Sound and Complete Sort Encodings for First-Order Logic

Jasmin Christian Blanchette and Andrei Popescu

Abstract

This is a formalization of the soundness and completeness properties for various efficient encodings of sorts in unsorted first-order logic used by Isabelle's Sledgehammer tool.

The results are reported in [1, §2,3], and the formalization itself is presented in [2, §3–5]. The encodings proceed as follows: a many-sorted problem is decorated with (as few as possible) tags or guards that make the problem monotonic; then sorts can be soundly erased. The proofs rely on monotonicity criteria recently introduced by Claessen, Lillieström, and Smallbone [3].

The development employs a formalization of many-sorted first-order logic in clausal form (clauses, structures, and the basic properties of the satisfaction relation), which could be of interest as the starting point for other formalizations of first-order logic metatheory.

References

[1] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. In N. Piterman and S. Smolka, editors, *TACAS 2013*, volume 7795 of *LNCS*, pages 493–507. Springer, 2013.

[2] J. C. Blanchette and A. Popescu. Mechanizing the metatheory of sledgehammer. To be presented at FroCoS 2013.

[3] K. Claessen, A. Lillieström, and N. Smallbone. Sort it out with monotonicity—Translating between many-sorted and unsorted first-order logic. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE-23*, volume 6803 of *LNAI*, pages 207–221. Springer, 2011.

Contents

1 Preliminaries

theory *Preliminaries*
imports *HOL−Cardinals.Cardinals*
HOL−Library.Countable-Set-Type
begin

1.1 Miscelanea

A fixed countable universe for interpreting countable models:

datatype *univ* = *UU nat*

lemma *infinite-univ*[*simp*]: *infinite* (*UNIV* :: *univ set*)
unfolding *infinite-iff-card-of-nat card-of-ordLeq*[*symmetric*]
unfolding *inj-on-def* **by** *auto*

lemma *countable-univ*[*simp*]: *countable* (*UNIV* :: *univ set*)
unfolding *countable-card-of-nat* **apply**(*rule surj-imp-ordLeq*[*of - UU*])
by (*metis subset-UNIV surj-def univ.exhaust*)

Picking an element from a nonempty set (Hilbert choice for sets):

definition *pick X* ≡ *SOME x. x* ∈ *X*

lemma *pick*[*simp*]: *x* ∈ *X* ⟹ *pick X* ∈ *X*
unfolding *pick-def* **by** (*metis someI-ex*)

lemma *pick-NE*[*simp*]: *X* ≠ {} ⟹ *pick X* ∈ *X* **by** *auto*

definition *sappend* (**infix** ‹@@› *60*) **where**
Al @@ Bl = {*al* @ *bl* | *al bl. al* ∈ *Al* ∧ *bl* ∈ *Bl*}

lemma *sappend-NE*[*simp*]: *A @@ B* ≠ {} ⟷ *A* ≠ {} ∧ *B* ≠ {}
unfolding *sappend-def* **by** *auto*

abbreviation *fst3* :: *'a* ∗ *'b* ∗ *'c* ⇒ *'a* **where** *fst3 abc* ≡ *fst abc*
abbreviation *snd3 abc* ≡ *fst* (*snd abc*)
abbreviation *trd3 abc* ≡ *snd* (*snd abc*)

hide-const *int*

abbreviation *any* ≡ *undefined*

Non-emptyness of predicates:

abbreviation (*input*) *NE* φ ≡ ∃ *a.* φ *a*

lemma *NE-NE*: *NE NE*
apply(*rule exI*[*of* - λ *a. True*]) **by** *auto*

lemma *length-Suc-0*:
length al = *Suc 0* ⟷ (∃ *a. al* = [*a*])
by (*metis* (*lifting*) *length-0-conv length-Suc-conv*)

1.2 List combinators

lemmas *list-all2-length* = *list-all2-conv-all-nth*
lemmas *list-eq-iff* = *list-eq-iff-nth-eq*
lemmas *list-all-iff*
lemmas *list-all-length*

definition *singl a* = [*a*]

lemma *length-singl*[*simp*]: *length* (*singl a*) = *Suc 0*
unfolding *singl-def* **by** *simp*

lemma *hd-singl*[*simp*]: *hd* (*singl a*) = *a*
unfolding *singl-def* **by** *simp*

lemma *hd-o-singl*[*simp*]: *hd o singl* = *id*
unfolding *comp-def fun-eq-iff* **by** *simp*

lemma *singl-hd*[*simp*]: *length al* = *Suc 0* ⟹ *singl* (*hd al*) = *al*
unfolding *singl-def* **by** (*cases al*, *auto*)

lemma *singl-inj*[*simp*]: *singl a* = *singl b* ⟷ *a* = *b*
unfolding *singl-def* **by** *simp*

definition *list A* ≡ *SOME al*. *distinct al* ∧ *set al* = *A*

lemma *distinct-set-list*:
finite A ⟹ *distinct* (*list A*) ∧ *set* (*list A*) = *A*
unfolding *list-def* **apply**(*rule someI-ex*) **using** *finite-distinct-list* **by** *auto*

lemmas *distinct-list*[*simp*] = *distinct-set-list*[*THEN conjunct1*]
lemmas *set-list*[*simp*] = *distinct-set-list*[*THEN conjunct2*]

lemma *set-list-set*[*simp*]: *set* (*list* (*set xl*)) = *set xl* **by** *auto*

lemma *length-list*[*simp*]: *finite A* ⟹ *length* (*list A*) = *card A*
by (*metis distinct-card distinct-set-list*)

lemma *list-all-mp*[*elim*]:
assumes *list-all* (λ *a*. *φ a* ⟶ *ψ a*) *al* **and** *list-all φ al*
shows *list-all ψ al*
using *assms* **unfolding** *list-all-iff* **by** *auto*

lemma *list-all-map*:
list-all φ (*map f al*) = *list-all* (*φ o f*) *al*
by (*metis* (*no-types*) *length-map list-all-length nth-map o-def*)

lemma *list-Emp*[*simp*]: *list* {} = []
by (*metis set-empty2 set-list-set*)

lemma *distinct-set-eq-Singl*[*simp*]: *distinct al* ⟹ *set al* = {*a*} ⟷ *al* = [*a*]
apply(*cases al*, *simp*)
by (*metis* (*lifting*) *List.set-simps distinct.simps*
distinct-singleton empty-not-insert insert-eq-iff set-empty2)

lemma *list-Singl*[*simp*]: *list* {*b*} = [*b*]
unfolding *list-def* **apply**(*rule some-equality*) **by** *auto*

lemma *list-insert*:
assumes *A*: *finite A* **and** *b*: *b* ∉ *A*
shows
∃ *al1 al2*.
A = *set* (*al1* @ *al2*) ∧ *distinct* (*al1* @ [*b*] @ *al2*) ∧
list (*insert b A*) = *al1* @ [*b*] @ *al2*
proof−

have $b \in set\ (list\ (insert\ b\ A))$ **using** A **by** *auto*
then obtain *al1 al2* **where** *0*: $list\ (insert\ b\ A) = al1\ @\ [b]\ @\ al2$
by (*metis append-Cons eq-Nil-appendI split-list-last*)
hence *1*: $distinct\ (al1\ @\ [b]\ @\ al2)$ **using** A
by (*metis distinct-set-list finite-insert*)
hence $b \notin set\ (al1\ @\ al2)$ **by** *simp*
moreover have $insert\ b\ A = insert\ b\ (set\ (al1\ @\ al2))$
using *0 set-list*[*OF finite.insertI*[*OF A, of b*]] **by** *auto*
ultimately have $A = set\ (al1\ @\ al2)$ **using** b **by** *auto*
thus *?thesis* **using** *0 1* **by** *auto*
qed

lemma *list-all-list*[*simp*]:
assumes *finite A* **shows** $list\text{-}all\ \varphi\ (list\ A) \longleftrightarrow (\forall\ a \in A.\ \varphi\ a)$
using *assms* **unfolding** *list-all-iff* **by** *simp*

lemma *list-ex-list*[*simp*]:
$finite\ A \Longrightarrow list\text{-}ex\ \varphi\ (list\ A) = (\exists\, a{\in}A.\ \varphi\ a)$
unfolding *list-ex-iff* **by** *simp*

list update:

fun *lupd* **where**
$lupd\ Nil\ Nil\ F = F$
|
$lupd\ (a\ \#\ al)\ (b\ \#\ bl)\ F = lupd\ al\ bl\ (F(a := b))$
|
$lupd\ \text{-}\ \text{-}\ F = any$

lemma *set-lupd*:
assumes $a \in set\ al \vee F1\ a = F2\ a$
shows $lupd\ al\ bl\ F1\ a = lupd\ al\ bl\ F2\ a$
using *assms* **apply**(*induct arbitrary*: *F1 F2 rule*: *list-induct2'*) **by** *auto*

lemma *lupd-map*:
assumes $length\ al = length\ bl$ **and** $a1 \in set\ al \vee G\ a1 = F\ (H\ a1)$
shows $lupd\ al\ (map\ F\ bl)\ G\ a1 = F\ (lupd\ al\ bl\ H\ a1)$
using *assms* **apply** (*induct arbitrary*: *F G H rule*: *list-induct2*) **by** *auto*

lemma *nth-map2*[*simp*]:
assumes $length\ bl = length\ al$ **and** $i < length\ al$
shows $(map2\ f\ al\ bl)\ !\ i = f\ (al!i)\ (bl!i)$
using *assms* **by** *auto*

lemma *list-all2-Nil-iff*:
assumes *list-all2 R xs ys*
shows $xs = [] \longleftrightarrow ys = []$
using *assms* **by** (*cases xs*, *cases ys*) *auto*

lemma *list-all2-NilL*[*simp*]:

list-all2 R [] *ys* ⟷ *ys* = []
using *list-all2-Nil-iff* **by** *auto*

lemma *list-all2-NilR*[*simp*]:
list-all2 R xs [] ⟷ *xs* = []
using *list-all2-Nil-iff* **by** *auto*

lemma *list-all2-ConsL*:
assumes *list-all2 R* (*x* # *xs'*) *ys*
shows ∃ *y ys'*. *ys* = *y* # *ys'* ∧ *R x y* ∧ *list-all2 R xs' ys'*
using *assms* **by**(*cases ys*) *auto*

lemma *list-all2-elimL*[*elim*, *consumes 2*, *case-names Cons*]:
assumes *xs*: *xs* = *x* # *xs'* **and** *h*: *list-all2 R xs ys*
and *Cons*: ⋀ *y ys'*. ⟦*ys* = *y* # *ys'*; *R x y*; *list-all2 R xs' ys'*⟧ ⟹ *phi*
shows *phi*
using *list-all2-ConsL assms* **by** *metis*

lemma *list-all2-elimL2*[*elim*, *consumes 1*, *case-names Cons*]:
assumes *h*: *list-all2 R* (*x* # *xs'*) *ys*
and *Cons*: ⋀ *y ys'*. ⟦*ys* = *y* # *ys'*; *R x y*; *list-all2 R xs' ys'*⟧ ⟹ *phi*
shows *phi*
using *list-all2-ConsL assms* **by** *metis*

lemma *list-all2-ConsR*:
assumes *list-all2 R xs* (*y* # *ys'*)
shows ∃ *x xs'*. *xs* = *x* # *xs'* ∧ *R x y* ∧ *list-all2 R xs' ys'*
using *assms* **by**(*cases xs*) *auto*

lemma *list-all2-elimR*[*elim*, *consumes 2*, *case-names Cons*]:
assumes *ys*: *ys* = *y* # *ys'* **and** *h*: *list-all2 R xs ys*
and *Cons*: ⋀ *x xs'*. ⟦*xs* = *x* # *xs'*; *R x y*; *list-all2 R xs' ys'*⟧ ⟹ *phi*
shows *phi*
using *list-all2-ConsR assms* **by** *metis*

lemma *list-all2-elimR2*[*elim*, *consumes 1*, *case-names Cons*]:
assumes *h*: *list-all2 R xs* (*y* # *ys'*)
and *Cons*: ⋀ *x xs'*. ⟦*xs* = *x* # *xs'*; *R x y*; *list-all2 R xs' ys'*⟧ ⟹ *phi*
shows *phi*
using *list-all2-ConsR assms* **by** *metis*

lemma *ex-list-all2*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ ∃ *y*. *f x y*
shows ∃ *ys*. *list-all2 f xs ys*
using *assms* **apply**(*induct xs*)
apply *fastforce*
by (*metis set-simps insertCI list-all2-Cons*)

lemma *list-all2-cong*[*fundef-cong*]:

assumes *xs1* = *ys1* **and** *xs2* = *ys2*
and $\bigwedge$ *i* . *i* < *length xs2* $\Longrightarrow$ *R* (*xs1*!*i*) (*xs2*!*i*) $\longleftrightarrow$ *R′* (*ys1*!*i*) (*ys2*!*i*)
shows *list-all2 R xs1 xs2* $\longleftrightarrow$ *list-all2 R′ ys1 ys2*
by (*metis assms list-all2-length*)

lemma *list-all2-o*: *list-all2* (*P o f*) *al bl* = *list-all2 P* (*map f al*) *bl*
unfolding *list-all2-map1 comp-def* **..**

lemma *set-size-list*:
assumes *x* ∈ *set xs*
shows *f x* ≤ *size-list f xs*
by (*metis assms size-list-estimation′ order-eq-refl*)

lemma *nth-size-list*:
assumes *i* < *length xs*
shows *f* (*xs*!*i*) ≤ *size-list f xs*
by (*metis assms nth-mem set-size-list*)

lemma *list-all2-list-all*[*simp*]:
list-all2 (λ *x*. *f*) *xs ys* $\longleftrightarrow$
length xs = *length ys* ∧ *list-all f ys*
by (*metis list-all2-length list-all-length*)

lemma *list-all2-list-allR*[*simp*]:
list-all2 (λ *x y*. *f x*) *xs ys* $\longleftrightarrow$
length xs = *length ys* ∧ *list-all f xs*
by (*metis* (*lifting*) *list-all2-length list-all-length*)

lemma *list-all2-list-all-2*[*simp*]:
list-all2 f xs xs $\longleftrightarrow$ *list-all* (λ *x*. *f x x*) *xs*
by (*auto simp add*: *list-all2-iff list-all-iff zip-same*)

lemma *list-all2-map-map*:
list-all2 φ (*map f Tl*) (*map g Tl*) =
list-all (λ*T*. φ (*f T*) (*g T*)) *Tl*
unfolding *list-all2-map1 list-all2-map2 list-all2-list-all-2* **..**

lemma *length-map2*[*simp*]:
assumes *length ys* = *length xs*
shows *length* (*map2 f xs ys*) = *length xs*
using *assms* **by** *auto*

lemma *listAll2-map2I*[*intro?*]:
assumes *length xs* = *length ys*
and $\bigwedge$ *i*. *i* < *length xs* $\Longrightarrow$ *R* (*xs*!*i*) (*f* (*xs*!*i*) (*ys*!*i*))
shows *list-all2 R xs* (*map2 f xs ys*)
apply(*rule list-all2I*) **using** *assms* **unfolding** *set-zip* **by** *auto*

lemma *set-incl-pred*:
$A \leq B \longleftrightarrow (\forall\ a.\ A\ a \longrightarrow B\ a)$
by (*metis predicate1D predicate1I*)

lemma *set-incl-pred2*:
$A \leq B \longleftrightarrow (\forall\ a1\ a2.\ A\ a1\ a2 \longrightarrow B\ a1\ a2)$
by (*metis predicate2I rev-predicate2D*)

lemma *set-incl-pred3*:
$A \leq B \longleftrightarrow (\forall\ a1\ a2\ a3.\ A\ a1\ a2\ a3 \longrightarrow B\ a1\ a2\ a3)$ (**is** - $\longleftrightarrow$ *?R*)
proof−
have $A \leq B \longleftrightarrow (\forall\ a1.\ A\ a1 \leq B\ a1)$ **by** (*metis le-funD le-funI*)
also have ... $\longleftrightarrow$ *?R* **apply**(*rule iff-allI*)
unfolding *set-incl-pred2* **..**
finally show *?thesis* **.**
qed

lemma *set-incl-pred4*:
$A \leq B \longleftrightarrow (\forall\ a1\ a2\ a3\ a4.\ A\ a1\ a2\ a3\ a4 \longrightarrow B\ a1\ a2\ a3\ a4)$ (**is** - $\longleftrightarrow$ *?R*)
proof−
have $A \leq B \longleftrightarrow (\forall\ a1.\ A\ a1 \leq B\ a1)$ **by** (*metis le-funD le-funI*)
also have ... $\longleftrightarrow$ *?R* **apply**(*rule iff-allI*)
unfolding *set-incl-pred3* **..**
finally show *?thesis* **.**
qed

lemma *list-all-mono*:
assumes $phi \leq chi$
shows *list-all phi* $\leq$ *list-all chi*
using *assms* **unfolding** *set-incl-pred list-all-iff* **by** *auto*

lemma *list-all2-mono*:
assumes $phi \leq chi$
shows *list-all2 phi* $\leq$ *list-all2 chi*
using *assms* **by** (*metis* (*full-types*) *list-all2-mono set-incl-pred2*)

1.3 Variables

The type of variables:

datatype *var* = *Variable nat*

lemma *card-of-var*: $|UNIV::var\ set| =o\ natLeq$
proof−
have $|UNIV::var\ set| =o\ |UNIV::nat\ set|$
apply(*rule ordIso-symmetric*,*rule card-of-ordIsoI*)
unfolding *bij-betw-def inj-on-def surj-def* **using** *var.exhaust* **by** *auto*
also have $|UNIV::nat\ set| =o\ natLeq$ **by**(*rule card-of-nat*)
finally show *?thesis* **.**

qed

lemma *infinite-var*[*simp*]: *infinite* (*UNIV* :: *var set*)
using *card-of-var* **by** (*rule ordIso-natLeq-infinite1*)

lemma *countable-var*: *countable* (*UNIV* :: *var set*)
proof−
have *0*: (*UNIV* :: *var set*) ≠ {} **by** *simp*
show *?thesis* **unfolding** *countable-card-of-nat* **unfolding** *card-of-ordLeq2*[*symmetric*, *OF 0*]
apply(*rule exI*[*of - Variable*]) **unfolding** *image-def* **apply** *auto* **by** (*case-tac x*, *auto*)
qed

lemma *countable-infinite*:
assumes *A*: *countable A* **and** *B*: *infinite B*
shows |*A*| ≤*o* |*B*|
proof−
have |*A*| ≤*o natLeq* **using** *A* **unfolding** *countable-card-le-natLeq* **.**
also have *natLeq* ≤*o* |*B*| **by** (*metis B infinite-iff-natLeq-ordLeq*)
finally show *?thesis* **.**
qed

definition *part12-pred V V1-V2* ≡
V = *fst V1-V2* ∪ *snd V1-V2* ∧ *fst V1-V2* ∩ *snd V1-V2* = {} ∧
infinite (*fst V1-V2*) ∧ *infinite* (*snd V1-V2*)

definition *part12 V* ≡ *SOME V1-V2*. *part12-pred V V1-V2*
definition *part1* = *fst o part12* **definition** *part2* = *snd o part12*

lemma *part12-pred*:
assumes *infinite* (*V*::*'a set*) **shows** ∃ *V1-V2*. *part12-pred V V1-V2*
proof−
obtain *f* :: *nat* ⇒ *'a* **where** *f*: *inj f* **and** *r*: *range f* ⊆ *V*
using *assms* **by** (*metis infinite-iff-countable-subset*)
let *?u* = λ *k*::*nat*. *2* ∗ *k* **let** *?v* = λ *k*::*nat*. *Suc* (*2* ∗ *k*)
let *?A* = *?u* ‘ *UNIV* **let** *?B* = *?v* ‘ *UNIV*
have *0*: *inj ?u* ∧ *inj ?v* **unfolding** *inj-on-def* **by** *auto*
hence *1*: *infinite ?A* ∧ *infinite ?B* **using** *finite-imageD* **by** *auto*
let *?V1* = *f* ‘ *?A* **let** *?V2* = *V* − *?V1*
have *V* = *?V1* ∪ *?V2* ∧ *?V1* ∩ *?V2* = {} **using** *r* **by** *blast*
moreover have *infinite ?V1* **using** *1 f*
by (*metis finite-imageD subset-inj-on top-greatest*)
moreover
{have *infinite* (*f* ‘ *?B*) **using** *1 f*
by (*metis finite-imageD subset-inj-on top-greatest*)

moreover have *f ‘ ?B ⊆ ?V2* **using** *r f* **by** (*auto simp*: *inj-eq*) *arith*
ultimately have *infinite ?V2* **by** (*metis infinite-super*)
}
ultimately show *?thesis* **unfolding** *part12-pred-def*
by (*intro exI*[*of* - (*?V1*,*?V2*)]) *auto*
qed

lemma *part12*: **assumes** *infinite V* **shows** *part12-pred V* (*part12 V*)
using *part12-pred*[*OF assms*] **unfolding** *part12-def* **by**(*rule someI-ex*)

lemma *part1-Un-part2*: *infinite V* ⟹ *part1 V* ∪ *part2 V* = *V*
using *part12* **unfolding** *part1-def part2-def part12-pred-def* **by** *auto*

lemma *part1-Int-part2*: *infinite V* ⟹ *part1 V* ∩ *part2 V* = {}
using *part12* **unfolding** *part1-def part2-def part12-pred-def* **by** *auto*

lemma *infinite-part1*: *infinite V* ⟹ *infinite* (*part1 V*)
using *part12* **unfolding** *part1-def part12-pred-def* **by** *auto*

lemma *part1-su*: *infinite V* ⟹ *part1 V* ⊆ *V*
using *part1-Un-part2* **by** *auto*

lemma *infinite-part2*: *infinite V* ⟹ *infinite* (*part2 V*)
using *part12* **unfolding** *part2-def part12-pred-def* **by** *auto*

lemma *part2-su*: *infinite V* ⟹ *part2 V* ⊆ *V*
using *part1-Un-part2* **by** *auto*

end

2 Syntax of Terms and Clauses

theory *TermsAndClauses*
imports *Preliminaries*
begin

These are used for both unsorted and many-sorted FOL, the difference being that, for the latter, the signature will fix a variable typing.

