Minsky Machines*

Bertram Felgenhauer

September 13, 2023

Abstract

We formalize undecidablity results for Minsky machines. To this end, we also formalize recursive inseparability.

We start by proving that Minsky machines can compute arbitrary primitive recursive and recursive functions. We then show that there is a deterministic Minsky machine with one argument (modeled by assigning the argument to register 0 in the initial configuration) and final states 0 and 1 such that the set of inputs that are accepted in state 0 is recursively inseparable from the set of inputs that are accepted in state 1.

As a corollary, the set of Minsky configurations that reach state 0 but not state 1 is recursively inseparable from the set of Minsky configurations that reach state 1 but not state 0. In particular both these sets are undecidable.

We do *not* prove that recursive functions can simulate Minsky machines.

Contents

1 Recursive inseperability **2**
- 1.1 Definition and basic facts . . . 2
- 1.2 Rice's theorem . . . 3

2 Minsky machines **4**
- 2.1 Deterministic relations . . . 4
- 2.2 Minsky machine definition . . . 5
- 2.3 Concrete Minsky machines . . . 7
- 2.4 Trivial building blocks . . . 7
- 2.5 Sequential composition . . . 8
- 2.6 Bounded loop . . . 8
- 2.7 Copying values . . . 9
- 2.8 Primitive recursive functions . . . 10

*This work was supported by FWF (Austrian Science Fund) project P30301.

2.9 Recursively enumerable sets as Minsky machines 11
2.10 Encoding of Minsky machines 12
2.11 Undecidablity results 13

1 Recursive inseperability

theory *Recursive-Inseparability*
imports *Recursion−Theory−I.RecEnSet*
begin

Two sets A and B are recursively inseparable if there is no computable set that contains A and is disjoint from B. In particular, a set is computable if the set and its complement are recursively inseparable. The terminology was introduced by Smullyan [4]. The underlying idea can be traced back to Rosser, who essentially showed that provable and disprovable sentences are *arithmetically* inseparable in Peano Arithmetic [3]; see also Kleene's symmetric version of Gödel's incompleteness theorem [1].

Here we formalize recursive inseparability on top of the `Recursion-Theory-I` AFP entry [2]. Our main result is a version of Rice' theorem that states that the index sets of any two given recursively enumerable sets are recursively inseparable.

1.1 Definition and basic facts

Two sets A and B are recursively inseparable if there are no decidable sets X such that A is a subset of X and X is disjoint from B.

definition *rec-inseparable* **where**
rec-inseparable A B $\equiv \forall X.\ A \subseteq X \wedge B \subseteq -X \longrightarrow \neg$ *computable X*

lemma *rec-inseparableI*:
$(\bigwedge X.\ A \subseteq X \Longrightarrow B \subseteq -X \Longrightarrow$ *computable X* $\Longrightarrow$ *False*$) \Longrightarrow$ *rec-inseparable A B*
⟨*proof*⟩

lemma *rec-inseparableD*:
rec-inseparable A B $\Longrightarrow A \subseteq X \Longrightarrow B \subseteq -X \Longrightarrow$ *computable X* $\Longrightarrow$ *False*
⟨*proof*⟩

Recursive inseperability is symmetric and enjoys a monotonicity property.

lemma *rec-inseparable-symmetric*:
rec-inseparable A B $\Longrightarrow$ *rec-inseparable B A*
⟨*proof*⟩

lemma *rec-inseparable-mono*:
rec-inseparable A B $\Longrightarrow A \subseteq A' \Longrightarrow B \subseteq B' \Longrightarrow$ *rec-inseparable A' B'*
⟨*proof*⟩

Many-to-one reductions apply to recursive inseparability as well.

lemma *rec-inseparable-many-reducible*:
assumes *total-recursive f rec-inseparable* $(f\ -`\ A)$ $(f\ -`\ B)$
shows *rec-inseparable A B*
⟨*proof*⟩

Recursive inseparability of A and B holds vacuously if A and B are not disjoint.

lemma *rec-inseparable-collapse*:
$A \cap B \neq \{\} \Longrightarrow$ *rec-inseparable A B*
⟨*proof*⟩

Recursive inseparability is intimately connected to non-computability.

lemma *rec-inseparable-non-computable*:
$A \cap B = \{\} \Longrightarrow$ *rec-inseparable A B* $\Longrightarrow \neg$ *computable A*
⟨*proof*⟩

lemma *computable-rec-inseparable-conv*:
computable A $\longleftrightarrow \neg$ *rec-inseparable A* $(-\ A)$
⟨*proof*⟩

1.2 Rice's theorem

We provide a stronger version of Rice's theorem compared to [2]. Unfolding the definition of recursive inseparability, it states that there are no decidable sets X such that

- there is a r.e. set such that all its indices are elements of X; and
- there is a r.e. set such that none of its indices are elements of X.

