

Diophantine Equations and the DPRM Theorem

Jonas Bayer*

Marco David*

Benedikt Stock*

Abhik Pal

Yuri Matiyasevich[†]

Dierk Schleicher

May 26, 2024

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

1	Diophantine Equations	5
1.1	Parametric Polynomials	5
1.2	Variable Assignments	6
1.3	Diophantine Relations and Predicates	10
1.4	Existential quantification is Diophantine	14
1.5	Mod is Diophantine	14
2	Exponentiation is Diophantine	15
2.1	Expressing Exponentiation in terms of the alpha function . .	15
2.1.1	2x2 matrices and operations	15
2.1.2	Properties of 2x2 matrices	16
2.1.3	Special second-order recurrent sequences	16
2.1.4	First order relation	17
2.1.5	Characteristic equation	18
2.1.6	Divisibility properties	18

*Equal contribution.

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

2.1.7	Divisibility properties (continued)	19
2.1.8	Congruence properties	20
2.1.9	Diophantine definition of a sequence α	22
2.1.10	Exponentiation is Diophantine	24
2.2	Diophantine description of α function	25
2.3	Exponentiation is a Diophantine Relation	26
2.4	Digit function is Diophantine	27
2.5	Binomial Coefficient is Diophantine	27
2.6	Binary orthogonality is Diophantine	28
2.7	Binary masking is Diophantine	28
2.8	Binary and is Diophantine	29
3	Register Machines	30
3.1	Register Machine Specification	30
3.1.1	Basic Datatype Definitions	30
3.1.2	Essential Functions to operate the Register Machine	31
3.1.3	Validity Checks and Assumptions	31
3.2	Simple Properties of Register Machines	33
3.2.1	From Configurations to a Protocol	33
3.2.2	Protocol Properties	34
3.3	Simulation of a Register Machine	35
3.4	Single step relations	39
3.4.1	Registers	39
3.4.2	States	41
3.5	Multiple step relations	41
3.5.1	Registers	41
3.5.2	States	43
3.6	Masking properties	45
4	Arithmetization of Register Machines	51
4.1	A first definition of the arithmetizing equations	51
4.2	Preliminary commutation relations	52
4.3	From multiple to single step relations	55
4.4	Arithmetizing equations are Diophantine	61
4.4.1	Preliminary: Register machine sums are Diophantine	62
4.4.2	Register Equations	65
4.4.3	State 0 equation	67
4.4.4	State d equation	68
4.4.5	State unique equations	69
4.4.6	Wrap-up: Combining all state equations	70
4.4.7	Equations for masking relations	71
4.4.8	Equations for arithmetization constants	73
4.4.9	Invariance of equations	74
4.4.10	Wrap-Up: Combining all equations	75

4.5	Equivalence of register machine and arithmetizing equations .	76
5	Proof of the DPRM theorem	77

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  $\Rightarrow$  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  $\Rightarrow$  assignment  $\Rightarrow$  assignment  $\Rightarrow$  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  $\Rightarrow$  bool where
  is-dioph-set A = ( $\exists$  P1 P2::ppolynomial.  $\forall$  a. (a  $\in$  A)
     $\longleftrightarrow$  ( $\exists$  v. ppeval P1 ( $\lambda$ x. a) v = ppeval P2 ( $\lambda$ x.
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* \Rightarrow *assignment* \Rightarrow *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* \Rightarrow *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* \Rightarrow *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* \Rightarrow *assignment* \Rightarrow (*nat* \Rightarrow *nat*) **where**
list-eval *PL* *a* = *nth* (*map* (λx . *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 necessary to e.g. prove that conjunction is diophantine.

theory *Assignments*

imports *Parametric-Polynomials*

begin

definition *shift* :: *nat list* \Rightarrow *nat* \Rightarrow *assignment* **where**
shift *l* *a* \equiv λi . *l* ! (*i* + *a*)

definition *push* :: *assignment* \Rightarrow *nat* \Rightarrow *assignment* **where**
push *a* *n* *i* = (*if* *i* = 0 *then* *n* *else* *a* (*i*-1))

definition *push-list* :: *assignment* \Rightarrow *nat list* \Rightarrow *nat* \Rightarrow *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* \implies *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* \Rightarrow *ppolynomial* \Rightarrow *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* \Rightarrow *nat set* **where**
var-set (*ppolynomial.Const* *c*) = {*c*} |
var-set (*ppolynomial.Param* *x*) = {*x*} |
var-set (*ppolynomial.Var* *x*) = {*x*} |
var-set (*D1* + *D2*) = *var-set* *D1* \cup *var-set* *D2* |
var-set (*D1* - *D2*) = *var-set* *D1* \cup *var-set* *D2* |
var-set (*D1* * *D2*) = *var-set* *D1* \cup *var-set* *D2*

definition *disjoint-var* :: *ppolynomial* \Rightarrow *ppolynomial* \Rightarrow *bool* **where**
disjoint-var *P Q* = (*var-set* *P* \cap *var-set* *Q* = {*c*})

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* ($P1 + P2$) $Q = (\text{disjoint-var } P1 \ Q \wedge \text{disjoint-var } P2 \ Q)$
<proof>

lemma *disjoint-var-diff*[*disjoint-vars*]: *disjoint-var* ($P1 - P2$) $Q = (\text{disjoint-var } P1 \ Q \wedge \text{disjoint-var } P2 \ Q)$
<proof>

lemma *disjoint-var-prod*[*disjoint-vars*]: *disjoint-var* ($P1 * P2$) $Q = (\text{disjoint-var } P1 \ Q \wedge \text{disjoint-var } P2 \ Q)$
<proof>

lemma *aux-var-set*:
assumes $\forall i \in \text{var-set } P. x \ i = y \ i$
shows $\text{ppeval } P \ a \ x = \text{ppeval } P \ a \ y$
<proof>

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

definition *zip-assignments* :: *ppolynomial* \Rightarrow *ppolynomial* \Rightarrow *assignment* \Rightarrow *assignment* \Rightarrow *assignment*
where *zip-assignments* $P \ Q \ v \ w \ i = (\text{if } i \in \text{var-set } P \ \text{then } v \ i \ \text{else } w \ i)$

lemma *help-eval-zip-assignments1*:
shows $\text{ppeval } P1 \ a \ (\lambda i. \text{if } i \in \text{var-set } P1 \cup \text{var-set } P2 \ \text{then } v \ i \ \text{else } w \ i)$
 $= \text{ppeval } P1 \ a \ (\lambda i. \text{if } i \in \text{var-set } P1 \ \text{then } v \ i \ \text{else } w \ i)$
<proof>

lemma *help-eval-zip-assignments2*:
shows $\text{ppeval } P2 \ a \ (\lambda i. \text{if } i \in \text{var-set } P1 \cup \text{var-set } P2 \ \text{then } v \ i \ \text{else } w \ i)$
 $= \text{ppeval } P2 \ a \ (\lambda i. \text{if } i \in \text{var-set } P2 \ \text{then } v \ i \ \text{else } w \ i)$
<proof>

lemma *eval-zip-assignments1*:
fixes $v \ w$
assumes *disjoint-var* $P \ Q$
defines $x \equiv \text{zip-assignments } P \ Q \ v \ w$
shows $\text{ppeval } P \ a \ v = \text{ppeval } P \ a \ x$
<proof>

lemma *eval-zip-assignments2*:
fixes $v \ w$
assumes *disjoint-var* $P \ Q$
defines $x \equiv \text{zip-assignments } P \ Q \ v \ w$
shows $\text{ppeval } Q \ a \ w = \text{ppeval } Q \ a \ x$
<proof>

lemma *zip-assignments-correct*:

assumes $\text{ppeval } P1 \ a \ v = \text{ppeval } P2 \ a \ v$ **and** $\text{ppeval } Q1 \ a \ w = \text{ppeval } Q2 \ a \ w$
and $\text{disjoint-var } (P1 + P2) \ (Q1 + Q2)$
defines $x \equiv \text{zip-assignments } (P1 + P2) \ (Q1 + Q2) \ v \ w$
shows $\text{ppeval } P1 \ a \ x = \text{ppeval } P2 \ a \ x$ **and** $\text{ppeval } Q1 \ a \ x = \text{ppeval } Q2 \ a \ x$
(proof)

lemma *disjoint-var-unifies*:

assumes $\exists v1. \text{ppeval } P1 \ a \ v1 = \text{ppeval } P2 \ a \ v1$ **and** $\exists v2. \text{ppeval } Q1 \ a \ v2 = \text{ppeval } Q2 \ a \ v2$
and $\text{disjoint-var } (P1 + P2) \ (Q1 + Q2)$
shows $\exists v. \text{ppeval } P1 \ a \ v = \text{ppeval } P2 \ a \ v \wedge \text{ppeval } Q1 \ a \ v = \text{ppeval } Q2 \ a \ v$
(proof)

A function to manipulate variables in ppolynomials

fun *push-var* :: $\text{ppolynomial} \Rightarrow \text{nat} \Rightarrow \text{ppolynomial}$ **where**
 $\text{push-var } (\text{ppolynomial.Var } x) \ n = \text{ppolynomial.Var } (x + n) \ |$
 $\text{push-var } (D1 + D2) \ n = \text{push-var } D1 \ n + \text{push-var } D2 \ n \ |$
 $\text{push-var } (D1 - D2) \ n = \text{push-var } D1 \ n - \text{push-var } D2 \ n \ |$
 $\text{push-var } (D1 * D2) \ n = \text{push-var } D1 \ n * \text{push-var } D2 \ n \ |$
 $\text{push-var } D \ n = D$

lemma *push-var-bound*: $x \in \text{var-set } (\text{push-var } P \ (\text{Suc } n)) \Longrightarrow x > n$
(proof)

definition *pull-assignment* :: $\text{assignment} \Rightarrow \text{nat} \Rightarrow \text{assignment}$ **where**
 $\text{pull-assignment } v \ n = (\lambda x. v \ (x+n))$

lemma *push-var-pull-assignment*:

shows $\text{ppeval } (\text{push-var } P \ n) \ a \ v = \text{ppeval } P \ a \ (\text{pull-assignment } v \ n)$
(proof)

lemma *max-set*: $\text{finite } A \Longrightarrow \forall x \in A. x \leq \text{Max } A$
(proof)

fun *push-param* :: $\text{polynomial} \Rightarrow \text{nat} \Rightarrow \text{polynomial}$ **where**

$\text{push-param } (\text{Const } c) \ n = \text{Const } c \ |$
 $\text{push-param } (\text{Param } x) \ n = \text{Param } (x + n) \ |$
 $\text{push-param } (\text{Sum } D1 \ D2) \ n = \text{Sum } (\text{push-param } D1 \ n) \ (\text{push-param } D2 \ n) \ |$
 $\text{push-param } (\text{NatDiff } D1 \ D2) \ n = \text{NatDiff } (\text{push-param } D1 \ n) \ (\text{push-param } D2 \ n) \ |$
 $\text{push-param } (\text{Prod } D1 \ D2) \ n = \text{Prod } (\text{push-param } D1 \ n) \ (\text{push-param } D2 \ n)$

definition *push-param-list* :: $\text{polynomial list} \Rightarrow \text{nat} \Rightarrow \text{polynomial list}$ **where**
 $\text{push-param-list } s \ k \equiv \text{map } (\lambda x. \text{push-param } x \ k) \ s$

lemma *push-param0*: $\text{push-param } P \ 0 = P$
 ⟨proof⟩

lemma *push-push-aux*: $\text{peval } (\text{push-param } P \ (\text{Suc } m)) \ (\text{push } a \ n) = \text{peval } (\text{push-param } P \ m) \ a$
 ⟨proof⟩

lemma *push-push*:
shows $\text{length } ns = n \implies \text{peval } (\text{push-param } P \ n) \ (\text{push-list } a \ ns) = \text{peval } P \ a$
 ⟨proof⟩

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

lemma *push-push1*: $\text{peval } (\text{push-param } P \ 1) \ (\text{push } a \ k) = \text{peval } P \ a$
 ⟨proof⟩

lemma *push-push-map*: $\text{length } ns = n \implies$
 $\text{list-eval } (\text{map } (\lambda x. \text{push-param } x \ n) \ ls) \ (\text{push-list } a \ ns) = \text{list-eval } ls \ a$
 ⟨proof⟩

lemma *push-push-map-i*: $\text{length } ns = n \implies i < \text{length } ls \implies$
 $\text{peval } (\text{map } (\lambda x. \text{push-param } x \ n) \ ls \ ! \ i) \ (\text{push-list } a \ ns) = \text{list-eval } ls \ a \ i$
 ⟨proof⟩

lemma *push-push-map1*: $i < \text{length } ls \implies$
 $\text{peval } (\text{map } (\lambda x. \text{push-param } x \ 1) \ ls \ ! \ i) \ (\text{push } a \ n) = \text{list-eval } ls \ a \ i$
 ⟨proof⟩

end

1.3 Diophantine Relations and Predicates

theory *Diophantine-Relations*

imports *Assignments*

begin

datatype *relation* =

NARY nat list \Rightarrow bool polynomial list
 | *AND relation relation (infixl [\wedge] 35)*
 | *OR relation relation (infixl [\vee] 30)*
 | *EXIST-LIST nat relation ((\exists -] - 10)*

fun *eval* :: *relation \Rightarrow assignment \Rightarrow bool* **where**

eval (NARY R PL) a = R (map ($\lambda P. \text{peval } P \ a$) PL)
 | *eval (AND D1 D2) a = (eval D1 a \wedge eval D2 a)*

| $eval (OR D1 D2) a = (eval D1 a \vee eval D2 a)$
| $eval ([\exists n] D) a = (\exists ks::nat\ list. n = length\ ks \wedge eval D (push-list\ a\ ks))$

definition *is-dioph-rel* :: *relation* \Rightarrow *bool* **where**

is-dioph-rel $DR = (\exists P_1 P_2::ppolynomial. \forall a. (eval\ DR\ a) \longleftrightarrow (\exists v. ppeval\ P_1\ a\ v = ppeval\ P_2\ a\ v))$

definition *UNARY* :: (*nat* \Rightarrow *bool*) \Rightarrow *polynomial* \Rightarrow *relation* **where**

UNARY $R\ P = NARY (\lambda l. R (!0)) [P]$

lemma *unary-eval*: $eval (UNARY\ R\ P) a = R (peval\ P\ a)$

<proof>

definition *BINARY* :: (*nat* \Rightarrow *nat* \Rightarrow *bool*) \Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *relation* **where**

