Minsky Machines*

Bertram Felgenhauer

April 18, 2024

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

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

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**
$\textit{rec-inseparable}\ A\ B \equiv \forall X.\ A \subseteq X \wedge B \subseteq -\ X \longrightarrow \neg\ \textit{computable}\ X$

lemma *rec-inseparableI*:
$(\bigwedge X.\ A \subseteq X \Longrightarrow B \subseteq -\ X \Longrightarrow \textit{computable}\ X \Longrightarrow \textit{False}) \Longrightarrow \textit{rec-inseparable}\ A\ B$
unfolding *rec-inseparable-def* **by** *blast*

lemma *rec-inseparableD*:
$\textit{rec-inseparable}\ A\ B \Longrightarrow A \subseteq X \Longrightarrow B \subseteq -\ X \Longrightarrow \textit{computable}\ X \Longrightarrow \textit{False}$
unfolding *rec-inseparable-def* **by** *blast*

Recursive inseperability is symmetric and enjoys a monotonicity property.

lemma *rec-inseparable-symmetric*:
$\textit{rec-inseparable}\ A\ B \Longrightarrow \textit{rec-inseparable}\ B\ A$
unfolding *rec-inseparable-def computable-def* **by** (*metis double-compl*)

lemma *rec-inseparable-mono*:
$\textit{rec-inseparable}\ A\ B \Longrightarrow A \subseteq A' \Longrightarrow B \subseteq B' \Longrightarrow \textit{rec-inseparable}\ A'\ B'$
unfolding *rec-inseparable-def* **by** (*meson subset-trans*)

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 (*intro rec-inseparableI*)
fix X **assume** $A \subseteq X$ $B \subseteq - X$ *computable* X
moreover have *many-reducible-to* $(f -` X)$ X **using** *assms*(*1*)
by (*auto simp*: *many-reducible-to-def many-reducible-to-via-def*)
ultimately have *computable* $(f -` X)$ **and** $(f -` A) \subseteq (f -` X)$ **and** $(f -` B) \subseteq - (f -` X)$
by (*auto dest*!: *m-red-to-comp*)
then show *False* **using** *assms*(*2*) **unfolding** *rec-inseparable-def* **by** *blast*
qed

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*
by (*auto simp*: *rec-inseparable-def*)

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*
by (*auto simp*: *rec-inseparable-def*)

lemma *computable-rec-inseparable-conv*:
computable A $\longleftrightarrow \neg$ *rec-inseparable A* $(- A)$
by (*auto simp*: *computable-def rec-inseparable-def*)

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*
by (*metis c-fst-of-c-pair c-snd-of-c-pair*)

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 (*intro rec-inseparableI*, *goal-cases*)
case (*1 X*)

Note that $[\![$*index-set ?A*; *?A* $\neq \{\}$; *?A* $\neq$ *UNIV*$]\!] \Longrightarrow \neg$ *computable ?A* is not applicable because X may not be an index set.

let *?Q* $= \{q.$ *s-ce q q* $\in X\} \times$ *nat-to-ce-set m* $\cup \{q.$ *s-ce q q* $\in - X\} \times$ *nat-to-ce-set n*
have *?Q* $\in$ *ce-rels*
using *1*(*3*) *ce-set-lm-5 comp2-1*[*OF s-ce-is-pr id1-1 id1-1*] **unfolding** *computable-def*
by (*intro ce-union*[*of ce-rel-to-set - ce-rel-to-set -*, *folded ce-rel-lm-32 ce-rel-lm-8*] *ce-rel-lm-29 nat-to-ce-set-into-ce*) *blast+*
then obtain *q* **where** *nat-to-ce-set q* $= \{$*c-pair q x* $|q\ x.$ $(q, x) \in$ *?Q*$\}$
unfolding *ce-rel-lm-8 ce-rel-to-set-def* **by** (*metis* (*no-types*, *lifting*) *nat-to-ce-set-srj*)
from *eqset-imp-iff*[*OF this*, *of c-pair q -*]
have *nat-to-ce-set* (*s-ce q q*) = (*if s-ce q q* $\in X$ *then nat-to-ce-set m else nat-to-ce-set n*)
by (*auto simp*: *s-lm c-pair-inj′ nat-to-ce-set-def fn-to-set-def pr-conv-1-to-2-def*)
then show *?case* **using** *1*(*1*,*2*)[*THEN subsetD*, *of s-ce q q*] **by** (*auto split*: *if-splits*)
qed

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 $R \Longrightarrow (x,\ y) \in R \Longrightarrow (x,\ z) \in R \Longrightarrow y = z$
by (*auto simp*: *deterministic-def*)

lemma *deterministic-empty* [*simp*]:
deterministic $\{\}$
by (*auto simp*: *deterministic-def*)

lemma *deterministic-singleton* [*simp*]:
deterministic $\{p\}$
by (*auto simp*: *deterministic-def*)

lemma *deterministic-imp-weak-diamond* [*intro*]:
deterministic $R \Longrightarrow w\Diamond\ R$
by (*auto simp*: *weak-diamond-def deterministic-def*)

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* $(S \cup R)$
by (*fastforce simp add*: *deterministic-def disjoint-iff-not-equal*)

lemma *deterministic-map*:
inj-on f (*fst* ‘ R) $\Longrightarrow$ *deterministic* $R \Longrightarrow$ *deterministic* (*map-prod f g* ‘ R)
by (*auto simp add*: *deterministic-def dest*!: *inj-onD*; *force*)

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 $('s,\ 'v)\ Op = Dec\ (op\text{-}var{:}\ 'v)\ 's\ 's \mid Inc\ (op\text{-}var{:}\ 'v)\ 's$

type-synonym $('s,\ 'v)\ minsky = ('s \times ('s,\ 'v)\ Op)\ set$

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* :: $('s,\ 'v)\ minsky \Rightarrow ('s \times ('v \Rightarrow nat))\ rel$ **for** M :: $('s,\ 'v)\ minsky$ **where**
inc: $(s,\ Inc\ v\ s') \in M \Longrightarrow ((s,\ vs),\ (s',\ \lambda x.\ if\ x = v\ then\ Suc\ (vs\ v)\ else\ vs\ x)) \in step\ M$
| *decn*: $(s,\ Dec\ v\ sn\ sz) \in M \Longrightarrow vs\ v = Suc\ n \Longrightarrow ((s,\ vs),\ (sn,\ \lambda x.\ if\ x = v\ then\ n\ else\ vs\ x)) \in step\ M$
| *decz*: $(s,\ Dec\ v\ sn\ sz) \in M \Longrightarrow vs\ v = 0 \Longrightarrow ((s,\ vs),\ (sz,\ vs)) \in step\ M$

lemma *step-mono*:
$M \subseteq M' \Longrightarrow step\ M \subseteq step\ M'$
by (*auto elim*: *step.cases intro*: *step.intros*)