This is true even if X is not an index set (i.e., if an index of a r.e. set is an element of X, then X contains all indices of that r.e. set), which is a requirement of Rice's theorem in [2].

lemma *c-pair-inj'*:
c-pair x1 y1 $=$ *c-pair x2 y2* $\longleftrightarrow$ *x1* $=$ *x2* $\wedge$ *y1* $=$ *y2*
⟨*proof*⟩

lemma *Rice-rec-inseparable*:
rec-inseparable $\{k.$ *nat-to-ce-set k* $=$ *nat-to-ce-set n*$\}$ $\{k.$ *nat-to-ce-set k* $=$ *nat-to-ce-set m*$\}$
⟨*proof*⟩

end

2 Minsky machines

theory *Minsky*
imports *Recursive-Inseparability Abstract−Rewriting.Abstract-Rewriting Pure−ex.Guess*
begin

We formalize Minksy machines, and relate them to recursive functions. In our flavor of Minsky machines, a machine has a set of registers and a set of labels, and a program is a set of labeled operations. There are two operations, *Inc* and *Dec*; the former takes a register and a label, and the latter takes a register and two labels. When an *Inc* instruction is executed, the register is incremented and execution continues at the provided label. The *Dec* instruction checks the register. If it is non-zero, the register and continues execution at the first label. Otherwise, the register remains at zero and execution continues at the second label.

We continue to show that Minksy machines can implement any primitive recursive function. Based on that, we encode recursively enumerable sets as Minsky machines, and finally show that

1. The set of Minsky configurations such that from state 1, state 0 can be reached, is undecidable;
2. There is a deterministic Minsky machine U such that the set of values x such that $(2, \lambda n.\,\text{if } n = 0 \text{ then } x \text{ else } 0)$ reach state 0 is recursively inseparable from those that reach state 1; and
3. As a corollary, the set of Minsky configurations that reach state 0 but not state 1 is recursively inseparable from the configurations that reach state 1 but not state 0.

2.1 Deterministic relations

A relation $\to$ is *deterministic* if $t \leftarrow s \to u'$ implies $t = u$. This abstract rewriting notion is useful for talking about deterministic Minsky machines.

definition
deterministic $R \longleftrightarrow R^{-1}\ O\ R \subseteq Id$

lemma *deterministicD*:
deterministic $R \Longrightarrow (x,\ y) \in R \Longrightarrow (x,\ z) \in R \Longrightarrow y = z$
⟨*proof*⟩

lemma *deterministic-empty* [*simp*]:
deterministic {}
⟨*proof*⟩

lemma *deterministic-singleton* [*simp*]:

deterministic $\{p\}$
⟨*proof*⟩

lemma *deterministic-imp-weak-diamond* [*intro*]:
deterministic $R \Longrightarrow w\Diamond\ R$
⟨*proof*⟩

lemmas *deterministic-imp-CR* = *deterministic-imp-weak-diamond*[*THEN weak-diamond-imp-CR*]

lemma *deterministic-union*:
fst ‘ $S \cap$ *fst* ‘ $R = \{\} \Longrightarrow$ *deterministic* $S \Longrightarrow$ *deterministic* $R \Longrightarrow$ *deterministic* $(S \cup R)$
⟨*proof*⟩

lemma *deterministic-map*:
inj-on f (*fst* ‘ R) $\Longrightarrow$ *deterministic* $R \Longrightarrow$ *deterministic* (*map-prod f g* ‘ R)
⟨*proof*⟩

2.2 Minsky machine definition

A Minsky operation either decrements a register (testing for zero, with two possible successor states), or increments a register (with one successor state). A Minsky machine is a set of pairs of states and operations.