BINARY $R\ P_1\ P_2 = NARY (\lambda l. R (!0) (!1)) [P_1, P_2]$

lemma *binary-eval*: $eval (BINARY\ R\ P_1\ P_2) a = R (peval\ P_1\ a) (peval\ P_2\ a)$

<proof>

definition *TERNARY* :: (*nat* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *bool*)

\Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *relation* **where**

TERNARY $R\ P_1\ P_2\ P_3 = NARY (\lambda l. R (!0) (!1) (!2)) [P_1, P_2, P_3]$

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* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *bool*)

\Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *relation* **where**

QUATERNARY $R\ P_1\ P_2\ P_3\ P_4 = NARY (\lambda l. R (!0) (!1) (!2) (!3)) [P_1, P_2, P_3, P_4]$

definition *EXIST* :: *relation* \Rightarrow *relation* ($[\exists]$ - 10) **where**

$([\exists] D) = ([\exists\ 1] D)$

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

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

fun *ALLC-LIST* :: *nat list* \Rightarrow (*nat* \Rightarrow *relation*) \Rightarrow *relation* ($[\forall\ in\ -]\ -$) **where**

$[\forall\ in\ []] DF = TRUE$ |

$[\forall\ in\ (l\ \# \ ls)] DF = (DF\ l\ [\wedge] [\forall\ in\ ls] DF)$

lemma *ALLC-LIST-eval-list-all*: $eval ([\forall\ in\ L] DF) a = list-all (\lambda l. eval (DF\ l) a) L$

<proof>

lemma *ALLC-LIST-eval*: $eval ([\forall \text{ in } L] DF) a = (\forall k < \text{length } L. eval (DF (L!k)) a)$

<proof>

definition *ALLC* :: $nat \Rightarrow (nat \Rightarrow relation) \Rightarrow relation ([\forall < -] -)$ **where**
 $[\forall < n] D \equiv [\forall \text{ in } [0..<n]] D$

lemma *ALLC-eval*: $eval ([\forall < n] DF) a = (\forall k < n. eval (DF k) a)$

<proof>

fun *concat* :: $'a \text{ list list} \Rightarrow 'a \text{ list}$ **where**

$concat [] = []$ |

$concat (l \# ls) = l @ concat ls$

fun *splits* :: $'a \text{ list} \Rightarrow nat \text{ list} \Rightarrow 'a \text{ list list}$ **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* :: $(nat \text{ list list} \Rightarrow bool) \Rightarrow (polynomial \text{ list list}) \Rightarrow relation$ **where**
 $LARY R PLL = NARY (\lambda l. R (splits l (map length PLL))) (concat PLL)$

lemma *LARY-eval*:

fixes *PLL* :: $polynomial \text{ list list}$

shows $eval (LARY R PLL) a = R (map (map (\lambda P. peval P a)) PLL)$

<proof>

lemma *or-dioph*:

assumes *is-dioph-rel A* **and** *is-dioph-rel B*

shows *is-dioph-rel (A [V] B)*

<proof>

lemma *exists-disjoint-vars*:

fixes *Q1 Q2* :: $ppolynomial$

fixes *A* :: $relation$

assumes *is-dioph-rel A*

shows $\exists P1 P2. disjoint\text{-}var (P1 + P2) (Q1 + Q2)$

$\wedge (\forall a. eval A a \longleftrightarrow (\exists v. ppeval P1 a v = ppeval P2 a v))$

<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* $Q R \equiv \text{BINARY } (=) Q R$
definition *lt* (**infix** [<] 50) **where** *lt* $Q R \equiv \text{BINARY } (<) Q R$
definition *le* (**infix** [≤] 50) **where** *le* $Q R \equiv Q [<] R \vee Q [=] R$
definition *gt* (**infix** [>] 50) **where** *gt* $Q R \equiv R [<] Q$
definition *ge* (**infix** [≥] 50) **where** *ge* $Q R \equiv Q [>] R \vee Q [=] R$

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* $Q \equiv \mathbf{0} [=] Q$
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*]: $\forall i < n. \text{is-dioph-rel } (DF \ i) \Rightarrow \text{is-dioph-rel } ([\forall < n]$
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 ( $[\exists n]$  D)
<proof>
```

```
lemma exist-dioph[dioph]:
  fixes D
  assumes is-dioph-rel D
  shows is-dioph-rel ( $[\exists]$  D)
<proof>
```

```
lemma exist-eval[defs]:
  shows eval ( $[\exists]$  D) a = ( $\exists k.$  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  $\equiv$  (BINARY (dvd) Q R)
```

```
lemma dvd-repr:
  fixes a b :: nat
  shows a dvd b  $\longleftrightarrow$  ( $\exists x.$  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  $\equiv$  (TERNARY ( $\lambda a b c.$  a mod b = c mod b) A B C)
declare mod-def[defs]
```

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

```

shows  $a \bmod b = c \bmod b \iff (\exists x y. c + x*b = a + y*b)$ 
  <proof>

lemma mod-dioph[dioph]:
  fixes  $A B C$ 
  defines  $D \equiv (MOD A B C)$ 
  shows is-dioph-rel  $D$ 
  <proof>

declare mod-def[defs]

end

```

2 Exponentiation is Diophantine

2.1 Expressing Exponentiation in terms of the alpha function

```

theory Exponentiation
  imports HOL-Library.Discrete
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  $\Rightarrow$  mat2  $\Rightarrow$  mat2 where
  mat-plus  $A B = \text{mat } (\text{mat-11 } A + \text{mat-11 } B) (\text{mat-12 } A + \text{mat-12 } B)$ 
    ( $\text{mat-21 } A + \text{mat-21 } B$ ) ( $\text{mat-22 } A + \text{mat-22 } B$ )

fun mat-mul:: mat2  $\Rightarrow$  mat2  $\Rightarrow$  mat2 where
  mat-mul  $A B = \text{mat } (\text{mat-11 } A * \text{mat-11 } B + \text{mat-12 } A * \text{mat-21 } B)$ 
    ( $\text{mat-11 } A * \text{mat-12 } B + \text{mat-12 } A * \text{mat-22 } B$ )
    ( $\text{mat-21 } A * \text{mat-11 } B + \text{mat-22 } A * \text{mat-21 } B$ )
    ( $\text{mat-21 } A * \text{mat-12 } B + \text{mat-22 } A * \text{mat-22 } B$ )

fun mat-pow:: nat  $\Rightarrow$  mat2  $\Rightarrow$  mat2 where
  mat-pow 0 = mat 1 0 0 1 |
  mat-pow  $n A = \text{mat-mul } A (\text{mat-pow } (n - 1) A)$ 

lemma mat-pow-2[simp]: mat-pow 2  $A = \text{mat-mul } A A$ 
  <proof>

fun mat-det::mat2  $\Rightarrow$  int where
  mat-det  $M = \text{mat-11 } M * \text{mat-22 } M - \text{mat-12 } M * \text{mat-21 } M$ 

```

fun *mat-scalar-mult*::*int* \Rightarrow *mat2* \Rightarrow *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* \Rightarrow *mat2* \Rightarrow *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* \Rightarrow *vec2* \Rightarrow *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-22 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* \Rightarrow *nat* \Rightarrow *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* \geq 2 \implies α *b n* < α *b (Suc n)* \wedge 0 < α *b (Suc n)*
<proof>

lemma *alpha-strictly-increasing-general*:
fixes *b n m*::*nat*

assumes $b > 2 \wedge m > n$
shows $\alpha b m > \alpha b n$
 $\langle proof \rangle$

Equation 3.4

lemma *alpha-superlinear*: $b > 2 \implies \text{int } n \leq \alpha b n$
 $\langle proof \rangle$

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

lemma *alpha-nonnegative*:
shows $b > 2 \implies \alpha b n \geq 0$
 $\langle proof \rangle$

Equation 3.5

lemma *alpha-linear*: $\alpha 2 n = n$
 $\langle proof \rangle$

Equation 3.6 (modified)

lemma *alpha-exponential-1*: $b > 0 \implies \text{int } b \wedge n \leq \alpha (b + 1) (n + 1)$
 $\langle proof \rangle$

lemma *alpha-exponential-2*: $\text{int } b > 2 \implies \alpha b (n + 1) \leq (\text{int } b) \wedge (n)$
 $\langle proof \rangle$

2.1.4 First order relation

Equation 3.7 - Definition of A

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

Equation 3.9 - Definition of B

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

declare $A.\text{simps}[simp \ del]$
declare $B.\text{simps}[simp \ del]$

Equation 3.8

lemma *A-rec*: $b > 2 \implies A b (\text{Suc } n) = \text{mat-mul } (A b n) (B b)$
 $\langle proof \rangle$

Equation 3.10

lemma *A-pow*: $b > 2 \implies A b n = \text{mat-pow } n (B b)$
 $\langle proof \rangle$

2.1.5 Characteristic equation

Equation 3.11

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

Equation 3.12

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

Equation 3.12

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

Equations 3.14 to 3.17

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

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

2.1.6 Divisibility properties

The following lemmas are needed in the proof of equation 3.25

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

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

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

defines $M \equiv \text{mat } a \ b \ c \ d$
assumes $\text{mat-21 } M \bmod m = 0$
shows $(\text{mat-21 } (\text{mat-pow } l \ M)) \bmod m = 0$ (**is** $?P \ l$)
 $\langle \text{proof} \rangle$

lemma *div-3253*:
fixes $a \ b \ c \ d \ m :: \text{int}$ **and** $l :: \text{nat}$
defines $M \equiv \text{mat } a \ b \ c \ d$
assumes $\text{mat-21 } M \bmod m = 0$
shows $((\text{mat-11 } (\text{mat-pow } l \ M)) - a^{\wedge} l) \bmod m = 0$ (**is** $?P \ l$)
 $\langle \text{proof} \rangle$

Equation 3.25

lemma *divisibility-lemma1*:
fixes $b \ k \ m :: \text{nat}$
assumes $b > 2$ **and** $k > 0$
defines $n \equiv m \bmod k$
defines $l \equiv (m - n) \text{ div } k$
shows $\alpha \ b \ m \bmod \alpha \ b \ k = \alpha \ b \ n * (\alpha \ b \ (k+1))^{\wedge} l \bmod \alpha \ b \ k$
 $\langle \text{proof} \rangle$

Prerequisite lemma for 3.27

lemma *div-coprime*:
assumes $b > 2 \ n \geq 0$
shows $\text{coprime } (\alpha \ b \ k) (\alpha \ b \ (k+1))$ (**is** $?P$)
 $\langle \text{proof} \rangle$

Equation 3.27

lemma *divisibility-lemma2*:
fixes $b \ k \ m :: \text{nat}$
assumes $b > 2$ **and** $k > 0$
defines $n \equiv m \bmod k$
defines $l \equiv (m - n) \text{ div } k$
assumes $\alpha \ b \ k \ \text{dvd} \ \alpha \ b \ m$
shows $\alpha \ b \ k \ \text{dvd} \ \alpha \ b \ n$
 $\langle \text{proof} \rangle$

Equation 3.23 - main result of this section

theorem *divisibility-alpha*:
assumes $b > 2$ **and** $k > 0$
shows $\alpha \ b \ k \ \text{dvd} \ \alpha \ b \ m \longleftrightarrow k \ \text{dvd} \ m$ (**is** $?P \longleftrightarrow ?Q$)
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *divisibility-cong*:

fixes $e\ f :: \text{int}$

fixes $l :: \text{nat}$

fixes $M :: \text{mat2}$

assumes $\text{mat-22 } M = 0\ \text{mat-21 } M = 1$

shows $(\text{mat-21 } (\text{mat-pow } l\ (\text{mat-minus } (\text{mat-scalar-mult } e\ M)\ (\text{mat-scalar-mult } f\ ID)))) \text{ mod } e^{\wedge}2 = (-1)^{\wedge}(l-1) * l * e * f^{\wedge}(l-1) * (\text{mat-21 } M) \text{ mod } e^{\wedge}2$

$\wedge \text{mat-22 } (\text{mat-pow } l\ (\text{mat-minus } (\text{mat-scalar-mult } e\ M)\ (\text{mat-scalar-mult } f\ ID))) \text{ mod } e^{\wedge}2 = (-1)^{\wedge}l * f^{\wedge}l \text{ mod } e^{\wedge}2$

(is ?P l \wedge ?Q l)

$\langle \text{proof} \rangle$

lemma *divisibility-congruence*:

assumes $m = k * l$ **and** $b > 2\ m > 0$

shows $\alpha\ b\ m \text{ mod } (\alpha\ b\ k)^{\wedge}2 = ((-1)^{\wedge}(l-1) * l * (\alpha\ b\ k) * (\alpha\ b\ (k-1))^{\wedge}(l-1)) \text{ mod } (\alpha\ b\ k)^{\wedge}2$

$\langle \text{proof} \rangle$

Main result section 3.5

theorem *divisibility-alpha2*:

assumes $b > 2\ m > 0$

shows $(\alpha\ b\ k)^{\wedge}2 \text{ dvd } (\alpha\ b\ m) \longleftrightarrow k * (\alpha\ b\ k) \text{ dvd } m$ **(is ?P \longleftrightarrow ?Q)**

$\langle \text{proof} \rangle$

2.1.8 Congruence properties

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

fun *A-inv* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{mat2}$ **where**

$A\text{-inv } b\ n = \text{mat } (-\alpha\ b\ (n-1))\ (\alpha\ b\ n)\ (-\alpha\ b\ n)\ (\alpha\ b\ (n+1))$

fun *B-inv* :: $\text{nat} \Rightarrow \text{mat2}$ **where**

$B\text{-inv } b = \text{mat } 0\ 1\ (-1)\ b$

lemma *A-inv-aux*: $b > 2 \implies n > 0 \implies \alpha\ b\ n * \alpha\ b\ n - \alpha\ b\ (\text{Suc } n) * \alpha\ b\ (n - \text{Suc } 0) = 1$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

Equation 3.33

lemma congruence:
assumes $b1 \text{ mod } q = b2 \text{ mod } q$
shows $\alpha \ b1 \ n \ \text{mod } q = \alpha \ b2 \ n \ \text{mod } q$
 $\langle \text{proof} \rangle$

Equation 3.34

lemma congruence2:
fixes $b1 :: \text{nat}$
assumes $b \geq 2$
shows $(\alpha \ b \ n) \ \text{mod } (b - 2) = n \ \text{mod } (b - 2)$
 $\langle \text{proof} \rangle$

lemma congruence-jpos:
fixes $b \ m \ j \ l :: \text{nat}$
assumes $b > 2$ **and** $2 * l * m + j > 0$
defines $n \equiv 2 * l * m + j$
shows $A \ b \ n = \text{mat-mul } (\text{mat-pow } l \ (\text{mat-pow } 2 \ (A \ b \ m))) \ (A \ b \ j)$
 $\langle \text{proof} \rangle$

lemma congruence-inverse: $b > 2 \implies \text{mat-pow } (n+1) \ (B\text{-inv } b) = A\text{-inv } b \ (n+1)$
 $\langle \text{proof} \rangle$

lemma congruence-inverse2:
fixes $n \ b :: \text{nat}$
assumes $b > 2$
shows $\text{mat-mul } (\text{mat-pow } n \ (B \ b)) \ (\text{mat-pow } n \ (B\text{-inv } b)) = \text{mat } 1 \ 0 \ 0 \ 1$
 $\langle \text{proof} \rangle$

lemma congruence-mult:
fixes $m :: \text{nat}$
assumes $b > 2$
shows $n > m \implies \text{mat-pow } (\text{nat}(\text{int } n - \text{int } m)) \ (B \ b) = \text{mat-mul } (\text{mat-pow } n \ (B \ b)) \ (\text{mat-pow } m \ (B\text{-inv } b))$
 $\langle \text{proof} \rangle$

lemma congruence-jneg:
fixes $b \ m \ j \ l :: \text{nat}$
assumes $b > 2$ **and** $2 * l * m > j$ **and** $j \geq 1$
defines $n \equiv \text{nat}(\text{int } 2 * l * m - \text{int } j)$
shows $A \ b \ n = \text{mat-mul } (\text{mat-pow } l \ (\text{mat-pow } 2 \ (A \ b \ m))) \ (A\text{-inv } b \ j)$
 $\langle \text{proof} \rangle$

lemma matrix-congruence:
fixes $Y \ Z :: \text{mat } 2$
fixes $b \ m \ j \ l :: \text{nat}$
assumes $b > 2$
defines $X \equiv \text{mat-mul } Y \ Z$
defines $a \equiv \text{mat-11 } Y$ **and** $b0 \equiv \text{mat-12 } Y$ **and** $c \equiv \text{mat-21 } Y$ **and** $d \equiv \text{mat-22}$

Y

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

3.38

lemma congruence-Abm:

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

3.36 requires two lemmas 361 and 362

lemma 361:

fixes $b m j l :: \text{nat}$
assumes $b > 2$
defines $n \equiv 2*l*m + j$
defines $v \equiv \alpha b (m+1) - \alpha b (m-1)$
shows $(\alpha b n) \text{ mod } v = ((-1)^\wedge l * \alpha b j) \text{ mod } v$
(proof)

lemma 362:

fixes $b m j l :: \text{nat}$
assumes $b > 2$ **and** $2*l*m > j$ **and** $j \geq 1$
defines $n \equiv 2*l*m - j$
defines $v \equiv \alpha b (m+1) - \alpha b (m-1)$
shows $(\alpha b n) \text{ mod } v = -((-1)^\wedge l * \alpha b j) \text{ mod } v$
(proof)

Equation 3.36

lemma 36:

fixes $b m j l :: \text{nat}$
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 $(\alpha b n) \text{ mod } v = \alpha b j \text{ mod } v \vee (\alpha b n) \text{ mod } v = -\alpha b j \text{ mod } v$ (proof)

2.1.9 Diophantine definition of a sequence alpha

definition alpha-equations $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

$\Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow$

bool where

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

The sufficiency

lemma *alpha-equiv-suff*:

fixes $a\ b\ c :: \text{nat}$

assumes $\exists r\ s\ t\ u\ v\ w\ x\ y. \text{ alpha-equations } a\ b\ c\ r\ s\ t\ u\ v\ w\ x\ y$

shows $3 < b \wedge \text{int } a = (\alpha\ b\ c)$

<proof>

3.7.2 The necessity

lemma *add-mod*:

fixes $p\ q :: \text{int}$

assumes $p \text{ mod } 2 = 0\ q \text{ mod } 2 = 0$

shows $(p+q) \text{ mod } 2 = 0 \wedge (p-q) \text{ mod } 2 = 0$

<proof>

lemma *one-odd*:

fixes $b\ n :: \text{nat}$

assumes $b > 2$

shows $(\alpha\ b\ n) \text{ mod } 2 = 1 \vee (\alpha\ b\ (n+1)) \text{ mod } 2 = 1$

<proof>

lemma *oneodd*:

fixes $b\ n :: \text{nat}$

assumes $b > 2$

shows $\text{odd } (\alpha\ b\ n) = \text{True} \vee \text{odd } (\alpha\ b\ (n+1)) = \text{True}$

<proof>

lemma *cong-solve-nat*: $a \neq 0 \implies \exists x. (a*x) \text{ mod } n = (\text{gcd } a\ n) \text{ mod } n$

for $a\ n :: \text{nat}$

<proof>

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

<proof>

lemma *chinese-remainder-aux-nat*:

fixes $m1\ m2 :: nat$

assumes $a:\text{coprime } m1\ m2$

shows $\exists b1\ b2. b1 \bmod m1 = 1 \bmod m1 \wedge b1 \bmod m2 = 0 \bmod m2 \wedge b2 \bmod m1 = 0 \bmod m1 \wedge b2 \bmod m2 = 1 \bmod m2$

<proof>

lemma *cong-scalar2-nat*: $a \bmod m = b \bmod m \implies (k*a) \bmod m = (k*b) \bmod m$

for $a\ b\ k :: nat$

<proof>

lemma *chinese-remainder-nat*:

fixes $m1\ m2 :: nat$

assumes $a:\text{coprime } m1\ m2$

shows $\exists x. x \bmod m1 = u1 \bmod m1 \wedge x \bmod m2 = u2 \bmod m2$

<proof>

lemma *nat-int1*: $\forall (w::nat) (u::int). u > 0 \implies (w \bmod nat\ u = 2 \bmod nat\ u \implies int\ w \bmod u = 2 \bmod u)$

<proof>

lemma *nat-int2*: $\forall (w::nat) (b::nat) (v::int). u > 0 \implies (w \bmod nat\ v = b \bmod nat\ v \implies int\ w \bmod v = int\ b \bmod v)$

<proof>

lemma *lem*:

fixes $u\ t::int$ **and** $b::nat$

assumes $u^2 - int\ b * u * t + t^2 = 1\ u \geq 0\ t \geq 0$

shows $(nat\ u)^2 + (nat\ t)^2 = 1 + b * (nat\ u) * (nat\ t)$

<proof>

The necessity

lemma *alpha-equiv-nec*:

$b > 3 \wedge a = \alpha\ b\ c \implies \exists r\ s\ t\ u\ v\ w\ x\ y. \text{alpha-equations } a\ b\ c\ r\ s\ t\ u\ v\ w\ x\ y$

<proof>

2.1.10 Exponentiation is Diophantine

Equations 3.80-3.83

lemma 86:

fixes $b\ r$ **and** $q::int$

defines $m \equiv b * q - q * q - 1$

shows $(q * \alpha\ b\ (r + 1) - \alpha\ b\ r) \bmod m = (q \wedge (r + 1)) \bmod m$

<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  $(q * \alpha\ b\ r - (int\ b * \alpha\ b\ r - \alpha\ b\ (Suc\ r)))\ mod\ m = (q \wedge r)\ mod\ m$ 
<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  $int\ q \wedge r < m$  and  $q > 0$ 
shows  $int\ p = int\ q \wedge r \longleftrightarrow int\ p < m \wedge (q * \alpha\ b\ r - (int\ b * \alpha\ b\ r - \alpha\ b\ (Suc\ r)))\ mod\ m = int\ p\ mod\ m$ 
<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  $int\ q \wedge r < m$ 
<proof>
end