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 −
{ **fix** *s vs s1 vs1 s2 vs2*
assume *s*: $((s,\ vs),\ (s1,\ vs1)) \in step\ M$ $((s,\ vs),\ (s2,\ vs2)) \in step\ M$
have $(s1,\ vs1) = (s2,\ vs2)$ **using** *deterministicD*[*OF assms*]
by (*cases rule*: *step.cases*[*OF s*(*1*)]; *cases rule*: *step.cases*[*OF s*(*2*)]) *fastforce+*
}
then show *?thesis* **by** (*auto simp*: *deterministic-def*)
qed

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

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

by (*auto intro*!: *no-step elim*!: *step.cases simp*: *rev-image-eqI*)

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*
by (*simp add*: *map-minsky-def Op.map-id0 map-prod.id*)

lemma *map-minsky-comp*:
map-minsky f g (*map-minsky f' g' M*) = *map-minsky* ($f \circ f'$) ($g \circ g'$) *M*
unfolding *map-minsky-def image-comp Op.map-comp map-prod.comp comp-def*[*of map-Op - -*] **..**

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'* $\circ$ *g* ((*s*, *vs*), (*t*, *ws*)) $\in$ *step M*
shows ((*f s*, *vs'*), (*f t*, λx. *if* $x \in$ *range g then ws* (*inv g x*) *else vs' x*)) $\in$ *step* (*map-minsky f g M*)
using *assms*(*3*)
proof (*cases rule*: *step.cases*)
case (*inc v*) **note** [*simp*] = *inc*(*1*)
let *?ws'* = λw. *if w* = *g v then Suc* (*vs'* (*g v*)) *else vs' w*
have ((*f s*, *vs'*), (*f t*, *?ws'*)) $\in$ *step* (*map-minsky f g M*)
using *inc*(*2*) *step.inc*[*of f s g v f t map-minsky f g M vs'*]
by (*force simp*: *map-minsky-def*)
moreover have (λx. *if* $x \in$ *range g then ws* (*inv g x*) *else vs' x*) = *?ws'*
using *assms*(*1*,*2*) **by** (*auto intro*!: *ext simp*: *injD image-def*)
ultimately show *?thesis* **by** *auto*
next
case (*decn v sz n*) **note** [*simp*] = *decn*(*1*)
let *?ws'* = λx. *if x* = *g v then n else vs' x*
have ((*f s*, *vs'*), (*f t*, *?ws'*)) $\in$ *step* (*map-minsky f g M*)
using *assms*(*2*) *decn*(*2*−) *step.decn*[*of f s g v f t f sz map-minsky f g M vs' n*]
by (*force simp*: *map-minsky-def*)
moreover have (λx. *if* $x \in$ *range g then ws* (*inv g x*) *else vs' x*) = *?ws'*
using *assms*(*1*,*2*) **by** (*auto intro*!: *ext simp*: *injD image-def*)
ultimately show *?thesis* **by** *auto*
next
case (*decz v sn*) **note** [*simp*] = *decz*(*1*)
have ((*f s*, *vs'*), (*f t*, *vs'*)) $\in$ *step* (*map-minsky f g M*)
using *assms*(*2*) *decz*(*2*−) *step.decz*[*of f s g v f sn f t map-minsky f g M vs'*]

by (*force simp*: *map-minsky-def*)
moreover have $(\lambda x.\ \mathit{if}\ x \in \mathit{range}\ g\ \mathit{then}\ \mathit{ws}\ (\mathit{inv}\ g\ x)\ \mathit{else}\ \mathit{vs}'\ x) = \mathit{vs}'$
using *assms*(*1*,*2*) **by** (*auto intro*!: *ext simp*: *injD image-def*)
ultimately show *?thesis* **by** *auto*
qed

lemma *map-steps*:
assumes $\mathit{inj}\ g\ \mathit{vs} = \mathit{ws} \circ g\ ((s,\ \mathit{vs}),\ (t,\ \mathit{vs}')) \in (\mathit{step}\ M)^*$
shows $((f\ s,\ \mathit{ws}),\ (f\ t,\ \lambda x.\ \mathit{if}\ x \in \mathit{range}\ g\ \mathit{then}\ \mathit{vs}'\ (\mathit{inv}\ g\ x)\ \mathit{else}\ \mathit{ws}\ x)) \in (\mathit{step}\ (\mathit{map\text{-}minsky}\ f\ g\ M))^*$
using *assms*(*3*,*2*)
proof (*induct* (*s*, *vs*) *arbitrary*: *s vs ws rule*: *converse-rtrancl-induct*)
case *base*
then have $(\lambda x.\ \mathit{if}\ x \in \mathit{range}\ g\ \mathit{then}\ \mathit{vs}'\ (\mathit{inv}\ g\ x)\ \mathit{else}\ \mathit{ws}\ x) = \mathit{ws}$
using *assms*(*1*) **by** (*auto intro*!: *ext simp*: *injD image-def*)
then show *?case* **by** *auto*
next
case (*step y*)
have $\mathit{snd}\ y = (\lambda x.\ \mathit{if}\ x \in \mathit{range}\ g\ \mathit{then}\ \mathit{snd}\ y\ (\mathit{inv}\ g\ x)\ \mathit{else}\ \mathit{ws}\ x) \circ g$ (**is** - = *?ys'* ∘ -)
using *assms*(*1*) **by** *auto*
then show *?case* **using** *map-step*[*OF assms*(*1*) *step*(*4*), *of s fst y snd y M f*] *step*(*1*)
step(*3*)[*OF prod.collapse*[*symmetric*], *of ?ys'*] **by** (*auto cong*: *if-cong*)
qed

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 $\mathit{mk\text{-}minsky\text{-}wit} :: (\mathit{nat},\ \mathit{nat})\ \mathit{minsky} \Rightarrow ((\mathit{nat} \Rightarrow \mathit{nat}) \Rightarrow (\mathit{nat} \Rightarrow \mathit{nat}) \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$ **where**
$\mathit{mk\text{-}minsky\text{-}wit}\ M\ P \equiv \mathit{finite}\ M \wedge \mathit{deterministic}\ M \wedge 0 \notin \mathit{fst}\ `\ M \wedge$
$(\forall \mathit{vs}.\ \exists \mathit{vs}'.\ ((\mathit{Suc}\ 0,\ \mathit{vs}),\ (0,\ \mathit{vs}')) \in (\mathit{step}\ M)^* \wedge P\ \mathit{vs}\ \mathit{vs}')$

abbreviation $\mathit{mk\text{-}minsky} :: ((\mathit{nat} \Rightarrow \mathit{nat}) \Rightarrow (\mathit{nat} \Rightarrow \mathit{nat}) \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$ **where**
$\mathit{mk\text{-}minsky}\ P \equiv \exists M.\ \mathit{mk\text{-}minsky\text{-}wit}\ M\ P$