datatype ('*s*, '*v*) *Op* = *Dec* (*op-var*: '*v*) '*s* '*s* | *Inc* (*op-var*: '*v*) '*s*

type-synonym ('*s*, '*v*) *minsky* = ('*s* × ('*s*, '*v*) *Op*) *set*

Semantics: A Minsky machine operates on pairs consisting of a state and an assignment of the registers; in each step, either a register is incremented, or a register is decremented, provided it is non-zero. We write α for assignments; $\alpha[v]$ for the value of the register v in α and $\alpha[v := n]$ for the update of v to n. Thus, the semantics is as follows:

1. if $(s, Inc\ v\ s') \in M$ then $(s, \alpha) \to (s', \alpha[v := \alpha[v] + 1])$;
2. if $(s, Dec\ v\ s_n\ s_z) \in M$ and $\alpha[v] > 0$ then $(s, \alpha) \to (s_n, \alpha[v := \alpha[v]-1])$; and
3. if $(s, Dec\ v\ s_n\ s_z) \in M$ and $\alpha[v] = 0$ then $(s, \alpha) \to (s_z, \alpha)$.

A state is finite if there is no operation associated with it.

inductive-set *step* :: ('*s*, '*v*) *minsky* ⇒ ('*s* × ('*v* ⇒ *nat*)) *rel* **for** *M* :: ('*s*, '*v*) *minsky* **where**
inc: (*s*, *Inc v s'*) ∈ *M* ⟹ ((*s*, *vs*), (*s'*, λ*x*. *if x* = *v then Suc* (*vs v*) *else vs x*)) ∈ *step M*
| *decn*: (*s*, *Dec v sn sz*) ∈ *M* ⟹ *vs v* = *Suc n* ⟹ ((*s*, *vs*), (*sn*, λ*x*. *if x* = *v then n else vs x*)) ∈ *step M*
| *decz*: (*s*, *Dec v sn sz*) ∈ *M* ⟹ *vs v* = *0* ⟹ ((*s*, *vs*), (*sz*, *vs*)) ∈ *step M*

lemma *step-mono*:
$M \subseteq M' \Longrightarrow step\ M \subseteq step\ M'$
⟨*proof*⟩

lemmas *steps-mono* = *rtrancl-mono*[*OF step-mono*]

A Minsky machine has deterministic steps if its defining relation between states and operations is deterministic.

lemma *deterministic-stepI* [*intro*]:
assumes *deterministic M* **shows** *deterministic* (*step M*)
⟨*proof*⟩

A Minksy machine halts when it reaches a state with no associated operation.

lemma *NF-stepI* [*intro*]:
$s \notin fst\ `\ M \Longrightarrow (s, vs) \in NF\ (step\ M)$
⟨*proof*⟩

Deterministic Minsky machines enjoy unique normal forms.

lemmas *deterministic-minsky-UN* =
join-NF-imp-eq[*OF CR-divergence-imp-join*[*OF deterministic-imp-CR*[*OF deterministic-stepI*]] *NF-stepI NF-stepI*]

We will rename states and variables.

definition *map-minsky* **where**
map-minsky f g M = *map-prod f* (*map-Op f g*) ` *M*

lemma *map-minsky-id*:
map-minsky id id M = *M*
⟨*proof*⟩

lemma *map-minsky-comp*:
map-minsky f g (*map-minsky f' g' M*) = *map-minsky* ($f \circ f'$) ($g \circ g'$) *M*
⟨*proof*⟩

When states and variables are renamed, computations carry over from the original machine, provided that variables are renamed injectively.

lemma *map-step*:
assumes *inj g vs* = *vs'* ∘ *g* ((*s*, *vs*), (*t*, *ws*)) ∈ *step M*
shows ((*f s*, *vs'*), (*f t*, λ*x*. *if x* ∈ *range g then ws* (*inv g x*) *else vs' x*)) ∈ *step* (*map-minsky f g M*)
⟨*proof*⟩

lemma *map-steps*:
assumes *inj g vs* = *ws* ∘ *g* ((*s*, *vs*), (*t*, *vs'*)) ∈ (*step M*)*
shows ((*f s*, *ws*), (*f t*, λ*x*. *if x* ∈ *range g then vs'* (*inv g x*) *else ws x*)) ∈ (*step* (*map-minsky f g M*))*
⟨*proof*⟩