```

The final equivalence

theorem exp-alpha:

```

fixes  $p\ q\ r::nat$ 
shows  $p = q \wedge r \longleftrightarrow ((q = 0 \wedge r = 0 \wedge p = 1) \vee$ 
 $(q = 0 \wedge 0 < r \wedge p = 0) \vee$ 
 $(q > 0 \wedge (\exists b\ m.$ 
 $b = Exp-Matrices.\alpha\ (q + 4)\ (r + 1) + q * q + 2 \wedge$ 
 $m = b * q - q * q - 1 \wedge$ 
 $p < m \wedge$ 
 $p\ mod\ m = ((q * Exp-Matrices.\alpha\ b\ r) - (int\ b * Exp-Matrices.\alpha$ 
 $b\ r - Exp-Matrices.\alpha\ b\ (r + 1)))\ mod\ m))$ 
<proof>

```

lemma alpha-equivalence:

```

fixes  $a\ b\ c$ 
shows  $\exists < b \wedge int\ a = Exp-Matrices.\alpha\ b\ c \longleftrightarrow (\exists r\ s\ t\ u\ v\ w\ x\ y. Exp-Matrices.alpha-equations$ 
 $a\ b\ c\ r\ s\ t\ u\ v\ w\ x\ y)$ 
<proof>

```

end

2.2 Diophantine description of alpha function

theory Alpha-Sequence

```

imports Modulo-Divisibility Exponentiation
begin

```

The alpha function is diophantine

definition *alpha* ($[- = \alpha -]$ 1000)

where $[X = \alpha B N] \equiv (TERNARY (\lambda b n x. b > 3 \wedge x = Exp-Matrices.\alpha b n) B N X)$

lemma *alpha-dioph*[*dioph*]:

fixes $B N X$

defines $D \equiv [X = \alpha B N]$

shows *is-dioph-rel* D

$\langle proof \rangle$

declare *alpha-def*[*defs*]

end

2.3 Exponentiation is a Diophantine Relation

theory *Exponential-Relation*

imports *Alpha-Sequence Exponentiation*

begin

definition *exp-equations* $:: nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**

exp-equations $p q r b m = (b = Exp-Matrices.\alpha (q + 4) (r + 1) + q * q + 2 \wedge$
 $m + q^2 + 1 = b * q \wedge$
 $p < m \wedge$

$(p + b * Exp-Matrices.\alpha b r) \bmod m = (q * Exp-Matrices.\alpha b (r + 1)) \bmod m)$

lemma *exp-repr*:

fixes $p q r :: nat$

shows $p = q^r \iff ((q = 0 \wedge r = 0 \wedge p = 1) \vee$

$(q = 0 \wedge 0 < r \wedge p = 0) \vee$

$(q > 0 \wedge (\exists b m :: nat. exp-equations p q r b m)))$ (**is** $?P$)

$\iff ?Q$)

$\langle proof \rangle$

definition *exp* ($[- = - ^ -]$ 1000)

where $[Q = R ^ S] \equiv (TERNARY (\lambda a b c. a = b ^ c) Q R S)$

lemma *exp-dioph*[*dioph*]:

fixes $P Q R :: polynomial$

defines $D \equiv [P = Q ^ R]$

shows *is-dioph-rel* D

$\langle proof \rangle$

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*] \equiv (*QUATERNARY* ($\lambda d a k b. b > 1$
 $\wedge d = \text{nth-digit } a k b$) *D AA K BASE*)

lemma *mod-dioph2*[*dioph*]:

fixes *A B C*

defines *D* \equiv (*MOD A B C*)

shows *is-dioph-rel D*

<proof>

lemma *digit-dioph*[*dioph*]:

fixes *D A B K* :: *polynomial*

defines *DR* \equiv [*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 $c = a \text{ choose } b \iff (\exists u. (u = 2^{\wedge}(\text{Suc } a) \wedge c = \text{nth-digit } ((u+1)^{\wedge} a) b u))$

<proof>

definition *binomial-coefficient* ([*- = - choose -*] 1000)

where [*A = B choose C*] \equiv (*TERNARY* ($\lambda a b c. a = b \text{ choose } c$) *A B C*)

lemma *binomial-coefficient-dioph*[*dioph*]:

fixes *A B C* :: *polynomial*

defines *DR* \equiv [*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 $odd\ a \longleftrightarrow (\exists x::nat. a = 2*x + 1)$

<proof>

definition *odd-lift* (*ODD* - [999] 1000)

where $ODD\ A \equiv (UNARY\ (odd)\ A)$

lemma *odd-dioph[dioph]*:

fixes A

defines $DR \equiv (ODD\ A)$

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 \perp b) \longleftrightarrow odd ((a + b) choose b) (is ?P \longleftrightarrow*

?Q)

<proof>

definition *orthogonal* (**infix** [\perp] 50)

where $P\ [\perp]\ Q \equiv (BINARY\ (\lambda a\ b. a\ \perp\ b)\ P\ Q)$

lemma *orthogonal-dioph[dioph]*:

fixes $P\ Q$

defines $DR \equiv (P\ [\perp]\ Q)$

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*: $(b \preceq c) \longleftrightarrow \text{odd } (c \text{ choose } b)$ (**is** $?P \longleftrightarrow ?Q$)
<proof>

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

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 $\text{bin-carry } a \ b \ k = \text{bin-carry } a \ b \ k \ \text{mod } 2$
<proof>

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

lemma *auxm1-lm0246*: $a \preceq (a+b) \implies (\forall n. \text{nth-bit } a \ n + \text{nth-bit } b \ n \leq 1)$ (**is** $?P \implies ?Q$)
<proof>

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

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

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

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

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

definition *binary-and* ($[- = - \&\& -]$ 1000)
 where $[A = B \&\& C] \equiv (\text{TERNARY } (\lambda a b c. a = b \&\& c) A B C)$

lemma *binary-and-dioph*[*dioph*]:
 fixes $A B C :: \text{polynomial}$
 defines $DR \equiv [A = B \&\& C]$
 shows *is-dioph-rel* DR
 ⟨proof⟩

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

definition *binary-and-attempt* :: $\text{polynomial} \Rightarrow \text{polynomial} \Rightarrow \text{polynomial} (- \&? -)$
 where
 $A \&? B \equiv \text{Const } 0$

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* \Rightarrow *program* \Rightarrow *state* \Rightarrow *nat*
where *read t p s* = *t ! (modifies (p!s))*

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

definition *update* :: *tape* \Rightarrow *instruction* \Rightarrow *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* \Rightarrow *program* \Rightarrow *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* \Rightarrow *program* \Rightarrow *nat* \Rightarrow *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* \Rightarrow *instruction* \Rightarrow *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*: $\text{steps } (\text{step } c \ p) \ p \ t = \text{step } (\text{steps } c \ p \ t) \ p$
 $\langle \text{proof} \rangle$

lemma *step-fetch-correct*:

fixes $t :: \text{nat}$

and $c :: \text{configuration}$

and $p :: \text{program}$

assumes *is-valid* $c \ p$

defines $ct \equiv (\text{steps } c \ p \ t)$

shows $\text{fst } (\text{steps } (\text{step } c \ p) \ p \ t) = \text{fetch } (\text{fst } ct) \ p \ (\text{read } (\text{snd } ct) \ p \ (\text{fst } ct))$