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

lemma *mk-minsky-mono*:
shows $\mathit{mk\text{-}minsky}\ P \Longrightarrow (\bigwedge \mathit{vs}\ \mathit{vs}'.\ P\ \mathit{vs}\ \mathit{vs}' \Longrightarrow Q\ \mathit{vs}\ \mathit{vs}') \Longrightarrow \mathit{mk\text{-}minsky}\ Q$
unfolding *mk-minsky-def* **by** *meson*

lemma *mk-minsky-sound*:
assumes $\mathit{mk\text{-}minsky\text{-}wit}\ M\ P\ ((\mathit{Suc}\ 0,\ \mathit{vs}),\ (0,\ \mathit{vs}')) \in (\mathit{step}\ M)^*$

shows $P\ vs\ vs'$
proof $-$
have M: $deterministic\ M\ 0 \notin fst\ `\ M\ \bigwedge vs.\ \exists vs'.\ ((Suc\ 0,\ vs),\ 0,\ vs') \in (step\ M)^* \wedge P\ vs\ vs'$
using $assms(1)$ **by** $(auto\ simp$: $mk\text{-}minsky\text{-}wit\text{-}def)$
obtain vs'' **where** vs'': $((Suc\ 0,\ vs),\ (0,\ vs'')) \in (step\ M)^*\ P\ vs\ vs''$ **using** $M(3)$ **by** $blast$
have $(0 :: nat,\ vs') = (0,\ vs'')$ **using** $M(1,2)$
by $(intro\ deterministic\text{-}minsky\text{-}UN[OF\ \text{-}\ assms(2)\ vs''(1)])$
then show $?thesis$ **using** $vs''(2)$ **by** $simp$
qed

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 $mk\text{-}minsky1$ **where**
$mk\text{-}minsky1\ f \equiv mk\text{-}minsky\ (\lambda vs\ vs'.\ vs'\ 0 = f\ (vs\ 1))$

abbreviation $mk\text{-}minsky2$ **where**
$mk\text{-}minsky2\ f \equiv mk\text{-}minsky\ (\lambda vs\ vs'.\ vs'\ 0 = f\ (vs\ 1)\ (vs\ 2))$

abbreviation $mk\text{-}minsky3$ **where**
$mk\text{-}minsky3\ f \equiv mk\text{-}minsky\ (\lambda vs\ vs'.\ vs'\ 0 = f\ (vs\ 1)\ (vs\ 2)\ (vs\ 3))$

2.4 Trivial building blocks

We can increment and decrement any register.

lemma $mk\text{-}minsky\text{-}inc$:
shows $mk\text{-}minsky\ (\lambda vs\ vs'.\ vs' = (\lambda x.\ if\ x = v\ then\ Suc\ (vs\ v)\ else\ vs\ x))$
using $step.inc[of\ Suc\ 0\ v\ 0]$
by $(auto\ simp$: $deterministic\text{-}def\ mk\text{-}minsky\text{-}def\ intro!$: $exI[of\ \text{-}\ \{(1,\ Inc\ v\ 0)\} :: (nat,\ nat)\ minsky])$

lemma $mk\text{-}minsky\text{-}dec$:
shows $mk\text{-}minsky\ (\lambda vs\ vs'.\ vs' = (\lambda x.\ if\ x = v\ then\ vs\ v - 1\ else\ vs\ x))$
proof $-$
let $?M = \{(1,\ Dec\ v\ 0\ 0)\} :: (nat,\ nat)\ minsky$
show $?thesis$ **unfolding** $mk\text{-}minsky\text{-}def$
proof $(intro\ exI[of\ \text{-}\ ?M]\ allI\ conjI,\ goal\text{-}cases)$
case $(4\ vs)$
have $[simp]$: $vs\ v = 0 \Longrightarrow (\lambda x.\ if\ x = v\ then\ 0\ else\ vs\ x) = vs$ **by** $auto$
show $?case$ **using** $step.decz[of\ Suc\ 0\ v\ 0\ 0\ ?M]\ step.decn[of\ Suc\ 0\ v\ 0\ 0\ ?M]$
by $(cases\ vs\ v)\ (auto\ cong$: $if\text{-}cong)$
qed $auto$
qed

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 −
  obtain *M* **where** *M*: *finite M deterministic M* $0 \notin fst$ ‘ *M*
    $\bigwedge vs.\ \exists vs'.\ ((Suc\ 0,\ vs),\ 0,\ vs') \in (step\ M)^* \wedge P\ vs\ vs'$
    using *assms*(*1*) **by** (*auto simp*: *mk-minsky-def*)
  obtain *N* **where** *N*: *finite N deterministic N* $0 \notin fst$ ‘ *N*
    $\bigwedge vs.\ \exists vs'.\ ((Suc\ 0,\ vs),\ 0,\ vs') \in (step\ N)^* \wedge Q\ vs\ vs'$
    using *assms*(*2*) **by** (*auto simp*: *mk-minsky-def*)
  let $?fM = \lambda s.\ if\ s = 0\ then\ 2\ else\ if\ s = 1\ then\ 1\ else\ 2 * s + 1$ — M: from 1 to 2
  let $?fN = \lambda s.\ 2 * s$ — N: from 2 to 0
  let $?M = map\text{-}minsky\ ?fM\ g\ M \cup map\text{-}minsky\ ?fN\ g\ N$
  show *?thesis* **unfolding** *mk-minsky-def*
  proof (*intro exI*[*of - ?M*] *conjI allI*, *goal-cases*)
    case *1* **show** *?case* **using** *M*(*1*) *N*(*1*) **by** (*auto simp*: *map-minsky-def*)
  next
    case *2* **show** *?case* **using** *M*(*2*,*3*) *N*(*2*) **unfolding** *map-minsky-def*
      by (*intro deterministic-union deterministic-map*)
        (*auto simp*: *inj-on-def rev-image-eqI Suc-double-not-eq-double split*: *if-splits*)
  next
    case *3* **show** *?case* **using** *N*(*3*) **by** (*auto simp*: *rev-image-eqI map-minsky-def split*: *if-splits*)
  next
    case (*4 vs*)
    obtain *vsM* **where** *M′*: $((Suc\ 0,\ vs \circ g),\ 0,\ vsM) \in (step\ M)^*\ P\ (vs \circ g)\ vsM$
      using *M*(*4*) **by** *blast*
    obtain *vsN* **where** *N′*: $((Suc\ 0,\ vsM),\ 0,\ vsN) \in (step\ N)^*\ Q\ vsM\ vsN$
      using *N*(*4*) **by** *blast*
    note $*$ = *subsetD*[*OF steps-mono*, *of - ?M*]
      map-steps[*OF - - M′*(*1*), *of g vs ?fM*, *simplified*]
      map-steps[*OF - - N′*(*1*), *of g - ?fN*, *simplified*]
    show *?case*
      using *assms*(*3*,*4*) *M′*(*2*) *N′*(*2*) *rtrancl-trans*[*OF* $*$(*1*)[*OF -* $*$(*2*)] $*$(*1*)[*OF -* $*$(*3*)]]
      by (*auto simp*: *comp-def*)
  qed
qed