Terms:

datatype *'fsym trm* =
Var var |
Fn 'fsym 'fsym trm list

Atomic formulas (atoms):

datatype (*'fsym*, *'psym*) *atm* =

Eq 'fsym trm 'fsym trm |
Pr 'psym 'fsym trm list

Literals:

datatype (*'fsym*, *'psym*) *lit* =
Pos (*'fsym*, *'psym*) *atm* |
Neg (*'fsym*, *'psym*) *atm*

Clauses:

type-synonym (*'fsym*, *'psym*) *cls* = (*'fsym*, *'psym*) *lit list*

Problems:

type-synonym (*'fsym*, *'psym*) *prob* = (*'fsym*, *'psym*) *cls set*

lemma *trm-induct*[*case-names Var Fn*, *induct type*: *trm*]:
assumes $\bigwedge$ *x*. φ (*Var x*)
and $\bigwedge$ *f Tl*. *list-all* φ *Tl* $\Longrightarrow$ φ (*Fn f Tl*)
shows φ *T*
using *assms* **unfolding** *list-all-iff* **by** (*rule trm.induct*) *metis*

fun *vars* **where**
vars (*Var x*) = {*x*}
|
vars (*Fn f Tl*) = $\bigcup$ (*vars* ‘ (*set Tl*))

fun *varsA* **where**
varsA (*Eq T1 T2*) = *vars T1* ∪ *vars T2*
|
varsA (*Pr p Tl*) = $\bigcup$ (*set* (*map vars Tl*))

fun *varsL* **where**
varsL (*Pos at*) = *varsA at*
|
varsL (*Neg at*) = *varsA at*

definition *varsC c* = $\bigcup$ (*set* (*map varsL c*))

definition *varsPB* Φ = $\bigcup$ {*varsC c* | *c*. *c* ∈ Φ}

Substitution:

fun *subst* **where**
subst π (*Var x*) = π *x*
|
subst π (*Fn f Tl*) = *Fn f* (*map* (*subst* π) *Tl*)

fun *substA* **where**
substA π (*Eq T1 T2*) = *Eq* (*subst* π *T1*) (*subst* π *T2*)
|

substA π (*Pr p Tl*) = *Pr p* (*map* (*subst* π) *Tl*)

fun *substL* **where**
substL π (*Pos at*) = *Pos* (*substA* π *at*)
|
substL π (*Neg at*) = *Neg* (*substA* π *at*)

definition *substC* π *c* = *map* (*substL* π) *c*

definition *substPB* π Φ = {*substC* π *c* | *c*. *c* $\in$ Φ}

lemma *subst-cong*:
assumes $\bigwedge$ *x*. *x* $\in$ *vars T* $\Longrightarrow$ π*1 x* = π*2 x*
shows *subst* π*1 T* = *subst* π*2 T*
using *assms* **by** (*induct T*, *auto simp add*: *list-all-iff*)

lemma *substA-congA*:
assumes $\bigwedge$ *x*. *x* $\in$ *varsA at* $\Longrightarrow$ π*1 x* = π*2 x*
shows *substA* π*1 at* = *substA* π*2 at*
using *assms subst-cong*[*of* - π*1* π*2*]
by (*cases at*, *fastforce*+)

lemma *substL-congL*:
assumes $\bigwedge$ *x*. *x* $\in$ *varsL l* $\Longrightarrow$ π*1 x* = π*2 x*
shows *substL* π*1 l* = *substL* π*2 l*
using *assms substA-congA*[*of* - π*1* π*2*] **by** (*cases l*, *auto*)

lemma *substC-congC*:
assumes $\bigwedge$ *x*. *x* $\in$ *varsC c* $\Longrightarrow$ π*1 x* = π*2 x*
shows *substC* π*1 c* = *substC* π*2 c*
using *substL-congL*[*of* - π*1* π*2*] *assms* **unfolding** *substC-def varsC-def*
by (*induct c*, *auto*)

lemma *substPB-congPB*:
assumes $\bigwedge$ *x*. *x* $\in$ *varsPB* Φ $\Longrightarrow$ π*1 x* = π*2 x*
shows *substPB* π*1* Φ = *substPB* π*2* Φ
using *substC-congC*[*of* - π*1* π*2*] *assms* **unfolding** *substPB-def varsPB-def*
by *simp-all* (*metis* (*lifting*) *substC-congC*)

lemma *vars-subst*:
vars (*subst* π *T*) = ($\bigcup$ *x* $\in$ *vars T*. *vars* (π *x*))
by (*induct T*) (*auto simp add*: *list-all-iff*)

lemma *varsA-substA*:
varsA (*substA* π *at*) = ($\bigcup$ *x* $\in$ *varsA at*. *vars* (π *x*))
using *vars-subst*[*of* π] **by** (*cases at*, *auto*)

lemma *varsL-substL*:
varsL (*substL* π *l*) = ($\bigcup$ *x* $\in$ *varsL l*. *vars* (π *x*))

using *varsA-substA*[*of* π] **by** (*cases l*, *auto*)

lemma *varsC-substC*:
varsC (*substC* π *c*) = ($\bigcup$ *x* $\in$ *varsC c*. *vars* (π *x*))
apply(*induct c*) **unfolding** *substC-def varsC-def*
apply *fastforce*
unfolding *substC-def varsC-def map-map set-map*
unfolding *comp-def varsL-substL* **by** *blast*

lemma *varsPB-Un*[*simp*]: *varsPB* (Φ*1* $\cup$ Φ*2*) = *varsPB* Φ*1* $\cup$ *varsPB* Φ*2*
unfolding *varsPB-def* **by** *auto*

lemma *varsC-append*[*simp*]: *varsC* (*c1* @ *c2*) = *varsC c1* $\cup$ *varsC c2*
unfolding *varsC-def* **by** *auto*

lemma *varsPB-sappend-incl*[*simp*]:
varsPB (Φ*1* @@ Φ*2*) $\subseteq$ *varsPB* Φ*1* $\cup$ *varsPB* Φ*2*
by (*unfold varsPB-def sappend-def*, *fastforce*)

lemma *varsPB-sappend*[*simp*]:
assumes *1*: Φ*1* $\neq$ {} **and** *2*: Φ*2* $\neq$ {}
shows *varsPB* (Φ*1* @@ Φ*2*) = *varsPB* Φ*1* $\cup$ *varsPB* Φ*2*
proof *safe*
fix *x*
{assume *x* $\in$ *varsPB* Φ*1*
then obtain *c1 c2* **where** *x* $\in$ *varsC c1* **and** *c1* $\in$ Φ*1* **and** *c2* $\in$ Φ*2*
using *2* **unfolding** *varsPB-def* **by** *auto*
thus *x* $\in$ *varsPB* (Φ*1* @@ Φ*2*) **unfolding** *sappend-def varsPB-def* **by** *fastforce*
}
{assume *x* $\in$ *varsPB* Φ*2*
then obtain *c1 c2* **where** *x* $\in$ *varsC c2* **and** *c1* $\in$ Φ*1* **and** *c2* $\in$ Φ*2*
using *1* **unfolding** *varsPB-def* **by** *auto*
thus *x* $\in$ *varsPB* (Φ*1* @@ Φ*2*) **unfolding** *sappend-def varsPB-def* **by** *fastforce*
}
qed(*unfold varsPB-def sappend-def*, *fastforce*)

lemma *varsPB-substPB*:
varsPB (*substPB* π Φ) = ($\bigcup$ *x* $\in$ *varsPB* Φ. *vars* (π *x*)) (**is** - = *?K*)
proof *safe*
fix *x* **assume** *x* $\in$ *varsPB* (*substPB* π Φ)
then obtain *c* **where** *c* $\in$ Φ **and** *x* $\in$ *varsC* (*substC* π *c*)
unfolding *varsPB-def substPB-def* **by** *auto*
thus *x* $\in$ *?K* **unfolding** *varsC-substC varsPB-def* **by** *auto*
next
fix *x y* **assume** *y* $\in$ *varsPB* Φ **and** *x*: *x* $\in$ *vars* (π *y*)
then obtain *c* **where** *c*: *c* $\in$ Φ **and** *y* $\in$ *varsC c* **unfolding** *varsPB-def* **by** *auto*
hence *x* $\in$ *varsC* (*substC* π *c*) **using** *x* **unfolding** *varsC-substC* **by** *auto*
thus *x* $\in$ *varsPB* (*substPB* π Φ) **using** *c* **unfolding** *varsPB-def substPB-def* **by** *auto*

qed

lemma *subst-o*:
subst (*subst* $\pi 1$ *o* $\pi 2$) *T* = *subst* $\pi 1$ (*subst* $\pi 2$ *T*)
apply(*induct T*) **by** (*auto simp add*: *list-all-iff*)

lemma *o-subst*:
subst $\pi 1$ *o subst* $\pi 2$ = *subst* (*subst* $\pi 1$ *o* $\pi 2$)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *subst-o*[*symmetric*] **..**

lemma *substA-o*:
substA (*subst* $\pi 1$ *o* $\pi 2$) *at* = *substA* $\pi 1$ (*substA* $\pi 2$ *at*)
using *subst-o*[*of* $\pi 1$ $\pi 2$] **by** (*cases at*, *auto*)

lemma *o-substA*:
substA $\pi 1$ *o substA* $\pi 2$ = *substA* (*subst* $\pi 1$ *o* $\pi 2$)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *substA-o*[*symmetric*] **..**

lemma *substL-o*:
substL (*subst* $\pi 1$ *o* $\pi 2$) *l* = *substL* $\pi 1$ (*substL* $\pi 2$ *l*)
using *substA-o*[*of* $\pi 1$ $\pi 2$] **by** (*cases l*, *auto*)

lemma *o-substL*:
substL $\pi 1$ *o substL* $\pi 2$ = *substL* (*subst* $\pi 1$ *o* $\pi 2$)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *substL-o*[*symmetric*] **..**

lemma *substC-o*:
substC (*subst* $\pi 1$ *o* $\pi 2$) *c* = *substC* $\pi 1$ (*substC* $\pi 2$ *c*)
using *substL-o*[*of* $\pi 1$ $\pi 2$] **unfolding** *substC-def* **by** (*induct c*, *auto*)

lemma *o-substC*:
substC $\pi 1$ *o substC* $\pi 2$ = *substC* (*subst* $\pi 1$ *o* $\pi 2$)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *substC-o*[*symmetric*] **..**

lemma *substPB-o*:
substPB (*subst* $\pi 1$ *o* $\pi 2$) Φ = *substPB* $\pi 1$ (*substPB* $\pi 2$ Φ)
using *substC-o*[*of* $\pi 1$ $\pi 2$] **unfolding** *substPB-def* **by** *auto*

lemma *o-substPB*:
substPB $\pi 1$ *o substPB* $\pi 2$ = *substPB* (*subst* $\pi 1$ *o* $\pi 2$)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *substPB-o*[*symmetric*] **..**

lemma *finite-vars*: *finite* (*vars T*)
by(*induct T*, *auto simp add*: *list-all-iff*)

lemma *finite-varsA*: *finite* (*varsA at*)
using *finite-vars* **by** (*cases at*, *auto*)

lemma *finite-varsL*: *finite* (*varsL l*)

using *finite-varsA* **by** (*cases l*, *auto*)

lemma *finite-varsC*: *finite* (*varsC c*)
using *finite-varsL* **unfolding** *varsC-def* **by** (*induct c*, *auto*)

lemma *finite-varsPB*: *finite* $\Phi \Longrightarrow$ *finite* (*varsPB* Φ)
using *finite-varsC* **unfolding** *varsPB-def* **by** (*auto intro*!: *finite-Union*)

end

3 Many-Typed (Many-Sorted) First-Order Logic

theory *Sig* **imports** *Preliminaries*
begin

In this formalization, we call “types" what the first-order logic community usually calls “sorts".

3.1 Signatures

locale *Signature* =
fixes
  wtFsym :: *'fsym* ⇒ *bool*
and *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* :: *'fsym* ⇒ *'tp*
and *parOf* :: *'psym* ⇒ *'tp list*
assumes
  countable-tp: *countable* (*UNIV* :: *'tp set*)
and *countable-wtFsym*: *countable* {*f*::*'fsym*. *wtFsym f*}
and *countable-wtPsym*: *countable* {*p*::*'psym*. *wtPsym p*}
begin

Partitioning of the variables in countable sets for each type:

definition *tpOfV-pred* :: (*var* ⇒ *'tp*) ⇒ *bool* **where**
tpOfV-pred f ≡ ∀ *σ*. *infinite* (*f* −‘ {*σ*})

definition *tpOfV* ≡ *SOME f*. *tpOfV-pred f*

lemma *infinite-fst-vimage*:
infinite ((*fst* :: *'a* × *nat* ⇒ *'a*) −‘ {*a*}) (**is** *infinite* (*?f* −‘ {*a*}))
proof−
  have *?f* −‘ {*a*} = {(*a*,*n*::*nat*) | *n* . *True*} (**is** - = *?A*) **by** *auto*
  moreover
  {have *0*: *?A* = *range* (*Pair a*) **by** *auto*
   have *infinite ?A* **unfolding** *0* **apply**(*rule range-inj-infinite*)
   unfolding *inj-on-def* **by** *auto*
  }

ultimately show *?thesis* **by** *auto*
qed

lemma *tpOfV-pred*: $\exists\ f.$ *tpOfV-pred f*
proof−
define *Ut Uv* **where** $Ut = (UNIV :: 'tp\ set)$ **and** $Uv = (UNIV :: var\ set)$
define *Unt* **where** $Unt = (UNIV :: nat\ set)$
define *U2* **where** $U2 = (UNIV :: ('tp \times nat)\ set)$
have *U2*: $U2 \equiv Ut \times Unt$ **unfolding** *Ut-def Unt-def U2-def UNIV-Times-UNIV*
.
have $|U2| =o |Unt \times Ut|$ **unfolding** *U2* **by** (*metis card-of-Times-commute*)
also have $|Unt \times Ut| =o |Unt|$
apply(*rule card-of-Times-infinite-simps*(*1*)) **unfolding** *Ut-def Unt-def*
apply (*metis nat-not-finite*)
apply (*metis UNIV-not-empty*)
by (*metis countable-card-of-nat countable-tp*)
also have $|Unt| =o |Uv|$ **apply**(*rule ordIso-symmetric*)
unfolding *Unt-def Uv-def* **using** *card-of-var card-of-nat*[*THEN ordIso-symmetric*]
by(*rule ordIso-transitive*)
finally have $|U2| =o |Uv|$ **.**
hence $|Uv| =o |U2|$ **by**(*rule ordIso-symmetric*)
then obtain *g* **where** *g*: *bij-betw g Uv U2* **unfolding** *card-of-ordIso*[*symmetric*]
by *blast*
show *?thesis* **apply**(*rule exI*[*of - fst o g*]) **unfolding** *tpOfV-pred-def* **apply** *safe*
unfolding *vimage-comp* [*symmetric*] **apply** (*drule finite-vimageD*)
using *g* **unfolding** *bij-betw-def Uv-def U2-def* **by** (*auto simp*: *infinite-fst-vimage*)
qed

lemma *tpOfV-pred-tpOfV*: *tpOfV-pred tpOfV*
using *tpOfV-pred* **unfolding** *tpOfV-def* **by** (*rule someI-ex*)

lemma *tpOfV*: *infinite* (*tpOfV* $-`\{\sigma\}$)
using *tpOfV-pred-tpOfV* **unfolding** *tpOfV-pred-def* **by** *auto*

definition *tpart1* $V \equiv \bigcup\ \sigma.$ *part1* $(V \cap tpOfV -`\{\sigma\})$
definition *tpart2* $V \equiv \bigcup\ \sigma.$ *part2* $(V \cap tpOfV -`\{\sigma\})$
definition *tinfinite* $V \equiv \forall\ \sigma.$ *infinite* $(V \cap tpOfV -`\{\sigma\})$

lemma *tinfinite-var*[*simp*,*intro*]: *tinfinite* $(UNIV :: var\ set)$
unfolding *tinfinite-def* **using** *tpOfV* **by** *auto*

lemma *tinifnite-singl*[*simp*]:
assumes *tinfinite V* **shows** *tinfinite* $(V - \{x\})$
unfolding *tinfinite-def* **proof** *safe*
fix σ **assume** *0*: *finite* $((V - \{x\}) \cap tpOfV -`\{\sigma\})$
have *finite* $((V \cap tpOfV -`\{\sigma\}) - \{x\})$ **apply**(*rule finite-subset*[*OF - 0*]) **by**
auto
thus *False* **using** *assms* **unfolding** *tinfinite-def* **by** *auto*
qed

lemma *tpart1-Un-tpart2*[*simp*]:
assumes *tinfinite V* **shows** *tpart1 V* ∪ *tpart2 V* = *V*
using *assms part1-Un-part2* **unfolding** *tinfinite-def tpart1-def tpart2-def*
unfolding *UN-Un-distrib*[*symmetric*] **by** *blast*

lemma *tpart1-Int-tpart2*[*simp*]:
assumes *tinfinite V* **shows** *tpart1 V* ∩ *tpart2 V* = {}
using *assms part1-Int-part2* **unfolding** *tinfinite-def tpart1-def tpart2-def*
unfolding *Int-UN-distrib2* **apply** *auto* **apply** (*case-tac xa* = *xb*, *auto*)
using *part1-su part2-su* **by** *blast*

lemma *tpart1-su*:
assumes *tinfinite V* **shows** *tpart1 V* ⊆ *V*
using *assms* **unfolding** *tinfinite-def tpart1-def*
using *part1-su* **by** (*auto intro*: *UN-least*)

lemma *tpart1-in*:
assumes *tinfinite V* **and** *x* ∈ *tpart1 V* **shows** *x* ∈ *V*
using *assms tpart1-su* **by** *auto*

lemma *tinfinite-tpart1*[*simp*]:
assumes *tinfinite V*
shows *tinfinite* (*tpart1 V*)
unfolding *tinfinite-def tpart1-def* **proof** *safe*
fix *σ* **assume**
finite ((⋃*σ'*. *part1* (*V* ∩ *tpOfV* −‘ {*σ'*})) ∩ *tpOfV* −‘ {*σ*}) (**is** *finite ?A*)
moreover have *?A* = *part1* (*V* ∩ *tpOfV* −‘ {*σ*})
using *assms part1-su* **unfolding** *tinfinite-def* **by** *auto*
moreover have *infinite ...*
using *assms infinite-part1* **unfolding** *tinfinite-def* **by** *auto*
ultimately show *False* **by** *auto*
qed

lemma *tinfinite-tpart2*[*simp*]:
assumes *tinfinite V*
shows *tinfinite* (*tpart2 V*)
unfolding *tinfinite-def tpart2-def* **proof** *safe*
fix *σ* **assume**
finite ((⋃*σ'*. *part2* (*V* ∩ *tpOfV* −‘ {*σ'*})) ∩ *tpOfV* −‘ {*σ*}) (**is** *finite ?A*)
moreover have *?A* = *part2* (*V* ∩ *tpOfV* −‘ {*σ*})
using *assms part2-su* **unfolding** *tinfinite-def* **by** *auto*
moreover have *infinite ...*
using *assms infinite-part2* **unfolding** *tinfinite-def* **by** *auto*
ultimately show *False* **by** *auto*
qed

lemma *tpart2-su*:
assumes *tinfinite V* **shows** *tpart2 V* ⊆ *V*

using *assms* **unfolding** *tinfinite-def tpart2-def*
using *part2-su* **by** (*auto intro*: *UN-least*)

lemma *tpart2-in*:
assumes *tinfinite V* **and** $x \in$ *tpart2 V* **shows** $x \in V$
using *assms tpart2-su* **by** *auto*

Typed-pick: picking a variable of a given type

definition *tpick* $\sigma\ V \equiv$ *pick* $(V \cap tpOfV\ -`\ \{\sigma\})$

lemma *tinfinite-ex*: *tinfinite* $V \Longrightarrow \exists\ x \in V.\ tpOfV\ x = \sigma$
unfolding *tinfinite-def* **using** *infinite-imp-nonempty* **by** *auto*

lemma *tpick*: **assumes** *tinfinite V* **shows** *tpick* $\sigma\ V \in V \cap tpOfV\ -`\ \{\sigma\}$
proof−
obtain x **where** $x \in V \wedge tpOfV\ x = \sigma$
using *tinfinite-ex*[*OF assms*] **by** *auto*
hence $x \in V \cap tpOfV\ -`\ \{\sigma\}$ **by** *blast*
thus *?thesis* **unfolding** *tpick-def* **by** (*rule pick*)
qed

lemma *tpick-in*[*simp*]: *tinfinite* $V \Longrightarrow$ *tpick* $\sigma\ V \in V$
and *tpOfV-tpick*[*simp*]: *tinfinite* $V \Longrightarrow$ *tpOfV* (*tpick* $\sigma\ V$) $= \sigma$
using *tpick* **by** *auto*

lemma *finite-tinfinite*:
assumes *finite V*
shows *tinfinite* (*UNIV* − *V*)
using *assms infinite-var* **unfolding** *tinfinite-def*
by (*metis Diff-Int2 Diff-Int-distrib2 Int-UNIV-left finite-Diff2 tpOfV*)

fun *getVars* **where**
getVars [] = []
|
getVars (σ # σl) =
(*let xl* = *getVars* σl *in* (*tpick* σ (*UNIV* − *set xl*)) # *xl*)

lemma *distinct-getVars*: *distinct* (*getVars* σl)
using *tpick-in*[*OF finite-tinfinite*] **by** (*induct* σl, *auto simp*: *Let-def*)

lemma *length-getVars*[*simp*]: *length* (*getVars* σl) = *length* σl
by(*induct* σl, *auto simp*: *Let-def*)

lemma *map-tpOfV-getVars*[*simp*]: *map tpOfV* (*getVars* σl) = σl
using *tpOfV-tpick*[*OF finite-tinfinite*] **by** (*induct* σl, *auto simp*: *Let-def*)

lemma *tpOfV-getVars-nth*[*simp*]:
assumes $i <$ *length* σl **shows** *tpOfV* (*getVars* σl ! i) = σl ! i

using *assms* **using** *map-tpOfV-getVars* **by** (*metis length-getVars nth-map*)

end

end
theory *M*
imports *TermsAndClauses Sig*
begin

3.2 Well-typed (well-formed) terms, clauses, literals and problems

context *Signature* **begin**

The type of a term

fun *tpOf* **where**
tpOf (*Var x*) = *tpOfV x*
|
tpOf (*Fn f Tl*) = *resOf f*

fun *wt* **where**
wt (*Var x*) ⟷ *True*
|
wt (*Fn f Tl*) ⟷
wtFsym f ∧ *list-all wt Tl* ∧ *arOf f* = *map tpOf Tl*

fun *wtA* **where**
wtA (*Eq T1 T2*) ⟷ *wt T1* ∧ *wt T2* ∧ *tpOf T1* = *tpOf T2*
|
wtA (*Pr p Tl*) ⟷
wtPsym p ∧ *list-all wt Tl* ∧ *parOf p* = *map tpOf Tl*

fun *wtL* **where**
wtL (*Pos a*) ⟷ *wtA a*
|
wtL (*Neg a*) ⟷ *wtA a*

definition *wtC* ≡ *list-all wtL*

lemma *wtC-append*[*simp*]: *wtC* (*c1* @ *c2*) ⟷ *wtC c1* ∧ *wtC c2*
unfolding *wtC-def* **by** *simp*

definition $wtPB\ \Phi \equiv \forall\ c \in \Phi.\ wtC\ c$

lemma $wtPB\text{-}Un[simp]$: $wtPB\ (\Phi 1 \cup \Phi 2) \longleftrightarrow wtPB\ \Phi 1 \wedge wtPB\ \Phi 2$
unfolding $wtPB\text{-}def$ **by** $auto$

lemma $wtPB\text{-}UN[simp]$: $wtPB\ (\bigcup\ i \in I.\ \Phi\ i) \longleftrightarrow (\forall\ i \in I.\ wtPB\ (\Phi\ i))$
unfolding $wtPB\text{-}def$ **by** $auto$

lemma $wtPB\text{-}sappend[simp]$:
assumes $wtPB\ \Phi 1$ **and** $wtPB\ \Phi 2$ **shows** $wtPB\ (\Phi 1\ @@\ \Phi 2)$
using $assms$ **unfolding** $wtPB\text{-}def\ sappend\text{-}def$ **by** $auto$

definition $wtSB\ \pi \equiv \forall\ x.\ wt\ (\pi\ x) \wedge tpOf\ (\pi\ x) = tpOfV\ x$

lemma $wtSB\text{-}wt[simp]$: $wtSB\ \pi \Longrightarrow wt\ (\pi\ x)$
unfolding $wtSB\text{-}def$ **by** $auto$

lemma $wtSB\text{-}tpOf[simp]$: $wtSB\ \pi \Longrightarrow tpOf\ (\pi\ x) = tpOfV\ x$
unfolding $wtSB\text{-}def$ **by** $auto$

lemma $wt\text{-}tpOf\text{-}subst$:
assumes $wtSB\ \pi$ **and** $wt\ T$
shows $wt\ (subst\ \pi\ T) \wedge tpOf\ (subst\ \pi\ T) = tpOf\ T$
using $assms$ **apply**$(induct\ T)$ **by** $(auto\ simp\ add{:}\ list\text{-}all\text{-}iff)$

lemmas $wt\text{-}subst[simp] = wt\text{-}tpOf\text{-}subst[THEN\ conjunct1]$
lemmas $tpOf\text{-}subst[simp] = wt\text{-}tpOf\text{-}subst[THEN\ conjunct2]$

lemma $wtSB\text{-}o$:
assumes $1{:}\ wtSB\ \pi 1$ **and** $2{:}\ wtSB\ \pi 2$
shows $wtSB\ (subst\ \pi 1\ o\ \pi 2)$
using 2 **unfolding** $wtSB\text{-}def$ **using** 1 **by** $auto$

definition $getTvars\ \sigma l \equiv map\ Var\ (getVars\ \sigma l)$

lemma $length\text{-}getTvars[simp]$: $length\ (getTvars\ \sigma l) = length\ \sigma l$
unfolding $getTvars\text{-}def$ **by** $auto$

lemma $wt\text{-}getTvars[simp]$: $list\text{-}all\ wt\ (getTvars\ \sigma l)$
unfolding $list\text{-}all\text{-}length\ getTvars\text{-}def$ **by** $simp$

lemma $wt\text{-}nth\text{-}getTvars[simp]$:
$i < length\ \sigma l \Longrightarrow wt\ (getTvars\ \sigma l\ !\ i)$
unfolding $getTvars\text{-}def$ **by** $auto$

lemma $map\text{-}tpOf\text{-}getTvars[simp]$: $map\ tpOf\ (getTvars\ \sigma l) = \sigma l$
unfolding $getTvars\text{-}def$ **unfolding** $list\text{-}eq\text{-}iff$ **by** $auto$

lemma *tpOf-nth-getTvars*[*simp*]:
$i < length\ \sigma l \Longrightarrow tpOf\ (getTvars\ \sigma l\ !\ i) = \sigma l\ !\ i$
unfolding *getTvars-def* **by** *auto*

end

3.3 Structures

We split a structre into a "type structure" that interprets the types and the rest of the structure that interprets the function and relation symbols.