$\langle \text{proof} \rangle$

3.2.1 From Configurations to a Protocol

Register Values

definition $R :: \text{configuration} \Rightarrow \text{program} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

where $R \ c \ p \ n \ t = (\text{snd } (\text{steps } c \ p \ t)) \ ! \ n$

fun $RL :: \text{configuration} \Rightarrow \text{program} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**

$RL \ c \ p \ b \ 0 \ l = ((\text{snd } c) \ ! \ l) \ |$

$RL \ c \ p \ b \ (\text{Suc } t) \ l = ((\text{snd } c) \ ! \ l) + b * (RL \ (\text{step } c \ p) \ p \ b \ t \ l)$

lemma *RL-simp-aux*:

$\langle \text{snd } c \ ! \ l + b * RL \ (\text{step } c \ p) \ p \ b \ t \ l =$

$RL \ c \ p \ b \ t \ l + b * (b \ ^ \ t * \text{snd } (\text{step } (\text{steps } c \ p \ t) \ p) \ ! \ l) \rangle$

$\langle \text{proof} \rangle$

declare $RL.\text{sims}[simp \ del]$

lemma *RL-simp*:

$RL \ c \ p \ b \ (\text{Suc } t) \ l = (\text{snd } (\text{steps } c \ p \ (\text{Suc } t)) \ ! \ l) * b \ ^ \ (\text{Suc } t) + (RL \ c \ p \ b \ t \ l)$

$\langle \text{proof} \rangle$

State Values

definition $S :: \text{configuration} \Rightarrow \text{program} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

where $S \ c \ p \ k \ t = (\text{if } (\text{fst } (\text{steps } c \ p \ t)) = k \ \text{then } (\text{Suc } 0) \ \text{else } 0)$

definition $S2 :: \text{configuration} \Rightarrow \text{nat} \Rightarrow \text{nat}$

where $S2 \ c \ k = (\text{if } (\text{fst } c) = k \ \text{then } 1 \ \text{else } 0)$

fun $SK :: \text{configuration} \Rightarrow \text{program} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

where $SK \ c \ p \ b \ 0 \ k = (S2 \ c \ k) \ |$

$SK \ c \ p \ b \ (\text{Suc } t) \ k = (S2 \ c \ k) + b * (SK \ (\text{step } c \ p) \ p \ b \ t \ k)$

lemma *SK-simp-aux*:

$\langle SK\ c\ p\ b\ (Suc\ (Suc\ t))\ k =$
 $S2\ (steps\ c\ p\ (Suc\ (Suc\ t)))\ k * b \wedge Suc\ (Suc\ t) + SK\ c\ p\ b\ (Suc\ t)\ k \rangle$
 $\langle proof \rangle$

declare *SK.simps*[*simp del*]

lemma *SK-simp*:

$SK\ c\ p\ b\ (Suc\ t)\ k = (S2\ (steps\ c\ p\ (Suc\ t))\ k) * b \wedge (Suc\ t) + (SK\ c\ p\ b\ t\ k)$
 $\langle proof \rangle$

Zero-Indicator Values

definition *Z* :: *configuration* \Rightarrow *program* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *nat* **where**

$Z\ c\ p\ n\ t = (if\ (R\ c\ p\ n\ t > 0)\ then\ 1\ else\ 0)$

definition *Z2* :: *configuration* \Rightarrow *nat* \Rightarrow *nat* **where**

$Z2\ c\ n = (if\ (snd\ c)!\ n > 0\ then\ 1\ else\ 0)$

fun *ZL* :: *configuration* \Rightarrow *program* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *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*:

$Z2\ c\ l + b * ZL\ (step\ c\ p)\ p\ b\ t\ l =$
 $ZL\ c\ p\ b\ t\ l + b * (b \wedge t * Z2\ (step\ (steps\ c\ p\ t))\ p)\ l)$
 $\langle proof \rangle$

declare *ZL.simps*[*simp del*]

lemma *ZL-simp*:

$ZL\ c\ p\ b\ (Suc\ t)\ l = (Z2\ (steps\ c\ p\ (Suc\ t))\ l) * b \wedge (Suc\ t) + (ZL\ c\ p\ b\ t\ l)$
 $\langle proof \rangle$

3.2.2 Protocol Properties

lemma *Z-bounded*: $Z\ c\ p\ l\ t \leq 1$

$\langle proof \rangle$

lemma *S-bounded*: $S\ c\ p\ k\ t \leq 1$

$\langle proof \rangle$

lemma *S-unique*: $\forall k \leq length\ p. (k \neq fst\ (steps\ c\ p\ t) \longrightarrow S\ c\ p\ k\ t = 0)$

$\langle proof \rangle$

fun *cells-bounded* :: *configuration* \Rightarrow *program* \Rightarrow *nat* \Rightarrow *bool* **where**

$cells-bounded\ conf\ p\ c = ((\forall l < (length\ (snd\ conf)). \forall t. 2 \wedge c > R\ conf\ p\ l\ t)$
 $\wedge (\forall k\ t. 2 \wedge c > S\ conf\ p\ k\ t)$
 $\wedge (\forall l\ t. 2 \wedge c > Z\ conf\ p\ l\ t))$

lemma *steps-tape-length-invar*: $\text{length} (\text{snd} (\text{steps } c \ p \ t)) = \text{length} (\text{snd } c)$
 <proof>

lemma *step-is-valid-invar*: $\text{is-valid } c \ p \implies \text{is-valid} (\text{step } c \ p) \ p$
 <proof>

fun *fetch-old*

where

(*fetch-old* $p \ s \ (\text{Add } r \ \text{next}) \ -$) = *next*
 | (*fetch-old* $p \ s \ (\text{Sub } r \ \text{next} \ \text{nextalt}) \ \text{val}$) = (if *val* = 0 then *nextalt* else *next*)
 | (*fetch-old* $p \ s \ \text{Halt } -$) = *s*

lemma *fetch-equiv*:

assumes $i = p!s$

shows $\text{fetch } s \ p \ v = \text{fetch-old } p \ s \ i \ v$

<proof>

lemma *p-contains*: $\text{is-valid-initial } ic \ p \ a \implies (\text{fst} (\text{steps } ic \ p \ t)) < \text{length } p$
 <proof>

lemma *steps-is-valid-invar*: $\text{is-valid } c \ p \implies \text{is-valid} (\text{steps } c \ p \ t) \ p$
 <proof>

lemma *terminates-halt-state*: $\text{terminates } ic \ p \ q \implies \text{is-valid-initial } ic \ p \ a$
 $\implies \text{ishalt } (p \ ! \ (\text{fst} (\text{steps } ic \ p \ q)))$

<proof>

lemma *R-termination*:

fixes $l :: \text{register}$ **and** $ic :: \text{configuration}$

assumes *is-val*: $\text{is-valid } ic \ p$ **and** *terminate*: $\text{terminates } ic \ p \ q$ **and** $l: l < \text{length} (\text{snd } ic)$

shows $\forall t \geq q. R \ ic \ p \ l \ t = 0$

<proof>

lemma *terminate-c-exists*: $\text{is-valid } ic \ p \implies \text{terminates } ic \ p \ q \implies \exists c > 1. \text{cells-bounded } ic \ p \ c$

<proof>

end

3.3 Simulation of a Register Machine

theory *RegisterMachineSimulation*

imports *RegisterMachineProperties Digit-Expansions.Binary-Operations*

begin

definition $B :: \text{nat} \Rightarrow \text{nat}$ **where**

$(B \ c) = \mathcal{L}^\sim(\text{Suc } c)$

definition $RLe\ c\ p\ b\ q\ l = (\sum t = 0..q. b^{\wedge}t * R\ c\ p\ l\ t)$

definition $SKe\ c\ p\ b\ q\ k = (\sum t = 0..q. b^{\wedge}t * S\ c\ p\ k\ t)$

definition $ZLe\ c\ p\ b\ q\ l = (\sum t = 0..q. b^{\wedge}t * Z\ c\ p\ l\ t)$

fun $sum-radd :: program \Rightarrow register \Rightarrow (nat \Rightarrow nat) \Rightarrow nat$

where $sum-radd\ p\ l\ f = (\sum k = 0..length\ p-1. if\ isadd\ (p!k) \wedge l = modifies\ (p!k)$
 $then\ f\ k\ else\ 0)$

abbreviation $sum-radd-abbrev\ (\sum R+ \dots [999, 999, 999] 1000)$

where $(\sum R+ p\ l\ f) \equiv (sum-radd\ p\ l\ f)$

fun $sum-rsub :: program \Rightarrow register \Rightarrow (nat \Rightarrow nat) \Rightarrow nat$

where $sum-rsub\ p\ l\ f = (\sum k = 0..length\ p-1. if\ issub\ (p!k) \wedge l = modifies\ (p!k)$
 $then\ f\ k\ else\ 0)$

abbreviation $sum-rsub-abbrev\ (\sum R- \dots [999, 999, 999] 1000)$

where $(\sum R- p\ l\ f) \equiv (sum-rsub\ p\ l\ f)$

fun $sum-sadd :: program \Rightarrow state \Rightarrow (nat \Rightarrow nat) \Rightarrow nat$

where $sum-sadd\ p\ d\ f = (\sum k = 0..length\ p-1. if\ isadd\ (p!k) \wedge d = goes-to\ (p!k)$
 $then\ f\ k\ else\ 0)$

abbreviation $sum-sadd-abbrev\ (\sum S+ \dots [999, 999, 999] 1000)$

where $(\sum S+ p\ d\ f) \equiv (sum-sadd\ p\ d\ f)$

fun $sum-ssub-nzero :: program \Rightarrow state \Rightarrow (nat \Rightarrow nat) \Rightarrow nat$

where $sum-ssub-nzero\ p\ d\ f = (\sum k = 0..length\ p-1. if\ issub\ (p!k) \wedge d = goes-to$
 $(p!k)\ then\ f\ k\ else\ 0)$

abbreviation $sum-ssub-nzero-abbrev\ (\sum S- \dots [999, 999, 999] 1000)$

where $(\sum S- p\ d\ f) \equiv (sum-ssub-nzero\ p\ d\ f)$

fun $sum-ssub-zero :: program \Rightarrow state \Rightarrow (nat \Rightarrow nat) \Rightarrow nat$

where $sum-ssub-zero\ p\ d\ f = (\sum k = 0..length\ p-1. if\ issub\ (p!k) \wedge d = goes-to-alt$
 $(p!k)\ then\ f\ k\ else\ 0)$

abbreviation $sum-ssub-zero-abbrev\ (\sum S0 \dots [999, 999, 999] 1000)$

where $(\sum S0 p\ d\ f) \equiv (sum-ssub-zero\ p\ d\ f)$

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 $\forall k. k \leq \text{length } p - 1 \wedge \text{isadd } (p!k) \wedge l = \text{goes-to } (p!k) \longrightarrow f k = g k$

shows $\sum S+ p l f = \sum S+ p l g$

<proof>

lemma *sum-ssub-nzero-cong*:

assumes $\forall k. k \leq \text{length } p - 1 \wedge \text{issub } (p!k) \wedge l = \text{goes-to } (p!k) \longrightarrow f k = g k$

shows $\sum S- p l f = \sum S- p l g$

<proof>

lemma *sum-ssub-zero-cong*:

assumes $\forall k. k \leq \text{length } p - 1 \wedge \text{issub } (p!k) \wedge l = \text{goes-to-alt } (p!k) \longrightarrow f k = g k$

shows $\sum S0 p l f = \sum S0 p l g$

<proof>

lemma *sum-radd-cong*:

assumes $\forall k. k \leq \text{length } p - 1 \wedge \text{isadd } (p!k) \wedge l = \text{modifies } (p!k) \longrightarrow f k = g k$

shows $\sum R+ p l f = \sum R+ p l g$

<proof>

lemma *sum-rsub-cong*:

assumes $\forall k. k \leq \text{length } p - 1 \wedge \text{issub } (p!k) \wedge l = \text{modifies } (p!k) \longrightarrow f k = g k$

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 $SX :: program \Rightarrow nat \Rightarrow (nat \Rightarrow nat) \Rightarrow nat$
and $p :: program$

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

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

lemma *sum-commutative*:

fixes $SX :: program \Rightarrow nat \Rightarrow (nat \Rightarrow nat) \Rightarrow nat$
and $p :: program$

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

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*: $c \leq a + b \implies \text{int}(a + b - c) = \text{int}(a) + \text{int}(b) - \text{int}(c)$
 ⟨*proof*⟩

lemma *ZLe-bounded*: $b > 2 \implies ZLe\ c\ p\ b\ q\ l < b \wedge (Suc\ q)$
 ⟨*proof*⟩

lemma *SKe-bounded*: $b > 2 \implies SKe\ c\ p\ b\ q\ k < b \wedge (Suc\ q)$
 ⟨*proof*⟩

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

shows $(\sum t=0..q. b^{\wedge}t * (Z ic p l t * S ic p k t))$
 $= ZLe ic p b q l \ \&\& \ SKe ic p b q k$
 $\langle proof \rangle$

lemma *sum-bt*:
fixes $b q :: nat$
assumes $b > 2$
shows $(\sum t = 0..q. b^{\wedge}t) < b^{\wedge}(Suc q)$
 $\langle proof \rangle$

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

shows $(\sum t=0..q. b^{\wedge}t * ((1 - Z ic p l t) * S ic p k t))$
 $= ((\sum t = 0..q. b^{\wedge}t) - ZLe ic p b q l) \ \&\& \ SKe ic p b q k$
 $\langle proof \rangle$