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*
using *mk-minsky-seq-map*[*OF assms*(*1*,*2*), *of id*] *assms*(*3*) **by** *simp*

lemma *mk-minsky-seq*′:
assumes *mk-minsky P mk-minsky Q*
shows *mk-minsky* $(\lambda vs\ vs''.\ (\exists\, vs'.\ P\ vs\ vs' \wedge Q\ vs'\ vs''))$
by (*intro mk-minsky-seq*[*OF assms*]) *blast*

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

lemma *mk-minsky-nop*:
mk-minsky $(\lambda vs\ vs'.\ vs = vs')$
by (*intro mk-minsky-seq*[*OF mk-minsky-inc mk-minsky-dec*]) *auto*

Renaming variables.

lemma *mk-minsky-map*:
assumes *mk-minsky P inj f*
shows *mk-minsky* $(\lambda vs\ vs'.\ P\ (vs \circ f)\ (vs' \circ f) \wedge (\forall\, x.\ x \notin range\ f \longrightarrow vs\ x = vs'\ x))$
using *mk-minsky-seq-map*[*OF assms*(*1*) *mk-minsky-nop assms*(*2*)] **by** *simp*

lemma *inj-shift* [*simp*]:
fixes $a\ b :: nat$
assumes $a < b$
shows *inj* $(\lambda x.\ if\ x = 0\ then\ a\ else\ x + b)$
using *assms* **by** (*auto simp*: *inj-on-def*)

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* (λ*x. if x* = *v then n else vs x*) *vs′* $\Longrightarrow$ *Q vs′ vs″* $\Longrightarrow$ *Q vs vs″*
shows *mk-minsky Q*
proof −
obtain *M* **where** *M*: *finite M deterministic M 0* $\notin$ *fst ‘ M*
$\bigwedge$*vs.* $\exists$ *vs′*. ((*Suc 0*, *vs*), *0*, *vs′*) $\in$ (*step M*)* $\wedge$ *P vs vs′*
using *assms*(*1*) **by** (*auto simp*: *mk-minsky-def*)
let *?M* = {(*1*, *Dec v 2 0*)} $\cup$ *map-minsky Suc id M*
show *?thesis* **unfolding** *mk-minsky-def*
proof (*intro exI*[*of - ?M*] *conjI allI*, *goal-cases*)
case *1* **show** *?case* **using** *M*(*1*) **by** (*auto simp*: *map-minsky-def*)
next
case *2* **show** *?case* **using** *M*(*2*,*3*) **unfolding** *map-minsky-def*
by (*intro deterministic-union deterministic-map*) (*auto simp*: *rev-image-eqI*)
next
case *3* **show** *?case* **by** (*auto simp*: *map-minsky-def*)
next
case (*4 vs*) **show** *?case*
proof (*induct vs v arbitrary*: *vs*)
case *0* **then show** *?case* **using** *assms*(*3*)[*of vs*] *step.decz*[*of 1 v 2 0 ?M vs*]
by (*auto simp*: *id-def*)
next
case (*Suc n*)
obtain *vs′* **where** *M′*: ((*Suc 0*, λ*x. if x* = *v then n else vs x*), *0*, *vs′*) $\in$ (*step M*)*
P (λ*x. if x* = *v then n else vs x*) *vs′* **using** *M*(*4*) **by** *blast*
obtain *vs″* **where** *D*: ((*Suc 0*, *vs′*), *0*, *vs″*) $\in$ (*step ?M*)* *Q vs′ vs″*
using *Suc*(*1*)[*of vs′*] *assms*(*2*)[*OF M′*(*2*)] **by** *auto*
note $*$ = *subsetD*[*OF steps-mono*, *of - ?M*]
r-into-rtrancl[*OF decn*[*of Suc 0 v 2 0 ?M vs n*]]
map-steps[*OF - - M′*(*1*), *of id - Suc*, *simplified*, *OF refl*, *simplified*, *folded numeral-2-eq-2*]
show *?case* **using** *rtrancl-trans*[*OF rtrancl-trans*, *OF* $*$(*2*) $*$(*1*)[*OF -* $*$(*3*)] *D*(*1*)]
D(*2*) *Suc*(*2*) *assms*(*4*)[*OF - M′*(*2*), *of vs″*] **by** *auto*
qed
qed
qed

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* $(\lambda vs\ vs'.\ vs' = (\lambda x.\ \mathit{if}\ x = v\ \mathit{then}\ 0\ \mathit{else}\ vs\ x))$
by (*intro mk-minsky-loop*[**where** $v = v$, *OF* — while v[v]−−:
mk-minsky-nop]) *auto* — pass

lemma *mk-minsky-add1*:
assumes $v \neq w$
shows *mk-minsky* $(\lambda vs\ vs'.\ vs' = (\lambda x.\ \mathit{if}\ x = v\ \mathit{then}\ 0\ \mathit{else}\ \mathit{if}\ x = w\ \mathit{then}\ vs\ v + vs\ w\ \mathit{else}\ vs\ x))$
using *assms* **by** (*intro mk-minsky-loop*[**where** $v = v$, *OF* — while v[v]−−:
mk-minsky-inc[*of w*]]) *auto* — v[w]++

lemma *mk-minsky-add2*:
assumes $u \neq v$ $u \neq w$ $v \neq w$
shows *mk-minsky* $(\lambda vs\ vs'.\ vs' =$
$(\lambda x.\ \mathit{if}\ x = u\ \mathit{then}\ 0\ \mathit{else}\ \mathit{if}\ x = v\ \mathit{then}\ vs\ u + vs\ v\ \mathit{else}\ \mathit{if}\ x = w\ \mathit{then}\ vs\ u + vs\ w\ \mathit{else}\ vs\ x))$
using *assms* **by** (*intro mk-minsky-loop*[**where** $v = u$, *OF mk-minsky-seq'*[*OF* — while v[u]−−:
mk-minsky-inc[*of v*] — v[v]++
mk-minsky-inc[*of w*]]]) *auto* — v[w]++

lemma *mk-minsky-copy1*:
assumes $v \neq w$
shows *mk-minsky* $(\lambda vs\ vs'.\ vs' = (\lambda x.\ \mathit{if}\ x = v\ \mathit{then}\ 0\ \mathit{else}\ \mathit{if}\ x = w\ \mathit{then}\ vs\ v\ \mathit{else}\ vs\ x))$
using *assms* **by** (*intro mk-minsky-seq*[*OF*
mk-minsky-zero[*of w*] — v[w] := 0
mk-minsky-add1[*of v w*]]) *auto* — v[w] := v[w] + v[v], v[v] := 0

lemma *mk-minsky-copy2*:
assumes $u \neq v$ $u \neq w$ $v \neq w$
shows *mk-minsky* $(\lambda vs\ vs'.\ vs' =$
$(\lambda x.\ \mathit{if}\ x = u\ \mathit{then}\ 0\ \mathit{else}\ \mathit{if}\ x = v\ \mathit{then}\ vs\ u\ \mathit{else}\ \mathit{if}\ x = w\ \mathit{then}\ vs\ u\ \mathit{else}\ vs\ x))$
using *assms* **by** (*intro mk-minsky-seq*[*OF mk-minsky-seq'*, *OF*
mk-minsky-zero[*of v*] — v[v] := 0
mk-minsky-zero[*of w*] — v[w] := 0
mk-minsky-add2[*of u v w*]]) *auto* — v[v] := v[v] + v[u], v[w] := v[w] + v[u], v[u] := 0