Type structures:

locale *Tstruct* =
fixes $intT :: 'tp \Rightarrow 'univ \Rightarrow bool$
assumes *NE-intT*: $NE\ (intT\ \sigma)$

Environment:

type-synonym $('tp,'univ)\ env = 'tp \Rightarrow var \Rightarrow 'univ$

Structures:

locale *Struct* = *Signature wtFsym wtPsym arOf resOf parOf* +
Tstruct intT
for *wtFsym* **and** *wtPsym*
and $arOf :: 'fsym \Rightarrow 'tp\ list$
and $resOf :: 'fsym \Rightarrow 'tp$
and $parOf :: 'psym \Rightarrow 'tp\ list$
and $intT :: 'tp \Rightarrow 'univ \Rightarrow bool$
+
fixes
$intF :: 'fsym \Rightarrow 'univ\ list \Rightarrow 'univ$
and $intP :: 'psym \Rightarrow 'univ\ list \Rightarrow bool$
assumes
intF: $[\![wtFsym\ f;\ list\text{-}all2\ intT\ (arOf\ f)\ al]\!] \Longrightarrow intT\ (resOf\ f)\ (intF\ f\ al)$
and
dummy: $intP = intP$
begin

Well-typed environment:

definition $wtE\ \xi \equiv \forall\ x.\ intT\ (tpOfV\ x)\ (\xi\ x)$

lemma *wtTE-intT*[*simp*]: $wtE\ \xi \Longrightarrow intT\ (tpOfV\ x)\ (\xi\ x)$
unfolding *wtE-def dom-def* **by** *auto*

definition $pickT\ \sigma \equiv SOME\ a.\ intT\ \sigma\ a$

lemma *pickT*[*simp*]: $intT\ \sigma\ (pickT\ \sigma)$

unfolding *pickT-def* **apply**(*rule someI-ex*) **using** *NE-intT* **by** *auto*

Picking a well-typed environment:

definition
pickE (*xl*::*var list*) *al* ≡
SOME ξ. *wtE* ξ ∧ (∀ *i* < *length xl*. ξ (*xl*!*i*) = *al*!*i*)

lemma *ex-pickE*:
assumes *length xl* = *length al*
and *distinct xl* **and** ⋀ *i*. *i* < *length xl* ⟹ *intT* (*tpOfV* (*xl*!*i*)) (*al*!*i*)
shows ∃ ξ. *wtE* ξ ∧ (∀ *i* < *length xl*. ξ (*xl*!*i*) = *al*!*i*)
using *assms*
proof(*induct rule*: *list-induct2*)
case *Nil* **show** *?case* **apply**(*rule exI*[*of* - λ *x*. *pickT* (*tpOfV x*)])
unfolding *wtE-def* **by** *auto*
next
case (*Cons x xl a al*)
then obtain ξ **where** *1*: *wtE* ξ **and** *2*: ∀ *i* < *length xl*. ξ (*xl*!*i*) = *al*!*i* **by** *force*
define ξ′ **where** ξ′ *x*′ = (*if x* = *x*′ *then a else* ξ *x*′) **for** *x*′
show *?case*
proof(*rule exI*[*of* - ξ′], *unfold wtE-def*, *safe*)
fix *x*′ **show** *intT* (*tpOfV x*′) (ξ′ *x*′)
using *1 Cons.prems*(*2*)[*of 0*] **unfolding** ξ′*-def* **by** *auto*
next
fix *i* **assume** *i*: *i* < *length* (*x* # *xl*)
thus ξ′ ((*x* # *xl*) ! *i*) = (*a* # *al*) ! *i*
proof(*cases i*)
case (*Suc j*) **hence** *j*: *j* < *length xl* **using** *i* **by** *auto*
have ¬ *x* = (*x* # *xl*) ! *i* **using** *Suc i Cons.prems*(*1*) **by** *auto*
thus *?thesis* **using** *Suc* **using** *j Cons.prems*(*1*) *Cons.hyps 2* **unfolding** ξ′*-def*
by *auto*
qed(*insert Cons.prems*(*1*) *Cons.hyps 2*, *unfold* ξ′*-def*, *simp*)
qed
qed

lemma *wtE-pickE-pickE*:
assumes *length xl* = *length al*
and *distinct xl* **and** ⋀ *i*. *i* < *length xl* ⟹ *intT* (*tpOfV* (*xl*!*i*)) (*al*!*i*)
shows *wtE* (*pickE xl al*) ∧ (∀ *i*. *i* < *length xl* ⟶ *pickE xl al* (*xl*!*i*) = *al*!*i*)
proof−
let *?phi* = λ ξ. *wtE* ξ ∧ (∀ *i* < *length xl*. ξ (*xl*!*i*) = *al*!*i*)
show *?thesis* **unfolding** *pickE-def* **apply**(*rule someI-ex*[*of ?phi*])
using *ex-pickE*[*OF assms*] **by** *simp*
qed

lemmas *wtE-pickE*[*simp*] = *wtE-pickE-pickE*[*THEN conjunct1*]

lemma *pickE*[*simp*]:
assumes *length xl* = *length al*

and *distinct xl* **and** $\bigwedge$ *i. i < length xl* $\Longrightarrow$ *intT* (*tpOfV* (*xl*!*i*)) (*al*!*i*)
and *i < length xl*
shows *pickE xl al* (*xl*!*i*) = *al*!*i*
using *assms wtE-pickE-pickE* **by** *auto*

definition *pickAnyE* ≡ *pickE* [] []

lemma *wtE-pickAnyE*[*simp*]: *wtE pickAnyE*
unfolding *pickAnyE-def* **by** (*rule wtE-pickE*) *auto*

fun *int* **where**
int ξ (*Var x*) = *ξ x*
|
int ξ (*Fn f Tl*) = *intF f* (*map* (*int ξ*) *Tl*)

fun *satA* **where**
satA ξ (*Eq T1 T2*) ⟷ *int ξ T1* = *int ξ T2*
|
satA ξ (*Pr p Tl*) ⟷ *intP p* (*map* (*int ξ*) *Tl*)

fun *satL* **where**
satL ξ (*Pos a*) ⟷ *satA ξ a*
|
satL ξ (*Neg a*) ⟷ ¬ *satA ξ a*

definition *satC ξ* ≡ *list-ex* (*satL ξ*)

lemma *satC-append*[*simp*]: *satC ξ* (*c1* @ *c2*) ⟷ *satC ξ c1* ∨ *satC ξ c2*
unfolding *satC-def* **by** *auto*

lemma *satC-iff-set*: *satC ξ c* ⟷ (∃ *l* ∈ *set c. satL ξ l*)
unfolding *satC-def Bex-set*[*symmetric*] **..**

definition *satPB ξ* Φ ≡ ∀ *c* ∈ Φ. *satC ξ c*

lemma *satPB-Un*[*simp*]: *satPB ξ* (Φ*1* ∪ Φ*2*) ⟷ *satPB ξ* Φ*1* ∧ *satPB ξ* Φ*2*
unfolding *satPB-def* **by** *auto*

lemma *satPB-UN*[*simp*]: *satPB ξ* (⋃ *i* ∈ *I*. Φ *i*) ⟷ (∀ *i* ∈ *I. satPB ξ* (Φ *i*))
unfolding *satPB-def* **by** *auto*

lemma *satPB-sappend*[*simp*]: *satPB ξ* (Φ*1* @@ Φ*2*) ⟷ *satPB ξ* Φ*1* ∨ *satPB ξ* Φ*2*
unfolding *satPB-def sappend-def* **by** (*fastforce simp*: *satC-append*)

definition $SAT\ \Phi \equiv \forall\ \xi.\ wtE\ \xi \longrightarrow satPB\ \xi\ \Phi$

lemma *SAT-UN*[*simp*]: $SAT\ (\bigcup\ i \in I.\ \Phi\ i) \longleftrightarrow (\forall\ i \in I.\ SAT\ (\Phi\ i))$
unfolding *SAT-def* **by** *auto*

Soundness of typing w.r.t. interpretation:

lemma *wt-int*:
assumes *wtE*: $wtE\ \xi$ **and** *wt*: $wt\ T$
shows $intT\ (tpOf\ T)\ (int\ \xi\ T)$
using *wt* **apply**(*induct T*) **using** *wtE*
by (*auto intro*!: *intF simp add*: *list-all2-map-map*)

lemma *int-cong*:
assumes $\bigwedge x.\ x \in vars\ T \Longrightarrow \xi 1\ x = \xi 2\ x$
shows $int\ \xi 1\ T = int\ \xi 2\ T$
using *assms* **proof**(*induct T*)
  case (*Fn f Tl*)
  hence *1*: $map\ (int\ \xi 1)\ Tl = map\ (int\ \xi 2)\ Tl$
  unfolding *list-all-iff* **by** (*auto intro*: *map-ext*)
  show *?case* **apply** *simp* **by** (*metis 1*)
qed *auto*

lemma *satA-cong*:
assumes $\bigwedge x.\ x \in varsA\ at \Longrightarrow \xi 1\ x = \xi 2\ x$
shows $satA\ \xi 1\ at \longleftrightarrow satA\ \xi 2\ at$
using *assms int-cong*[*of - ξ1 ξ2*]
apply(*cases at*) **apply**(*fastforce intro*!: *int-cong*[*of - ξ1 ξ2*])
apply *simp* **by** (*metis* (*opaque-lifting*, *mono-tags*) *map-eq-conv*)

lemma *satL-cong*:
assumes $\bigwedge\ x.\ x \in varsL\ l \Longrightarrow \xi 1\ x = \xi 2\ x$
shows $satL\ \xi 1\ l \longleftrightarrow satL\ \xi 2\ l$
using *assms satA-cong*[*of - ξ1 ξ2*] **by** (*cases l*, *auto*)

lemma *satC-cong*:
assumes $\bigwedge\ x.\ x \in varsC\ c \Longrightarrow \xi 1\ x = \xi 2\ x$
shows $satC\ \xi 1\ c \longleftrightarrow satC\ \xi 2\ c$
using *assms satL-cong*[*of - ξ1 ξ2*] **unfolding** *satC-def varsC-def*
apply (*induct c*) **by** (*fastforce intro*!: *satL-cong*[*of - ξ1 ξ2*])+

lemma *satPB-cong*:
assumes $\bigwedge\ x.\ x \in varsPB\ \Phi \Longrightarrow \xi 1\ x = \xi 2\ x$
shows $satPB\ \xi 1\ \Phi \longleftrightarrow satPB\ \xi 2\ \Phi$
by (*force simp*: *satPB-def varsPB-def intro*!: *satC-cong ball-cong assms*)

lemma *int-o*:
$int\ (int\ \xi\ o\ \varrho)\ T = int\ \xi\ (subst\ \varrho\ T)$
apply(*induct T*) **apply** *simp-all* **unfolding** *list-all-iff o-def*

using *map-ext* **by** (*metis* (*lifting*, *no-types*))

lemmas *int-subst* = *int-o*[*symmetric*]

lemma *int-o-subst*:
int ξ *o subst* ϱ = *int* (*int* ξ *o* ϱ)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *int-o*[*symmetric*] **..**

lemma *satA-o*:
satA (*int* ξ *o* ϱ) *at* = *satA* ξ (*substA* ϱ *at*)
by (*cases at*, *simp-all add*: *int-o-subst int-o*[*of* ξ ϱ])

lemmas *satA-subst* = *satA-o*[*symmetric*]

lemma *satA-o-subst*:
satA ξ *o substA* ϱ = *satA* (*int* ξ *o* ϱ)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *satA-o*[*symmetric*] **..**

lemma *satL-o*:
satL (*int* ξ *o* ϱ) *l* = *satL* ξ (*substL* ϱ *l*)
using *satA-o*[*of* ξ ϱ] **by** (*cases l*, *simp-all*)

lemmas *satL-subst* = *satL-o*[*symmetric*]

lemma *satL-o-subst*:
satL ξ *o substL* ϱ = *satL* (*int* ξ *o* ϱ)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *satL-o*[*symmetric*] **..**

lemma *satC-o*:
satC (*int* ξ *o* ϱ) *c* = *satC* ξ (*substC* ϱ *c*)
using *satL-o*[*of* ξ ϱ] **unfolding** *satC-def substC-def* **by** (*induct c*, *auto*)

lemmas *satC-subst* = *satC-o*[*symmetric*]

lemma *satC-o-subst*:
satC ξ *o substC* ϱ = *satC* (*int* ξ *o* ϱ)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *satC-o*[*symmetric*] **..**

lemma *satPB-o*:
satPB (*int* ξ *o* ϱ) Φ = *satPB* ξ (*substPB* ϱ Φ)
using *satC-o*[*of* ξ ϱ] **unfolding** *satPB-def substPB-def* **by** *auto*

lemmas *satPB-subst* = *satPB-o*[*symmetric*]

lemma *satPB-o-subst*:
satPB ξ *o substPB* ϱ = *satPB* (*int* ξ *o* ϱ)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *satPB-o*[*symmetric*] **..**

lemma *wtE-o*:

assumes *1*: *wtE* ξ **and** *2*: *wtSB* ϱ
shows *wtE* (*int* ξ *o* ϱ)
unfolding *wtE-def* **proof**
fix *x* **have** *0*: *tpOfV x* = *tpOf* (ϱ *x*) **using** *2* **by** *auto*
show *intT* (*tpOfV x*) ((*int* ξ ∘ ϱ) *x*) **apply**(*subst 0*) **unfolding** *comp-def*
apply(*rule wt-int*[*OF 1*]) **using** *2* **by** *auto*
qed

definition *compE* ϱ ξ *x* ≡ *int* ξ (ϱ *x*)

lemma *wtE-compE*:
assumes *wtSB* ϱ **and** *wtE* ξ **shows** *wtE* (*compE* ϱ ξ)
unfolding *wtE-def* **using** *assms wt-int*
unfolding *wtSB-def compE-def* **by** *fastforce*

lemma *compE-upd*: *compE* (ϱ (*x* := *T*)) ξ = (*compE* ϱ ξ) (*x* := *int* ξ *T*)
unfolding *compE-def*[*abs-def*] **by** *auto*

end

context *Signature* **begin**

fun *fsyms* **where**
fsyms (*Var x*) = {}
|
fsyms (*Fn f Tl*) = {*f*} ∪ ⋃ (*set* (*map fsyms Tl*))

fun *fsymsA* **where**
fsymsA (*Eq T1 T2*) = *fsyms T1* ∪ *fsyms T2*
|
fsymsA (*Pr p Tl*) = ⋃ (*set* (*map fsyms Tl*))

fun *fsymsL* **where**
fsymsL (*Pos at*) = *fsymsA at*
|
fsymsL (*Neg at*) = *fsymsA at*

definition *fsymsC c* = ⋃ (*set* (*map fsymsL c*))

definition *fsymsPB* Φ = ⋃ {*fsymsC c* | *c*. *c* ∈ Φ}

lemma *fsyms-int-cong*:
assumes *S1*: *Struct wtFsym wtPsym arOf resOf intT intF1 intP*
and *S2*: *Struct wtFsym wtPsym arOf resOf intT intF2 intP*
and *0*: ⋀ *f*. *f* ∈ *fsyms T* ⟹ *intF1 f* = *intF2 f*

shows *Struct.int intF1 ξ T = Struct.int intF2 ξ T*
using *0* **proof**(*induct T*)
case (*Fn f Tl*)
hence *1*: *map* (*Struct.int intF1 ξ*) *Tl* = *map* (*Struct.int intF2 ξ*) *Tl*
unfolding *list-all-iff map-ext* **by** *auto*
show *?case*
using *Fn Struct.int.simps*[*OF S1*, *of ξ*] *Struct.int.simps*[*OF S2*, *of ξ*] **apply** *simp*
using *1* **by** *metis*
qed (*auto simp*: *Struct.int.simps*[*OF S1*, *of ξ*] *Struct.int.simps*[*OF S2*, *of ξ*])

lemma *fsyms-satA-cong*:
assumes *S1*: *Struct wtFsym wtPsym arOf resOf intT intF1 intP*
and *S2*: *Struct wtFsym wtPsym arOf resOf intT intF2 intP*
and *0*: ⋀ *f*. *f* ∈ *fsymsA at* ⟹ *intF1 f* = *intF2 f*
shows *Struct.satA intF1 intP ξ at* ⟷ *Struct.satA intF2 intP ξ at*
using *0 fsyms-int-cong*[*OF S1 S2*]
apply(*cases at*)
apply(*fastforce intro*!: *fsyms-int-cong*[*OF S1 S2*, *of - ξ*]
simp: *Struct.satA.simps*[*OF S1*, *of ξ*] *Struct.satA.simps*[*OF S2*, *of ξ*])
apply (*simp add*: *Struct.satA.simps*[*OF S1*, *of ξ*] *Struct.satA.simps*[*OF S2*, *of ξ*])
by (*metis* (*opaque-lifting*, *mono-tags*) *map-eq-conv*)

lemma *fsyms-satL-cong*:
assumes *S1*: *Struct wtFsym wtPsym arOf resOf intT intF1 intP*
and *S2*: *Struct wtFsym wtPsym arOf resOf intT intF2 intP*
and *0*: ⋀ *f*. *f* ∈ *fsymsL l* ⟹ *intF1 f* = *intF2 f*
shows *Struct.satL intF1 intP ξ l* ⟷ *Struct.satL intF2 intP ξ l*
using *0 fsyms-satA-cong*[*OF S1 S2*]
by (*cases l*, *auto simp*: *Struct.satL.simps*[*OF S1*, *of ξ*] *Struct.satL.simps*[*OF S2*, *of ξ*])

lemma *fsyms-satC-cong*:
assumes *S1*: *Struct wtFsym wtPsym arOf resOf intT intF1 intP*
and *S2*: *Struct wtFsym wtPsym arOf resOf intT intF2 intP*
and *0*: ⋀ *f*. *f* ∈ *fsymsC c* ⟹ *intF1 f* = *intF2 f*
shows *Struct.satC intF1 intP ξ c* ⟷ *Struct.satC intF2 intP ξ c*
using *0 fsyms-satL-cong*[*OF S1 S2*]
unfolding *Struct.satC-def*[*OF S1*] *Struct.satC-def*[*OF S2*] *fsymsC-def*
apply (*induct c*) **by** (*fastforce intro*!: *fsyms-satL-cong*[*OF S1 S2*])+

lemma *fsyms-satPB-cong*:
assumes *S1*: *Struct wtFsym wtPsym arOf resOf intT intF1 intP*
and *S2*: *Struct wtFsym wtPsym arOf resOf intT intF2 intP*
and *0*: ⋀ *f*. *f* ∈ *fsymsPB* Φ ⟹ *intF1 f* = *intF2 f*
shows *Struct.satPB intF1 intP ξ* Φ ⟷ *Struct.satPB intF2 intP ξ* Φ
by (*force simp*: *Struct.satPB-def*[*OF S1*] *Struct.satPB-def*[*OF S2*] *fsymsPB-def*
intro!: *fsyms-satC-cong*[*OF S1 S2*] *ball-cong 0*)

lemma *fsymsPB-Un*[*simp*]: *fsymsPB* ($\Phi 1 \cup \Phi 2$) = *fsymsPB* $\Phi 1 \cup$ *fsymsPB* $\Phi 2$
unfolding *fsymsPB-def* **by** *auto*

lemma *fsymsC-append*[*simp*]: *fsymsC* (*c1* @ *c2*) = *fsymsC c1* ∪ *fsymsC c2*
unfolding *fsymsC-def* **by** *auto*

lemma *fsymsPB-sappend-incl*[*simp*]:
fsymsPB ($\Phi 1$ @@ $\Phi 2$) ⊆ *fsymsPB* $\Phi 1 \cup$ *fsymsPB* $\Phi 2$
by (*unfold fsymsPB-def sappend-def*, *fastforce*)

lemma *fsymsPB-sappend*[*simp*]:
assumes *1*: $\Phi 1 \neq \{\}$ **and** *2*: $\Phi 2 \neq \{\}$
shows *fsymsPB* ($\Phi 1$ @@ $\Phi 2$) = *fsymsPB* $\Phi 1 \cup$ *fsymsPB* $\Phi 2$
proof *safe*
fix *x*
{assume *x* ∈ *fsymsPB* $\Phi 1$
then obtain *c1 c2* **where** *x* ∈ *fsymsC c1* **and** *c1* ∈ $\Phi 1$ **and** *c2* ∈ $\Phi 2$
using *2* **unfolding** *fsymsPB-def* **by** *auto*
thus *x* ∈ *fsymsPB* ($\Phi 1$ @@ $\Phi 2$) **unfolding** *sappend-def fsymsPB-def* **by** *fastforce*
}
{assume *x* ∈ *fsymsPB* $\Phi 2$
then obtain *c1 c2* **where** *x* ∈ *fsymsC c2* **and** *c1* ∈ $\Phi 1$ **and** *c2* ∈ $\Phi 2$
using *1* **unfolding** *fsymsPB-def* **by** *auto*
thus *x* ∈ *fsymsPB* ($\Phi 1$ @@ $\Phi 2$) **unfolding** *sappend-def fsymsPB-def* **by** *fastforce*
}
qed(*unfold fsymsPB-def sappend-def*, *fastforce*)

lemma *Struct-upd*:
assumes *Struct wtFsym wtPsym arOf resOf intT intF intP*
and ⋀ *al*. *list-all2 intT* (*arOf ef*) *al* ⟹ *intT* (*resOf ef*) (*EF al*)
shows *Struct wtFsym wtPsym arOf resOf intT* (*intF* (*ef* := *EF*)) *intP*
apply *standard* **using** *assms*
unfolding *Struct-def Struct-axioms-def Tstruct-def* **by** *auto*

end

3.4 Problems

A problem is a potentially infinitary formula in clausal form, i.e., a potentially infinite conjunction of clauses.