2.3 Concrete Minsky machines

The following definition expresses when a Minsky machine M implements a specification P. We adopt the convention that computations always start out in state 1 and end in state 0, which must be a final state. The specification P relates initial assignments to final assignments.

definition *mk-minsky-wit* :: (*nat*, *nat*) *minsky* ⇒ ((*nat* ⇒ *nat*) ⇒ (*nat* ⇒ *nat*) ⇒ *bool*) ⇒ *bool* **where**
mk-minsky-wit M P ≡ *finite M* ∧ *deterministic M* ∧ *0* ∉ *fst* ‘ *M* ∧
(∀ *vs*. ∃ *vs′*. ((*Suc 0*, *vs*), (*0*, *vs′*)) ∈ (*step M*)* ∧ *P vs vs′*)

abbreviation *mk-minsky* :: ((*nat* ⇒ *nat*) ⇒ (*nat* ⇒ *nat*) ⇒ *bool*) ⇒ *bool* **where**
mk-minsky P ≡ ∃ *M*. *mk-minsky-wit M P*

lemmas *mk-minsky-def* = *mk-minsky-wit-def*

lemma *mk-minsky-mono*:
shows *mk-minsky P* ⟹ (⋀*vs vs′*. *P vs vs′* ⟹ *Q vs vs′*) ⟹ *mk-minsky Q*
⟨*proof*⟩

lemma *mk-minsky-sound*:
assumes *mk-minsky-wit M P* ((*Suc 0*, *vs*), (*0*, *vs′*)) ∈ (*step M*)*
shows *P vs vs′*
⟨*proof*⟩

Realizability of n-ary functions for $n = 1 \ldots 3$. Here we use the convention that the arguments are passed in registers $1 \ldots 3$, and the result is stored in register 0.

abbreviation *mk-minsky1* **where**
mk-minsky1 f ≡ *mk-minsky* (λ*vs vs′*. *vs′ 0* = *f* (*vs 1*))

abbreviation *mk-minsky2* **where**
mk-minsky2 f ≡ *mk-minsky* (λ*vs vs′*. *vs′ 0* = *f* (*vs 1*) (*vs 2*))

abbreviation *mk-minsky3* **where**
mk-minsky3 f ≡ *mk-minsky* (λ*vs vs′*. *vs′ 0* = *f* (*vs 1*) (*vs 2*) (*vs 3*))

2.4 Trivial building blocks

We can increment and decrement any register.

lemma *mk-minsky-inc*:
shows *mk-minsky* (λ*vs vs′*. *vs′* = (λ*x*. *if x* = *v then Suc* (*vs v*) *else vs x*))
⟨*proof*⟩

lemma *mk-minsky-dec*:
shows *mk-minsky* (λ*vs vs′*. *vs′* = (λ*x*. *if x* = *v then vs v* − *1 else vs x*))
⟨*proof*⟩

2.5 Sequential composition

The following lemma has two useful corollaries (which we prove simultaneously because they share much of the proof structure): First, if P and Q are realizable, then so is $P \circ Q$. Secondly, if we rename variables by an injective function f in a Minksy machine, then the variables outside the range of f remain unchanged.

lemma *mk-minsky-seq-map*:
assumes *mk-minsky P mk-minsky Q inj g*
$\bigwedge vs\ vs'\ vs''.\ P\ vs\ vs' \Longrightarrow Q\ vs'\ vs'' \Longrightarrow R\ vs\ vs''$
shows *mk-minsky* ($\lambda vs\ vs'.\ R\ (vs \circ g)\ (vs' \circ g) \wedge (\forall x.\ x \notin range\ g \longrightarrow vs\ x = vs'\ x)$)
⟨*proof*⟩

Sequential composition.