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* $vs\ x$))
using *assms* **by** (*intro mk-minsky-seq*[*OF*
mk-minsky-copy2 [*of u v w*] — v[v] := v[u], v[w] := v[u], v[u] := 0
mk-minsky-copy1 [*of w u*]]) *auto* — v[u] := v[w], v[w] := 0

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* $vs\ x$))
using *assms*(*2*−)
by (*intro mk-minsky-seq*[*OF mk-minsky-seq'*[*OF mk-minsky-seq'*], *OF*
mk-minsky-copy[*of u 1 + k k*] — v[1+k] := v[u]
mk-minsky-copy[*of v 2 + k k*] — v[2+k] := v[v]
mk-minsky-copy[*of w 3 + k k*] — v[3+k] := v[w]
mk-minsky-map[*OF assms*(*1*), *of* $\lambda x.$ *if* $x = 0$ *then* t *else* $x + k$]]) (*auto 0 2*)
— v[t] := f v[1+k] v[2+k] v[3+k]

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* ($\lambda x\ y\ z.\ f\ (g\ x\ y\ z)\ (h\ x\ y\ z)\ (k\ x\ y\ z)$)
by (*rule mk-minsky-seq*[*OF mk-minsky-seq'*[*OF mk-minsky-seq'*], *OF*
mk-minsky-apply3 [*OF assms*(*2*), *of 4 7 1 2 3*] — v[4] := g v[1] v[2] v[3]
mk-minsky-apply3 [*OF assms*(*3*), *of 5 7 1 2 3*] — v[5] := h v[1] v[2] v[3]
mk-minsky-apply3 [*OF assms*(*4*), *of 6 7 1 2 3*] — v[6] := k v[1] v[2] v[3]
mk-minsky-apply3 [*OF assms*(*1*), *of 0 7 4 5 6*]]) *auto* — v[0] := f v[4] v[5] v[6]

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*)
by (*intro mk-minsky-seq*[*OF mk-minsky-seq'*, *OF*
mk-minsky-apply3 [*OF assms*(*1*), *of 0 4 2 2 2*] — v[0] := g v[2]
mk-minsky-zero[*of 3*] — v[3] := 0
mk-minsky-loop[**where** $v = 1$, *OF mk-minsky-seq'*, *OF* — while v[1]−−:
mk-minsky-apply3 [*OF assms*(*2*), *of 0 4 3 0 2*] — v[0] := h v[3] v[0] v[2]
mk-minsky-inc[*of 3*], — v[3]++

of λ*vs vs'. vs 0* = *PrimRecOp g h* (*vs 3*) (*vs 2*) ⟶ *vs' 0* = *PrimRecOp g h* (*vs 3* + *vs 1*) (*vs 2*)
]]) *auto*

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 (*goal-cases*)
have ∗: (*f* ∈ *PrimRec1* ⟶ *mk-minsky1 f*) ∧ (*g* ∈ *PrimRec2* ⟶ *mk-minsky2 g*) ∧ (*h* ∈ *PrimRec3* ⟶ *mk-minsky3 h*)
proof (*induction rule*: *PrimRec1-PrimRec2-PrimRec3.induct*)
case *zero* **show** *?case* **by** (*intro mk-minsky-mono*[*OF mk-minsky-zero*]) *auto*
next
case *suc* **show** *?case* **by** (*intro mk-minsky-seq*[*OF mk-minsky-copy1*[*of 1 0*] *mk-minsky-inc*[*of 0*]]) *auto*
next
case *id1-1* **show** *?case* **by** (*intro mk-minsky-mono*[*OF mk-minsky-copy1*[*of 1 0*]]) *auto*
next
case *id2-1* **show** *?case* **by** (*intro mk-minsky-mono*[*OF mk-minsky-copy1*[*of 1 0*]]) *auto*
next
case *id2-2* **show** *?case* **by** (*intro mk-minsky-mono*[*OF mk-minsky-copy1*[*of 2 0*]]) *auto*
next
case *id3-1* **show** *?case* **by** (*intro mk-minsky-mono*[*OF mk-minsky-copy1*[*of 1 0*]]) *auto*
next
case *id3-2* **show** *?case* **by** (*intro mk-minsky-mono*[*OF mk-minsky-copy1*[*of 2 0*]]) *auto*
next
case *id3-3* **show** *?case* **by** (*intro mk-minsky-mono*[*OF mk-minsky-copy1*[*of 3 0*]]) *auto*
next
case (*comp1-1 f g*) **then show** *?case* **using** *mk-minsky-comp3-3* **by** *fast*
next
case (*comp1-2 f g*) **then show** *?case* **using** *mk-minsky-comp3-3* **by** *fast*
next
case (*comp1-3 f g*) **then show** *?case* **using** *mk-minsky-comp3-3* **by** *fast*
next
case (*comp2-1 f g h*) **then show** *?case* **using** *mk-minsky-comp3-3* **by** *fast*
next
case (*comp3-1 f g h k*) **then show** *?case* **using** *mk-minsky-comp3-3* **by** *fast*
next
case (*comp2-2 f g h*) **then show** *?case* **using** *mk-minsky-comp3-3* **by** *fast*
next

case (*comp2-3 f g h*) **then show** *?case* **using** *mk-minsky-comp3-3* **by** *fast*
next
case (*comp3-2 f g h k*) **then show** *?case* **using** *mk-minsky-comp3-3* **by** *fast*
next
case (*comp3-3 f g h k*) **then show** *?case* **using** *mk-minsky-comp3-3* **by** *fast*
next
case (*prim-rec g h*) **then show** *?case* **using** *mk-minsky-prim-rec* **by** *blast*
qed
{ case *1* **thus** *?case* **using** $*$ **by** *blast* **next**
case *2* **thus** *?case* **using** $*$ **by** *blast* **next**
case *3* **thus** *?case* **using** $*$ **by** *blast* **}**
qed