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 $l < length\ tape$

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

lemma *single-step-sub*:

```

fixes  $c :: \text{configuration}$ 
and  $p :: \text{program}$ 
and  $l :: \text{register}$ 
and  $t a :: \text{nat}$ 

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

assumes  $\text{is-val: is-valid-initial } c \ p \ a$ 

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

lemma  $\text{lm04-06-one-step-relation-register-old:}$ 
fixes  $l :: \text{register}$ 
and  $ic :: \text{configuration}$ 
and  $p :: \text{program}$ 

defines  $s \equiv \text{fst } ic$ 
and  $\text{tape} \equiv \text{snd } ic$ 

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

assumes  $\text{is-val: is-valid } ic \ p$ 
and  $l: \langle l < \text{length } \text{tape} \rangle$ 

shows  $(\text{tape}'!l) = (\text{tape}!l) + (\text{if isadd } (p!s) \wedge l = \text{modifies } (p!s) \text{ then } 1 \text{ else } 0)$ 
 $- Z \ ic \ p \ l \ 0 * (\text{if issub } (p!s) \wedge l = \text{modifies } (p!s) \text{ then } 1$ 
 $\text{else } 0)$ 
 $\langle \text{proof} \rangle$ 

lemma  $\text{lm04-06-one-step-relation-register:}$ 
fixes  $l :: \text{register}$ 
and  $c :: \text{configuration}$ 
and  $p :: \text{program}$ 
and  $t :: \text{nat}$ 
and  $a :: \text{nat}$ 

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

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

shows  $r \ l \ (Suc \ t) = r \ l \ t + (\sum R+ \ p \ l \ (\lambda k. s \ k \ t))$ 
 $- (\sum R- \ p \ l \ (\lambda k. (Z \ c \ p \ l \ t) * s \ k \ t))$ 
 $\langle \text{proof} \rangle$ 

```


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 $n == \text{length } (\text{snd } ic)$

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 $m == \text{length } p$
and $n == \text{length } (\text{snd } ic)$

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 $c :: \text{nat}$
 and $l :: \text{register}$
 and $ic :: \text{configuration}$
 and $p :: \text{program}$
 and $q :: \text{nat}$
 and $a :: \text{nat}$

defines $b == B\ c$
 and $m == \text{length}\ p$

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 $e \equiv \sum t = 0..q. b^{\wedge}t$

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

<proof>

lemma *lm04-25-multiple-step-state1:*

fixes $c :: \text{nat}$
 and $l :: \text{register}$
 and $ic :: \text{configuration}$
 and $p :: \text{program}$
 and $q :: \text{nat}$
 and $a :: \text{nat}$

defines $b == B\ c$

and $m == \text{length } p$

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

shows $s \ d = 1 + b * (\sum S+ \ p \ d \ s)$
 $+ b * (\sum S- \ p \ d \ (\lambda k. z \ (\text{modifies } (p!k)) \ \&\& \ s \ k))$
 $+ b * (\sum S0 \ p \ d \ (\lambda k. (e - z \ (\text{modifies } (p!k))) \ \&\& \ s \ k))$
<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 $m == \text{length } p - 1$

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

assumes *terminate*: *terminates ic p q*

shows $SKe \text{ ic } p \ b \ q \ m = b \ \hat{q}$
<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 $m == \text{length } p - 1$
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 $SKe\ ic\ p\ b\ q\ k < b \wedge q$
 $\langle \text{proof} \rangle$

end

3.6 Masking properties

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

definition $E :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $(E\ q\ b) = (\sum t = 0..q. b \wedge t)$

lemma *e-geom-series:*
assumes $b \geq 2$
shows $(E\ q\ b = e) \longleftrightarrow ((b-1) * e = b \wedge (\text{Suc } q) - 1)$ (**is** $?P \longleftrightarrow ?Q$)
 $\langle \text{proof} \rangle$

definition $D :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $(D\ q\ c\ b) = (\sum t = 0..q. (2 \wedge c - 1) * b \wedge t)$

lemma *d-geom-series:*
assumes $b = 2 \wedge (\text{Suc } c)$
shows $(D\ q\ c\ b = d) \longleftrightarrow ((b-1) * d = (2 \wedge c - 1) * (b \wedge (\text{Suc } q) - 1))$ (**is** $?P \longleftrightarrow ?Q$)
 $\langle \text{proof} \rangle$

definition $F :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $(F\ q\ c\ b) = (\sum t = 0..q. 2 \wedge c * b \wedge t)$

lemma *f-geom-series:*
assumes $b = 2 \wedge (\text{Suc } c)$
shows $(F\ q\ c\ b = f) \longleftrightarrow ((b-1) * f = 2 \wedge c * (b \wedge (\text{Suc } q) - 1))$
 $\langle \text{proof} \rangle$

lemma *aux-lt-implies-mask*:

assumes $a < 2^k$

shows $\forall r \geq k. a \text{ j } r = 0$

<proof>

lemma *lt-implies-mask*:

fixes $a b :: \text{nat}$

assumes $\exists k. a < 2^k \wedge (\forall r < k. \text{nth-bit } b \ r = 1)$

shows $a \preceq b$

<proof>

lemma *mask-conversed-shift*:

fixes $a b k :: \text{nat}$

assumes *asm*: $a \preceq b$

shows $a * 2^k \preceq b * 2^k$

<proof>

lemma *base-summation-bound*:

fixes $c q :: \text{nat}$

and $f :: (\text{nat} \Rightarrow \text{nat})$

defines $b: b \equiv B \ c$

assumes *bound*: $\forall t. f \ t < 2^{\text{Suc } c} - (1 :: \text{nat})$

shows $(\sum t = 0..q. f \ t * b^t) < b^{\text{Suc } q}$

<proof>

lemma *mask-conserved-sum*:

fixes $y c q :: \text{nat}$

and $x :: (\text{nat} \Rightarrow \text{nat})$

defines $b: b \equiv B \ c$

assumes *mask*: $\forall t. x \ t \preceq y$

assumes *xlt*: $\forall t. x \ t \leq 2^{\text{Suc } c} - \text{Suc } 0$

assumes *ygt*: $y \leq 2^{\text{Suc } c} - \text{Suc } 0$

shows $(\sum t = 0..q. x \ t * b^t) \preceq (\sum t = 0..q. y * b^t)$

<proof>

lemma *aux-powertwo-digits*:

fixes $k c :: \text{nat}$

assumes $k < c$

shows $\text{nth-bit } (2^c) \ k = 0$

<proof>

lemma *obtain-digit-rep*:

fixes $x c :: \text{nat}$

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

<proof>

lemma *nth-digit-bitAND-equiv:*

fixes $x\ c :: \text{nat}$

shows $2^{\wedge}c * \text{nth-bit } x\ c = (x \ \&\& \ 2^{\wedge}c)$

<proof>

lemma *bitAND-single-digit:*

fixes $x\ c :: \text{nat}$

assumes $2^{\wedge}c \leq x$

assumes $x < 2^{\wedge} \text{Suc } c$

shows $\text{nth-bit } x\ c = 1$

<proof>

lemma *aux-bitAND-distrib:* $2 * (a \ \&\& \ b) = (2 * a) \ \&\& \ (2 * b)$

<proof>

lemma *bitAND-distrib:* $2^{\wedge}c * (a \ \&\& \ b) = (2^{\wedge}c * a) \ \&\& \ (2^{\wedge}c * b)$

<proof>

lemma *bitAND-linear-sum:*

fixes $x\ y :: \text{nat} \Rightarrow \text{nat}$

and $c :: \text{nat}$

and $q :: \text{nat}$

defines $b: b == 2^{\wedge} \text{Suc } c$

assumes $xb: \forall t. x\ t < 2^{\wedge} \text{Suc } c - 1$

assumes $yb: \forall t. y\ t < 2^{\wedge} \text{Suc } c - 1$

shows $(\sum t = 0..q. (x\ t \ \&\& \ y\ t) * b^{\wedge}t) =$

$(\sum t = 0..q. x\ t * b^{\wedge}t) \ \&\& \ (\sum t = 0..q. y\ t * b^{\wedge}t)$

<proof>

lemma *dmask-aux0:*

fixes $x :: \text{nat}$

assumes $x > 0$

shows $(2^{\wedge}x - \text{Suc } 0) \ \text{div } 2 = 2^{\wedge}(x - 1) - \text{Suc } 0$

<proof>

lemma *dmask-aux:*

fixes $c :: \text{nat}$

shows $d < c \implies (2^{\wedge}c - \text{Suc } 0) \ \text{div } 2^{\wedge}d = 2^{\wedge}(c - d) - \text{Suc } 0$

<proof>

lemma *register-cells-masked:*

```

fixes  $l :: \text{register}$ 
and  $t :: \text{nat}$ 
and  $ic :: \text{configuration}$ 
and  $p :: \text{program}$ 

assumes cells-bounded: cells-bounded ic p c
assumes  $l: l < \text{length} (\text{snd } ic)$ 

shows  $R \text{ ic } p \ l \ t \preceq 2^{\hat{c}} - 1$ 
 $\langle \text{proof} \rangle$ 

lemma lm04-15-register-masking:
fixes  $c :: \text{nat}$ 
and  $l :: \text{register}$ 
and  $ic :: \text{configuration}$ 
and  $p :: \text{program}$ 
and  $q :: \text{nat}$ 

defines  $b == B \ c$ 
defines  $d == D \ q \ c \ b$ 

assumes cells-bounded: cells-bounded ic p c
assumes  $l: l < \text{length} (\text{snd } ic)$ 

defines  $r == RLe \ ic \ p \ b \ q$ 

shows  $r \ l \preceq d$ 
 $\langle \text{proof} \rangle$ 

lemma zero-cells-masked:
fixes  $l :: \text{register}$ 
and  $t :: \text{nat}$ 
and  $ic :: \text{configuration}$ 
and  $p :: \text{program}$ 

assumes  $l: l < \text{length} (\text{snd } ic)$ 

shows  $Z \text{ ic } p \ l \ t \preceq 1$ 
 $\langle \text{proof} \rangle$ 

lemma lm04-15-zero-masking:
fixes  $c :: \text{nat}$ 
and  $l :: \text{register}$ 
and  $ic :: \text{configuration}$ 
and  $p :: \text{program}$ 
and  $q :: \text{nat}$ 

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 2c * z l t = (r l t + 2c - 1) && 2c
⟨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 2c * z l = (r l + d) && f
⟨proof⟩

