diophantine/Binary-And

begin

context *rm-eq-fixes*
begin

Equation 4.14

definition *constant-b* :: *bool* **where**
constant-b ≡ *b* = *B c*

Equation 4.16

definition *constant-d* :: *bool* **where**
constant-d ≡ *d* = *D q c b*

Equation 4.18

definition *constant-e* :: *bool* **where**
constant-e ≡ *e* = *E q b*

Equation 4.21

definition *constant-f* :: *bool* **where**
constant-f ≡ *f* = *F q c b*

Equation not in the book

definition *c-gt-0* :: *bool* **where**
c-gt-0 ≡ *c* > *0*

Equation 4.26

definition *a-bound* :: *bool* **where**

a-bound ≡ *a* < *2* ^ *c*

Equation not in the book

definition *q-gt-0* :: *bool* **where**
q-gt-0 ≡ *q* > *0*

definition *constants-equations* :: *bool* **where**
constants-equations ≡ *constant-b* ∧ *constant-d* ∧ *constant-e* ∧ *constant-f*

definition *miscellaneous-equations* :: *bool* **where**
miscellaneous-equations ≡ *c-gt-0* ∧ *a-bound* ∧ *q-gt-0*

end

context *register-machine*
begin

definition *rm-constant-equations* ::
polynomial ⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial* ⇒ *relation*
(‹[*CONST*] - - - - - -›) **where**
[*CONST*] *b c d e f q* ≡ *NARY* (λ*l*. *rm-eq-fixes.constants-equations*
(*l*!*0*) (*l*!*1*) (*l*!*2*) (*l*!*3*) (*l*!*4*) (*l*!*5*)) [*b*, *c*, *d*, *e*, *f*, *q*]

definition *rm-miscellaneous-equations* ::
polynomial ⇒ *polynomial* ⇒ *polynomial* ⇒ *relation*
(‹[*MISC*] - - -›) **where**
[*MISC*] *c a q* ≡ *NARY* (λ*l*. *rm-eq-fixes.miscellaneous-equations*
(*l*!*0*) (*l*!*1*) (*l*!*2*)) [*c*, *a*, *q*]

lemma *rm-constant-equations-dioph*:
fixes *b c d e f q*
defines *DR* ≡ [*CONST*] *b c d e f q*
shows *is-dioph-rel DR*
⟨*proof*⟩

lemma *rm-miscellaneous-equations-dioph*:
fixes *c a q*
defines *DR* ≡ [*MISC*] *a c q*
shows *is-dioph-rel DR*
⟨*proof*⟩

end

end

4.4.9 Invariance of equations

theory *All-Equations-Invariance*
imports *Register-Equations All-State-Equations Mask-Equations Constants-Equations*

begin

context *register-machine*
begin

definition *all-equations* **where**
all-equations a q b c d e f r z s
≡ *rm-eq-fixes.register-equations p n a b q r z s*
∧ *rm-eq-fixes.state-equations p b e q z s*
∧ *rm-eq-fixes.mask-equations n c d e f r z*
∧ *rm-eq-fixes.constants-equations b c d e f q*
∧ *rm-eq-fixes.miscellaneous-equations a c q*

lemma *all-equations-invariance*:
fixes $r\ z\ s :: nat \Rightarrow nat$
and $r'\ z'\ s' :: nat \Rightarrow nat$
assumes $\forall\, i{<}n.\ r\ i = r'\ i$ **and** $\forall\, i{<}n.\ z\ i = z'\ i$ **and** $\forall\, i{<}Suc\ m.\ s\ i = s'\ i$
shows *all-equations a q b c d e f r z s* = *all-equations a q b c d e f r′ z′ s′*
⟨*proof*⟩

end

end

4.4.10 Wrap-Up: Combining all equations

theory *All-Equations*
imports *All-Equations-Invariance*

begin

context *register-machine*
begin

definition *all-equations-relation* :: *polynomial* ⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial*
⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial* ⇒ *polynomial list* ⇒ *polynomial list* ⇒ *polynomial list*
⇒ *relation* (‹[*ALLEQ*] - - - - - - - - - -›) **where**
[*ALLEQ*] *a q b c d e f r z s*
≡ *LARY* (λ*ll. all-equations* (*ll*!*0*!*0*) (*ll*!*0*!*1*) (*ll*!*0*!*2*) (*ll*!*0*!*3*) (*ll*!*0*!*4*) (*ll*!*0*!*5*) (*ll*!*0*!*6*)
(*nth* (*ll*!*1*)) (*nth* (*ll*!*2*)) (*nth* (*ll*!*3*)))
[[*a*, *q*, *b*, *c*, *d*, *e*, *f*], *r*, *z*, *s*]