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 $A \in$ *ce-sets* $B \in$ *ce-sets*
obtains M :: (*nat*, *nat*) *minsky* **where**
finite M deterministic M $0 \notin$ *fst* ‘ *M Suc* $0 \notin$ *fst* ‘ *M*
$\bigwedge x\ vs.\ vs\ 0 = x \Longrightarrow vs\ 1 = 0 \Longrightarrow x \in A \cup B \Longrightarrow$
$\exists\, vs'.\ ((2,\ vs),\ (0,\ vs')) \in (step\ M)^* \lor ((2,\ vs),\ (Suc\ 0,\ vs')) \in (step\ M)^*$
$\bigwedge x\ vs\ vs'.\ vs\ 0 = x \Longrightarrow vs\ 1 = 0 \Longrightarrow ((2,\ vs),\ (0,\ vs')) \in (step\ M)^* \Longrightarrow x \in A$
$\bigwedge x\ vs\ vs'.\ vs\ 0 = x \Longrightarrow vs\ 1 = 0 \Longrightarrow ((2,\ vs),\ (Suc\ 0,\ vs')) \in (step\ M)^* \Longrightarrow x \in B$
proof −
obtain g **where** g: $g \in PrimRec2\ \bigwedge x.\ x \in A \longleftrightarrow (\exists\, y.\ g\ x\ y = 0)$
using *assms*(*1*) **by** (*auto simp*: *ce-sets-def fn-to-set-def*)
obtain h **where** h: $h \in PrimRec2\ \bigwedge x.\ x \in B \longleftrightarrow (\exists\, y.\ h\ x\ y = 0)$
using *assms*(*2*) **by** (*auto simp*: *ce-sets-def fn-to-set-def*)

have *mk-minsky* ($\lambda vs\ vs'.\ vs'\ 0 = vs\ 0 \wedge vs'\ 1 = vs\ 1 \wedge vs'\ 2 = g\ (vs\ 0)\ (vs\ 1)$)
using *mk-minsky-seq*[*OF*
mk-minsky-apply3[*OF mk-minsky-PrimRec*(*2*)[*OF g*(*1*)], *of 2 3 0 1 0*] — v[2] := g v[0] v[1]
mk-minsky-nop] **by** *auto* — pass
then obtain M :: (*nat*, *nat*) *minsky* **where** M: *finite M deterministic M* $0 \notin$ *fst* ‘ M
$\bigwedge vs.\ \exists vs'.\ ((Suc\ 0,\ vs),\ 0,\ vs') \in (step\ M)^* \wedge$
$vs'\ 0 = vs\ 0 \wedge vs'\ 1 = vs\ 1 \wedge vs'\ 2 = g\ (vs\ 0)\ (vs\ 1)$
unfolding *mk-minsky-def* **by** *blast*
have *mk-minsky* ($\lambda vs\ vs'.\ vs'\ 0 = vs\ 0 \wedge vs'\ 1 = vs\ 1 + 1 \wedge vs'\ 2 = h\ (vs\ 0)\ (vs\ 1)$)
using *mk-minsky-seq*[*OF*
mk-minsky-apply3[*OF mk-minsky-PrimRec*(*2*)[*OF h*(*1*)], *of 2 3 0 1 0*] — v[2] := h v[0] v[1]
mk-minsky-inc[*of 1*]] **by** *auto* — v[1] := v[1] + 1
then obtain N :: (*nat*, *nat*) *minsky* **where** N: *finite N deterministic N* $0 \notin$ *fst* ‘ N
$\bigwedge vs.\ \exists vs'.\ ((Suc\ 0,\ vs),\ 0,\ vs') \in (step\ N)^* \wedge$
$vs'\ 0 = vs\ 0 \wedge vs'\ 1 = vs\ 1 + 1 \wedge vs'\ 2 = h\ (vs\ 0)\ (vs\ 1)$
unfolding *mk-minsky-def* **by** *blast*
let $?f = \lambda s.\ if\ s = 0\ then\ 3\ else\ 2 * s$ — M: from state 4 to state 3
let $?g = \lambda s.\ 2 * s + 5$ — N: from state 7 to state 5
define X **where** $X = map\text{-}minsky\ ?f\ id\ M \cup map\text{-}minsky\ ?g\ id\ N \cup \{(3,\ Dec\ 2\ 7\ 0)\} \cup \{(5,\ Dec\ 2\ 2\ 1)\}$
have *MX*: *map-minsky ?f id M* $\subseteq X$ **by** (*auto simp*: *X-def*)
have *NX*: *map-minsky ?g id N* $\subseteq X$ **by** (*auto simp*: *X-def*)
have *DX*: $(3,\ Dec\ 2\ 7\ 0) \in X$ $(5,\ Dec\ 2\ 2\ 1) \in X$ **by** (*auto simp*: *X-def*)
have *X1*: *finite X* **using** *M*(*1*) *N*(*1*) **by** (*auto simp*: *map-minsky-def X-def*)
have *X2*: *deterministic X* **unfolding** *X-def* **using** *M*(*2*,*3*) *N*(*2*,*3*)
apply (*intro deterministic-union*)
by (*auto simp*: *map-minsky-def rev-image-eqI inj-on-def split*: *if-splits*
intro!: *deterministic-map*) *presburger*+
have *X3*: $0 \notin$ *fst* ‘ X *Suc* $0 \notin$ *fst* ‘ X **using** *M*(*3*) *N*(*3*)
by (*auto simp*: *X-def map-minsky-def split*: *if-splits*)
have *X4*: $\exists vs'.\ g\ (vs\ 0)\ (vs\ 1) = 0 \wedge ((2,\ vs),\ (0,\ vs')) \in (step\ X)^* \vee$
$h\ (vs\ 0)\ (vs\ 1) = 0 \wedge ((2,\ vs),\ (1,\ vs')) \in (step\ X)^* \vee$
$g\ (vs\ 0)\ (vs\ 1) \neq 0 \wedge h\ (vs\ 0)\ (vs\ 1) \neq 0 \wedge vs'\ 0 = vs\ 0 \wedge vs'\ 1 = vs\ 1 + 1 \wedge$
$((2,\ vs),\ (2,\ vs')) \in (step\ X)^+$ **for** *vs*
proof −
guess vs' **using** *M*(*4*)[*of vs*] **by** (*elim exE conjE*) **note** $vs' = this$
have *1*: $((2,\ vs),\ (3,\ vs')) \in (step\ X)^*$
using *subsetD*[*OF steps-mono*[*OF MX*], *OF map-steps*[*OF - - vs'*(*1*), *of id vs ?f*]] **by** *simp*
show *?thesis*
proof (*cases vs' 2*)
case *0* **then show** *?thesis* **using** *decz*[*OF DX*(*1*), *of vs'*] *vs' 1*