locale *Problem* = *Signature wtFsym wtPsym arOf resOf parOf*
for *wtFsym wtPsym*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* :: *'fsym* ⇒ *'tp*
and *parOf* :: *'psym* ⇒ *'tp list*
+
fixes Φ :: (*'fsym*, *'psym*) *prob*

assumes *wt-Φ*: *wtPB* Φ

3.5 Models of a problem

Model of a problem:

locale *Model* = *Problem* + *Struct* +
assumes *SAT*: *SAT* Φ
begin
lemma *sat-Φ*: *wtE* ξ ⟹ *satPB* ξ Φ
using *SAT* **unfolding** *SAT-def* **by** *auto*
end

end

theory *CM*
imports *M*
begin

locale *Tstruct* = *M.Tstruct intT*
for *intT* :: *'tp* ⇒ *univ* ⇒ *bool*

locale *Struct* = *M.Struct wtFsym wtPsym arOf resOf parOf intT intF intP*
for *wtFsym* **and** *wtPsym*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* :: *'psym* ⇒ *'tp list*
and *intT* :: *'tp* ⇒ *univ* ⇒ *bool*
and *intF* **and** *intP*

locale *Model* = *M.Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP*
for *wtFsym* **and** *wtPsym*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* :: *'psym* ⇒ *'tp list* **and** Φ
and *intT* :: *'tp* ⇒ *univ* ⇒ *bool*
and *intF* **and** *intP*

sublocale *Struct* < *Tstruct* **..**
sublocale *Model* < *Struct* **..**

end

4 Monotonicity

theory *Mono* **imports** *CM* **begin**

4.1 Fullness and infiniteness

In a structure, a full type is one that contains all elements of univ (the fixed countable universe):

definition (**in** *Tstruct*) *full* $\sigma \equiv \forall\ d.\ intT\ \sigma\ d$

locale *FullStruct* = *F?* : *Struct* +
assumes *full*: *full* σ
begin
lemma *full2*[*simp*]: *intT* σ *d*
using *full* **unfolding** *full-def* **by** *simp*

lemma *full-True*: *intT* = ($\lambda\ \sigma\ D.$ *True*)
apply(*intro ext*) **by** *auto*
end

locale *FullModel* =
F? : *Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP* +
F? : *FullStruct wtFsym wtPsym arOf resOf parOf intT intF intP*
for *wtFsym* :: *'fsym* $\Rightarrow$ *bool* **and** *wtPsym* :: *'psym* $\Rightarrow$ *bool*
and *arOf* :: *'fsym* $\Rightarrow$ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *intT* **and** *intF* **and** *intP*

An infinite structure is one with all carriers infinite:

locale *InfStruct* = *I?* : *Struct* +
assumes *inf*: *infinite* {*a. intT* σ *a*}

locale *InfModel* =
I? : *Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP* +
I? : *InfStruct wtFsym wtPsym arOf resOf parOf intT intF intP*
for *wtFsym* :: *'fsym* $\Rightarrow$ *bool* **and** *wtPsym* :: *'psym* $\Rightarrow$ *bool*
and *arOf* :: *'fsym* $\Rightarrow$ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *intT* **and** *intF* **and** *intP*

context *Problem* **begin**
abbreviation *SStruct* $\equiv$ *Struct wtFsym wtPsym arOf resOf*
abbreviation *FFullStruct* $\equiv$ *FullStruct wtFsym wtPsym arOf resOf*
abbreviation *IInfStruct* $\equiv$ *InfStruct wtFsym wtPsym arOf resOf*

abbreviation *MModel* $\equiv$ *Model wtFsym wtPsym arOf resOf parOf* Φ
abbreviation *FFullModel* $\equiv$ *FullModel wtFsym wtPsym arOf resOf parOf* Φ
abbreviation *IInfModel* $\equiv$ *InfModel wtFsym wtPsym arOf resOf parOf* Φ
end

Problem that deduces some infiniteness constraints:

locale *ProblemIk* = *Ik?* : *Problem wtFsym wtPsym arOf resOf parOf* Φ
for *wtFsym* :: *'fsym* $\Rightarrow$ *bool* **and** *wtPsym* :: *'psym* $\Rightarrow$ *bool*
and *arOf* :: *'fsym* $\Rightarrow$ *'tp list*
and *resOf* **and** *parOf* **and** Φ

+
fixes *infTp* :: *'tp* ⇒ *bool*

assumes *infTp*:
⋀ *σ intT intF intP* (*a*::*univ*). ⟦*infTp σ*; *MModel intT intF intP*⟧ ⟹ *infinite* {*a*. *intT σ a*}

locale *ModelIk* =
Ik? : *ProblemIk wtFsym wtPsym arOf resOf parOf* Φ *infTp* +
Ik? : *Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP*
for *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp* **and** *intT* **and** *intF* **and** *intP*
begin
lemma *infTp-infinite*[*simp*]: *infTp σ* ⟹ *infinite* {*a*. *intT σ a*}
apply(*rule ProblemIk.infTp*[*of wtFsym wtPsym arOf resOf parOf* Φ *infTp*])
apply *unfold-locales* **by** *simp*
end

4.2 Monotonicity

context *Problem* **begin**

definition
monot ≡
(∃ *intT intF intP*. *MModel intT intF intP*)
⟶
(∃ *intTI intFI intPI*. *IInfModel intTI intFI intPI*)
end

locale *MonotProblem* = *Problem* +
assumes *monot*: *monot*

locale *MonotProblemIk* =
MonotProblem wtFsym wtPsym arOf resOf parOf Φ +
ProblemIk wtFsym wtPsym arOf resOf parOf Φ *infTp*
for *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list* **and** *resOf* **and** *parOf* **and** Φ **and** *infTp*

context *MonotProblem*
begin

definition *MI-pred K* ≡ *IInfModel* (*fst3 K*) (*snd3 K*) (*trd3 K*)

definition *MI* ≡ *SOME K*. *MI-pred K*

lemma *MI-pred*:
assumes *MModel intT intF intP*
shows ∃ *K*. *MI-pred K*

proof−
  obtain *T F R* **where** *MI-pred* (*T*,*F*,*R*)
  using *monot assms* **unfolding** *monot-def MI-pred-def* **by** *auto*
  thus *?thesis* **by** *blast*
qed

lemma *MI-pred-MI*:
assumes *MModel intT intF intP*
shows *MI-pred MI*
using *MI-pred*[*OF assms*] **unfolding** *MI-def* **by** (*rule someI-ex*)

definition *intTI* ≡ *fst3 MI*
definition *intFI* ≡ *snd3 MI*
definition *intPI* ≡ *trd3 MI*

lemma *InfModel-intTI-intFI-intPI*:
assumes *MModel intT intF intP*
shows *IInfModel intTI intFI intPI*
using *MI-pred-MI*[*OF assms*]
unfolding *MI-pred-def intFI-def intPI-def intTI-def* **.**

end

locale *MonotModel* = *M?* : *MonotProblem* + *M?* : *Model*

context *MonotModel* **begin**

lemma *InfModelI*: *IInfModel intTI intFI intPI*
apply(*rule MonotProblem.InfModel-intTI-intFI-intPI*)
apply *standard*
done

end

sublocale *MonotModel* < *InfModel* **where**
intT = *intTI* **and** *intF* = *intFI* **and** *intP* = *intPI*
using *InfModelI* **.**

context *InfModel* **begin**

definition *toFull* σ ≡ *SOME F. bij-betw F* {*a*::*univ. intT* σ *a*} (*UNIV*::*univ set*)
definition *fromFull* σ ≡ *inv-into* {*a*::*univ. intT* σ *a*} (*toFull* σ)

definition *intTF* σ *a* ≡ *True*
definition *intFF f al* ≡ *toFull* (*resOf f*) (*intF f* (*map2 fromFull* (*arOf f*) *al*))
definition *intPF p al* ≡ *intP p* (*map2 fromFull* (*parOf p*) *al*)

lemma *intTF*: *intTF* σ *a*
unfolding *intTF-def* **by** *auto*

lemma *ex-toFull*: $\exists$ *F*. *bij-betw F* {*a*::*univ*. *intT* σ *a*} (*UNIV*::*univ set*)
by (*metis inf card-of-ordIso card-of-UNIV countable-univ UnE*
countable-infinite not-ordLeq-ordLess ordLeq-ordLess-Un-ordIso)

lemma *toFull*: *bij-betw* (*toFull* σ) {*a*. *intT* σ *a*} *UNIV*
by (*metis* (*lifting*) *ex-toFull someI-ex toFull-def*)

lemma *toFull-fromFull*[*simp*]: *toFull* σ (*fromFull* σ *a*) = *a*
by (*metis UNIV-I bij-betw-inv-into-right fromFull-def toFull*)

lemma *fromFull-toFull*[*simp*]: *intT* σ *a* $\Longrightarrow$ *fromFull* σ (*toFull* σ *a*) = *a*
by (*metis CollectI bij-betw-inv-into-left toFull fromFull-def*)

lemma *fromFull-inj*[*simp*]: *fromFull* σ *a* = *fromFull* σ *b* $\longleftrightarrow$ *a* = *b*
by (*metis toFull-fromFull*)

lemma *toFull-inj*[*simp*]:
assumes *intT* σ *a* **and** *intT* σ *b*
shows *toFull* σ *a* = *toFull* σ *b* $\longleftrightarrow$ *a* = *b*
by (*metis assms fromFull-toFull*)

lemma *fromFull*[*simp*]: *intT* σ (*fromFull* σ *a*)
unfolding *fromFull-def*
apply(*rule inv-into-into*[*of a toFull* σ {*a*. *intT* σ *a*}, *simplified*])
using *toFull* **unfolding** *bij-betw-def* **by** *auto*

lemma *toFull-iff-fromFull*:
assumes *intT* σ *a*
shows *toFull* σ *a* = *b* $\longleftrightarrow$ *a* = *fromFull* σ *b*
by (*metis assms fromFull-toFull toFull-fromFull*)

lemma *Tstruct*: *Tstruct intTF*
apply(*unfold-locales*) **unfolding** *intTF-def* **by** *simp*

lemma *FullStruct*: *FullStruct wtFsym wtPsym arOf resOf intTF intFF intPF*
apply (*unfold-locales*) **unfolding** *intTF-def Tstruct.full-def*[*OF Tstruct*] **by** *auto*

end

sublocale *InfModel* < *FullStruct*
where *intT* = *intTF* **and** *intF* = *intFF* **and** *intP* = *intPF*
using *FullStruct* **.**

context *InfModel* **begin**

definition $kE\ \xi \equiv \lambda\ x.\ fromFull\ (tpOfV\ x)\ (\xi\ x)$

lemma $kE[simp]$: $kE\ \xi\ x = fromFull\ (tpOfV\ x)\ (\xi\ x)$
unfolding $kE\text{-}def$ **by** $simp$

lemma $wtE[simp]$: $F.wtE\ \xi$
unfolding $F.wtE\text{-}def$ **by** $simp$

lemma $kE\text{-}wtE[simp]$: $I.wtE\ (kE\ \xi)$
unfolding $I.wtE\text{-}def\ kE\text{-}def$ **by** $simp$

lemma $kE\text{-}int\text{-}toFull$:
assumes ξ: $I.wtE\ (kE\ \xi)$ **and** T: $wt\ T$
shows $toFull\ (tpOf\ T)\ (I.int\ (kE\ \xi)\ T) = F.int\ \xi\ T$
using T **proof**$(induct\ T)$
 case $(Fn\ f\ Tl)$
 have 0: $map\ (I.int\ (kE\ \xi))\ Tl =$
 $map2\ fromFull\ (arOf\ f)\ (map\ (F.int\ \xi)\ Tl)$
 (**is** $map\ ?F\ Tl = map2\ fromFull\ (arOf\ f)\ (map\ ?H\ Tl)$
 is $?L = ?R$)
 proof$(rule\ nth\text{-}equalityI)$
 have l: $length\ Tl = length\ (arOf\ f)$ **using** Fn **by** $simp$
 thus $length\ ?L = length\ ?R$ **by** $simp$
 fix i **assume** $i < length\ ?L$ **hence** i: $i < length\ Tl$ **by** $simp$
 let $?toFull = toFull\ (arOf\ f!i)$ **let** $?fromFull = fromFull\ (arOf\ f!i)$
 have tp: $tpOf\ (Tl\ !\ i) = arOf\ f\ !\ i$ **using** $Fn(2)\ i$ **unfolding** $list\text{-}all\text{-}length$ **by** $auto$
 have wt: $wt\ (Tl!i)$ **using** $Fn\ i$ **by** $(auto\ simp{:}\ list\text{-}all\text{-}iff)$
 have $intT\ (arOf\ f!i)\ (?F\ (Tl!i))$ **using** $I.wt\text{-}int[OF\ \xi\ wt]$ **unfolding** tp **.**
 moreover have $?toFull\ (?F\ (Tl!i)) = ?H\ (Tl!i)$
 using $Fn\ tp\ i$ **by** $(auto\ simp{:}\ list\text{-}all\text{-}iff\ kE\text{-}def)$
 ultimately have $?F\ (Tl!i) = fromFull\ (arOf\ f!i)\ (?H\ (Tl!i))$
 using $toFull\text{-}iff\text{-}fromFull$ **by** $blast$
 thus $?L!i = ?R!i$ **using** $l\ i$ **by** $simp$
 qed
 show $?case$ **unfolding** $I.int.simps\ F.int.simps\ tpOf.simps$ **unfolding** $intFF\text{-}def\ 0$ **..**
qed $simp$

lemma $kE\text{-}int[simp]$:
assumes ξ: $I.wtE\ (kE\ \xi)$ **and** T: $wt\ T$
shows $I.int\ (kE\ \xi)\ T = fromFull\ (tpOf\ T)\ (F.int\ \xi\ T)$
using $kE\text{-}int\text{-}toFull[OF\ assms]$
unfolding $toFull\text{-}iff\text{-}fromFull[OF\ I.wt\text{-}int[OF\ \xi\ T]]$ **.**

lemma $map\text{-}kE\text{-}int[simp]$:
 assumes ξ: $I.wtE\ (kE\ \xi)$ **and** T: $list\text{-}all\ wt\ Tl$
 shows $map\ (I.int\ (kE\ \xi))\ Tl = map2\ fromFull\ (map\ tpOf\ Tl)\ (map\ (F.int\ \xi)\ Tl)$

apply(*rule nth-equalityI*)
apply (*metis* (*lifting*) *length-map length-map2*)
by (*metis T ξ kE-int length-map list-all-length nth-map nth-map2*)

lemma *kE-satA*[*simp*]:
assumes *at*: *wtA at* **and** *ξ*: *I.wtE* (*kE ξ*)
shows *I.satA* (*kE ξ*) *at* ⟷ *F.satA ξ at*
using *assms* **by** (*cases at*, *auto simp add*: *intPF-def*)

lemma *kE-satL*[*simp*]:
assumes *l*: *wtL l* **and** *ξ*: *I.wtE* (*kE ξ*)
shows *I.satL* (*kE ξ*) *l* ⟷ *F.satL ξ l*
using *assms* **by** (*cases l*, *auto*)

lemma *kE-satC*[*simp*]:
assumes *c*: *wtC c* **and** *ξ*: *I.wtE* (*kE ξ*)
shows *I.satC* (*kE ξ*) *c* ⟷ *F.satC ξ c*
unfolding *I.satC-def F.satC-def*
using *assms* **by**(*induct c*, *auto simp add*: *wtC-def*)

lemma *kE-satPB*:
assumes *ξ*: *I.wtE* (*kE ξ*) **shows** *F.satPB ξ* Φ
using *I.sat-*Φ[*OF assms*]
using *wt-*Φ *assms* **unfolding** *I.satPB-def F.satPB-def*
by (*auto simp add*: *wtPB-def*)

lemma *F-SAT*: *F.SAT* Φ
unfolding *F.SAT-def* **using** *kE-satPB kE-wtE* **by** *auto*

lemma *FullModel*: *FullModel wtFsym wtPsym arOf resOf parOf* Φ *intTF intFF intPF*
apply (*unfold-locales*) **using** *F-SAT* **.**

end

sublocale *InfModel* < *FullModel* **where**
intT = *intTF* **and** *intF* = *intFF* **and** *intP* = *intPF*
using *FullModel* **.**

context *MonotProblem* **begin**

definition *intTF* ≡ *InfModel.intTF*
definition *intFF* ≡ *InfModel.intFF arOf resOf intTI intFI*
definition *intPF* ≡ *InfModel.intPF parOf intTI intPI*

Strengthening of the infiniteness condition for monotonicity, replacing infininetess by fullness:

theorem *FullModel-intTF-intFF-intPF*:

assumes *MModel intT intF intP*
shows *FFullModel intTF intFF intPF*
unfolding *intTF-def intFF-def intPF-def*
apply(*rule InfModel.FullModel*) **using** *InfModel-intTI-intFI-intPI*[*OF assms*] **.**

end

sublocale *MonotModel* < *FullModel* **where**
intT = *intTF* **and** *intF* = *intFF* **and** *intP* = *intPF*
using *FullModel* **.**

end

5 The First Monotonicity Calculus

theory *Mcalc*
imports *Mono*
begin

context *ProblemIk* **begin**

5.1 Naked variables

fun *nvT* **where**
nvT (*Var x*) = {*x*}
|
nvT (*Fn f Tl*) = {}

fun *nvA* **where**
nvA (*Eq T1 T2*) = *nvT T1* ∪ *nvT T2*
|
nvA (*Pr p Tl*) = {}

fun *nvL* **where**
nvL (*Pos at*) = *nvA at*
|
nvL (*Neg at*) = {}

definition *nvC c* ≡ ⋃ (*set* (*map nvL c*))

definition *nvPB* ≡ ⋃ *c* ∈ Φ. *nvC c*

lemma *nvT-vars*[*simp*]: *x* ∈ *nvT T* ⟹ *x* ∈ *vars T*
by (*induct T*) (*auto split*: *if-splits*)

lemma *nvA-varsA*[*simp*]: *x* ∈ *nvA at* ⟹ *x* ∈ *varsA at*
by (*cases at*, *auto*)

lemma *nvL-varsL*[*simp*]: *x* ∈ *nvL l* ⟹ *x* ∈ *varsL l*

by (*cases l*, *auto*)

lemma *nvC-varsC*[*simp*]: $x \in nvC\ c \Longrightarrow x \in varsC\ c$
unfolding *varsC-def nvC-def* **by**(*induct c*, *auto*)

lemma *nvPB-varsPB*[*simp*]: $x \in nvPB \Longrightarrow x \in varsPB\ \Phi$
unfolding *varsPB-def nvPB-def* **by** *auto*

5.2 The calculus

inductive *mcalc* (**infix** $\langle\vdash\rangle$ *40*) **where**
[*simp*]: $infTp\ \sigma \Longrightarrow \sigma \vdash c$
|[*simp*]: $(\forall\ x \in nvC\ c.\ tpOfV\ x \neq \sigma) \Longrightarrow \sigma \vdash c$

lemma *mcalc-iff*: $\sigma \vdash c \longleftrightarrow infTp\ \sigma \vee (\forall\ x \in nvC\ c.\ tpOfV\ x \neq \sigma)$
unfolding *mcalc.simps* **by** *simp*

end

locale *ProblemIkMcalc* = *ProblemIk wtFsym wtPsym arOf resOf parOf* Φ *infTp*
for *wtFsym* :: *'fsym* $\Rightarrow$ *bool* **and** *wtPsym* :: *'psym* $\Rightarrow$ *bool*
and *arOf* :: *'fsym* $\Rightarrow$ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp*
+ **assumes** *mcalc*: $\bigwedge \sigma\ c.\ c \in \Phi \Longrightarrow \sigma \vdash c$

locale *ModelIkMcalc* =
ModelIk wtFsym wtPsym arOf resOf parOf Φ *infTp intT intF intP* +
ProblemIkMcalc wtFsym wtPsym arOf resOf parOf Φ *infTp*
for *wtFsym* :: *'fsym* $\Rightarrow$ *bool* **and** *wtPsym* :: *'psym* $\Rightarrow$ *bool*
and *arOf* :: *'fsym* $\Rightarrow$ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp* **and** *intT* **and** *intF* **and** *intP*

5.3 Extension of a structure to an infinite structure by adding indistinguishable elements

context *ModelIkMcalc* **begin**

The projection from univ to a structure:

definition *proj* **where** $proj\ \sigma\ a \equiv if\ intT\ \sigma\ a\ then\ a\ else\ pickT\ \sigma$

lemma *intT-proj*[*simp*]: $intT\ \sigma\ (proj\ \sigma\ a)$
unfolding *proj-def* **using** *pickT* **by** *auto*

lemma *proj-id*[*simp*]: $intT\ \sigma\ a \Longrightarrow proj\ \sigma\ a = a$
unfolding *proj-def* **by** *auto*

lemma *surj-proj*:
assumes $intT\ \sigma\ a$ **shows** $\exists\ b.\ proj\ \sigma\ b = a$
using *assms* **by** (*intro exI*[*of - a*]) *simp*

definition *I-intT* σ (*a*::*univ*) $\equiv$ *infTp* $\sigma \longrightarrow$ *intT* σ *a*
definition *I-intF f al* $\equiv$ *intF f* (*map2 proj* (*arOf f*) *al*)
definition *I-intP p al* $\equiv$ *intP p* (*map2 proj* (*parOf p*) *al*)

lemma *not-infTp-I-intT*[*simp*]: $\neg$ *infTp* $\sigma \Longrightarrow$ *I-intT* σ *a* **unfolding** *I-intT-def*
by *simp*

lemma *infTp-I-intT*[*simp*]: *infTp* $\sigma \Longrightarrow$ *I-intT* σ *a* = *intT* σ *a* **unfolding** *I-intT-def*
by *simp*

lemma *NE-I-intT*: *NE* (*I-intT* σ)
using *NE-intT* **by** (*cases infTp* σ, *auto*)

lemma *I-intF*:
assumes *f*: *wtFsym f* **and** *al*: *list-all2 I-intT* (*arOf f*) *al*
shows *I-intT* (*resOf f*) (*I-intF f al*)
unfolding *I-intT-def I-intF-def* **apply** *safe* **apply**(*rule intF*[*OF f*])
using *al* **unfolding** *list-all2-length* **by** *auto*

lemma *Tstruct-I-intT*: *Tstruct I-intT*
by *standard* (*rule NE-I-intT*)

lemma *inf-I-intT*: *infinite* {*a*. *I-intT* σ *a*}
by (*cases infTp* σ, *auto*)

lemma *InfStruct*: *IInfStruct I-intT I-intF I-intP*
apply *standard* **using** *NE-I-intT I-intF Tstruct-I-intT inf-I-intT* **by** *auto*

end

sublocale *ModelIkMcalc* < *InfStruct* **where**
intT = *I-intT* **and** *intF* = *I-intF* **and** *intP* = *I-intP*
using *InfStruct* **.**