lemma *mk-minsky-seq*:
assumes *mk-minsky P mk-minsky Q*
$\bigwedge vs\ vs'\ vs''.\ P\ vs\ vs' \Longrightarrow Q\ vs'\ vs'' \Longrightarrow R\ vs\ vs''$
shows *mk-minsky R*
⟨*proof*⟩

lemma *mk-minsky-seq'*:
assumes *mk-minsky P mk-minsky Q*
shows *mk-minsky* ($\lambda vs\ vs''.\ (\exists vs'.\ P\ vs\ vs' \wedge Q\ vs'\ vs'')$)
⟨*proof*⟩

We can do nothing (besides transitioning from state 1 to state 0).

lemma *mk-minsky-nop*:
mk-minsky ($\lambda vs\ vs'.\ vs = vs'$)
⟨*proof*⟩

Renaming variables.

lemma *mk-minsky-map*:
assumes *mk-minsky P inj f*
shows *mk-minsky* ($\lambda vs\ vs'.\ P\ (vs \circ f)\ (vs' \circ f) \wedge (\forall x.\ x \notin range\ f \longrightarrow vs\ x = vs'\ x)$)
⟨*proof*⟩

lemma *inj-shift* [*simp*]:
fixes *a b* :: *nat*
assumes $a < b$
shows *inj* ($\lambda x.$ *if* $x = 0$ *then a else* $x + b$)
⟨*proof*⟩

2.6 Bounded loop

In the following lemma, P is the specification of a loop body, and Q the specification of the loop itself (a loop invariant). The loop variable is v. Q

can be realized provided that

1. P can be realized;
2. P ensures that the loop variable is not changed by the loop body; and
3. Q follows by induction on the loop variable:
   (a) $\alpha\, Q\, \alpha$ holds when $\alpha[v] = 0$; and
   (b) $\alpha[v := n]\, P\, \alpha'$ and $\alpha'\, Q\, \alpha''$ imply $\alpha\, Q\, alpha''$ when $\alpha[v] = n + 1$.

lemma *mk-minsky-loop*:
assumes *mk-minsky P*
$\bigwedge vs\ vs'.\ P\ vs\ vs' \Longrightarrow vs'\ v = vs\ v$
$\bigwedge vs.\ vs\ v = 0 \Longrightarrow Q\ vs\ vs$
$\bigwedge n\ vs\ vs'\ vs''.\ vs\ v = Suc\ n \Longrightarrow P\ (\lambda x.\ if\ x = v\ then\ n\ else\ vs\ x)\ vs' \Longrightarrow Q\ vs'\ vs'' \Longrightarrow Q\ vs\ vs''$
shows *mk-minsky Q*
⟨*proof*⟩

2.7 Copying values

We work up to copying values in several steps.

1. Clear a register. This is a loop that decrements the register until it reaches 0.
2. Add a register to another one. This is a loop that decrements one register, and increments the other register, until the first register reaches 0.
3. Add a register to two others. This is the same, except that two registers are incremented.
4. Move a register: set a register to 0, then add another register to it.
5. Copy a register destructively: clear two registers, then add another register to them.

lemma *mk-minsky-zero*:
shows *mk-minsky* $(\lambda vs\ vs'.\ vs' = (\lambda x.\ if\ x = v\ then\ 0\ else\ vs\ x))$
⟨*proof*⟩

lemma *mk-minsky-add1*:
assumes $v \neq w$
shows *mk-minsky* $(\lambda vs\ vs'.\ vs' = (\lambda x.\ if\ x = v\ then\ 0\ else\ if\ x = w\ then\ vs\ v + vs\ w\ else\ vs\ x))$
⟨*proof*⟩

lemma *mk-minsky-add2*:
assumes $u \neq v$ $u \neq w$ $v \neq w$
shows *mk-minsky* ($\lambda vs\ vs'.\ vs' =$
($\lambda x.$ *if* $x = u$ *then 0 else if* $x = v$ *then vs* u + *vs* v *else if* $x = w$ *then vs* u + *vs* w *else vs* x))
⟨*proof*⟩

lemma *mk-minsky-copy1*:
assumes $v \neq w$
shows *mk-minsky* ($\lambda vs\ vs'.\ vs' = (\lambda x.$ *if* $x = v$ *then 0 else if* $x = w$ *then vs* v *else vs* x))
⟨*proof*⟩

lemma *mk-minsky-copy2*:
assumes $u \neq v$ $u \neq w$ $v \neq w$
shows *mk-minsky* ($\lambda vs\ vs'.\ vs' =$
($\lambda x.$ *if* $x = u$ *then 0 else if* $x = v$ *then vs* u *else if* $x = w$ *then vs* u *else vs* x))
⟨*proof*⟩

lemma *mk-minsky-copy*:
assumes $u \neq v$ $u \neq w$ $v \neq w$
shows *mk-minsky* ($\lambda vs\ vs'.\ vs' = (\lambda x.$ *if* $x = v$ *then vs* u *else if* $x = w$ *then 0 else vs* x))
⟨*proof*⟩