by (*auto intro*: *rtrancl-into-rtrancl*)
next
case (*Suc n*) **note** $Suc' = Suc$
let $?vs = \lambda x.\ if\ x = 2\ then\ n\ else\ vs'\ x$
have *2*: $((2,\ vs),\ (7,\ ?vs)) \in (step\ X)^*$
using *1 decn*[*OF DX*(*1*), *of vs'*] *Suc* **by** (*auto intro*: *rtrancl-into-rtrancl*)
guess vs'' **using** *N*(*4*)[*of ?vs*] **by** (*elim exE conjE*) **note** $vs'' = this$
have *3*: $((2,\ vs),\ (5,\ vs'')) \in (step\ X)^*$
using *2 subsetD*[*OF steps-mono*[*OF NX*], *OF map-steps*[*OF - - vs''*(*1*), *of id ?vs ?g*]] **by** *simp*
show *?thesis*
proof (*cases* vs'' *2*)
case *0* **then show** *?thesis* **using** *3 decz*[*OF DX*(*2*), *of vs''*] $vs''(2-)$ $vs'(2-)$
by (*auto intro*: *rtrancl-into-rtrancl*)
next
case (*Suc m*)
let $?vs = \lambda x.\ if\ x = 2\ then\ m\ else\ vs''\ x$
have *4*: $((2,\ vs),\ (2,\ ?vs)) \in (step\ X)^+$ **using** *3 decn*[*OF DX*(*2*), *of vs'' m*] *Suc* **by** *auto*
then show *?thesis* **using** $vs''(2-)$ $vs'(2-)$ *Suc Suc'* **by** (*auto intro*!: *exI*[*of - ?vs*])
qed
qed
qed
have $*$: $vs\ 1 \le y \Longrightarrow g\ (vs\ 0)\ y = 0 \vee h\ (vs\ 0)\ y = 0 \Longrightarrow$
$\exists vs'.\ ((2,\ vs),\ (0,\ vs')) \in (step\ X)^* \vee ((2,\ vs),\ (1,\ vs')) \in (step\ X)^*$ **for** *vs y*
proof (*induct vs 1 arbitrary*: *vs rule*: *inc-induct*, *goal-cases base step*)
case (*base vs*) **then show** *?case* **using** *X4*[*of vs*] **by** *auto*
next
case (*step vs*)
guess vs' **using** *X4*[*of vs*] **by** (*elim exE*)
then show *?case* **unfolding** *ex-disj-distrib* **using** *step*(*4*) *step*(*3*)[*of vs'*]
by (*auto dest*!: *trancl-into-rtrancl*) (*meson rtrancl-trans*)+
qed
have $**$: $((s,\ vs),\ (t,\ ws)) \in (step\ X)^* \Longrightarrow t \in \{0,\ 1\} \Longrightarrow ((s,\ vs),\ (2,\ ws')) \in (step\ X)^* \Longrightarrow$
$\exists y.\ if\ t = 0\ then\ g\ (ws'\ 0)\ y = 0\ else\ h\ (ws'\ 0)\ y = 0$ **for** *s t vs ws' ws*
proof (*induct arbitrary*: ws' *rule*: *converse-rtrancl-induct2*)
case *refl* **show** *?case* **using** *refl*(*1*) *NF-not-suc*[*OF refl*(*2*) *NF-stepI*] *X3* **by** *auto*
next
case (*step s vs s' vs'*)
show *?case* **using** *step*(*5*)
proof (*cases rule*: *converse-rtranclE*[*case-names base' step'*])
case *base'*
note $***$ = *deterministic-minsky-UN*[*OF X2 - - X3*]
show *?thesis* **using** *X4*[*of ws'*]
proof (*elim exE disjE conjE*, *goal-cases*)
case (*1* vs'') **then show** *?case* **using** *step*(*1*,*2*,*4*) $***$[*of* (*2*,ws') vs'' *ws*]

by (*auto simp*: *base′ intro*: *converse-rtrancl-into-rtrancl*)
next
case (*2 vs″*) **then show** *?case* **using** *step*(*1*,*2*,*4*) ∗∗∗[*of* (*2*,*ws′*) *ws vs″*]
by (*auto simp*: *base′ intro*: *converse-rtrancl-into-rtrancl*)
next
case (*3 vs″*) **then show** *?case* **using** *step*(*2*) *step*(*3*)[*of vs″*, *OF step*(*4*)]
deterministicD[*OF deterministic-stepI*[*OF X2*], *OF - step*(*1*)]
by (*auto simp*: *base′ if-bool-eq-conj trancl-unfold-left*)
qed
next
case (*step′ y*) **then show** *?thesis*
by (*metis deterministicD*[*OF deterministic-stepI*[*OF X2*]] *step*(*1*) *step*(*3*)[*OF step*(*4*)]])
qed
qed
show *?thesis*
proof (*intro that*[*of X*] *X1 X2 X3*, *goal-cases*)
case (*1 x vs*) **then show** *?case* **using** ∗[*of vs*] **by** (*auto simp*: *g*(*2*) *h*(*2*))
next
case (*2 x vs vs′*) **then show** *?case* **using** ∗∗[*of 2 vs 0 vs′ vs*] **by** (*auto simp*: *g*(*2*) *h*(*2*))
next
case (*3 x vs vs′*) **then show** *?case* **using** ∗∗[*of 2 vs 1 vs′ vs*] **by** (*auto simp*: *g*(*2*) *h*(*2*))
qed
qed

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 $A \in$ *ce-sets*
obtains M :: (*nat*, *nat*) *minsky* **where**
finite M deterministic M $0 \notin$ *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 −
guess *M* **using** *ce-set-pair-by-minsky*[*OF assms*(*1*) *ce-empty*] **. note** *M* = *this*
let *?f* = λ*s*. *if s* = *1 then 2 else if s* = *2 then 1 else s* — swap states 1 and 2
have *?f* ∘ *?f* = *id* **by** *auto*
define *N* **where** *N* = *map-minsky ?f id M*
have *M-def*: *M* = *map-minsky ?f id N*
unfolding *N-def map-minsky-comp* ‹*?f* ∘ *?f* = *id*› *map-minsky-id o-id* **..**
show *?thesis* **using** *M*(*1*−*3*)
proof (*intro that*[*of N*], *goal-cases*)
case (*4 x vs*) **show** *?case* **using** *M*(*5*)[*OF 4*(*4*,*5*)] *4*(*6*) *M*(*7*)[*OF 4*(*4*,*5*)]
map-steps[*of id vs vs 2 0 - M ?f*] **by** (*auto simp*: *N-def*)
next

case (*5 x vs vs'*) **show** *?case*
using *M(6)[OF 5(4,5)] 5(6) map-steps[of id vs vs 1 0 - N ?f]* **by** (*auto simp*: *M-def*)
qed (*auto simp*: *N-def map-minsky-def inj-on-def rev-image-eqI deterministic-map split*: *if-splits*)
qed

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*
by (*cases x*) (*auto simp*: *decode-Op-def*)