lemma *all-equations-dioph*:
  fixes *A f e d c b q* :: *polynomial*
  fixes *r z s* :: *polynomial list*
  assumes *length r* = *n length z* = *n length s* = *Suc m*
  defines *DR* ≡ [*ALLEQ*] *A q b c d e f r z s*
  shows *is-dioph-rel DR*
⟨*proof*⟩

definition *rm-equations* :: *nat* ⇒ *bool* **where**
  rm-equations a ≡ ∃ *q* :: *nat*.
                ∃ *b c d e f* :: *nat*.
                ∃ *r z* :: *register* ⇒ *nat*.
                ∃ *s* :: *state* ⇒ *nat*.
                all-equations a q b c d e f r z s

definition *rm-equations-relation* :: *polynomial* ⇒ *relation* (‹[*RM*] -›) **where**
  [*RM*] *A* ≡ *UNARY* (*rm-equations*) *A*

lemma *rm-dioph*:
  fixes *A*
  fixes *ic* :: *configuration*
  defines *DR* ≡ [*RM*] *A*
  shows *is-dioph-rel DR*
⟨*proof*⟩

end

end

4.5 Equivalence of register machine and arithmetizing equations

theory *Machine-Equation-Equivalence* **imports** *All-Equations*
  ../Register-Machine/MachineEquations
  ../Register-Machine/MultipleToSingleSteps

begin

context *register-machine*
begin

lemma *conclusion-4-5*:
  assumes *is-val*: *is-valid-initial ic p a*
  and *n-def*: *n* ≡ *length* (*snd ic*)
  shows (∃ *q*. *terminates ic p q*) = *rm-equations a*
⟨*proof*⟩

end

end

5 Proof of the DPRM theorem

theory *DPRM*
  imports *Machine-Equations/Machine-Equation-Equivalence*
begin

definition *is-recenum* :: *nat set* ⇒ *bool* **where**
  is-recenum A =
    (∃ *p* :: *program*.
     ∃ *n* :: *nat*.
     ∀ *a* :: *nat*. ∃ *ic*. *ic* = *initial-config n a* ∧ *is-valid-initial ic p a* ∧
    (*a* ∈ *A*) = (∃ *q*::*nat*. *terminates ic p q*))

theorem *DPRM*: *is-recenum A* ⟹ *is-dioph-set A*
⟨*proof*⟩

end

References

[1] J. Bayer, M. David, A. Pal, and B. Stock. Beginners' quest to formalize mathematics: A feasibility study in Isabelle. In C. Kaliszyk, E. Brady, A. Kohlhase, and C. Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 16–27, Cham, 2019. Springer International Publishing.

[2] J. Bayer, M. David, A. Pal, B. Stock, and D. Schleicher. The DPRM Theorem in Isabelle (Short Paper). In J. Harrison, J. O'Leary, and A. Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:7, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] M. Carneiro. A Lean formalization of Matiyasevič's theorem. https://arxiv.org/abs/1802.01795v1, 02 2018.

[4] D. Larchey-Wendling and Y. Forster. Hilbert's Tenth Problem in Coq. In H. Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:20, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] Y. Matiyasevich. On Hilbert's tenth problem. In M. Lamoureux, editor, *PIMS Distinguished Chair Lectures*, volume 1. Pacific Institute for the Mathematical Sciences, 2000.

[6] K. Pak. Diophantine sets. preliminaries. *Formalized Mathematics*, 26(1):81–90, 2018.

[7] K. Pak. The Matiyasevich theorem. preliminaries. *Formalized Mathematics*, 25(4):315–322, 2018.

[8] J. Xu, X. Zhang, and C. Urban. Mechanising Turing machines and computability theory in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving. ITP 2013.*, volume 7998 of *Lecture Notes in Computer Science*, pages 147–162. Springer Berlin, Heidelberg, 2013.

[9] J. Xu, X. Zhang, C. Urban, and S. J. C. Joosten. Universal Turing machine. *Archive of Formal Proofs*, Feb. 2019. https://isa-afp.org/entries/Universal_Turing_Machine.html, Formal proof development.