2.8 Primitive recursive functions

Nondestructive apply: compute f on arguments $\alpha[u]$, $\alpha[v]$, $\alpha[w]$, storing the result in $\alpha[t]$ and preserving all other registers below k. This is easy now that we can copy values.

lemma *mk-minsky-apply3*:
assumes *mk-minsky3 f* $t < k$ $u < k$ $v < k$ $w < k$
shows *mk-minsky* ($\lambda vs\ vs'.\ \forall x < k.\ vs'\ x =$ (*if* $x = t$ *then f* (*vs* u) (*vs* v) (*vs* w) *else vs* x))
⟨*proof*⟩

Composition is just four non-destructive applies.

lemma *mk-minsky-comp3-3*:
assumes *mk-minsky3 f mk-minsky3 g mk-minsky3 h mk-minsky3 k*
shows *mk-minsky3* ($\lambda x\ y\ z.$ *f* (*g x y z*) (*h x y z*) (*k x y z*))
⟨*proof*⟩

Primitive recursion is a non-destructive apply followed by a loop with another non-destructive apply. The key to the proof is the loop invariant, which we can specify as part of composing the various *mk-minsky-*$*$ lemmas.

lemma *mk-minsky-prim-rec*:
assumes *mk-minsky1 g mk-minsky3 h*
shows *mk-minsky2* (*PrimRecOp g h*)

⟨*proof*⟩

With these building blocks we can easily show that all primitive recursive functions can be realized by a Minsky machine.

lemma *mk-minsky-PrimRec*:
f ∈ *PrimRec1* ⟹ *mk-minsky1 f*
g ∈ *PrimRec2* ⟹ *mk-minsky2 g*
h ∈ *PrimRec3* ⟹ *mk-minsky3 h*
⟨*proof*⟩

2.9 Recursively enumerable sets as Minsky machines

The following is the most complicated lemma of this theory: Given two r.e. sets A and B we want to construct a Minsky machine that reaches the final state 0 for input x if $x \in A$ and final state 1 if $x \in B$, and never reaches either of these states if $x \notin A \cup B$. (If $x \in A \cap B$, then either state 0 or state 1 may be reached.) We consider two r.e. sets rather than one because we target recursive inseparability.

For the r.e. set A, there is a primitive recursive function f such that $x \in A \iff \exists y.\, f(x, y) = 0$. Similarly there is a primitive recursive function g for B such that $x \in B \iff \exists y.\, f(x, y) = 0$. Our Minsky machine takes x in register 0 and y in register 1 (initially 0) and works as follows.

1. evaluate $f(x, y)$; if the result is 0, transition to state 0; otherwise,
2. evaluate $g(x, y)$; if the result is 0, transition to state 1; otherwise,
3. increment y and start over.

lemma *ce-set-pair-by-minsky*:
assumes *A* ∈ *ce-sets B* ∈ *ce-sets*
obtains *M* :: (*nat*, *nat*) *minsky* **where**
finite M deterministic M 0 ∉ *fst* ‘ *M Suc 0* ∉ *fst* ‘ *M*
⋀*x vs*. *vs 0* = *x* ⟹ *vs 1* = *0* ⟹ *x* ∈ *A* ∪ *B* ⟹
∃ *vs′*. ((*2*, *vs*), (*0*, *vs′*)) ∈ (*step M*)* ∨ ((*2*, *vs*), (*Suc 0*, *vs′*)) ∈ (*step M*)*
⋀*x vs vs′*. *vs 0* = *x* ⟹ *vs 1* = *0* ⟹ ((*2*, *vs*), (*0*, *vs′*)) ∈ (*step M*)* ⟹ *x* ∈ *A*
⋀*x vs vs′*. *vs 0* = *x* ⟹ *vs 1* = *0* ⟹ ((*2*, *vs*), (*Suc 0*, *vs′*)) ∈ (*step M*)* ⟹ *x* ∈ *B*
⟨*proof*⟩