```

```

lemma state-mask:
fixes  $c :: \text{nat}$ 
  and  $l :: \text{register}$ 
  and  $ic :: \text{configuration}$ 
  and  $p :: \text{program}$ 
  and  $q :: \text{nat}$ 
  and  $a :: \text{nat}$ 

defines  $b \equiv B\ c$ 
  and  $m \equiv \text{length } p - 1$ 

defines  $e \equiv E\ q\ b$ 

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

assumes terminate: terminates  $ic\ p\ q$ 
shows  $SKe\ ic\ p\ b\ q\ k \preceq e$ 
 $\langle \text{proof} \rangle$ 

lemma state-sum-mask:
fixes  $c :: \text{nat}$ 
  and  $l :: \text{register}$ 
  and  $ic :: \text{configuration}$ 
  and  $p :: \text{program}$ 
  and  $q :: \text{nat}$ 
  and  $a :: \text{nat}$ 

defines  $b \equiv B\ c$ 
  and  $m \equiv \text{length } p - 1$ 

defines  $e \equiv E\ q\ b$ 

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

assumes  $M \leq m$ 

assumes terminate: terminates  $ic\ p\ q$ 
shows  $(\sum k \leq M. SKe\ ic\ p\ b\ q\ k) \preceq e$ 
 $\langle \text{proof} \rangle$ 

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* :: $nat \Rightarrow (register \Rightarrow nat) \Rightarrow (register \Rightarrow nat) \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$

where (*mask-equations* $n\ r\ z\ c\ d\ e\ f$) = $((\forall l < n. (r\ l) \preceq d) \wedge (\forall l < n. (z\ l) \preceq e) \wedge (\forall l < n. 2^{\wedge c} * (z\ l) = (r\ l + d) \ \&\& \ f))$

definition *reg-equations* :: $program \Rightarrow (register \Rightarrow nat) \Rightarrow (register \Rightarrow nat) \Rightarrow (state \Rightarrow nat) \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**

(*reg-equations* $p\ r\ z\ s\ b\ a\ n\ q$) = (
 — 4.22 $(\forall l > 0. l < n \longrightarrow r\ l = b * r\ l + b * \sum R+ p\ l (\lambda k. s\ k) - b * \sum R- p\ l (\lambda k. s\ k \ \&\& \ z\ l))$
 \wedge — 4.23 $(r\ 0 = a + b * r\ 0 + b * \sum R+ p\ 0 (\lambda k. s\ k) - b * \sum R- p\ 0 (\lambda k. s\ k \ \&\& \ z\ 0))$
 $\wedge (\forall l < n. r\ l < b \wedge q)$ — Extra equation not in Matiyasevich's book. Needed to show that all registers are empty at time q

definition *state-equations* :: $program \Rightarrow (state \Rightarrow nat) \Rightarrow (register \Rightarrow nat) \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**

(*state-equations* $p\ s\ z\ b\ e\ q\ m$) = (
 — 4.24 $(\forall d > 0. d \leq m \longrightarrow s\ d = b * \sum S+ p\ d (\lambda k. s\ k) + b * \sum S- p\ d (\lambda k. s\ k \ \&\& \ z\ (modifies\ (p!k))) + b * \sum SO\ p\ d (\lambda k. s\ k \ \&\& \ (e - z\ (modifies\ (p!k))))))$
 \wedge — 4.25 $(s\ 0 = 1 + b * \sum S+ p\ 0 (\lambda k. s\ k) + b * \sum S- p\ 0 (\lambda k. s\ k \ \&\& \ z\ (modifies\ (p!k))) + b * \sum SO\ p\ 0 (\lambda k. s\ k \ \&\& \ (e - z\ (modifies\ (p!k))))))$
 \wedge — 4.27 $(s\ m = b \wedge q)$
 $\wedge (\forall k \leq m. s\ k \preceq e) \wedge (\forall k < m. s\ k < b \wedge q)$ — these equations are not from the book
 $\wedge (\forall M \leq m. (\sum k \leq M. s\ k) \preceq e)$ — this equation is added, too)

definition *state-unique-equations* :: $program \Rightarrow (state \Rightarrow nat) \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**

state-unique-equations $p\ s\ m\ e = ((\sum k = 0..m. s\ k) \preceq e \wedge (\forall k \leq m. s\ k \preceq e))$

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[^]*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 *fct* :: *nat* ⇒ *nat*
shows $\forall i \leq N. \forall j \leq N. i \neq j \longrightarrow (\forall k. (fct\ i)\ i\ k * (fct\ j)\ i\ k = 0)$
 $\implies (\sum k \leq N. fct\ k \ \&\&\ C) = (\sum k \leq N. fct\ k) \ \&\&\ C$
 ⟨*proof*⟩

lemma *aux-commute-bitAND-sum-if*:

fixes *N const* :: *nat*
assumes *nocarry*: $\forall i \leq N. \forall j \leq N. i \neq j \longrightarrow (\forall k. (fct\ i)\ i\ k * (fct\ j)\ i\ k = 0)$
shows $(\sum k \leq N. \text{if cond } k \text{ then } fct\ k \ \&\&\ const \text{ else } 0)$
 $= (\sum k \leq N. \text{if cond } k \text{ then } fct\ k \text{ else } 0) \ \&\&\ const$
 ⟨*proof*⟩

lemma *mod-mod*:

fixes *x a b* :: *nat*
shows $x \bmod 2^{\wedge} a \bmod 2^{\wedge} b = x \bmod 2^{\wedge} (\min\ a\ b)$
 ⟨*proof*⟩

lemma *carry-gen-pow2-reduct*:

assumes $c > 0$
defines $b: b \equiv 2^{\wedge} (Suc\ c)$
assumes $nth_digit\ x\ (t-1)\ (2^{\wedge}Suc\ c)\ \vdash\ c = 0$
and $nth_digit\ y\ (t-1)\ (2^{\wedge}Suc\ c)\ \vdash\ c = 0$
shows $k \leq c \implies bin_carry\ (nth_digit\ x\ t\ b)\ (nth_digit\ y\ t\ b)\ k$
 $= bin_carry\ x\ y\ (Suc\ c * t + k)$
 $\langle proof \rangle$

lemma *nth-digit-bound*:
fixes c **defines** $b \equiv 2^{\wedge} (Suc\ c)$
shows $nth_digit\ x\ t\ b < 2^{\wedge} (Suc\ c)$
 $\langle proof \rangle$

lemma *digit-wise-block-additivity*:
fixes c
defines $b \equiv 2^{\wedge} Suc\ c$
assumes $nth_digit\ x\ (t-1)\ (2^{\wedge}Suc\ c)\ \vdash\ c = 0$
and $nth_digit\ y\ (t-1)\ (2^{\wedge}Suc\ c)\ \vdash\ c = 0$
and $k \leq c$
and $c > 0$
shows $nth_digit\ (x+y)\ t\ b\ \vdash\ k = (nth_digit\ x\ t\ b + nth_digit\ y\ t\ b)\ \vdash\ k$
 $\langle proof \rangle$

lemma *block-additivity*:
assumes $c > 0$
defines $b \equiv 2^{\wedge} Suc\ c$
assumes $nth_digit\ x\ (t-1)\ b\ \vdash\ c = 0$
and $nth_digit\ y\ (t-1)\ b\ \vdash\ c = 0$
and $nth_digit\ x\ t\ b\ \vdash\ c = 0$
and $nth_digit\ y\ t\ b\ \vdash\ c = 0$

shows $nth_digit\ (x+y)\ t\ b = nth_digit\ x\ t\ b + nth_digit\ y\ t\ b$
 $\langle proof \rangle$

lemma *block-to-sum*:
assumes $c > 0$
defines $b: b \equiv 2^{\wedge} (Suc\ c)$
assumes *yltx-digits*: $\forall t'. nth_digit\ y\ t'\ b \leq nth_digit\ x\ t'\ b$
shows $y\ mod\ b^{\wedge}t \leq x\ mod\ b^{\wedge}t$
 $\langle proof \rangle$

lemma *narry-gen-pow2-reduct*:
assumes $c > 0$
defines $b: b \equiv 2^{\wedge} (Suc\ c)$
assumes *yltx-digits*: $\forall t'. nth_digit\ y\ t'\ b \leq nth_digit\ x\ t'\ b$
shows $k \leq c \implies bin_narry\ (nth_digit\ x\ t\ b)\ (nth_digit\ y\ t\ b)\ k$
 $= bin_narry\ x\ y\ (Suc\ c * t + k)$
 $\langle proof \rangle$

lemma *digit-wise-block-subtractivity*:

fixes c
defines $b \equiv 2^{\wedge} \text{Suc } c$
assumes *yltx-digits*: $\forall t'. \text{nth-digit } y \ t' \ b \leq \text{nth-digit } x \ t' \ b$
and $k \leq c$
and $c > 0$
shows $\text{nth-digit } (x - y) \ t \ b \ \text{i} \ k = (\text{nth-digit } x \ t \ b - \text{nth-digit } y \ t \ b) \ \text{i} \ k$
<proof>

lemma *block-subtractivity*:

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

lemma *bitAND-nth-digit-commute*:

assumes *b-def*: $b = 2^{\wedge}(\text{Suc } c)$
shows $\text{nth-digit } (x \ \&\& \ y) \ t \ b = \text{nth-digit } x \ t \ b \ \&\& \ \text{nth-digit } y \ t \ b$
<proof>

lemma *bx-aux*:

shows $b > 1 \implies \text{nth-digit } (b^{\wedge} x) \ t' \ b = (\text{if } x = t' \ \text{then } 1 \ \text{else } 0)$
<proof>

context

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

lemma *b-gt1*: $b > 1$ *<proof>*

Commutation relations with sums

lemma *finite-sum-nth-digit-commute*:

fixes $M :: \text{nat}$
shows $\forall t. \forall k \leq M. \text{nth-digit } (\text{fct } k) \ t \ b < 2^{\wedge} c \implies$
 $\forall t. (\sum i = 0..M. \text{nth-digit } (\text{fct } i) \ t \ b) < 2^{\wedge} c \implies$
 $\text{nth-digit } (\sum i = 0..M. \text{fct } i) \ t \ b = (\sum i = 0..M. (\text{nth-digit } (\text{fct } i) \ t \ b))$
<proof>

lemma *sum-nth-digit-commute-aux*:

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

<proof>

lemma *sum-nth-digit-commute*:

fixes g

defines *SX-def*: $SX \equiv \lambda p l (fct :: nat \Rightarrow nat). (\sum k = 0..length\ p - 1. \text{if } g\ l\ k \text{ then } fct\ k \text{ else } 0)$

assumes *nocarry*: $\forall t. \forall k \leq length\ p - 1. nth_digit\ (fct\ k)\ t\ b < 2^{\wedge}c$

and *nocarry-sum*: $\forall t. (SX\ p\ l\ (\lambda k. nth_digit\ (fct\ k)\ t\ b)) < 2^{\wedge}c$

shows $nth_digit\ (SX\ p\ l\ fct)\ t\ b = SX\ p\ l\ (\lambda k. nth_digit\ (fct\ k)\ t\ b)$

<proof>

Commute inside, need assumption for all partial sums

lemma *finite-sum-nth-digit-commute2*:

fixes $M :: nat$

shows $\forall t. \forall k \leq M. nth_digit\ (fct\ k)\ t\ b < 2^{\wedge}c \implies$

$\forall t. \forall m \leq M. nth_digit\ (\sum i=0..m. fct\ i)\ t\ b < 2^{\wedge}c \implies$

$nth_digit\ (\sum i=0..M. fct\ i)\ t\ b = (\sum i=0..M. (nth_digit\ (fct\ i)\ t\ b))$

<proof>

lemma *sum-nth-digit-commute-aux2*:

fixes g

defines *SX-def*: $SX \equiv \lambda l m (fct :: nat \Rightarrow nat). (\sum k = 0..m. \text{if } g\ l\ k \text{ then } fct\ k \text{ else } 0)$

assumes *nocarry*: $\forall t. \forall k \leq M. nth_digit\ (fct\ k)\ t\ b < 2^{\wedge}c$

and *nocarry-sum*: $\forall t. \forall m \leq M. nth_digit\ (SX\ l\ m\ fct)\ t\ b < 2^{\wedge}c$

shows $nth_digit\ (SX\ l\ M\ fct)\ t\ b = SX\ l\ M\ (\lambda k. nth_digit\ (fct\ k)\ t\ b)$

<proof>

lemma *sum-nth-digit-commute2*:

fixes $g\ p$

defines *SX-def*: $SX \equiv \lambda p l (fct :: nat \Rightarrow nat). (\sum k = 0..length\ p - 1. \text{if } g\ l\ k \text{ then } fct\ k \text{ else } 0)$

assumes *nocarry*: $\forall t. \forall k \leq length\ p - 1. nth_digit\ (fct\ k)\ t\ b < 2^{\wedge}c$

and *nocarry-sum*: $\forall t. \forall m \leq length\ p - 1. nth_digit\ (SX\ (take\ (Suc\ m)\ p)\ l\ fct)\ t\ b < 2^{\wedge}c$

shows $nth_digit\ (SX\ p\ l\ fct)\ t\ b = SX\ p\ l\ (\lambda k. nth_digit\ (fct\ k)\ t\ b)$

<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 \leftarrow 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 $r z :: register \Rightarrow nat$
and $s :: state \Rightarrow nat$
and $b c d e f :: nat$
and $m n :: nat$
and $Req Seq Zeq$

assumes $m-def: m \equiv length\ p - 1$
and $n-def: n \equiv length\ (snd\ ic)$

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 $Seq-def: Seq = (\lambda k\ t. nth-digit\ (s\ k)\ t\ b)$
and $Req-def: Req = (\lambda l\ t. nth-digit\ (r\ l)\ t\ b)$
and $Zeq-def: Zeq = (\lambda l\ t. nth-digit\ (z\ l)\ t\ b)$

begin

Basic properties

lemma $n-gt0: n > 0$
 $\langle proof \rangle$

lemma $f-def: f = (\sum\ t = 0..q. 2^{\wedge}c * b^{\wedge}t)$
 $\langle proof \rangle$

lemma $e-def: e = (\sum\ t = 0..q. b^{\wedge}t)$
 $\langle proof \rangle$

lemma $d-def: d = (\sum\ t = 0..q. (2^{\wedge}c - 1) * b^{\wedge}t)$
 $\langle proof \rangle$

lemma $b-def: b = 2^{\wedge}(Suc\ c)$
 $\langle proof \rangle$

lemma $b-gt1: b > 1 \langle proof \rangle$

lemma *c-gt0*: $c > 0$ \langle proof \rangle

lemma *h0*: $1 < (2::nat)^c$

\langle proof \rangle

lemma *rl-fst-digit-zero*:

assumes $l < n$

shows $\text{nth-digit } (r\ l)\ t\ b\ i\ c = 0$

\langle proof \rangle

lemma *e-mask-bound*:

assumes $x \preceq e$

shows $\text{nth-digit } x\ t\ b \leq 1$

\langle proof \rangle

lemma *sk-bound*:

shows $\forall t\ k. k \leq \text{length } p - 1 \longrightarrow \text{nth-digit } (s\ k)\ t\ b \leq 1$

\langle proof \rangle

lemma *sk-bitAND-bound*:

shows $\forall t\ k. k \leq \text{length } p - 1 \longrightarrow \text{nth-digit } (s\ k\ \&\&\ x\ k)\ t\ b \leq 1$

\langle proof \rangle

lemma *s-bound*:

shows $\forall j < m. s\ j < b \wedge q$

\langle proof \rangle

lemma *sk-sum-masked*:

shows $\forall M \leq m. (\sum k \leq M. s\ k) \preceq e$

\langle proof \rangle

lemma *sk-sum-bound*:

shows $\forall t\ M. M \leq \text{length } p - 1 \longrightarrow \text{nth-digit } (\sum k \leq M. s\ k)\ t\ b \leq 1$

\langle proof \rangle

lemma *sum-sk-bound*:

shows $(\sum k \leq \text{length } p - 1. \text{nth-digit } (s\ k)\ t\ b) \leq 1$

\langle proof \rangle

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

$\leq (\sum k \leq \text{length } p - 1. \text{Seq } k\ t)$

\langle proof \rangle

lemma *states-unique-RAW*:

$\forall k \leq m. \text{Seq } k\ t = 1 \longrightarrow (\forall j \leq m. j \neq k \longrightarrow \text{Seq } j\ t = 0)$

\langle proof \rangle

lemma *block-sum-radd-bound:*

shows $\forall t. (\sum R+ p l (\lambda k. nth-digit (s k) t b)) \leq 1$
<proof>

lemma *block-sum-rsub-bound:*

shows $\forall t. (\sum R- p l (\lambda k. nth-digit (s k \&\& z l) t b)) \leq 1$
<proof>

lemma *block-sum-rsub-special-bound:*