5.4 The soundness of the calculus

In what follows, "Ik" stands for the original (augmented with infiniteness-knowledge) and "I" for the infinite structure constructed from it through the above sublocale statement.

context *ModelIkMcalc* **begin**

The environment translation along the projection:

definition *transE* $\xi \equiv \lambda$ *x*. *proj* (*tpOfV x*) (ξ *x*)

lemma *transE*[*simp*]: *transE* ξ *x* = *proj* (*tpOfV x*) (ξ *x*)
unfolding *transE-def* **by** *simp*

lemma *wtE-transE*[*simp*]: *I.wtE* $\xi \Longrightarrow$ *Ik.wtE* (*transE* ξ)
unfolding *Ik.wtE-def I.wtE-def transE-def* **by** *auto*

abbreviation *Ik-intT* $\equiv$ *intT*
abbreviation *Ik-intF* $\equiv$ *intF*
abbreviation *Ik-intP* $\equiv$ *intP*

lemma *Ik-intT-int*:
assumes *wt*: *Ik.wt T* **and** ξ: *I.wtE* ξ
and *snv*: $\bigwedge \sigma$. *infTp* $\sigma \vee (\forall\ x \in nvT\ T.\ tpOfV\ x \neq \sigma)$
shows *Ik-intT* (*tpOf T*) (*I.int* ξ *T*)
proof(*cases* $\exists\ x.\ T = Var\ x$)
  case *True* **then obtain** *x* **where** *T*: $T = Var\ x$ **by** *auto*
  show *?thesis* **proof**(*cases infTp* (*tpOf T*))
    case *True* **thus** *?thesis* **using** *T* **using** *wtE-transE*[*OF* ξ]
    by (*metis I.wt-int Ik-intT-def* ξ *wt*)
  next
    case *False* **hence** $\forall\ x \in nvT\ T.\ tpOfV\ x \neq tpOf\ T$ **using** *snv* **by** *auto*
    hence *Ik.full* (*tpOf T*) **using** *T* **by** (*cases T*, *simp-all*)
    thus *?thesis* **unfolding** *Ik.full-def* **by** *simp*
  qed
next
  case *False* **hence** *nonVar*: $\neg\ (\exists\ x.\ T = Var\ x)$ **by** (*cases T*, *auto*)
  thus *?thesis* **using** *nonVar wt* **apply**(*induct T*, *force*)
  unfolding *Ik-intF-def tpOf.simps int.simps*
  apply(*rule Ik.intF*, *simp*) **apply**(*rule listAll2-map2I*) **by** *auto*
qed

lemma *int-transE-proj*:
  assumes *wt*: *Ik.wt T*
  shows *Ik.int* (*transE* ξ) *T* = *proj* (*tpOf T*) (*I.int* ξ *T*)
  using *wt* **proof** (*induct T*)
  case (*Fn f Tl*)
  have *0*: *Ik-intT* (*resOf f*) (*Ik-intF f* (*map* (*int* ξ) *Tl*)) (**is** *Ik-intT ?σ ?a*)
    unfolding *Ik-intF-def* **apply**(*rule Ik.intF*)
    using *Fn* **unfolding** *list-all2-length list-all-iff* **by** *auto*
  have *1*: *proj ?σ ?a* = *?a* **using** *proj-id*[*OF 0*] **.**
  show *?case*
    using [[*unfold-abs-def* = *false*]]
    unfolding *Ik.int.simps int.simps tpOf.simps 1*
    unfolding *Ik-intF-def* **apply**(*rule arg-cong*[*of - - intF f*])
  proof (*rule nth-equalityI*)
    have *l*[*simp*]: *length* (*arOf f*) = *length Tl* **using** *Fn* **by** *simp*
    fix *i* **assume** *i* < *length* (*map* (*Ik.int* (*transE* ξ)) *Tl*)
    hence *i*[*simp*]: *i* < *length Tl* **by** *simp*
    have *0*: *arOf f* ! *i* = *tpOf* (*Tl* ! *i*) **using** *Fn* **by** *simp*
    have [*simp*]: *Ik.int* (*transE* ξ) (*Tl* ! *i*) = *proj* (*arOf f* ! *i*) (*I.int* ξ (*Tl* ! *i*))
      unfolding *0* **using** *Fn* **by** (*auto simp*: *list-all-length transE-def*)
    show *map* (*Ik.int* (*transE* ξ)) *Tl* ! *i* =

map2 proj (*arOf f*) (*map* (*I.int* ξ) *Tl*) ! *i*
using *Fn* **unfolding** *list-all-length* **by** *simp*
qed(*use Fn* **in** *simp*)
qed *simp*

lemma *int-transE-snv*[*simp*]:
assumes *wt*: *Ik.wt T* **and** ξ: *I.wtE* ξ **and** *snv*: $\bigwedge \sigma$. *infTp* $\sigma \vee (\forall\ x \in nvT\ T.\ tpOfV\ x \neq \sigma)$
shows *Ik.int* (*transE* ξ) *T* = *I.int* ξ *T*
unfolding *int-transE-proj*[*OF wt*] **apply**(*rule proj-id*)
using *Ik-intT-int*[*OF wt* ξ *snv*] **.**

lemma *int-transE-Fn*:
assumes *wt*: *list-all wt Tl* **and** *f*: *wtFsym f* **and** ξ: *I.wtE* ξ
and *ar*: *arOf f* = *map tpOf Tl*
shows *Ik.int* (*transE* ξ) (*Fn f Tl*) = *I.int* ξ (*Fn f Tl*)
apply(*rule int-transE-snv*) **using** *assms* **by** *auto*

lemma *intP-transE*[*simp*]:
assumes *wt*: *list-all wt Tl* **and** *p*: *wtPsym p* **and** *ar*: *parOf p* = *map tpOf Tl*
shows *Ik-intP p* (*map* (*Ik.int* (*transE* ξ)) *Tl*) = *I-intP p* (*map* (*I.int* ξ) *Tl*)
unfolding *I-intP-def* **apply**(*rule arg-cong*[*of - - Ik-intP p*])
apply(*rule nth-equalityI*) **using** *assms*
using *int-transE-proj* **unfolding** *list-all-length* **by** *auto*

lemma *satA-snvA-transE*[*simp*]:
assumes *wtA*: *Ik.wtA at* **and** ξ: *I.wtE* ξ
and *pA*: $\bigwedge \sigma$. *infTp* $\sigma \vee (\forall\ x \in nvA\ at.\ tpOfV\ x \neq \sigma)$
shows *Ik.satA* (*transE* ξ) *at* $\longleftrightarrow$ *I.satA* ξ *at*
using *assms* **apply** (*cases at*, *simp-all add*: *ball-Un*) **by** (*metis int-transE-snv*)

lemma *satA-transE*[*simp*]:
assumes *wtA*: *Ik.wtA at* **and** *I.satA* ξ *at*
shows *Ik.satA* (*transE* ξ) *at*
using *assms* **apply**(*cases at*) **using** *int-transE-proj* **by** *auto*

lemma *satL-snvL-transE*[*simp*]:
assumes *wtL*: *Ik.wtL l* **and** ξ: *I.wtE* ξ
and *pL*: $\bigwedge \sigma$. *infTp* $\sigma \vee (\forall\ x \in nvL\ l.\ tpOfV\ x \neq \sigma)$ **and** *Ik.satL* (*transE* ξ) *l*
shows *I.satL* ξ *l*
using *assms* **by** (*cases l*) *auto*

lemma *satC-snvC-transE*[*simp*]:
assumes *wtC*: *Ik.wtC c* **and** ξ: *I.wtE* ξ
and *pC*: $\bigwedge \sigma$. $\sigma \vdash c$ **and** *Ik.satC* (*transE* ξ) *c*
shows *I.satC* ξ *c*
using *assms* **unfolding** *Ik.mcalc-iff Ik.satC-def satC-def Ik.wtC-def nvC-def*
unfolding *list-all-iff list-ex-iff* **apply** *simp* **by** (*metis nth-mem satL-snvL-transE*)

lemma *satPB-snvPB-transE*[*simp*]:
assumes ξ: *I.wtE* ξ **shows** *I.satPB* ξ Φ
using *Ik.wt-*Φ *Ik.sat-*Φ[*OF wtE-transE*[*OF* ξ]]
using *mcalc* ξ **unfolding** *Ik.satPB-def satPB-def Ik.wtPB-def nvPB-def* **by** *auto*

lemma *I-SAT*: *I.SAT* Φ **unfolding** *I.SAT-def* **by** *auto*

lemma *InfModel*: *IInfModel I-intT I-intF I-intP*
by *standard* (*rule I-SAT*)

end

sublocale *ModelIkMcalc* < *inf?*: *InfModel* **where**
intT = *I-intT* **and** *intF* = *I-intF* **and** *intP* = *I-intP*
using *InfModel* **.**

context *ProblemIkMcalc* **begin**

abbreviation *MModelIkMcalc* ≡ *ModelIkMcalc wtFsym wtPsym arOf resOf parOf* Φ *infTp*

theorem *monot*: *monot*
unfolding *monot-def* **proof** *safe*
  fix *intT intF intP* **assume** *MModel intT intF intP*
  hence *M*: *MModelIkMcalc intT intF intP*
  unfolding *ModelIkMcalc-def ModelIk-def* **apply** *safe* **..**
  show ∃ *intTI intFI intPI*. *IInfModel intTI intFI intPI*
  using *ModelIkMcalc.InfModel*[*OF M*] **by** *auto*
qed

end

Final theorem in sublocale form: Any problem that passes the monotonicity calculus is monotonic:

sublocale *ProblemIkMcalc* < *MonotProblem*
by *standard* (*rule monot*)

end

theory *T-G-Prelim*
imports *Mcalc*
begin

locale *ProblemIkTpart* =
Ik? : *ProblemIk wtFsym wtPsym arOf resOf parOf* Φ *infTp*

for *wtFsym* :: *'fsym* ⇒ *bool*
and *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp* +
fixes
prot :: *'tp* ⇒ *bool*
and *protFw* :: *'tp* ⇒ *bool*
assumes
tp-disj: ⋀ σ. ¬ *prot* σ ∨ ¬ *protFw* σ
and *tp-mcalc*: ⋀ σ. *prot* σ ∨ *protFw* σ ∨ (∀ *c* ∈ Φ. σ ⊢ *c*)
begin

definition *isRes* **where** *isRes* σ ≡ ∃ *f*. *wtFsym f* ∧ *resOf f* = σ

definition *unprot* σ ≡ ¬ *prot* σ ∧ ¬ *protFw* σ

lemma *unprot-mcalc*[*simp*]: ⟦*unprot* σ; *c* ∈ Φ⟧ ⟹ σ ⊢ *c*
using *tp-mcalc* **unfolding** *unprot-def* **by** *auto*

end

locale *ModelIkTpart* =
Ik? : *ProblemIkTpart wtFsym wtPsym arOf resOf parOf* Φ *infTp prot protFw* +
Ik? : *Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP*
for *wtFsym* :: *'fsym* ⇒ *bool*
and *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp* **and** *prot* **and** *protFw*
and *intT* **and** *intF* **and** *intP*

end

6 The Second Monotonicity Calculus

theory *Mcalc2*
imports *Mono*
begin

6.1 Extension policies

Extension policy: copy, true or false extension:

datatype *epol* = *Cext* | *Text* | *Fext*

Problem with infinite knowledge and predicate-extension policy:

locale *ProblemIkPol* = *ProblemIk wtFsym wtPsym arOf resOf parOf* Φ *infTp*
for *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp*
+ **fixes** *pol* :: *'tp* ⇒ *'psym* ⇒ *epol*
and
grdOf ::
(*'fsym*, *'psym*) *cls* ⇒ (*'fsym*, *'psym*) *lit* ⇒ *var* ⇒ (*'fsym*, *'psym*) *lit*

context *ProblemIkPol* **begin**

6.2 Naked variables

fun *nv2T* **where**
nv2T (*Var x*) = {*x*}
|
nv2T (*Fn f Tl*) = {}

fun *nv2L* **where**
nv2L (*Pos* (*Eq T1 T2*)) = *nv2T T1* ∪ *nv2T T2*
|
nv2L (*Neg* (*Eq T1 T2*)) = {}
|
nv2L (*Pos* (*Pr p Tl*)) = {*x* ∈ ⋃ (*set* (*map nv2T Tl*)) . *pol* (*tpOfV x*) *p* = *Fext*}
|
nv2L (*Neg* (*Pr p Tl*)) = {*x* ∈ ⋃ (*set* (*map nv2T Tl*)) . *pol* (*tpOfV x*) *p* = *Text*}

definition *nv2C c* ≡ ⋃ (*set* (*map nv2L c*))

lemma *in-nv2T*: *x* ∈ *nv2T T* ⟷ *T* = *Var x*
apply(*cases T*)
apply (*metis equals0D nv2T.simps singleton-iff*)
by *simp*

lemma *nv2T-vars*[*simp*]: *x* ∈ *nv2T T* ⟹ *x* ∈ *vars T*
by (*induct T*, *auto split*: *if-splits*)

lemma *nv2L-varsL*[*simp*]:
assumes *x* ∈ *nv2L l* **shows** *x* ∈ *varsL l*
proof (*cases l*)
case (*Pos at*)
show *?thesis* **proof**(*cases at*)
case (*Pr p Tl*) **thus** *?thesis* **using** *assms* **unfolding** *Pos*
apply(*cases pol* (*tpOfV x*) *p*, *simp-all*) **by** (*metis nv2T-vars*)
qed(*insert assms*, *unfold Pos*, *auto*)
next
case (*Neg at*)
show *?thesis* **proof**(*cases at*)

case (*Pr p Tl*) **thus** *?thesis* **using** *assms* **unfolding** *Neg*
apply(*cases pol* (*tpOfV x*) *p*, *simp-all*) **by** (*metis nv2T-vars*)
qed(*insert assms*, *unfold Neg*, *auto*)
qed

lemma *nv2C-varsC*[*simp*]: $x \in nv2C\ c \Longrightarrow x \in varsC\ c$
unfolding *varsC-def nv2C-def* **by** (*induct c*, *auto*)

6.3 The calculus

The notion of a literal being a guard for a (typed) variable:

fun *isGuard* :: *var* ⇒ (*'fsym*,*'psym*) *lit* ⇒ *bool* **where**
$isGuard\ x\ (Pos\ (Eq\ T1\ T2)) \longleftrightarrow False$
|
$isGuard\ x\ (Neg\ (Eq\ T1\ T2)) \longleftrightarrow$
$(T1 = Var\ x \wedge (\exists\ f\ Tl.\ T2 = Fn\ f\ Tl)) \vee$
$(T2 = Var\ x \wedge (\exists\ f\ Tl.\ T1 = Fn\ f\ Tl))$
|
$isGuard\ x\ (Pos\ (Pr\ p\ Tl)) \longleftrightarrow x \in \bigcup\ (set\ (map\ nv2T\ Tl)) \wedge pol\ (tpOfV\ x)\ p = Text$
|
$isGuard\ x\ (Neg\ (Pr\ p\ Tl)) \longleftrightarrow x \in \bigcup\ (set\ (map\ nv2T\ Tl)) \wedge pol\ (tpOfV\ x)\ p = Fext$

The monotonicity calculus from the Classen et. al. paper, applied to non-infinite types only: it checks that any variable in any literal of any clause is indeed guarded by its guard:

inductive *mcalc2* (**infix** ‹⊢2› *40*) **where**
[*simp*]: $infTp\ \sigma \Longrightarrow \sigma \vdash 2\ c$
|[*simp*]: $(\bigwedge\ l\ x.\ [\![l \in set\ c;\ x \in nv2L\ l;\ tpOfV\ x = \sigma]\!]$
$\Longrightarrow isGuard\ x\ (grdOf\ c\ l\ x)) \Longrightarrow \sigma \vdash 2\ c$

lemma *mcalc2-iff*:
$\sigma \vdash 2\ c \longleftrightarrow$
$infTp\ \sigma \vee (\forall\ l\ x.\ l \in set\ c \wedge x \in nv2L\ l \wedge tpOfV\ x = \sigma \longrightarrow isGuard\ x\ (grdOf\ c\ l\ x))$
unfolding *mcalc2.simps* **by** *auto*

end

locale *ProblemIkPolMcalc2* = *ProblemIkPol wtFsym wtPsym arOf resOf parOf* Φ *infTp pol grdOf*
for *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp* **and** *pol* **and** *grdOf*
+ **assumes**

grdOf: $\bigwedge\ c\ l\ x.\ [\![c \in \Phi;\ l \in set\ c;\ x \in nv2L\ l;\ \neg\ infTp\ (tpOfV\ x)]\!]$

$\Longrightarrow$ *grdOf c l x* $\in$ *set c*
and *mcalc2*: $\bigwedge \sigma\ c.\ c \in \Phi \Longrightarrow \sigma \vdash 2\ c$
begin
lemma *wtL-grdOf*[*simp*]:
assumes $c \in \Phi$ **and** $l \in$ *set c* **and** $x \in$ *nv2L l* **and** $\neg$ *infTp* (*tpOfV x*)
shows *wtL* (*grdOf c l x*)
using *grdOf*[*OF assms*] **by** (*metis assms list-all-iff wtC-def wtPB-def wt-*Φ)
end

locale *ModelIkPolMcalc2* =
ModelIk wtFsym wtPsym arOf resOf parOf Φ *infTp intT intF intP* +
ProblemIkPolMcalc2 wtFsym wtPsym arOf resOf parOf Φ *infTp pol grdOf*
for *wtFsym* :: *'fsym* $\Rightarrow$ *bool* **and** *wtPsym* :: *'psym* $\Rightarrow$ *bool*
and *arOf* :: *'fsym* $\Rightarrow$ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp pol* **and** *grdOf* **and** *intT* **and** *intF* **and** *intP*

end

theory *Mcalc2C*
imports *Mcalc2*
begin

6.4 Constant policy on types

Currently our soundness proof only covers the case of the calculus having different extension policies for different predicates, but not for differnt types versus the same predicate. This is sufficient for our purpose of proving soundness of the guard encodings.

locale *ProblemIkPolMcalc2C* =
ProblemIkPolMcalc2 wtFsym wtPsym arOf resOf parOf Φ *infTp pol grdOf*
for *wtFsym* :: *'fsym* $\Rightarrow$ *bool* **and** *wtPsym* :: *'psym* $\Rightarrow$ *bool*
and *arOf* :: *'fsym* $\Rightarrow$ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp* **and** *pol* **and** *grdOf*
+ **assumes** *pol-ct*: *pol* $\sigma 1\ P$ = *pol* $\sigma 2\ P$

context *ProblemIkPolMcalc2C* **begin**

definition *polC* $\equiv$ *pol any*

lemma *pol-polC*: *pol* $\sigma\ P$ = *polC P*
unfolding *polC-def* **using** *pol-ct* **by** *auto*

lemma *nv2L-simps*[*simp*]:
nv2L (*Pos* (*Pr p Tl*)) = (*case polC p of Fext* $\Rightarrow \bigcup$ (*set* (*map nv2T Tl*)) |- $\Rightarrow$ {})
nv2L (*Neg* (*Pr p Tl*)) = (*case polC p of Text* $\Rightarrow \bigcup$ (*set* (*map nv2T Tl*)) |- $\Rightarrow$ {})
by (*auto split*: *epol.splits simp*: *pol-polC*)

declare *nv2L.simps*(*3*,*4*)[*simp del*]

lemma *isGuard-simps*[*simp*]:
isGuard x (*Pos* (*Pr p Tl*)) ⟷ *x* ∈ ⋃ (*set* (*map nv2T Tl*)) ∧ *polC p* = *Text*
isGuard x (*Neg* (*Pr p Tl*)) ⟷ *x* ∈ ⋃ (*set* (*map nv2T Tl*)) ∧ *polC p* = *Fext*
by (*auto simp*: *pol-polC*)

declare *isGuard.simps*(*3*,*4*)[*simp del*]

end

locale *ModelIkPolMcalc2C* =
ModelIk wtFsym wtPsym arOf resOf parOf Φ *infTp intT intF intP* +
ProblemIkPolMcalc2C wtFsym wtPsym arOf resOf parOf Φ *infTp pol grdOf*
for *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp pol* **and** *grdOf* **and** *intT* **and** *intF* **and** *intP*

6.5 Extension of a structure to an infinte structure by adding indistinguishable elements

context *ModelIkPolMcalc2C* **begin**

definition *proj* **where** *proj* σ *a* ≡ *if intT* σ *a then a else pickT* σ

lemma *intT-proj*[*simp*]: *intT* σ (*proj* σ *a*)
unfolding *proj-def* **using** *pickT* **by** *auto*

lemma *proj-id*[*simp*]: *intT* σ *a* ⟹ *proj* σ *a* = *a*
unfolding *proj-def* **by** *auto*

lemma *map-proj-id*[*simp*]:
assumes *list-all2 intT* σ*l al*
shows *map2 proj* σ*l al* = *al*
apply(*rule nth-equalityI*)
using *assms* **unfolding** *list-all2-length* **by** *auto*

lemma *surj-proj*:
assumes *intT* σ *a* **shows** ∃ *b*. *proj* σ *b* = *a*
using *assms* **by** (*intro exI*[*of - a*]) *simp*

definition *I-intT* σ (*a*::*univ*) ≡ *infTp* σ ⟶ *intT* σ *a*
definition *I-intF f al* ≡ *intF f* (*map2 proj* (*arOf f*) *al*)
definition