For r.e. sets we obtain the following lemma as a special case (taking $B = \varnothing$, and swapping states 1 and 2).

lemma *ce-set-by-minsky*:
assumes *A* ∈ *ce-sets*
obtains *M* :: (*nat*, *nat*) *minsky* **where**
finite M deterministic M 0 ∉ *fst* ‘ *M*

$\bigwedge x\ vs.\ vs\ 0 = x \Longrightarrow vs\ 1 = 0 \Longrightarrow x \in A \Longrightarrow \exists\, vs'.\ ((Suc\ 0,\ vs),\ (0,\ vs')) \in (step\ M)^*$
$\bigwedge x\ vs\ vs'.\ vs\ 0 = x \Longrightarrow vs\ 1 = 0 \Longrightarrow ((Suc\ 0,\ vs),\ (0,\ vs')) \in (step\ M)^* \Longrightarrow x \in A$
⟨*proof*⟩

2.10 Encoding of Minsky machines

So far, Minsky machines have been sets of pairs of states and operations. We now provide an encoding of Minsky machines as natural numbers, so that we can talk about them as r.e. or computable sets. First we encode operations.

primrec *encode-Op* :: (*nat*, *nat*) *Op* ⇒ *nat* **where**
encode-Op (*Dec v s s'*) = *c-pair 0* (*c-pair v* (*c-pair s s'*))
| *encode-Op* (*Inc v s*) = *c-pair 1* (*c-pair v s*)

definition *decode-Op* :: *nat* ⇒ (*nat*, *nat*) *Op* **where**
decode-Op n = (*if c-fst n* = *0*
then Dec (*c-fst* (*c-snd n*)) (*c-fst* (*c-snd* (*c-snd n*))) (*c-snd* (*c-snd* (*c-snd n*)))
else Inc (*c-fst* (*c-snd n*)) (*c-snd* (*c-snd n*)))

lemma *encode-Op-inv* [*simp*]:
decode-Op (*encode-Op x*) = *x*
⟨*proof*⟩

Minsky machines are encoded via lists of pairs of states and operations.

definition *encode-minsky* :: (*nat* × (*nat*, *nat*) *Op*) *list* ⇒ *nat* **where**
encode-minsky M = *list-to-nat* (*map* (λ*x*. *c-pair* (*fst x*) (*encode-Op* (*snd x*))) *M*)

definition *decode-minsky* :: *nat* ⇒ (*nat* × (*nat*, *nat*) *Op*) *list* **where**
decode-minsky n = *map* (λ*n*. (*c-fst n*, *decode-Op* (*c-snd n*))) (*nat-to-list n*)

lemma *encode-minsky-inv* [*simp*]:
decode-minsky (*encode-minsky M*) = *M*
⟨*proof*⟩

Assignments are stored as lists (starting with register 0).

definition *decode-regs* :: *nat* ⇒ (*nat* ⇒ *nat*) **where**
decode-regs n = (λ*i*. *let xs* = *nat-to-list n in if i* < *length xs then nat-to-list n* ! *i else 0*)

The undecidability results talk about Minsky configurations (pairs of Minsky machines and assignments). This means that we do not have to construct any recursive functions that modify Minsky machines (for example in order to initialize variables), keeping the proofs simple.

definition *decode-minsky-state* :: *nat* ⇒ ((*nat*, *nat*) *minsky* × (*nat* ⇒ *nat*)) **where**
decode-minsky-state n = (*set* (*decode-minsky* (*c-fst n*)), (*decode-regs* (*c-snd n*)))

2.11 Undecidablity results

We conclude with some undecidability results. First we show that it is undecidable whether a Minksy machine starting at state 1 terminates in state 0.

definition *minsky-reaching-0* **where**
minsky-reaching-0 = $\{n \mid n\ M\ vs\ vs'.\ (M,\ vs) = \textit{decode-minsky-state}\ n \wedge ((\textit{Suc}\ 0,\ vs),\ (0,\ vs')) \in (\textit{step}\ M)^*\}$

lemma *minsky-reaching-0-not-computable*:
$\neg$ *computable minsky-reaching-0*
⟨*proof*⟩