shows $\forall t. (\sum R- p l (\lambda k. nth-digit (s k) t b)) \leq 1$
<proof>

lemma *block-sum-sadd-bound:*

shows $\forall t. (\sum S+ p j (\lambda k. nth-digit (s k) t b)) \leq 1$
<proof>

lemma *block-sum-ssub-bound:*

shows $\forall t. (\sum S- p j (\lambda k. nth-digit (s k \&\& z (l k)) t b)) \leq 1$
<proof>

lemma *block-sum-szero-bound:*

shows $\forall t. (\sum S0 p j (\lambda k. nth-digit (s k \&\& (e - z (l k))) t b)) \leq 1$
<proof>

lemma *sum-radd-nth-digit-commute:*

shows $nth-digit (\sum R+ p l (\lambda k. s k)) t b = \sum R+ p l (\lambda k. nth-digit (s k) t b)$
<proof>

lemma *sum-rsub-nth-digit-commute:*

shows $nth-digit (\sum R- p l (\lambda k. s k \&\& z l)) t b$
 $= \sum R- p l (\lambda k. nth-digit (s k \&\& z l) t b)$
<proof>

lemma *sum-sadd-nth-digit-commute:*

shows $nth-digit (\sum S+ p j (\lambda k. s k)) t b = \sum S+ p j (\lambda k. nth-digit (s k) t b)$
<proof>

lemma *sum-ssub-nth-digit-commute:*

shows $nth-digit (\sum S- p j (\lambda k. s k \&\& z (l k))) t b$
 $= \sum S- p j (\lambda k. nth-digit (s k \&\& z (l k)) t b)$
<proof>

lemma *sum-szero-nth-digit-commute:*

shows $nth-digit (\sum S0 p j (\lambda k. s k \&\& (e - z (l k)))) t b$
 $= \sum S0 p j (\lambda k. nth-digit (s k \&\& (e - z (l k))) t b)$
<proof>

lemma *block-bound-impl-fst-digit-zero:*

assumes $nth-digit x t b \leq 1$

shows $(nth_digit\ x\ t\ b)\ i\ c = 0$
 $\langle proof \rangle$

lemma *sum-radd-block-bound*:
shows $nth_digit\ (\sum R+ p\ l\ (\lambda k. s\ k))\ t\ b \leq 1$
 $\langle proof \rangle$

lemma *sum-radd-fst-digit-zero*:
shows $(nth_digit\ (\sum R+ p\ l\ s)\ t\ b)\ i\ c = 0$
 $\langle proof \rangle$

lemma *sum-sadd-block-bound*:
shows $nth_digit\ (\sum S+ p\ j\ (\lambda k. s\ k))\ t\ b \leq 1$
 $\langle proof \rangle$

lemma *sum-sadd-fst-digit-zero*:
shows $(nth_digit\ (\sum S+ p\ j\ s)\ t\ b)\ i\ c = 0$
 $\langle proof \rangle$

lemma *sum-ssub-block-bound*:
shows $nth_digit\ (\sum S- p\ j\ (\lambda k. s\ k\ \&\&\ z\ (l\ k)))\ t\ b \leq 1$
 $\langle proof \rangle$

lemma *sum-ssub-fst-digit-zero*:
shows $(nth_digit\ (\sum S- p\ j\ (\lambda k. s\ k\ \&\&\ z\ (l\ k)))\ t\ b)\ i\ c = 0$
 $\langle proof \rangle$

lemma *sum-szero-block-bound*:
shows $nth_digit\ (\sum S0\ p\ j\ (\lambda k. s\ k\ \&\&\ (e - z\ (l\ k))))\ t\ b \leq 1$
 $\langle proof \rangle$

lemma *sum-szero-fst-digit-zero*:
shows $(nth_digit\ (\sum S0\ p\ j\ (\lambda k. s\ k\ \&\&\ (e - z\ (l\ k))))\ t\ b)\ i\ c = 0$
 $\langle proof \rangle$

lemma *sum-rsub-special-block-bound*:
shows $nth_digit\ (\sum R- p\ l\ (\lambda k. s\ k))\ t\ b \leq 1$
 $\langle proof \rangle$

lemma *sum-state-special-block-bound*:
shows $nth_digit\ (\sum S+ p\ j\ (\lambda k. s\ k)$
 $+ \sum S0\ p\ j\ (\lambda k. s\ k\ \&\&\ (e - z\ (l\ k))))\ t\ b \leq 1$
 $\langle proof \rangle$

lemma *sum-state-special-fst-digit-zero*:
shows $(nth_digit\ (\sum S+ p\ j\ (\lambda k. s\ k)$
 $+ \sum S0\ p\ j\ (\lambda k. s\ k\ \&\&\ (e - z\ (modifies\ (p!k)))))\ t\ b)\ i\ c = 0$
 $\langle proof \rangle$

Main three redution lemmas: Zero Indicators, Registers, States

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

lemma *zl-le-rl*: $l < n \implies z\ l \leq r\ l$ for l

<proof>

lemma *modifies-valid*: $\forall k \leq m. \text{modifies } (p!k) < n$

<proof>

lemma *seq-bound*: $k \leq \text{length } p - 1 \implies \text{Seq } k\ t \leq 1$

<proof>

lemma *skzl-bitAND-to-mult*:

assumes $k \leq \text{length } p - 1$

assumes $l < n$

shows $\text{nth-digit } (z\ l)\ t\ b \ \&\& \ \text{nth-digit } (s\ k)\ t\ b = (\text{Zeq } l\ t) * \text{Seq } k\ t$

<proof>

lemma *skzl-bitAND-to-mult2*:

assumes $k \leq \text{length } p - 1$

assumes $\forall k \leq \text{length } p - 1. l\ k < n$

shows $(1 - \text{nth-digit } (z\ (l\ k))\ t\ b) \ \&\& \ \text{nth-digit } (s\ k)\ t\ b$
 $= (1 - \text{Zeq } (l\ k)\ t) * \text{Seq } k\ t$

<proof>

lemma *state-equations-digit-commute*:

assumes $t < q$ and $j \leq m$

defines $l \equiv \lambda k. \text{modifies } (p!k)$

shows $\text{nth-digit } (s\ j)\ (\text{Suc } t)\ b =$

$$\begin{aligned} & (\sum S+ p\ j\ (\lambda k. \text{Seq } k\ t)) \\ & + (\sum S- p\ j\ (\lambda k. \text{Zeq } (l\ k)\ t * \text{Seq } k\ t)) \\ & + (\sum S0\ p\ j\ (\lambda k. (1 - \text{Zeq } (l\ k)\ t) * \text{Seq } k\ t)) \end{aligned}$$

<proof>

lemma *aux-nocarry-sk*:

assumes $t \leq q$

shows $i \neq j \implies i \leq m \implies j \leq m \implies \text{nth-digit } (s\ i)\ t\ b * \text{nth-digit } (s\ j)\ t\ b = 0$

<proof>

lemma *nocarry-sk*:

assumes $i \neq j$ and $i \leq m$ and $j \leq m$

shows $(s\ i)\ i\ k * (s\ j)\ i\ k = 0$

<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*: $l < n \implies \sum R- p\ l\ (\lambda k. s\ k \ \&\& \ z\ l) \leq r\ l + \sum R+ p\ l\ s$

<proof>

Obtaining single step register relations from multiple step register relations

lemma *mult-to-single-reg*:

$$c > 0 \implies l < n \implies \text{Req } l \text{ (Suc } t) = \text{Req } l \text{ } t + (\sum R+ \text{ } p \text{ } l \text{ } (\lambda k. \text{Seq } k \text{ } t)) \\ - (\sum R- \text{ } p \text{ } l \text{ } (\lambda k. (\text{Zeq } l \text{ } t) * \text{Seq } k \text{ } t)) \text{ for } l \text{ } t$$

<proof>

Obtaining single step state relations from multiple step state relations

lemma *mult-to-single-state*:

fixes $t \text{ } j :: \text{nat}$

defines $l \equiv \lambda k. \text{modifies } (p!k)$

$$\text{shows } j \leq m \implies t < q \implies \text{Seq } j \text{ (Suc } t) = (\sum S+ \text{ } p \text{ } j \text{ } (\lambda k. \text{Seq } k \text{ } t)) \\ + (\sum S- \text{ } p \text{ } j \text{ } (\lambda k. \text{Zeq } (l \text{ } k) \text{ } t * \text{Seq } k \text{ } t)) \\ + (\sum S0 \text{ } p \text{ } j \text{ } (\lambda k. (1 - \text{Zeq } (l \text{ } k) \text{ } t) * \text{Seq } k \text{ } t))$$

<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*:

$$l < n \implies j \leq m \implies t \leq q \implies R \text{ ic } p \text{ } l \text{ } t = \text{Req } l \text{ } t \wedge Z \text{ ic } p \text{ } l \text{ } t = \text{Zeq } l \text{ } t \wedge S \text{ ic } p \text{ } j \text{ } t \\ = \text{Seq } j \text{ } t$$

<proof>

end

end

4.4 Arithmetizing equations are Diophantine

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

begin

locale *register-machine* =

fixes $p :: \text{program}$

and $n :: \text{nat}$

assumes *p-nonempty*: $\text{length } p > 0$

and *valid-program*: *program-register-check* $p \text{ } n$

assumes *n-gt-0*: $n > 0$

begin

definition $m :: \text{nat}$ **where**

$$m \equiv \text{length } p - 1$$

lemma *modifies-yields-valid-register*:
assumes $k < \text{length } p$
shows $\text{modifies } (p!k) < n$
 $\langle \text{proof} \rangle$

end

locale *rm-eq-fixes* = *register-machine* +
fixes $a b c d e f :: \text{nat}$
and $q :: \text{nat}$
and $r z :: \text{register} \Rightarrow \text{nat}$
and $s :: \text{state} \Rightarrow \text{nat}$

end

4.4.1 Preliminary: Register machine sums are Diophantine

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

begin

fun *sum-polynomial* :: $(\text{nat} \Rightarrow \text{polynomial}) \Rightarrow \text{nat list} \Rightarrow \text{polynomial}$ **where**
sum-polynomial $f [] = \text{Const } 0$ |
sum-polynomial $f (i\#\text{idxs}) = f i [+]$ *sum-polynomial* $f \text{idxs}$

lemma *sum-polynomial-eval*:
 $\text{peval } (\text{sum-polynomial } f \text{idxs}) a = (\sum k=0..<\text{length } \text{idxs}. \text{peval } (f (i\#k)) a)$
 $\langle \text{proof} \rangle$

definition *sum-program* :: $\text{program} \Rightarrow (\text{nat} \Rightarrow \text{polynomial}) \Rightarrow \text{polynomial}$
 $([\sum -] - [100, 100] 100)$ **where**
 $[\sum p] f \equiv \text{sum-polynomial } f [0..<\text{length } p]$

lemma *sum-program-push*: $m = \text{length } ns \Longrightarrow \text{length } l = \text{length } p \Longrightarrow$
 $\text{peval } ([\sum p] (\lambda k. \text{if } g \text{ } k \text{ then } \text{map } (\lambda x. \text{push-param } x \text{ } m) l ! k \text{ else } h \text{ } k)) (\text{push-list } a \text{ } ns)$
 $= \text{peval } ([\sum p] (\lambda k. \text{if } g \text{ } k \text{ then } l ! k \text{ else } h \text{ } k)) a$
 $\langle \text{proof} \rangle$

definition *sum-radd-polynomial* :: $\text{program} \Rightarrow \text{register} \Rightarrow (\text{nat} \Rightarrow \text{polynomial}) \Rightarrow$
 polynomial
 $([\sum R+] - - -)$ **where**
 $[\sum R+] p l f \equiv [\sum p] (\lambda k. \text{if } \text{isadd } (p!k) \wedge l = \text{modifies } (p!k) \text{ then } f \text{ } k \text{ else } \text{Const } 0)$

lemma *sum-radd-polynomial-eval*[*defs*]:

assumes $length\ p > 0$
shows $peval\ ([\sum R+] p\ l\ f)\ a = (\sum R+ p\ l\ (\lambda x. peval\ (f\ x)\ a))$
 $\langle proof \rangle$

definition $sum-rsub-polynomial :: program \Rightarrow register \Rightarrow (nat \Rightarrow polynomial) \Rightarrow polynomial$
 $([\sum R-] - - -)$ **where**
 $[\sum R-] p\ l\ f \equiv [\sum p] (\lambda k. if\ issub\ (p!k) \wedge l = modifies\ (p!k)\ then\ f\ k\ else\ Const\ 0)$

lemma $sum-rsub-polynomial-eval[defs]$:
assumes $length\ p > 0$
shows $peval\ ([\sum R-] p\ l\ f)\ a = (\sum R- p\ l\ (\lambda x. peval\ (f\ x)\ a))$
 $\langle proof \rangle$

definition $sum-sadd-polynomial :: program \Rightarrow state \Rightarrow (nat \Rightarrow polynomial) \Rightarrow polynomial$
 $([\sum S+] - - -)$ **where**
 $[\sum S+] p\ d\ f \equiv [\sum p] (\lambda k. if\ isadd\ (p!k) \wedge d = goes-to\ (p!k)\ then\ f\ k\ else\ Const\ 0)$

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

definition $sum-ssub-nzero-polynomial :: program \Rightarrow state \Rightarrow (nat \Rightarrow polynomial) \Rightarrow polynomial$
 $([\sum S-] - - -)$ **where**
 $[\sum S-] p\ d\ f \equiv [\sum p] (\lambda k. if\ issub\ (p!k) \wedge d = goes-to\ (p!k)\ then\ f\ k\ else\ Const\ 0)$

lemma $sum-ssub-nzero-polynomial-eval[defs]$:
assumes $length\ p > 0$
shows $peval\ ([\sum S-] p\ d\ f)\ a = (\sum S- p\ d\ (\lambda x. peval\ (f\ x)\ a))$
 $\langle proof \rangle$

definition $sum-ssub-zero-polynomial :: program \Rightarrow state \Rightarrow (nat \Rightarrow polynomial) \Rightarrow polynomial$
 $([\sum S0] - - -)$ **where**
 $[\sum S0] p\ d\ f \equiv [\sum p] (\lambda k. if\ issub\ (p!k) \wedge d = goes-to-alt\ (p!k)\ then\ f\ k\ else\ Const\ 0)$

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

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* \Rightarrow *nat* \Rightarrow *polynomial list* \Rightarrow
polynomial list

\Rightarrow *relation*

$([- = \sum S - - '(- \&\& -)'])$ **where**
 $[x = \sum S - d (s \&\& z)] \equiv \text{let } x' = \text{push-param } x \text{ (length } p);$
 $s' = \text{push-param-list } s \text{ (length } p);$
 $z' = \text{push-param-list } z \text{ (length } p)$
in $[\exists \text{ length } p] [\forall < \text{length } p] (\lambda i. [\text{Param } i = s^!i \&\& z^!i])$
 $[\wedge] x' [=] ([\sum S -] p d \text{ Param})$

lemma *sum-ssub-nzero-of-bit-and-dioph*[*dioph*]:
fixes $s z :: \text{polynomial list}$ **and** $d :: \text{nat}$ **and** x
shows *is-dioph-rel* $[x = \sum S - d (s \&\& z)]$
<proof>

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

definition *sum-ssub-zero-of-bit-and* :: *polynomial* \Rightarrow *nat* \Rightarrow *polynomial list* \Rightarrow
polynomial list

\Rightarrow *relation*

$([- = \sum S0 - '(- \&\& -)'])$ **where**
 $[x = \sum S0 d (s \&\& z)] \equiv \text{let } x' = \text{push-param } x \text{ (length } p);$
 $s' = \text{push-param-list } s \text{ (length } p);$
 $z' = \text{push-param-list } z \text{ (length } p)$
in $[\exists \text{ length } p] [\forall < \text{length } p] (\lambda i. [\text{Param } i = s^!i \&\& z^!i])$
 $[\wedge] x' [=] [\sum S0] p d \text{ Param}$

lemma *sum-ssub-zero-of-bit-and-dioph*[*dioph*]:
fixes $s z :: \text{polynomial list}$ **and** $d :: \text{nat}$ **and** x
shows *is-dioph-rel* $[x = \sum S0 d (s \&\& z)]$
<proof>

lemma *sum-rsub-zero-of-bit-and-eval*:
fixes $z s :: \text{polynomial list}$ **and** $d :: \text{nat}$ **and** $x :: \text{polynomial}$