I-intP p al ≡
case polC p of
Cext ⇒ *intP p* (*map2 proj* (*parOf p*) *al*)
|*Text* ⇒ *if list-all2 intT* (*parOf p*) *al then intP p al else True*
|*Fext* ⇒ *if list-all2 intT* (*parOf p*) *al then intP p al else False*

lemma *not-infTp-I-intT*[*simp*]: ¬ *infTp σ* ⟹ *I-intT σ a*
unfolding *I-intT-def* **by** *simp*

lemma *infTp-I-intT*[*simp*]: *infTp σ* ⟹ *I-intT σ a* = *intT σ a*
unfolding *I-intT-def* **by** *simp*

lemma *NE-I-intT*: *NE* (*I-intT σ*)
using *NE-intT* **by** (*cases infTp σ*, *auto*)

lemma *I-intP-Cext*[*simp*]:
polC p = *Cext* ⟹ *I-intP p al* = *intP p* (*map2 proj* (*parOf p*) *al*)
unfolding *I-intP-def* **by** *simp*

lemma *I-intP-Text-imp*[*simp*]:
assumes *polC p* = *Text* **and** *intP p al*
shows *I-intP p al*
using *assms* **unfolding** *I-intP-def* **by** *auto*

lemma *I-intP-Fext-imp*[*simp*]:
assumes *polC p* = *Fext* **and** ¬ *intP p al*
shows ¬ *I-intP p al*
using *assms* **unfolding** *I-intP-def* **by** (*cases list-all2 intT* (*parOf p*) *al*, *auto*)

lemma *I-intP-intT*[*simp*]:
assumes *list-all2 intT* (*parOf p*) *al*
shows *I-intP p al* = *intP p al*
using *assms* **unfolding** *I-intP-def* **by** (*cases polC p*) *auto*

lemma *I-intP-Text-not-intT*[*simp*]:
assumes *polC p* = *Text* **and** ¬ *list-all2 intT* (*parOf p*) *al*
shows *I-intP p al*
using *assms* **unfolding** *I-intP-def* **by** *auto*

lemma *I-intP-Fext-not-intT*[*simp*]:
assumes *polC p* = *Fext* **and** ¬ *list-all2 intT* (*parOf p*) *al*
shows ¬ *I-intP p al*
using *assms* **unfolding** *I-intP-def* **by** *auto*

lemma *I-intF*:
assumes *f*: *wtFsym f* **and** *al*: *list-all2 I-intT* (*arOf f*) *al*
shows *I-intT* (*resOf f*) (*I-intF f al*)
unfolding *I-intT-def I-intF-def* **apply** *safe* **apply**(*rule intF*[*OF f*])
using *al* **unfolding** *list-all2-length* **by** *auto*

lemma *Tstruct-I-intT*: *Tstruct I-intT*
by *standard* (*rule NE-I-intT*)

lemma *inf-I-intT*: *infinite* {*a*. *I-intT* σ *a*}
by (*cases infTp* σ, *auto*)

lemma *InfStruct*: *IInfStruct I-intT I-intF I-intP*
apply *standard* **using** *NE-I-intT I-intF Tstruct-I-intT inf-I-intT* **by** *auto*

end

sublocale *ModelIkPolMcalc2C* < *InfStruct* **where**
intT = *I-intT* **and** *intF* = *I-intF* **and** *intP* = *I-intP*
using *InfStruct* **.**

6.6 The soundness of the calculus

context *ModelIkPolMcalc2C* **begin**

definition *transE* $\xi \equiv \lambda$ *x*. *proj* (*tpOfV x*) (ξ *x*)

lemma *transE*[*simp*]: *transE* ξ *x* = *proj* (*tpOfV x*) (ξ *x*)
unfolding *transE-def* **by** *simp*

lemma *wtE-transE*[*simp*]: *I.wtE* $\xi \Longrightarrow$ *Ik.wtE* (*transE* ξ)
unfolding *Ik.wtE-def I.wtE-def transE-def* **by** *auto*

abbreviation *Ik-intT* ≡ *intT*
abbreviation *Ik-intF* ≡ *intF*
abbreviation *Ik-intP* ≡ *intP*

lemma *Ik-intT-int*:
assumes *wt*: *Ik.wt T* **and** ξ: *I.wtE* ξ
and *nv2T*: *infTp* (*Ik.tpOf T*) ∨ ($\forall$ *x* ∈ *nv2T T*. *tpOfV x* ≠ *Ik.tpOf T*)
shows *Ik-intT* (*Ik.tpOf T*) (*I.int* ξ *T*)
proof(*cases* $\exists$ *x*. *T* = *Var x*)
  case *True* **then obtain** *x* **where** *T*: *T* = *Var x* **by** *auto*
  show *?thesis* **proof**(*cases infTp* (*tpOf T*))
    case *True* **thus** *?thesis* **using** *T* **using** *wtE-transE*[*OF* ξ]
    by (*metis I.wt-int I-intT-def* ξ *wt*)
  next
    case *False* **hence** $\forall$ *x* ∈ *nv2T T*. *tpOfV x* ≠ *tpOf T* **using** *nv2T* **by** *auto*
    hence *Ik.full* (*tpOf T*) **using** *T* **by** (*cases T*, *simp-all*)
    thus *?thesis* **unfolding** *Ik.full-def* **by** *simp*
  qed
next
  case *False* **hence** *nonVar*: ¬ ($\exists$ *x*. *T* = *Var x*) **by** (*cases T*, *auto*)
  thus *?thesis* **using** *nonVar wt* **apply**(*induct T*, *force*)

unfolding *I-intF-def tpOf.simps int.simps*
apply(*rule Ik.intF*, *simp*) **apply**(*rule listAll2-map2I*) **by** *auto*
qed

lemma *int-transE-proj*:
assumes *wt*: *Ik.wt T*
shows *Ik.int* (*transE ξ*) *T* = *proj* (*tpOf T*) (*I.int ξ T*)
using *wt* **proof** (*induct T*)
case (*Fn f Tl*)
have *0*: *Ik-intT* (*resOf f*) (*I-intF f* (*map* (*int ξ*) *Tl*)) (**is** *Ik-intT ?σ ?a*)
unfolding *I-intF-def* **apply**(*rule Ik.intF*)
using *Fn* **unfolding** *list-all2-length list-all-iff* **by** *auto*
have *1*: *proj ?σ ?a* = *?a* **using** *proj-id*[*OF 0*] **.**
show *?case*
using [[*unfold-abs-def* = *false*]]
unfolding *Ik.int.simps int.simps tpOf.simps 1*
unfolding *I-intF-def* **apply**(*rule arg-cong*[*of - - intF f*])
proof (*rule nth-equalityI*)
have *l*[*simp*]: *length* (*arOf f*) = *length Tl* **using** *Fn* **by** *simp*
fix *i* **assume** *i* < *length* (*map* (*Ik.int* (*transE ξ*)) *Tl*)
hence *i*[*simp*]: *i* < *length Tl* **by** *simp*
have *0*: *arOf f* ! *i* = *tpOf* (*Tl* ! *i*) **using** *Fn* **by** *simp*
have [*simp*]: *Ik.int* (*transE ξ*) (*Tl* ! *i*) = *proj* (*arOf f* ! *i*) (*I.int ξ* (*Tl* ! *i*))
unfolding *0* **using** *Fn* **by** (*auto simp*: *list-all-length transE-def*)
show *map* (*Ik.int* (*transE ξ*)) *Tl* ! *i* =
map2 proj (*arOf f*) (*map* (*I.int ξ*) *Tl*) ! *i*
using *Fn* **unfolding** *list-all-length* **by** *simp*
qed(*insert Fn*, *simp*)
qed *simp*

lemma *int-transE-nv2T*:
assumes *wt*: *Ik.wt T* **and** *ξ*: *I.wtE ξ*
and *nv2T*: *infTp* (*Ik.tpOf T*) ∨ (∀ *x* ∈ *nv2T T*. *tpOfV x* ≠ *Ik.tpOf T*)
shows *Ik.int* (*transE ξ*) *T* = *I.int ξ T*
unfolding *int-transE-proj*[*OF wt*] **apply**(*rule proj-id*)
using *Ik-intT-int*[*OF wt ξ nv2T*] **.**

lemma *isGuard-not-satL-intT*:
assumes *wtL*: *Ik.wtL l*
and *ns*: ¬ *I.satL ξ l*
and *g*: *isGuard x l* **and** *ξ*: *I.wtE ξ*
shows *Ik-intT* (*tpOfV x*) (*ξ x*) (**is** *Ik-intT ?σ* (*ξ x*))

proof(*cases l*)
case (*Pos at*) **show** *?thesis* **proof**(*cases at*)
case (*Pr p Tl*)
then obtain *T* **where** *Tin*: *T* ∈ *set Tl* **and** *x*: *x* ∈ *nv2T T* **and** *pol*: *polC p*
= *Text*

```
    using g unfolding Pos Pr by auto
    hence T: T = Var x by (simp add: in-nv2T)
    obtain i where i: i < length Tl and Ti: T = Tl!i using Tin
    by (metis in-set-conv-nth)
    hence 0 : wt T parOf p ! i = ?σ using wtL unfolding Pos Pr T
    apply (simp-all add: list-all-iff) by (metis T x in-nv2T tpOf.simps)
    have list-all2 Ik-intT (parOf p) (map (I.int ξ) Tl) (is ?phi)
    using ns unfolding Pos Pr using pol by (cases ?phi, auto)
    hence Ik-intT ?σ (I.int ξ T)
   using ns 0 i Ti unfolding Pos Pr by (auto simp add: list-all2-length nth-map)
    thus ?thesis unfolding T by simp
  qed(insert g, unfold Pos, simp)
next
  case (Neg at)  show ?thesis proof(cases at)
    case (Eq T1 T2)
    hence 0: T1 = Var x ∨ T2 = Var x using g unfolding Neg by auto
    hence wt1: Ik.wt T1 Ik.tpOf T1 = tpOfV x
    and wt2: Ik.wt T2 Ik.tpOf T2 = tpOfV x
    using wtL unfolding Neg Eq by auto
    have eq: I.int ξ T1 = I.int ξ T2 using ns unfolding Neg Eq by simp
    show ?thesis proof(cases T1 = Var x)
      case True note T1 = True obtain f Tl where T2 = Fn f Tl
      using g T1 Eq unfolding Neg by auto
      hence ⋀ σ. infTp σ ∨ (∀ x ∈ nv2T T2. tpOfV x ≠ σ) by auto
     hence 1: I.int ξ T2 = Ik.int (transE ξ) T2 using int-transE-nv2T wt2 ξ by
auto
      have Ik-intT ?σ (I.int ξ T1) unfolding eq 1 using wt2 ξ Ik.wt-int by force
      thus ?thesis unfolding T1 by simp
    next
      case False then obtain f Tl where T2: T2 = Var x and T1 = Fn f Tl
      using Eq Neg g by auto
      hence ⋀ σ. infTp σ ∨ (∀ x ∈ nv2T T1. tpOfV x ≠ σ) by simp
     hence 1: I.int ξ T1 = Ik.int (transE ξ) T1 using int-transE-nv2T wt1 ξ by
auto
      have Ik-intT ?σ (I.int ξ T2) unfolding eq[symmetric] 1
      using wt1 ξ Ik.wt-int by force
      thus ?thesis unfolding T2 by simp
    qed
  next
    case (Pr p Tl)
    then obtain T where Tin: T ∈ set Tl and x: x ∈ nv2T T and pol: polC p
= Fext
    using g unfolding Neg Pr by auto
    hence T: T = Var x by (simp add: in-nv2T)
    obtain i where i: i < length Tl and Ti: T = Tl!i using Tin
    by (metis in-set-conv-nth)
    hence 0 : wt T parOf p ! i = ?σ using wtL unfolding Neg Pr T
    apply (simp-all add: list-all-iff) by (metis T x in-nv2T tpOf.simps)
    have list-all2 Ik-intT (parOf p) (map (I.int ξ) Tl) (is ?phi)
```

```
    using ns unfolding Neg Pr using pol by (cases ?phi, auto)
    hence Ik-intT ?σ (I.int ξ T)
    using ns 0 i Ti unfolding Neg Pr by (auto simp add: list-all2-length nth-map)
    thus ?thesis unfolding T by simp
  qed
qed

lemma int-transE[simp]:
assumes wt: Ik.wt T and ξ: I.wtE ξ and
nv2T: ⋀ x. ⟦¬ infTp (tpOfV x); x ∈ nv2T T⟧ ⟹
         ∃ l. Ik.wtL l ∧ ¬ I.satL ξ l ∧ isGuard x l
shows Ik.int (transE ξ) T = I.int ξ T
proof(cases infTp (Ik.tpOf T) ∨ (∀ x ∈ nv2T T. tpOfV x ≠ Ik.tpOf T))
  case True
  thus ?thesis using int-transE-nv2T[OF wt ξ] by auto
next
  define σ where σ = Ik.tpOf T
  case False then obtain x where i: ¬ infTp σ and x: x ∈ nv2T T
  unfolding σ-def by auto
  hence T: T = Var x by (simp add: in-nv2T)
  hence σ: σ = tpOfV x unfolding σ-def by simp
  obtain l where 0: Ik.wtL l ¬ I.satL ξ l isGuard x l
  using nv2T[OF i[unfolded σ] x] by auto
  show ?thesis unfolding T using isGuard-not-satL-intT[OF 0 ξ] by simp
qed

lemma intT-int-transE[simp]:
assumes wt: Ik.wt T and ξ: I.wtE ξ and
nv2T: ⋀ x. ⟦¬ infTp (tpOfV x); x ∈ nv2T T⟧ ⟹
         ∃ l. Ik.wtL l ∧ ¬ I.satL ξ l ∧ isGuard x l
shows Ik-intT (Ik.tpOf T) (I.int ξ T)
proof−
  have 0: I.int ξ T = Ik.int (transE ξ) T using int-transE[OF assms] by simp
  show ?thesis unfolding 0 using Ik.wt-int[OF wtE-transE[OF ξ] wt] .
qed

lemma map-int-transE-nv2T[simp]:
assumes wt: list-all Ik.wt Tl and ξ: I.wtE ξ and
nv2T: ⋀ x. ⟦¬ infTp (tpOfV x); ∃ T∈set Tl. x ∈ nv2T T⟧ ⟹
         ∃ l. Ik.wtL l ∧ ¬ I.satL ξ l ∧ isGuard x l
shows map (Ik.int (transE ξ)) Tl = map (I.int ξ) Tl
apply(rule nth-equalityI) using assms by (force simp: list-all-iff intro: int-transE)+

lemma list-all2-intT-int-transE-nv2T[simp]:
assumes wt: list-all Ik.wt Tl and ξ: I.wtE ξ and
nv2T: ⋀ x. ⟦¬ infTp (tpOfV x); ∃ T∈set Tl. x ∈ nv2T T⟧ ⟹
         ∃ l. Ik.wtL l ∧ ¬ I.satL ξ l ∧ isGuard x l
shows list-all2 Ik-intT (map Ik.tpOf Tl) (map (I.int ξ) Tl)
unfolding list-all2-length using assms
```

unfolding *list-all-iff* **apply** *simp-all* **by** (*metis intT-int-transE nth-mem*)

lemma *map-proj-transE*[*simp*]:
assumes *wt*: *list-all wt Tl*
shows *map* (*Ik.int* (*transE* ξ)) *Tl* =
  map2 proj (*map tpOf Tl*) (*map* (*I.int* ξ) *Tl*)
apply(*rule nth-equalityI*) **using** *assms*
using *int-transE-proj* **unfolding** *list-all-length* **by** *auto*

lemma *satL-transE*[*simp*]:
assumes *wtL*: *Ik.wtL l* **and** ξ: *I.wtE* ξ **and**
nv2T: $\bigwedge$ *x*. $[\![\neg$ *infTp* (*tpOfV x*); *x* $\in$ *nv2L l*$]\!] \Longrightarrow$
  $\exists$ *l'*. *Ik.wtL l'* $\wedge \neg$ *I.satL* ξ *l'* $\wedge$ *isGuard x l'*
and *Ik.satL* (*transE* ξ) *l*
shows *I.satL* ξ *l*
proof(*cases l*)
 case (*Pos at*) **show** *?thesis* **proof** (*cases at*)
  case (*Pr p Tl*) **show** *?thesis* **using** *assms* **unfolding** *Pos Pr*
  apply(*cases polC p*)
   apply *force*
   apply(*cases list-all2 intT* (*map Ik.tpOf Tl*) (*map* (*I.int* ξ) *Tl*), *force*, *force*)
   by *simp*
 qed(*insert assms*, *unfold Pos*, *simp*)
next
 case (*Neg at*) **show** *?thesis* **proof** (*cases at*)
  case (*Pr p Tl*) **show** *?thesis* **using** *assms* **unfolding** *Neg Pr*
  apply(*cases polC p*)
   apply *force* **apply** *force*
   by (*cases list-all2 intT* (*map Ik.tpOf Tl*) (*map* (*I.int* ξ) *Tl*), *force*, *force*)
 qed(*insert assms int-transE-proj*, *unfold Neg*, *auto*)
qed

lemma *satPB-transE*[*simp*]:
assumes ξ: *I.wtE* ξ **shows** *I.satPB* ξ Φ
unfolding *I.satPB-def* **proof** *safe*
 fix *c* **assume** *cin*: *c* $\in$ Φ **let** *?thesis* = *I.satC* ξ *c*
 have *mc*: $\bigwedge$ σ. $\sigma \vdash 2$ *c* **using** *mcalc2*[*OF cin*] **.**
 have *c*: *Ik.satC* (*transE* ξ) *c*
 using *sat-*Φ *wtE-transE*[*OF* ξ] *cin* **unfolding** *Ik.satPB-def* **by** *auto*
 have *wtC*: *Ik.wtC c* **using** *wt-*Φ *cin* **unfolding** *wtPB-def* **by** *auto*
 obtain *l* **where** *lin*: *l* $\in$ *set c* **and** *l*: *Ik.satL* (*transE* ξ) *l*
 using *c* **unfolding** *Ik.satC-iff-set* **by** *auto*
 have *wtL*: *Ik.wtL l* **using** *wtC* **unfolding** *wtC-def*
 by (*metis* (*lifting*) *lin list-all-iff*)
 {assume $\neg$ *?thesis*
  hence *0*: $\bigwedge$ *l'*. *l'* $\in$ *set c* $\Longrightarrow \neg$ *I.satL* ξ *l'* **unfolding** *I.satC-iff-set* **by** *auto*
  have *I.satL* ξ *l*
  proof (*rule satL-transE*[*OF wtL* ξ *- l*])
   fix *x* **let** *?*σ = *tpOfV x*

```
    assume σ: ¬ infTp ?σ and x: x ∈ nv2L l
     hence g: isGuard x (grdOf c l x) using mc[of ?σ] lin unfolding mcalc2-iff
by simp
    show ∃ l'. Ik.wtL l' ∧ ¬ I.satL ξ l' ∧ isGuard x l'
    apply(rule exI[of - grdOf c l x]) apply safe
    using g σ cin lin wtL-grdOf x 0 grdOf x by auto
   qed
   hence False using 0 lin by auto
   hence ?thesis by simp
  }
  thus ?thesis by auto
qed

lemma I-SAT: I.SAT Φ
unfolding I.SAT-def by simp

lemma InfModel: IInfModel I-intT I-intF I-intP
  by standard (rule I-SAT)

end

sublocale ModelIkPolMcalc2C < inf?: InfModel where
intT = I-intT and intF = I-intF and intP = I-intP
using InfModel .

context ProblemIkPolMcalc2C begin

abbreviation
MModelIkPolMcalc2C ≡ ModelIkPolMcalc2C wtFsym wtPsym arOf resOf parOf Φ
infTp pol grdOf

theorem monot: monot
unfolding monot-def proof safe
  fix intT intF intP assume MModel intT intF intP
  hence M: MModelIkPolMcalc2C intT intF intP
  unfolding ModelIkPolMcalc2C-def ModelIk-def apply safe ..
  show ∃ intTI intFI intPI. IInfModel intTI intFI intPI
  using ModelIkPolMcalc2C.InfModel[OF M] by auto
qed

end
```

Final theorem in sublocale form: Any problem that passes the monotonicity calculus is monotonic:

```
sublocale ProblemIkPolMcalc2C < MonotProblem
by standard (rule monot)

end
```

7 Guard-Based Encodings

theory *G*
imports *T-G-Prelim Mcalc2C*
begin

7.1 The guard translation

The extension of the function symbols with type witnesses:

datatype (*'fsym*,*'tp*) *efsym* = *Oldf 'fsym* | *Wit 'tp*

The extension of the predicate symbols with type guards:

datatype (*'psym*,*'tp*) *epsym* = *Oldp 'psym* | *Guard 'tp*

Extension of the partitioned infinitely augmented problem for dealing with guards:

locale *ProblemIkTpartG* =
Ik? : *ProblemIkTpart wtFsym wtPsym arOf resOf parOf* Φ *infTp prot protFw*
for *wtFsym* :: *'fsym* ⇒ *bool*
and *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp* **and** *prot* **and** *protFw* +
fixes
protCl :: *'tp* ⇒ *bool*
assumes
protCl-prot[*simp*]: ⋀ σ. *protCl* σ ⟹ *prot* σ

and *protCl-fsym*: ⋀ *f*. *protCl* (*resPf f*) ⟹ *list-all protCl* (*arOf f*)

locale *ModelIkTpartG* =
Ik? : *ProblemIkTpartG wtFsym wtPsym arOf resOf parOf* Φ *infTp prot protFw protCl* +
Ik? : *ModelIkTpart wtFsym wtPsym arOf resOf parOf* Φ *infTp prot protFw intT intF intP*
for *wtFsym* :: *'fsym* ⇒ *bool*
and *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp* **and** *prot* **and** *protFw* **and** *protCl*
and *intT* **and** *intF* **and** *intP*

context *ProblemIkTpartG*
begin

lemma *protCl-resOf-arOf*[*simp*]:
assumes *protCl* (*resOf f*) **and** *i* < *length* (*arOf f*)
shows *protCl* (*arOf f* ! *i*)
using *assms protCl-fsym* **unfolding** *list-all-length* **by** *auto*

"GE" stands for "guard encoding":

fun *GE-wtFsym* **where**
GE-wtFsym (*Oldf f*) $\longleftrightarrow$ *wtFsym f*
|*GE-wtFsym* (*Wit σ*) $\longleftrightarrow$ ¬ *isRes σ* ∨ *protCl σ*

fun *GE-arOf* **where**
GE-arOf (*Oldf f*) = *arOf f*
|*GE-arOf* (*Wit σ*) = []

fun *GE-resOf* **where**
GE-resOf (*Oldf f*) = *resOf f*
|*GE-resOf* (*Wit σ*) = *σ*

fun *GE-wtPsym* **where**
GE-wtPsym (*Oldp p*) $\longleftrightarrow$ *wtPsym p*
|*GE-wtPsym* (*Guard σ*) $\longleftrightarrow$ ¬ *unprot σ*

fun *GE-parOf* **where**
GE-parOf (*Oldp p*) = *parOf p*
|*GE-parOf* (*Guard σ*) = [*σ*]

lemma *countable-GE-wtFsym*: *countable* (*Collect GE-wtFsym*) (**is** *countable ?K*)
proof−
let *?F* = λ *ef*. *case ef of Oldf f* ⇒ *Inl f* | *Wit σ* ⇒ *Inr σ*
let *?U* = *UNIV*::*'tp set* **let** *?L* = (*Collect wtFsym*) <+> *?U*
have *inj-on ?F ?K* **unfolding** *inj-on-def* **apply** *clarify*
apply(*case-tac x*, *simp-all*) **by** (*case-tac y*, *simp-all*)+
moreover have *?F* ‘ *?K* ⊆ *?L* **apply** *clarify* **by** (*case-tac ef*, *auto*)
ultimately have |*?K*| ≤*o* |*?L*| **unfolding** *card-of-ordLeq*[*symmetric*] **by** *auto*
moreover have *countable ?L* **using** *countable-wtFsym countable-tp*
by (*metis countable-Plus*)
ultimately show *?thesis* **by**(*rule countable-ordLeq*)
qed

lemma *countable-GE-wtPsym*: *countable* (*Collect GE-wtPsym*) (**is** *countable ?K*)
proof−
let *?F* = λ *ep*. *case ep of Oldp p* ⇒ *Inl p* | *Guard σ* ⇒ *Inr σ*
let *?U* = *UNIV*::*'tp set* **let** *?L* = (*Collect wtPsym*) <+> *?U*
have *inj-on ?F ?K* **unfolding** *inj-on-def* **apply** *clarify*
apply(*case-tac x*, *simp-all*) **by** (*case-tac y*, *simp-all*)+
moreover have *?F* ‘ *?K* ⊆ *?L* **apply** *clarify* **by** (*case-tac ep*, *auto*)
ultimately have |*?K*| ≤*o* |*?L*| **unfolding** *card-of-ordLeq*[*symmetric*] **by** *auto*
moreover have *countable ?L* **using** *countable-wtPsym countable-tp*
by (*metis countable-Plus*)
ultimately show *?thesis* **by**(*rule countable-ordLeq*)
qed

end

sublocale *ProblemIkTpartG* < *GE?* : *Signature*
where *wtFsym* = *GE-wtFsym* **and** *wtPsym* = *GE-wtPsym*
and *arOf* = *GE-arOf* **and** *resOf* = *GE-resOf* **and** *parOf* = *GE-parOf*
apply *standard*
using *countable-tp countable-GE-wtFsym countable-GE-wtPsym* **by** *auto*

context *ProblemIkTpartG*
begin

The guarding literal of a variable:

definition *grdLit* :: *var* ⇒ ((*'fsym*, *'tp*) *efsym*, (*'psym*, *'tp*) *epsym*) *lit*
where *grdLit x* ≡ *Neg* (*Pr* (*Guard* (*tpOfV x*)) [*Var x*])