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*
by (*auto simp*: *encode-minsky-def decode-minsky-def comp-def*)

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

lemma *minsky-reaching-0-not-computable*:
$\neg$ *computable minsky-reaching-0*
proof –
guess *U* **using** *ce-set-by-minsky*[*OF univ-is-ce*] **. note** *U = this*
obtain *us* **where** [*simp*]: *set us = U* **using** *finite-list*[*OF U(1)*] **by** *blast*
let *?f* = λn. *c-pair* (*encode-minsky us*) (*c-cons n 0*)
have *?f* $\in$ *PrimRec1*
using *comp2-1*[*OF c-pair-is-pr const-is-pr comp2-1*[*OF c-cons-is-pr id1-1 const-is-pr*]] **by** *simp*
moreover have *?f x* $\in$ *minsky-reaching-0* $\longleftrightarrow$ *x* $\in$ *univ-ce* **for** *x*
using *U(4,5)*[*of* λi. *if i = 0 then x else 0*]
by (*auto simp*: *minsky-reaching-0-def decode-minsky-state-def decode-regs-def c-cons-def cong*: *if-cong*)
ultimately have *many-reducible-to univ-ce minsky-reaching-0*
by (*auto simp*: *many-reducible-to-def many-reducible-to-via-def dest*: *pr-is-total-rec*)
then show *?thesis* **by** (*rule many-reducible-lm-1*)
qed

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 $\{p.\ 0 \in nat\text{-}to\text{-}ce\text{-}set\ p \wedge 1 \notin nat\text{-}to\text{-}ce\text{-}set\ p\}$ $\{p.\ 0 \notin nat\text{-}to\text{-}ce\text{-}set\ p \wedge 1 \in nat\text{-}to\text{-}ce\text{-}set\ p\}$
proof –
obtain *n* **where** *n*: *nat-to-ce-set n* = $\{0\}$ **using** *nat-to-ce-set-srj*[*OF ce-finite*[*of* $\{0\}$]] **by** *auto*
obtain *m* **where** *m*: *nat-to-ce-set m* = $\{1\}$ **using** *nat-to-ce-set-srj*[*OF ce-finite*[*of* $\{1\}$]] **by** *auto*
show *?thesis* **by** (*rule rec-inseparable-mono*[*OF Rice-rec-inseparable*[*of n m*]]) (*auto simp*: *n m*)
qed

lemma *ce-sets-containing-n-ce*:
$\{p.\ n \in nat\text{-}to\text{-}ce\text{-}set\ p\} \in ce\text{-}sets$
using *ce-set-lm-5*[*OF univ-is-ce comp2-1*[*OF c-pair-is-pr id1-1 const-is-pr*[*of n*]]]
by (*auto simp*: *univ-ce-lm-1*)

lemma *rec-inseparable-fixed-minsky-reaching-0-1*:
obtains U :: (*nat*, *nat*) *minsky* **where**
finite U deterministic U 0 $\notin$ *fst* ‘ *U 1* $\notin$ *fst* ‘ *U*
rec-inseparable $\{x \mid x\ vs'.\ ((2, (\lambda n.\ if\ n = 0\ then\ x\ else\ 0)), (0, vs')) \in (step\ U)^*\}$
$\{x \mid x\ vs'.\ ((2, (\lambda n.\ if\ n = 0\ then\ x\ else\ 0)), (1, vs')) \in (step\ U)^*\}$
proof −
guess U **using** *ce-set-pair-by-minsky*[*OF ce-sets-containing-n-ce ce-sets-containing-n-ce*, *of 0 1*] **.**
from *this*(*1*−*4*) *this*(*5*−*7*)[*of* $\lambda n.$ *if n* = *0 then - else 0*]
show *?thesis* **by** (*auto 0 0 intro*!: *that*[*of U*] *rec-inseparable-mono*[*OF rec-inseparable-0not1-1not0*]
pr-is-total-rec simp: *rev-image-eqI cong*: *if-cong*) *meson*+
qed

Consequently, it is impossible to separate Minsky configurations with deter-mistic 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 = $\{m \mid M\ m\ vs\ vs'.\ (M, vs) = decode\text{-}minsky\text{-}state\ m \wedge$
$deterministic\ M \wedge 0 \notin fst\ `\ M \wedge 1 \notin fst\ `\ M \wedge ((2, vs), (s, vs')) \in (step\ M)^*\}$

lemma *rec-inseparable-minsky-reaching-0-1*:
rec-inseparable (*minsky-reaching-s 0*) (*minsky-reaching-s 1*)
proof −
guess U **using** *rec-inseparable-fixed-minsky-reaching-0-1* **. note** U = *this*
obtain *us* **where** [*simp*]: *set us* = *U* **using** *finite-list*[*OF U*(*1*)] **by** *blast*
let *?f* = $\lambda n.$ *c-pair* (*encode-minsky us*) (*c-cons n 0*)
have *?f* $\in$ *PrimRec1*
using *comp2-1*[*OF c-pair-is-pr const-is-pr comp2-1*[*OF c-cons-is-pr id1-1 const-is-pr*]]
by *simp*
then show *?thesis*
using *U*(*1*−*4*) *rec-inseparable-many-reducible*[*of ?f*, *OF - rec-inseparable-mono*[*OF U*(*5*)]]
by (*auto simp*: *pr-is-total-rec minsky-reaching-s-def decode-minsky-state-def rev-image-eqI*
decode-regs-def c-cons-def cong: *if-cong*)
qed

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 = $\{m \mid M\ m\ vs\ vs'.\ (M, vs) = decode\text{-}minsky\text{-}state\ m\ \wedge$
$((2, vs), (s, vs')) \in (step\ M)^* \wedge ((2, vs), (t, vs')) \notin (step\ M)^*\}$

lemma *minsky-reaching-s-imp-minsky-reaching-s-not-t*:
assumes $s \in \{0,1\}$ $t \in \{0,1\}$ $s \neq t$
shows *minsky-reaching-s s* $\subseteq$ *minsky-reaching-s-not-t s t*
proof −

have [*dest*!]: $((2,\ vs),\ (0,\ vs')) \notin (step\ M)^* \vee ((2,\ vs),\ (1,\ vs')) \notin (step\ M)^*$
if *deterministic* $M\ 0 \notin fst$ ' $M\ 1 \notin fst$ ' M **for** M :: (*nat*, *nat*) *minsky* **and** *vs* *vs'*
using *deterministic-minsky-UN*[*OF that*(*1*) - - *that*(*2*,*3*)] **by** *auto*
show *?thesis* **using** *assms*
by (*auto simp*: *minsky-reaching-s-def minsky-reaching-s-not-t-def rev-image-eqI*)
qed

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*)
by (*intro rec-inseparable-mono*[*OF rec-inseparable-minsky-reaching-0-1*]
minsky-reaching-s-imp-minsky-reaching-s-not-t) *simp-all*

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.