```

assumes length s = Suc m length z = Suc m length p > 0
shows eval [x =  $\sum SO\ d\ (s\ \&\&\ z)$ ] a
       $\longleftrightarrow$  peval x a =  $\sum SO\ p\ d\ (\lambda k. peval\ (s!k)\ a\ \&\&\ peval\ (z!k)\ a)$  (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  $\equiv r\ 0 = a + b * r\ 0 + b * \sum R+ p\ 0\ s - b * \sum R- p\ 0\ 0$  ( $\lambda k. s\ k\ \&\&\ z\ 0$ )

```

Equation 4.23

```

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

```

Extra equation not in Matiyasevich's book

```

definition register-bound :: bool where
  register-bound  $\equiv \forall l < n. r\ l < b \wedge q$ 

```

```

definition register-equations :: bool where
  register-equations  $\equiv register-0 \wedge register-l \wedge register-bound$ 

```

```

end

```

```

context register-machine
begin

```

```

definition sum-rsub-of-bit-and :: polynomial  $\Rightarrow$  nat  $\Rightarrow$  polynomial list  $\Rightarrow$  polynomial

```

```

       $\Rightarrow$  relation
  ([- =  $\sum R- -\ '(-\ \&\&\ -')$ ]) where
  [x =  $\sum R- d\ (s\ \&\&\ zl)$ ]  $\equiv$  let x' = push-param x (length p);
      s' = push-param-list s (length p);
      zl' = push-param zl (length p)

```

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

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

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

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

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

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

definition *register-equations-relation* $:: \text{polynomial} \Rightarrow \text{polynomial} \Rightarrow \text{polynomial} \Rightarrow \text{polynomial list} \Rightarrow \text{polynomial list} \Rightarrow \text{polynomial list} \Rightarrow \text{relation} \ ([REG] \ - \ - \ -)$

```

- - -) where
  [REG] a b q r z s ≡ LARY (λll. 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 (λk. s k && z (modifies
  (p!k)))
    + b*∑ S0 p 0 (λk. 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 (λll. 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**
 $state-d \equiv \forall d > 0. d \leq m \longrightarrow s\ d = b * \sum S + p\ d\ s + b * \sum S - p\ d\ (\lambda k. s\ k \ \&\&\ z$
 (*modifies* (*p!k*)))
 $+ b * \sum S\ 0\ p\ d\ (\lambda k. s\ k \ \&\&\ (e - z\ (\textit{modifies}$
 (*p!k*)))

Combining the two

definition *state-relations-from-recursion* :: *bool* **where**
 $state-relations-from-recursion \equiv state-0 \wedge state-d$

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 $DR \equiv LARY\ (\lambda ll. rm-eq-fixes.state-d\ p\ (ll!0!0)\ (ll!0!1)$
 $(nth\ (ll!1))\ (nth\ (ll!2)))$

$[[b, e], z, s]$

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 $DR \equiv LARY\ (\lambda ll. rm-eq-fixes.state-relations-from-recursion\ p\ (ll!0!0)$
 $(ll!0!1)$

```

                                                                    (nth (ll!1)) (nth (ll!2)))
    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  $\equiv \forall k \leq m. s\ k \preceq e$ 

```

```

definition state-bound :: bool where
  state-bound  $\equiv \forall k < m. s\ k < b \wedge q$ 

```

```

definition state-unique-equations :: bool where
  state-unique-equations  $\equiv state\_mask \wedge state\_bound$ 

```

```

end

```

```

context register-machine
begin

```

```

lemma state-mask-dioph:
  fixes e :: polynomial
  fixes s :: polynomial list
  assumes length s = Suc m
  defines DR  $\equiv$  LARY ( $\lambda ll. rm\_eq\_fixes.state\_mask\ p\ (ll!0!0)\ (nth\ (ll!1))\ [[e],\ s]$ )
  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 (λll. 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
      (λll. 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 $b\ q :: \text{polynomial}$
fixes $s :: \text{polynomial list}$
assumes $\text{length } s = \text{Suc } m$
defines $DR \equiv LARY (\lambda ll. \text{rm-eq-fixes.state-m } p (ll!0!0) (ll!0!1) (\text{nth } (ll!1))) [[b,$
 $q], s]$
shows *is-dioph-rel* DR
 $\langle \text{proof} \rangle$

lemma *state-partial-sum-mask-dioph*:
fixes $e :: \text{polynomial}$
fixes $s :: \text{polynomial list}$
assumes $\text{length } s = \text{Suc } m$
defines $DR \equiv LARY (\lambda ll. \text{rm-eq-fixes.state-partial-sum-mask } p (ll!0!0) (\text{nth}$
 $(ll!1))) [[e], s]$
shows *is-dioph-rel* DR
 $\langle \text{proof} \rangle$

definition *state-equations-relation* $:: \text{polynomial} \Rightarrow \text{polynomial} \Rightarrow \text{polynomial} \Rightarrow$
 polynomial list
 $\Rightarrow \text{polynomial list} \Rightarrow \text{relation } ([STATE] \text{ - - - -})$ **where**
 $[STATE] \ b\ e\ q\ z\ s \equiv LARY (\lambda ll. \text{rm-eq-fixes.state-equations } p (ll!0!0) (ll!0!1)$
 $(ll!0!2)$
 $(\text{nth } (ll!1) (\text{nth } (ll!2)))$
 $[[b, e, q], z, s]$

lemma *state-equations-dioph*:
fixes $b\ e\ q :: \text{polynomial}$
fixes $s\ z :: \text{polynomial list}$
assumes $\text{length } s = \text{Suc } m \ \text{length } z = n$
defines $DR \equiv [STATE] \ b\ e\ q\ z\ s$
shows *is-dioph-rel* DR
 $\langle \text{proof} \rangle$

end

end

4.4.7 Equations for masking relations

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

abbrevs $mb = \preceq$

begin

context *rm-eq-fixes*
begin

Equation 4.15

definition *register-mask* :: *bool* **where**
register-mask $\equiv \forall l < n. r\ l \preceq d$

Equation 4.17

definition *zero-indicator-mask* :: *bool* **where**
zero-indicator-mask $\equiv \forall l < n. z\ l \preceq e$

Equation 4.20

definition *zero-indicator-0-or-1* :: *bool* **where**
zero-indicator-0-or-1 $\equiv \forall l < n. 2^{\wedge}c * z\ l = (r\ l + d) \&\& f$

definition *mask-equations* :: *bool* **where**
mask-equations $\equiv \text{register-mask} \wedge \text{zero-indicator-mask} \wedge \text{zero-indicator-0-or-1}$

end

context *register-machine*
begin

lemma *register-mask-dioph*:

fixes *d r*
assumes *n = length r*
defines *DR* $\equiv (\text{NARY } (\lambda l. \text{rm-eq-fixes.register-mask } n\ (!0)\ (\text{shift } l\ 1))\ ([d\ @\ r]))$
shows *is-dioph-rel DR*
<proof>

lemma *zero-indicator-mask-dioph*:

fixes *e z*
assumes *n = length z*
defines *DR* $\equiv (\text{NARY } (\lambda l. \text{rm-eq-fixes.zero-indicator-mask } n\ (!0)\ (\text{shift } l\ 1))\ ([e\ @\ z]))$
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* $\equiv \text{LARY } (\lambda ll. \text{rm-eq-fixes.zero-indicator-0-or-1 } n\ (!0!0)\ (!0!1)\ (!0!2)\ (\text{nth } (!0!1))\ (\text{nth } (!0!2)))\ [[c, d, f], r, z]$
shows *is-dioph-rel DR*
<proof>

definition *mask-equations-relation* (*[MASK]* - - - - -) **where**

[MASK] *c d e f r z* $\equiv \text{LARY } (\lambda ll. \text{rm-eq-fixes.mask-equations } n\ (!0!0)\ (!0!1)\ (!0!2)\ (!0!3)\ (\text{nth } (!0!1))\ (\text{nth } (!0!2)))$

$[[c, d, e, f], r, z]$

lemma *mask-equations-relation-dioph*:
 fixes $c\ d\ e\ f\ r\ z$
 assumes $n = \text{length } r$ **and** $n = \text{length } z$
 defines $DR \equiv [MASK]\ c\ d\ e\ f\ r\ z$
 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* $:: \text{bool}$ **where**
 $\text{constant-b} \equiv b = B\ c$

Equation 4.16

definition *constant-d* $:: \text{bool}$ **where**
 $\text{constant-d} \equiv d = D\ q\ c\ b$

Equation 4.18

definition *constant-e* $:: \text{bool}$ **where**
 $\text{constant-e} \equiv e = E\ q\ b$

Equation 4.21

definition *constant-f* $:: \text{bool}$ **where**
 $\text{constant-f} \equiv f = F\ q\ c\ b$

Equation not in the book

definition *c-gt-0* $:: \text{bool}$ **where**
 $\text{c-gt-0} \equiv c > 0$

Equation 4.26

definition *a-bound* $:: \text{bool}$ **where**
 $\text{a-bound} \equiv a < 2 \wedge c$

Equation not in the book

definition *q-gt-0* :: *bool* **where**
q-gt-0 \equiv $q > 0$

definition *constants-equations* :: *bool* **where**
constants-equations \equiv *constant-b* \wedge *constant-d* \wedge *constant-e* \wedge *constant-f*

definition *miscellaneous-equations* :: *bool* **where**
miscellaneous-equations \equiv *c-gt-0* \wedge *a-bound* \wedge *q-gt-0*

end

context *register-machine*
begin

definition *rm-constant-equations* ::
polynomial \Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *relation*
 ([CONST] - - - - -) **where**
 [CONST] *b c d e f q* \equiv *NARY* ($\lambda l.$ *rm-eq-fixes.constants-equations*
 (!0) (!1) (!2) (!3) (!4) (!5)) [*b, c, d, e, f, q*]

definition *rm-miscellaneous-equations* ::
polynomial \Rightarrow *polynomial* \Rightarrow *polynomial* \Rightarrow *relation*
 ([MISC] - - -) **where**
 [MISC] *c a q* \equiv *NARY* ($\lambda l.$ *rm-eq-fixes.miscellaneous-equations*
 (!0) (!1) (!2)) [*c, a, q*]

lemma *rm-constant-equations-dioph*:
fixes *b c d e f q*
defines *DR* \equiv [CONST] *b c d e f q*
shows *is-dioph-rel DR*
 <proof>

lemma *rm-miscellaneous-equations-dioph*:
fixes *c a q*
defines *DR* \equiv [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 $a\ q\ b\ c\ d\ e\ f\ r\ z\ s$
 \equiv *rm-eq-fixes.register-equations* $p\ n\ a\ b\ q\ r\ z\ s$
 \wedge *rm-eq-fixes.state-equations* $p\ b\ e\ q\ z\ s$
 \wedge *rm-eq-fixes.mask-equations* $n\ c\ d\ e\ f\ r\ z$
 \wedge *rm-eq-fixes.constants-equations* $b\ c\ d\ e\ f\ q$
 \wedge *rm-eq-fixes.miscellaneous-equations* $a\ c\ q$

lemma *all-equations-invariance*:

fixes $r\ z\ s :: \text{nat} \Rightarrow \text{nat}$

and $r'\ z'\ s' :: \text{nat} \Rightarrow \text{nat}$

assumes $\forall i < n. r\ i = r'\ i$ **and** $\forall i < n. z\ i = z'\ i$ **and** $\forall i < \text{Suc}\ m. s\ i = s'\ i$

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* $:: \text{polynomial} \Rightarrow \text{polynomial} \Rightarrow \text{polynomial} \Rightarrow$
polynomial

$\Rightarrow \text{polynomial} \Rightarrow \text{polynomial} \Rightarrow \text{polynomial} \Rightarrow \text{polynomial list} \Rightarrow \text{polynomial list}$
 $\Rightarrow \text{polynomial list}$

\Rightarrow *relation* (*[ALLEQ]* - - - - -) **where**

[ALLEQ] $a\ q\ b\ c\ d\ e\ f\ r\ z\ s$

\equiv *LARY* ($\lambda ll. \text{all-equations } (ll!0!0) (ll!0!1) (ll!0!2) (ll!0!3) (ll!0!4) (ll!0!5)$
 $(ll!0!6)$

$(\text{nth } (ll!1)) (\text{nth } (ll!2)) (\text{nth } (ll!3))$

$[[a, q, b, c, d, e, f], r, z, s]$

lemma *all-equations-dioph*:

fixes $A\ f\ e\ d\ c\ b\ q :: \text{polynomial}$

```

fixes  $r z s :: \text{polynomial list}$ 
assumes  $\text{length } r = n \text{ length } z = n \text{ length } s = \text{Suc } m$ 
defines  $DR \equiv [ALLEQ] A q b c d e f r z s$ 
shows  $\text{is-dioph-rel } DR$ 
<proof>

```

```

definition  $\text{rm-equations} :: \text{nat} \Rightarrow \text{bool}$  where
   $\text{rm-equations } a \equiv \exists q :: \text{nat}.$ 
     $\exists b c d e f :: \text{nat}.$ 
     $\exists r z :: \text{register} \Rightarrow \text{nat}.$ 
     $\exists s :: \text{state} \Rightarrow \text{nat}.$ 
     $\text{all-equations } a q b c d e f r z s$ 

```

```

definition  $\text{rm-equations-relation} :: \text{polynomial} \Rightarrow \text{relation}$   $([RM] \ -)$  where
   $[RM] A \equiv \text{UNARY } (\text{rm-equations}) A$ 

```

```

lemma  $\text{rm-dioph}$ :
  fixes  $A$ 
  fixes  $ic :: \text{configuration}$ 
  defines  $DR \equiv [RM] A$ 
  shows  $\text{is-dioph-rel } DR$ 
<proof>

```

end

end

4.5 Equivalence of register machine and arithmetizing equations

```

theory  $\text{Machine-Equation-Equivalence}$  imports  $\text{All-Equations}$ 
   $\dots/ \text{Register-Machine/ MachineEquations}$ 
   $\dots/ \text{Register-Machine/ MultipleToSingleSteps}$ 

```

begin

```

context  $\text{register-machine}$ 
begin

```

```

lemma  $\text{conclusion-4-5}$ :
  assumes  $\text{is-val: is-valid-initial } ic p a$ 
  and  $n\text{-def: } n \equiv \text{length } (\text{snd } ic)$ 
  shows  $(\exists q. \text{terminates } ic p q) = \text{rm-equations } a$ 
<proof>

```

end

end

5 Proof of the DPRM theorem

theory *DPRM*

imports *Machine-Equations/Machine-Equation-Equivalence*

begin

definition *is-recenum* :: *nat set* \Rightarrow *bool* **where**

is-recenum *A* =

(\exists *p* :: *program*.

\exists *n* :: *nat*.

\forall *a* :: *nat*. \exists *ic*. *ic* = *initial-config* *n a* \wedge *is-valid-initial* *ic p a* \wedge

(*a* \in *A*) = (\exists *q*::*nat*. *terminates* *ic p q*)

theorem *DPRM*: *is-recenum* *A* \implies *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.