The (set of) guarding literals of a literal and of a clause:

fun *glitOfL* ::
(*'fsym*, *'psym*) *lit* ⇒ ((*'fsym*, *'tp*) *efsym*, (*'psym*, *'tp*) *epsym*) *lit set*
where
glitOfL (*Pos at*) =
{*grdLit x* | *x*. *x* ∈ *varsA at* ∧ (*prot* (*tpOfV x*) ∨ (*protFw* (*tpOfV x*) ∧ *x* ∈ *nvA at*))}
|
glitOfL (*Neg at*) = {*grdLit x* | *x*. *x* ∈ *varsA at* ∧ *prot* (*tpOfV x*)}

definition *glitOfC c* ≡ ⋃ (*set* (*map glitOfL c*))

lemma *finite-glitOfL*[*simp*]: *finite* (*glitOfL l*)
proof−
have *glitOfL l* ⊆ *grdLit* ' {*x* . *x* ∈ *varsL l*} **by** (*cases l*, *auto*)
thus *?thesis* **by** (*metis Collect-mem-eq finite-surj finite-varsL*)
qed

lemma *finite-glitOfC*[*simp*]: *finite* (*glitOfC c*)
unfolding *glitOfC-def* **apply**(*rule finite-Union*) **using** *finite-glitOfL* **by** *auto*

fun *gT* **where**
gT (*Var x*) = *Var x*
|
gT (*Fn f Tl*) = *Fn* (*Oldf f*) (*map gT Tl*)

fun *gA* **where**
gA (*Eq T1 T2*) = *Eq* (*gT T1*) (*gT T2*)
|
gA (*Pr p Tl*) = *Pr* (*Oldp p*) (*map gT Tl*)

fun *gL* **where**
gL (*Pos at*) = *Pos* (*gA at*)

|
gL (Neg at) = Neg (gA at)

definition *gC c ≡ (map gL c) @ (list (glitOfC c))*

lemma *set-gC*[*simp*]: *set (gC c) = gL ‘ (set c) ∪ glitOfC c*
unfolding *gC-def* **by** *simp*

The extra axioms:

The function axioms:

definition *cOfFax f = Pr (Guard (resOf f)) [Fn (Oldf f) (getTvars (arOf f))]*

definition *hOfFax f = map2 (Pr o Guard) (arOf f) (map singl (getTvars (arOf f)))*

definition *fax f ≡ [Pos (cOfFax f)]*

definition *faxCD f ≡ map Neg (hOfFax f) @ fax f*

definition
Fax ≡ {fax f | f. wtFsym f ∧ ¬ unprot (resOf f) ∧ ¬ protCl (resOf f)} ∪
{faxCD f | f. wtFsym f ∧ protCl (resOf f)}

The witness axioms:

definition *wax σ ≡ [Pos (Pr (Guard σ) [Fn (Wit σ) []])]*

definition *Wax ≡ {wax σ | σ. ¬ unprot σ ∧ (¬ isRes σ ∨ protCl σ)}*

definition *gPB = gC ‘ Φ ∪ Fax ∪ Wax*

Well-typedness of the translation:

lemma *tpOf-g*[*simp*]: *GE.tpOf (gT T) = Ik.tpOf T*
by (*cases T*) *auto*

lemma *wt-g*[*simp*]: *Ik.wt T ⟹ GE.wt (gT T)*
by (*induct T*, *auto simp add*: *list-all-iff*)

lemma *wtA-gA*[*simp*]: *Ik.wtA at ⟹ GE.wtA (gA at)*
by (*cases at*, *auto simp add*: *list-all-iff*)

lemma *wtL-gL*[*simp*]: *Ik.wtL l ⟹ GE.wtL (gL l)*
by (*cases l*, *auto*)

lemma *wtC-map-gL*[*simp*]: *Ik.wtC c ⟹ GE.wtC (map gL c)*
unfolding *Ik.wtC-def GE.wtC-def* **by** (*induct c*, *auto*)

lemma *wtL-grdLit-unprot*[*simp*]: *¬ unprot (tpOfV x) ⟹ GE.wtL (grdLit x)*
unfolding *grdLit-def* **by** *auto*

lemma *wtL-grdLit*[*simp*]: *prot* (*tpOfV x*) ∨ *protFw* (*tpOfV x*) ⟹ *GE.wtL* (*grdLit x*)
apply(*rule wtL-grdLit-unprot*) **unfolding** *unprot-def* **by** *auto*

lemma *wtL-glitOfL*[*simp*]: *l'* ∈ *glitOfL l* ⟹ *GE.wtL l'*
by (*cases l*, *auto*)

lemma *wtL-glitOfC*[*simp*]: *l'* ∈ *glitOfC c* ⟹ *GE.wtL l'*
unfolding *glitOfC-def GE.wtC-def* **by** (*induct c*, *auto*)

lemma *wtC-list-glitOfC*[*simp*]: *GE.wtC* (*list* (*glitOfC c*))
unfolding *glitOfC-def GE.wtC-def* **by** *auto*

lemma *wtC-gC*[*simp*]: *Ik.wtC c* ⟹ *GE.wtC* (*gC c*)
unfolding *gC-def* **by** *simp*

lemma *wtA-cOfFax-unprot*[*simp*]: ⟦*wtFsym f*; ¬ *unprot* (*resOf f*)⟧ ⟹ *GE.wtA* (*cOfFax f*)
unfolding *cOfFax-def* **by** *simp*

lemma *wtA-cOfFax*[*simp*]:
⟦*wtFsym f*; *prot* (*resOf f*) ∨ *protFw* (*resOf f*)⟧ ⟹ *GE.wtA* (*cOfFax f*)
apply(*rule wtA-cOfFax-unprot*) **unfolding** *unprot-def* **by** *auto*

lemma *wtA-hOfFax*[*simp*]:
⟦*wtFsym f*; *protCl* (*resOf f*)⟧ ⟹ *list-all GE.wtA* (*hOfFax f*)
unfolding *hOfFax-def* **unfolding** *list-all-length*
by (*auto simp add*: *singl-def unprot-def*)

lemma *wtC-fax-unprot*[*simp*]: ⟦*wtFsym f*; ¬ *unprot* (*resOf f*)⟧ ⟹ *GE.wtC* (*fax f*)
unfolding *fax-def GE.wtC-def* **by** *auto*

lemma *wtC-fax*[*simp*]: ⟦*wtFsym f*; *prot* (*resOf f*) ∨ *protFw* (*resOf f*)⟧ ⟹ *GE.wtC* (*fax f*)
apply(*rule wtC-fax-unprot*) **unfolding** *unprot-def* **by** *auto*

lemma *wtC-faxCD*[*simp*]: ⟦*wtFsym f*; *protCl* (*resOf f*)⟧ ⟹ *GE.wtC* (*faxCD f*)
unfolding *faxCD-def GE.wtC-append* **apply**(*rule conjI*)
using *wtA-hOfFax*[*unfolded list-all-length*] **apply**(*simp add*: *GE.wtC-def list-all-length*)
by *simp*

lemma *wtPB-Fax*[*simp*]: *GE.wtPB Fax*
unfolding *Fax-def GE.wtPB-def* **by** *auto*

lemma *wtC-wax-unprot*[*simp*]: ⟦¬ *unprot σ*; ¬ *isRes σ* ∨ *protCl σ*⟧ ⟹ *GE.wtC* (*wax σ*)
unfolding *wax-def GE.wtC-def* **by** *simp*

lemma *wtC-wax*[*simp*]: ⟦*prot σ* ∨ *protFw σ*; ¬ *isRes σ* ∨ *protCl σ*⟧ ⟹ *GE.wtC* (*wax σ*)
apply(*rule wtC-wax-unprot*) **unfolding** *unprot-def* **by** *auto*

lemma *wtPB-Wax*[*simp*]: *GE.wtPB Wax*
unfolding *Wax-def GE.wtPB-def* **by** *auto*

lemma *wtPB-gC-Φ*[*simp*]: *GE.wtPB* (*gC* ‘ Φ)
using *Ik.wt-Φ* **unfolding** *Ik.wtPB-def GE.wtPB-def* **by** *auto*

lemma *wtPB-tPB*[*simp*]: *GE.wtPB gPB* **unfolding** *gPB-def* **by** *simp*

lemma *wtA-Guard*:
assumes *GE.wtA* (*Pr* (*Guard σ*) *Tl*)
shows ∃ *T*. *Tl* = [*T*] ∧ *GE.wt T* ∧ *tpOf T* = *σ*
using *assms* **by** *simp* (*metis* (*opaque-lifting*, *no-types*) *list.inject map-eq-Cons-conv*

list-all-simps map-is-Nil-conv neq-Nil-conv)

lemma *wt-Wit*:
assumes *GE.wt* (*Fn* (*Wit σ*) *Tl*) **shows** *Tl* = []
using *assms* **by** *simp*

lemma *tpOf-Wit*: *GE.tpOf* (*Fn* (*Wit σ*) *Tl*) = *σ* **by** *simp*

end

7.2 Soundness

context *ModelIkTpartG* **begin**

fun *GE-intF* **where**
GE-intF (*Oldf f*) *al* = *intF f al*
|*GE-intF* (*Wit σ*) *al* = *pickT σ*

fun *GE-intP* **where**
GE-intP (*Oldp p*) *al* = *intP p al*
|*GE-intP* (*Guard σ*) *al* = *True*

end

sublocale *ModelIkTpartG* < *GE?* : *Struct*
where *wtFsym* = *GE-wtFsym* **and** *wtPsym* = *GE-wtPsym* **and**
arOf = *GE-arOf* **and** *resOf* = *GE-resOf* **and** *parOf* = *GE-parOf*

and *intF* = *GE-intF* **and** *intP* = *GE-intP*
proof
  fix *ef al* **assume** *GE-wtFsym ef* **and** *list-all2 intT* (*GE-arOf ef*) *al*
  thus *intT* (*GE-resOf ef*) (*GE-intF ef al*)
  using *intF* **by** (*cases ef*, *auto*)
qed *auto*

context *ModelIkTpartG* **begin**

lemma *g-int*[*simp*]: *GE.int* ξ (*gT T*) = *Ik.int* ξ *T*
proof (*induct T*)
  case (*Fn f Tl*)
  hence *0*: *map* (*GE.int* ξ ∘ *gT*) *Tl* = *map* (*Ik.int* ξ) *Tl*
  unfolding *list-eq-iff list-all-iff* **by** *auto*
  show *?case* **by** (*simp add*: *0*)
qed *auto*

lemma *map-g-int*[*simp*]: *map* (*GE.int* ξ ∘ *gT*) *Tl* = *map* (*Ik.int* ξ) *Tl*
unfolding *list-eq-iff list-all-iff* **by** *auto*

lemma *gA-satA*[*simp*]: *GE.satA* ξ (*gA at*) ⟷ *Ik.satA* ξ *at*
apply(*cases at*) **by** *auto*

lemma *gL-satL*[*simp*]: *GE.satL* ξ (*gL l*) ⟷ *Ik.satL* ξ *l*
apply(*cases l*) **by** *auto*

lemma *map-gL-satC*[*simp*]: *GE.satC* ξ (*map gL c*) ⟷ *Ik.satC* ξ *c*
unfolding *GE.satC-def Ik.satC-def* **by** (*induct c*, *auto*)

lemma *gC-satC*[*simp*]:
assumes *Ik.satC* ξ *c* **shows** *GE.satC* ξ (*gC c*)
using *assms* **unfolding** *gC-def* **by** *simp*

lemma *gC-Φ-satPB*[*simp*]:
assumes *Ik.satPB* ξ Φ **shows** *GE.satPB* ξ (*gC* ‘ Φ)
using *assms* **unfolding** *GE.satPB-def Ik.satPB-def* **by** *auto*

lemma *Fax-Wax-satPB*:
GE.satPB ξ (*Fax*) ∧ *GE.satPB* ξ (*Wax*)
unfolding *GE.satPB-def GE.satC-def Fax-def Wax-def*
by (*auto simp add*: *cOfFax-def hOfFax-def fax-def faxCD-def wax-def*)

lemmas *Fax-satPB*[*simp*] = *Fax-Wax-satPB*[*THEN conjunct1*]
lemmas *Wax-satPB*[*simp*] = *Fax-Wax-satPB*[*THEN conjunct2*]

lemma *soundness*: *GE.SAT gPB*
unfolding *GE.SAT-def gPB-def* **using** *SAT* **unfolding** *Ik.SAT-def* **by** *auto*

theorem *G-soundness*:

Model GE-wtFsym GE-wtPsym GE-arOf GE-resOf GE-parOf gPB intT GE-intF GE-intP
apply *standard* **using** *wtPB-tPB soundness* **by** *auto*

end

sublocale *ModelIkTpartG* < *GE?* : *Model*
where *wtFsym* = *GE-wtFsym* **and** *wtPsym* = *GE-wtPsym* **and**
arOf = *GE-arOf* **and** *resOf* = *GE-resOf* **and** *parOf* = *GE-parOf*
and Φ = *gPB* **and** *intF* = *GE-intF* **and** *intP* = *GE-intP*
using *G-soundness* **.**

7.3 Completeness

locale *ProblemIkTpartG-GEModel* =
Ik? : *ProblemIkTpartG wtFsym wtPsym arOf resOf parOf* Φ *infTp prot protFw protCl* +
GE? : *Model ProblemIkTpartG.GE-wtFsym wtFsym resOf protCl*
ProblemIkTpartG.GE-wtPsym wtPsym prot protFw
ProblemIkTpartG.GE-arOf arOf ProblemIkTpartG.GE-resOf resOf
ProblemIkTpartG.GE-parOf parOf
gPB eintT eintF eintP
for *wtFsym* :: *'fsym* ⇒ *bool*
and *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp* **and** *prot* **and** *protFw* **and** *protCl*
and *eintT* **and** *eintF* **and** *eintP*

context *ProblemIkTpartG-GEModel* **begin**

The reduct structure of a given structure in the guard-extended signature:

definition
intT σ a ≡
if unprot σ then eintT σ a
else eintT σ a ∧ *eintP* (*Guard σ*) [*a*]
definition
intF f al ≡ *eintF* (*Oldf f*) *al*
definition
intP p al ≡ *eintP* (*Oldp p*) *al*

lemma *GE-Guard-all*:
assumes *f*: *wtFsym f*
and *al*: *list-all2 eintT* (*arOf f*) *al*
shows
(¬ *unprot* (*resOf f*) ∧ ¬ *protCl* (*resOf f*)
⟶ *eintP* (*Guard* (*resOf f*)) [*eintF* (*Oldf f*) *al*])
∧

```
 (protCl (resOf f) ⟶
  list-all2 (eintP ∘ Guard) (arOf f) (map singl al)
  ⟶ eintP (Guard (resOf f)) [eintF (Oldf f) al])
(is (?A1 ⟶ ?C) ∧
    (?A2 ⟶ ?H2 ⟶ ?C))
proof(intro conjI impI)
  define xl where xl = getVars (arOf f)
  have l[simp]: length xl = length al length al = length (arOf f)
              length (getTvars (arOf f)) = length (arOf f)
  unfolding xl-def using al unfolding list-all2-iff by auto
  have 1[simp]: ⋀ i. i < length (arOf f) ⟹ tpOfV (xl!i) = (arOf f)!i
  unfolding xl-def by auto
  have xl[simp]: distinct xl unfolding xl-def using distinct-getVars by auto
  define ξ where ξ = pickE xl al
 have ξ: GE.wtE ξ unfolding ξ-def apply(rule wtE-pickE) using al list-all2-nthD
by auto
  have [simp]: ⋀ i. i < length (arOf f) ⟹ ξ (xl ! i) = al ! i
  using al unfolding ξ-def by (auto simp: list-all2-length intro: pickE)
  have 0: map (GE.int ξ) (getTvars (arOf f)) = al
  apply(rule nth-equalityI)
  using al by (auto simp: list-all2-length getTvars-def xl-def[symmetric])
  have 1:
  GE.satC ξ (map Neg (map2 (Pr ∘ Guard) (arOf f) (map singl (getTvars (arOf
f))))) ⟷
   ¬ list-all2 (eintP ∘ Guard) (arOf f) (map singl al)
  unfolding GE.satC-def list-ex-length list-all2-map2 singl-def
  unfolding list-all2-length
  by (auto simp add: map-zip-map2 singl-def getTvars-def xl-def[symmetric])
 have Fax: GE.satPB ξ Fax using GE.sat-Φ[OF ξ] unfolding gPB-def by simp
  {assume ?A1
   hence GE.satC ξ (fax f) using f Fax unfolding GE.satPB-def Fax-def by
auto
   thus ?C unfolding fax-def cOfFax-def GE.satC-def by (simp add: 0)
  }
  {assume ?A2 and ?H2
   hence GE.satC ξ (faxCD f) using f Fax unfolding GE.satPB-def Fax-def by
auto
   thus ?C using ‹?H2›
   unfolding faxCD-def fax-def cOfFax-def hOfFax-def
   unfolding GE.satC-append 1 unfolding GE.satC-def by (simp add: 0)
  }
qed

lemma GE-Guard-not-unprot-protCl:
assumes f: wtFsym f and f2: ¬ unprot (resOf f) ¬ protCl (resOf f)
and al: list-all2 eintT (arOf f) al
shows eintP (Guard (resOf f)) [intF f al]
using GE-Guard-all[OF f al] f2 unfolding intF-def by auto
```

lemma *GE-Guard-protCl*:
assumes *f*: *wtFsym f* **and** *f2*: *protCl* (*resOf f*) **and** *al*: *list-all2 eintT* (*arOf f*) *al*
and *H*: *list-all2* (*eintP* ∘ *Guard*) (*arOf f*) (*map singl al*)
shows *eintP* (*Guard* (*resOf f*)) [*intF f al*]
using *GE-Guard-all*[*OF f al*] *f2 H* **unfolding** *intF-def* **by** *auto*

lemma *GE-Guard-not-unprot*:
assumes *f*: *wtFsym f* **and** *f2*: ¬ *unprot* (*resOf f*) **and** *al*: *list-all2 eintT* (*arOf f*) *al*
and *H*: *list-all2* (*eintP* ∘ *Guard*) (*arOf f*) (*map singl al*)
shows *eintP* (*Guard* (*resOf f*)) [*intF f al*]
apply(*cases protCl* (*resOf f*))
using *GE-Guard-protCl*[*OF f - al H*] **apply** *fastforce*
using *GE-Guard-not-unprot-protCl*[*OF f f2 - al*] **by** *simp*

lemma *GE-Wit*:
assumes *σ*: ¬ *unprot σ* ¬ *isRes σ* ∨ *protCl σ*
shows *eintP* (*Guard σ*) [*eintF* (*Wit σ*) []]
proof−
define *ξ* **where** *ξ* = *pickE* [] []
have *ξ*: *GE.wtE ξ* **unfolding** *ξ-def* **apply**(*rule wtE-pickE*) **by** *auto*
have *GE.satPB ξ Wax* **using** *GE.sat-Φ*[*OF ξ*] **unfolding** *gPB-def* **by** *simp*
hence *GE.satC ξ* (*wax σ*) **unfolding** *Wax-def GE.satPB-def* **using** *σ* **by** *auto*
thus *?thesis* **unfolding** *satC-def wax-def* **by** *simp*
qed

lemma *NE-intT-forget*: *NE* (*intT σ*)
proof−
obtain *a* **where** *a*: *eintT σ a* **using** *GE.NE-intT* **by** *blast*
show *?thesis*
proof(*cases unprot σ*)
case *True*
thus *?thesis* **using** *a* **unfolding** *intT-def* **by** *auto*
next
case *False* **note** *unprot* = *False*
show *?thesis* **proof**(*cases isRes σ*)
case *True* **then obtain** *f* **where** *f*: *wtFsym f* **and** *σ*: *σ* = *resOf f*
unfolding *isRes-def* **by** *auto*
define *al* **where** *al* = *map pickT* (*arOf f*)
have *al*: *list-all2 eintT* (*arOf f*) *al*
unfolding *al-def list-all2-map2* **unfolding** *list-all2-length* **by** *auto*
define *a* **where** *a* = *intF f al*
have *a*: *eintT σ a* **unfolding** *a-def σ intF-def* **using** *f al*
by (*metis GE-arOf.simps GE-resOf.simps GE-wtFsym.simps GE.intF*)
show *?thesis* **proof** (*cases protCl σ*)
case *True*
define *a* **where** *a* = *eintF* (*Wit σ*) []
have *eintT σ a* **unfolding** *a-def*
by (*metis True GE-arOf.simps GE-resOf.simps GE-wtFsym.simps intF*

list-all2-Nil)
      moreover have *eintP* (*Guard* σ) [*a*]
      unfolding *a-def* **using** *GE-Wit*[*OF unprot*] *True* **by** *simp*
      ultimately show *?thesis* **using** *unprot* **unfolding** *intT-def* **by** *auto*
     next
      case *False*
      hence *eintP* (*Guard* σ) [*a*]
       using *unprot GE-Guard-not-unprot-protCl*[*OF f - - al*] **unfolding** σ *a-def*
by *simp*
      thus *?thesis* **using** *unprot a* **unfolding** *intT-def* **by** *auto*
     qed
   next
     case *False*
     define *a* **where** *a* = *eintF* (*Wit* σ) []
     have *eintT* σ *a* **unfolding** *a-def*
       by (*metis False GE-arOf.simps GE-resOf.simps GE-wtFsym.simps intF*
list-all2-Nil)
     moreover have *eintP* (*Guard* σ) [*a*]
     unfolding *a-def* **using** *GE-Wit*[*OF unprot*] *False* **by** *simp*
     ultimately show *?thesis* **using** *unprot* **unfolding** *intT-def* **by** *auto*
   qed
 qed
qed