The remaining results are resursive inseparability results. We start be showing that there is a Minksy machine U with final states 0 and 1 such that it is not possible to recursively separate inputs reaching state 0 from inputs reaching state 1.

lemma *rec-inseparable-0not1-1not0*:
rec-inseparable $\{p.\ 0 \in \textit{nat-to-ce-set}\ p \wedge 1 \notin \textit{nat-to-ce-set}\ p\}$ $\{p.\ 0 \notin \textit{nat-to-ce-set}\ p \wedge 1 \in \textit{nat-to-ce-set}\ p\}$
⟨*proof*⟩

lemma *ce-sets-containing-n-ce*:
$\{p.\ n \in \textit{nat-to-ce-set}\ p\} \in \textit{ce-sets}$
⟨*proof*⟩

lemma *rec-inseparable-fixed-minsky-reaching-0-1*:
obtains $U :: (\textit{nat}, \textit{nat})\ \textit{minsky}$ **where**
finite U deterministic U $0 \notin \textit{fst}\ `\ U\ 1 \notin \textit{fst}\ `\ U$
rec-inseparable $\{x \mid x\ vs'.\ ((2,\ (\lambda n.\ \textit{if}\ n = 0\ \textit{then}\ x\ \textit{else}\ 0)),\ (0,\ vs')) \in (\textit{step}\ U)^*\}$
$\{x \mid x\ vs'.\ ((2,\ (\lambda n.\ \textit{if}\ n = 0\ \textit{then}\ x\ \textit{else}\ 0)),\ (1,\ vs')) \in (\textit{step}\ U)^*\}$
⟨*proof*⟩

Consequently, it is impossible to separate Minsky configurations with deterministic machines and final states 0 and 1 that reach state 0 from those that reach state 1.

definition *minsky-reaching-s* **where**
minsky-reaching-s $s = \{m \mid M\ m\ vs\ vs'.\ (M,\ vs) = \textit{decode-minsky-state}\ m \wedge$
$\textit{deterministic}\ M \wedge 0 \notin \textit{fst}\ `\ M \wedge 1 \notin \textit{fst}\ `\ M \wedge ((2,\ vs),\ (s,\ vs')) \in (\textit{step}\ M)^*\}$

lemma *rec-inseparable-minsky-reaching-0-1*:
rec-inseparable (*minsky-reaching-s 0*) (*minsky-reaching-s 1*)
⟨*proof*⟩

As a corollary, it is impossible to separate Minsky configurations that reach state 0 but not state 1 from those that reach state 1 but not state 0.

definition *minsky-reaching-s-not-t* **where**

minsky-reaching-s-not-t s t = {*m* |*M m vs vs′*. (*M*, *vs*) = *decode-minsky-state m*
∧
((*2*, *vs*), (*s*, *vs′*)) ∈ (*step M*)* ∧ ((*2*, *vs*), (*t*, *vs′*)) ∉ (*step M*)*}

lemma *minsky-reaching-s-imp-minsky-reaching-s-not-t*:
assumes *s* ∈ {*0*,*1*} *t* ∈ {*0*,*1*} *s* ≠ *t*
shows *minsky-reaching-s s* ⊆ *minsky-reaching-s-not-t s t*
⟨*proof*⟩

lemma *rec-inseparable-minsky-reaching-0-not-1-1-not-0*:
rec-inseparable (*minsky-reaching-s-not-t 0 1*) (*minsky-reaching-s-not-t 1 0*)
⟨*proof*⟩

end

References

[1] S. C. Kleene. *Introduction to metamathematics*. North-Holland, 1952.

[2] M. Nedzelsky. Recursion theory I. *Archive of Formal Proofs*, Apr. 2008. http://isa-afp.org/entries/Recursion-Theory-I.html, Formal proof development.

[3] J. B. Rosser. Extensions of some theorems of gödel and church. *Journal of Symbolic Logic*, 1:87–91, 1936.

[4] R. M. Smullyan. Undecidability and recursive inseparability. *Mathematical Logic Quarterly*, 4(7-11):143–147, 1958.