lemma *wt-intF*:
assumes *f*: *wtFsym f* **and** *al*: *list-all2 intT* (*arOf f*) *al*
shows *intT* (*resOf f*) (*intF f al*)
proof−
 have *0*: *list-all2 eintT* (*arOf f*) *al*
 using *al* **unfolding** *intT-def*[*abs-def*] *list-all2-length* **by** *metis*
 hence *eintT* (*resOf f*) (*eintF* (*Oldf f*) *al*)
 by (*metis GE-arOf.simps GE-resOf.simps GE-wtFsym.simps f al GE.intF*)
 hence *1*: *eintT* (*resOf f*) (*intF f al*) **unfolding** *intF-def* **by** *simp*
 show *?thesis* **proof**(*cases unprot* (*resOf f*))
  case *True* **thus** *?thesis* **unfolding** *intT-def* **by** (*simp add*: *1*)
 next
  case *False* **note** *unprot* = *False*
  have *eintP* (*Guard* (*resOf f*)) [*intF f al*]
  proof(*cases protCl* (*resOf f*))
   case *False* **show** *?thesis* **using** *GE-Guard-not-unprot-protCl*[*OF f unprot False*
0] **.**
  next
   case *True*
   hence *list-all protCl* (*arOf f*) **using** *protCl-fsym* **by** *simp*
   hence *list-all* (λ σ. ¬ *unprot* σ) (*arOf f*)
   unfolding *list-all-length unprot-def* **by** *auto*
   hence *2*: *list-all2* (*eintP* ∘ *Guard*) (*arOf f*) (*map singl al*)
   using *al* **unfolding** *intT-def*[*abs-def*] *list-all2-length list-all-length*
   singl-def[*abs-def*] **by** *auto*

```
    show ?thesis using GE-Guard-protCl[OF f True 0 2] .
   qed
   thus ?thesis using unprot unfolding intT-def by (simp add: 1)
  qed
qed

lemma Struct: Struct wtFsym wtPsym arOf resOf intT intF intP
apply standard using NE-intT-forget wt-intF by auto

end

sublocale ProblemIkTpartG-GEModel < Ik? : Struct
where intT = intT and intF = intF and intP = intP
using Struct .


context ProblemIkTpartG-GEModel begin

lemma wtE[simp]: Ik.wtE ξ ⟹ GE.wtE ξ
unfolding Ik.wtE-def GE.wtE-def intT-def by metis

lemma int-g[simp]: GE.int ξ (gT T) = Ik.int ξ T
proof (induct T)
 case (Fn f Tl)
 let ?ar = arOf f let ?r = resOf f
 have 0: map (Ik.int ξ) Tl = map (GE.int ξ ∘ gT) Tl
 apply(rule nth-equalityI) using Fn unfolding list-all-length by auto
 show ?case
 using [[unfold-abs-def = false]]
 unfolding Ik.int.simps GE.int.simps gT.simps unfolding intF-def
 using Fn by (simp add: 0)
qed auto

lemma map-int-g[simp]:
map (Ik.int ξ) Tl = map (GE.int ξ ∘ gT) Tl
apply(rule nth-equalityI) unfolding list-all-length by auto

lemma satA-gA[simp]: GE.satA ξ (gA at) ⟷ Ik.satA ξ at
by (cases at) (auto simp add: intP-def)

lemma satL-gL[simp]: GE.satL ξ (gL l) ⟷ Ik.satL ξ l
by (cases l) auto

lemma satC-map-gL[simp]: GE.satC ξ (map gL c) ⟷ Ik.satC ξ c
unfolding GE.satC-def Ik.satC-def by (induct c) auto

lemma wtE-not-grdLit-unprot[simp]:
assumes Ik.wtE ξ and ¬ unprot (tpOfV x)
shows ¬ GE.satL ξ (grdLit x)
```

using *assms* **unfolding** *Ik.wtE-def intT-def grdLit-def* **by** *simp*

lemma *wtE-not-grdLit*[*simp*]:
assumes *Ik.wtE* ξ **and** *prot* (*tpOfV x*) $\vee$ *protFw* (*tpOfV x*)
shows $\neg$ *GE.satL* ξ (*grdLit x*)
apply(*rule wtE-not-grdLit-unprot*) **using** *assms* **unfolding** *unprot-def* **by** *auto*

lemma *wtE-not-glitOfL*[*simp*]:
assumes *Ik.wtE* ξ
shows $\neg$ *GE.satC* ξ (*list* (*glitOfL l*))
using *assms* **unfolding** *GE.satC-def list-ex-list*[*OF finite-glitOfL*]
by (*cases l*, *auto*)

lemma *wtE-not-glitOfC*[*simp*]:
assumes *Ik.wtE* ξ
shows $\neg$ *GE.satC* ξ (*list* (*glitOfC c*))
using *wtE-not-glitOfL*[*OF assms*]
unfolding *GE.satC-def list-ex-list*[*OF finite-glitOfC*] *list-ex-list*[*OF finite-glitOfL*]
unfolding *glitOfC-def* **by** *auto*

lemma *satC-gC*[*simp*]:
assumes *Ik.wtE* ξ **and** *GE.satC* ξ (*gC c*)
shows *Ik.satC* ξ *c*
using *assms* **unfolding** *gC-def* **by** *simp*

lemma *satPB-gPB*[*simp*]:
assumes *Ik.wtE* ξ **and** *GE.satPB* ξ (*gC* ' Φ)
shows *Ik.satPB* ξ Φ
using *Ik.wt-*Φ *assms* **unfolding** *GE.satPB-def Ik.satPB-def* **by** (*auto simp add*: *Ik.wtPB-def*)

lemma *completeness*: *Ik.SAT* Φ
unfolding *Ik.SAT-def* **proof** *safe*
  fix ξ **assume** ξ: *Ik.wtE* ξ **hence** *GE.wtE* ξ **by** *simp*
  hence *GE.satPB* ξ *gPB* **by** (*rule GE.sat-*Φ)
  hence *GE.satPB* ξ (*gC* ' Φ) **unfolding** *gPB-def* **by** *simp*
  thus *Ik.satPB* ξ Φ **using** ξ **by** *simp*
qed

lemma *G-completeness*: *Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP*
apply *standard* **using** *completeness* **.**

end

sublocale *ProblemIkTpartG-GEModel* < *Ik?* : *Model*
where *intT* = *intT* **and** *intF* = *intF* **and** *intP* = *intP*

using *G-completeness* **.**

7.4 The result of the guard translation is an infiniteness-augmented problem

sublocale *ProblemIkTpartG* < *GE?* : *Problem*
where *wtFsym* = *GE-wtFsym* **and** *wtPsym* = *GE-wtPsym*
and *arOf* = *GE-arOf* **and** *resOf* = *GE-resOf* **and** *parOf* = *GE-parOf*
and Φ = *gPB*
apply *standard*
apply *auto*
done

sublocale *ProblemIkTpartG* < *GE?* : *ProblemIk*
where *wtFsym* = *GE-wtFsym* **and** *wtPsym* = *GE-wtPsym*
and *arOf* = *GE-arOf* **and** *resOf* = *GE-resOf* **and** *parOf* = *GE-parOf*
and Φ = *gPB*
proof
fix *σ eintT eintF eintP a* **assume** *σ*: *infTp σ*
assume *M*: *Model GE-wtFsym GE-wtPsym GE-arOf GE-resOf GE-parOf gPB eintT eintF eintP*
let *?GE-intT* = *ProblemIkTpartG-GEModel.intT prot protFw eintT eintP*
let *?GE-intF* = *ProblemIkTpartG-GEModel.intF eintF*
let *?GE-intP* = *ProblemIkTpartG-GEModel.intP eintP*
have *0*: *ProblemIkTpartG-GEModel wtFsym wtPsym arOf resOf parOf Φ infTp prot protFw protCl eintT eintF eintP*
using *M* **unfolding** *ProblemIkTpartG-GEModel-def* **apply** *safe* **..**
hence *MM*: *Ik.MModel ?GE-intT ?GE-intF ?GE-intP*
by (*rule ProblemIkTpartG-GEModel.G-completeness*)
have *infinite* {*a*. *?GE-intT σ a*} **using** *infTp*[*OF σ MM*] **.**
moreover have {*a*. *?GE-intT σ a*} ⊆ {*a*. *eintT σ a*}
using *ProblemIkTpartG-GEModel.intT-def*[*OF 0*] **by** *auto*
ultimately show *infinite* {*a*. *eintT σ a*} **using** *infinite-super* **by** *blast*
qed

7.5 The verification of the second monotonicity calculus criterion for the guarded problem

context *ProblemIkTpartG* **begin**

fun *pol* **where**
pol - (*Oldp p*) = *Cext*
|
pol - (*Guard σ*) = *Fext*

lemma *pol-ct*: *pol σ1 p* = *pol σ2 p*
by(*cases p*, *auto*)

definition *grdOf c l x* = *grdLit x*

end

sublocale *ProblemIkTpartG* < *GE?*: *ProblemIkPol*
where *wtFsym* = *GE-wtFsym* **and** *wtPsym* = *GE-wtPsym*
and *arOf* = *GE-arOf* **and** *resOf* = *GE-resOf* **and** *parOf* = *GE-parOf*
and Φ = *gPB* **and** *pol* = *pol* **and** *grdOf* = *grdOf* **..**

context *ProblemIkTpartG* **begin**

lemma *nv2-nv*[*simp*]: *GE.nv2T* (*gT T*) = *GE.nvT T*
apply (*induct T*) **by** *auto*

lemma *nv2L-nvL*[*simp*]: *GE.nv2L* (*gL l*) = *GE.nvL l*
proof(*cases l*)
case (*Pos at*) **thus** *?thesis* **by** (*cases at*, *simp-all*)
next
case (*Neg at*) **thus** *?thesis* **by** (*cases at*, *auto*)
qed

lemma *nv2L*:
assumes *l* ∈ *set c* **and** *mc*: *GE.mcalc* σ *c*
shows *infTp* σ ∨ (∀ *x* ∈ *GE.nv2L* (*gL l*). *tpOfV x* ≠ σ)
using *assms mc nv2L-nvL* **unfolding** *GE.mcalc-iff GE.nvC-def* **apply** *simp*
using *nv2L-nvL*[*of l*]
by (*metis empty-subsetI equalityI nv2L-nvL*)

lemma *isGuard-grdLit*[*simp*]: *GE.isGuard x* (*grdLit x*)
unfolding *grdLit-def* **by** *auto*

lemma *nv2L-grdLit*[*simp*]: *GE.nv2L* (*grdLit x*) = {}
unfolding *grdLit-def* **by** *auto*

lemma *mcalc-mcalc2*: *GE.mcalc* σ *c* ⟹ *GE.mcalc2* σ (*gC c*)
using *nv2L* **unfolding** *GE.mcalc2-iff gC-def glitOfC-def grdOf-def* **by** *auto*

lemma *nv2L-wax*[*simp*]: *l'* ∈ *set* (*wax* σ) ⟹ *GE.nv2L l'* = {}
unfolding *wax-def* **by** *auto*

lemma *nv2L-Wax*:
assumes *c'* ∈ *Wax* **and** *l'* ∈ *set c'*
shows *GE.nv2L l'* = {}
using *assms* **unfolding** *Wax-def* **by** *auto*

lemma *nv2L-cOfFax*[*simp*]: *GE.nv2L* (*Pos* (*cOfFax* σ)) = {}
unfolding *cOfFax-def* **by** *auto*

lemma *nv2L-hOfFax*[*simp*]:
assumes *at* ∈ *set* (*hOfFax* σ)

shows *GE.nv2L* (*Neg at*) = {}
using *assms* **unfolding** *hOfFax-def* **by** *auto*

lemma *nv2L-fax*[*simp*]: $l \in$ *set* (*fax* σ) $\Longrightarrow$ *GE.nv2L* l = {}
unfolding *fax-def* **by** *auto*

lemma *nv2L-faxCD*[*simp*]: $l \in$ *set* (*faxCD* σ) $\Longrightarrow$ *GE.nv2L* l = {}
unfolding *faxCD-def* **by** *auto*

lemma *nv2L-Fax*:
assumes $c' \in$ *Fax* **and** $l' \in$ *set* c'
shows *GE.nv2L* l' = {}
using *assms* **unfolding** *Fax-def* **by** *auto*

lemma *grdOf*:
assumes c': $c' \in$ *gPB* **and** l': $l' \in$ *set* c'
and x: $x \in$ *GE.nv2L* l' **and** i: $\neg$ *infTp* (*tpOfV* x)
shows *grdOf* c' l' $x \in$ *set* $c' \wedge$ *GE.isGuard* x (*grdOf* c' l' x)
proof(*cases* $c' \in$ *Fax* $\cup$ *Wax*)
case *True* **thus** *?thesis* **using** x l' *nv2L-Wax nv2L-Fax* **by** *force*
next
case *False* **then obtain** c **where** c': c' = *gC* c **and** c: $c \in \Phi$
using c' **unfolding** *gPB-def* **by** *auto*
show *?thesis*
proof(*cases* $l' \in$ *glitOfC* c)
case *True* **then obtain** l **where** l: $l \in$ *set* c **and** l': $l' \in$ *glitOfL* l
unfolding *glitOfC-def* **by** *auto*
then obtain *x1* **where** l' = *grdLit x1* **using** l' **by** (*cases* l *rule*: *lit.exhaust*)
auto
hence *GE.nv2L* l' = {} **by** *simp*
thus *?thesis* **using** x **by** *simp*
next
let *?σ* = *tpOfV* x
case *False* **then obtain** l **where** l: $l \in$ *set* c **and** l': l' = *gL* l
using l' **unfolding** c' *gC-def* **by** *auto*
hence x: $x \in$ *GE.nvL* l **using** x **by** *simp*
hence $x \in$ *GE.nvC* c **using** l **unfolding** *GE.nvC-def* **by** *auto*
hence $\neg$ *GE.mcalc ?σ* c **using** i **unfolding** *GE.mcalc-iff* **by** *auto*
hence *tp*: *prot ?σ* $\vee$ *protFw ?σ* **using** *unprot-mcalc*[*OF* - c] **unfolding** *unprot-def* **by** *auto*
moreover obtain *at* **where** *l-at*: l = *Pos at* **using** x **by**(*cases* l, *auto*)
moreover have $x \in$ *varsA at* **using** x **unfolding** *l-at* **by** *auto*
ultimately have *grdLit* $x \in$ *glitOfL* l **using** x **unfolding** *l-at* **by** *force*
thus *?thesis* **using** l x **unfolding** *grdOf-def* c' *gC-def glitOfC-def* **by** *auto*
qed
qed

lemma *mcalc2*:
assumes c': $c' \in$ *gPB*

shows *GE.mcalc2* σ c'
proof(*cases* $c' \in Fax \cup Wax$)
case *True* **thus** *?thesis* **using** *nv2L-Wax nv2L-Fax*
unfolding *GE.mcalc2-iff* **by** *fastforce*
next
case *False* **hence** c': $c' \in gPB$ **using** c' **unfolding** *gPB-def* **by** *auto*
show *?thesis* **unfolding** *GE.mcalc2-iff* **using** *grdOf*[*OF* c'] **by** *auto*
qed

end

sublocale *ProblemIkTpartG* < *GE?*: *ProblemIkPolMcalc2C*
where *wtFsym* = *GE-wtFsym* **and** *wtPsym* = *GE-wtPsym*
and *arOf* = *GE-arOf* **and** *resOf* = *GE-resOf* **and** *parOf* = *GE-parOf*
and Φ = *gPB* **and** *pol* = *pol* **and** *grdOf* = *grdOf*
apply *standard* **using** *grdOf mcalc2*
apply (*auto simp*: *pol-ct*)
done

context *ProblemIkTpartG* **begin**

theorem *G-monotonic*:
MonotProblem GE-wtFsym GE-wtPsym GE-arOf GE-resOf GE-parOf gPB **..**

end

sublocale *ProblemIkTpartG* < *GE?*: *MonotProblem*
where *wtFsym* = *GE-wtFsym* **and** *wtPsym* = *GE-wtPsym*
and *arOf* = *GE-arOf* **and** *resOf* = *GE-resOf* **and** *parOf* = *GE-parOf*
and Φ = *gPB*
using *G-monotonic* **.**

end

8 Tag-Based Encodings

theory *T*
imports *T-G-Prelim*
begin

8.1 The tag translation

The extension of the function symbols with type tags and type witnesses:

datatype (*'fsym*,*'tp*) *efsym* = *Oldf 'fsym* | *Tag 'tp* | *Wit 'tp*

context *ProblemIkTpart*
begin

"TE" stands for "tag encoding"

fun *TE-wtFsym* **where**
TE-wtFsym (*Oldf f*) ⟷ *wtFsym f*
|*TE-wtFsym* (*Tag σ*) ⟷ *True*
|*TE-wtFsym* (*Wit σ*) ⟷ ¬ *isRes σ*

fun *TE-arOf* **where**
TE-arOf (*Oldf f*) = *arOf f*
|*TE-arOf* (*Tag σ*) = [*σ*]
|*TE-arOf* (*Wit σ*) = []

fun *TE-resOf* **where**
TE-resOf (*Oldf f*) = *resOf f*
|*TE-resOf* (*Tag σ*) = *σ*
|*TE-resOf* (*Wit σ*) = *σ*

lemma *countable-TE-wtFsym*: *countable* (*Collect TE-wtFsym*) (**is** *countable ?K*)
proof−
let *?F* = *λ ef*. *case ef of Oldf f* ⇒ *Inl f* | *Tag σ* ⇒ *Inr* (*Inl σ*) | *Wit σ* ⇒ *Inr* (*Inr σ*)
let *?U* = (*UNIV*::*'tp set*) <+> (*UNIV*::*'tp set*)
let *?L* = (*Collect wtFsym*) <+> *?U*
have *inj-on ?F ?K* **unfolding** *inj-on-def* **apply** *clarify*
apply(*case-tac x*, *simp-all*) **by** (*case-tac y*, *simp-all*)+
moreover have *?F* ‘ *?K* ⊆ *?L* **apply** *clarify* **by** (*case-tac ef*, *auto*)
ultimately have |*?K*| ≤*o* |*?L*| **unfolding** *card-of-ordLeq*[*symmetric*] **by** *auto*
moreover have *countable ?L* **using** *countable-wtFsym countable-tp*
by (*metis countable-Plus*)
ultimately show *?thesis* **by**(*rule countable-ordLeq*)
qed

end

sublocale *ProblemIkTpart* < *TE?* : *Signature*
where *wtFsym* = *TE-wtFsym* **and** *arOf* = *TE-arOf* **and** *resOf* = *TE-resOf*
apply *standard*
using *countable-tp countable-TE-wtFsym countable-wtPsym* **by** *auto*

context *ProblemIkTpart*

begin

fun *tNN* **where**
tNN (*Var x*) =
(*if unprot* (*tpOfV x*) ∨ *protFw* (*tpOfV x*) *then Var x else Fn* (*Tag* (*tpOfV x*)) [*Var x*])
|
tNN (*Fn f Tl*) = (*if unprot* (*resOf f*) ∨ *protFw* (*resOf f*)
then Fn (*Oldf f*) (*map tNN Tl*)
else Fn (*Tag* (*resOf f*)) [*Fn* (*Oldf f*) (*map tNN Tl*)])

fun *tT* **where**
tT (*Var x*) = (*if unprot* (*tpOfV x*) *then Var x else Fn* (*Tag* (*tpOfV x*)) [*Var x*])
|
tT t = *tNN t*

fun *tL* **where**
tL (*Pos* (*Eq T1 T2*)) = *Pos* (*Eq* (*tT T1*) (*tT T2*))
|
tL (*Neg* (*Eq T1 T2*)) = *Neg* (*Eq* (*tNN T1*) (*tNN T2*))
|
tL (*Pos* (*Pr p Tl*)) = *Pos* (*Pr p* (*map tNN Tl*))
|
tL (*Neg* (*Pr p Tl*)) = *Neg* (*Pr p* (*map tNN Tl*))

definition *tC* ≡ *map tL*

definition *rOfFax f* = *Fn* (*Oldf f*) (*getTvars* (*arOf f*))

definition *lOfFax f* = *Fn* (*Tag* (*resOf f*)) [*rOfFax f*]
definition *Fax* ≡ {[*Pos* (*Eq* (*lOfFax f*) (*rOfFax f*))] | *f*. *wtFsym f*}

definition *rOfWax σ* = *Fn* (*Wit σ*) []

definition *lOfWax σ* = *Fn* (*Tag σ*) [*rOfWax σ*]
definition *Wax* ≡ {[*Pos* (*Eq* (*lOfWax σ*) (*rOfWax σ*))] | *σ*. ¬ *isRes σ* ∧ *protFw σ*}

definition *tPB* = *tC* ‘ Φ ∪ *Fax* ∪ *Wax*

lemma *tpOf-tNN*[*simp*]: *TE.tpOf* (*tNN T*) = *Ik.tpOf T*

by (*induct T*) *auto*

lemma *tpOf-t*[*simp*]: *TE.tpOf* (*tT T*) = *Ik.tpOf T*
by (*cases T*) *auto*

lemma *wt-tNN*[*simp*]: *Ik.wt T* ⟹ *TE.wt* (*tNN T*)
by (*induct T*, *auto simp add*: *list-all-iff*)

lemma *wt-t*[*simp*]: *Ik.wt T* ⟹ *TE.wt* (*tT T*)
by (*cases T*, *auto simp add*: *list-all-iff*)

lemma *wtL-tL*[*simp*]: *Ik.wtL l* ⟹ *TE.wtL* (*tL l*)
apply(*cases l*) **apply** (*rename-tac* [!] *atm*) **apply**(*case-tac* [!] *atm*)
by (*auto simp add*: *list-all-iff*)

lemma *wtC-tC*[*simp*]: *Ik.wtC c* ⟹ *TE.wtC* (*tC c*)
unfolding *tC-def Ik.wtC-def TE.wtC-def* **by** (*induct c*, *auto*)

lemma *tpOf-rOfFax*[*simp*]: *TE.tpOf* (*rOfFax f*) = *resOf f*
unfolding *rOfFax-def* **by** *simp*

lemma *tpOf-lOfFax*[*simp*]: *TE.tpOf* (*lOfFax f*) = *resOf f*
unfolding *lOfFax-def* **by** *simp*

lemma *tpOf-rOfWax*[*simp*]: *TE.tpOf* (*rOfWax σ*) = *σ*
unfolding *rOfWax-def* **by** *simp*

lemma *tpOf-lOfWax*[*simp*]: *TE.tpOf* (*lOfWax σ*) = *σ*
unfolding *lOfWax-def* **by** *simp*

lemma *wt-rOfFax*[*simp*]: *wtFsym f* ⟹ *TE.wt* (*rOfFax f*)
unfolding *rOfFax-def* **by** *simp*

lemma *wt-lOfFax*[*simp*]: *wtFsym f* ⟹ *TE.wt* (*lOfFax f*)
unfolding *lOfFax-def* **by** *simp*

lemma *wt-rOfWax*[*simp*]: ¬ *isRes σ* ⟹ *TE.wt* (*rOfWax σ*)
unfolding *rOfWax-def* **by** *simp*

lemma *wt-lOfWax*[*simp*]: ¬ *isRes σ* ⟹ *TE.wt* (*lOfWax σ*)
unfolding *lOfWax-def* **by** *simp*

lemma *wtPB-Fax*[*simp*]: *TE.wtPB Fax* **unfolding** *Fax-def TE.wtPB-def TE.wtC-def*
by *auto*

lemma *wtPB-Wax*[*simp*]: *TE.wtPB Wax* **unfolding** *Wax-def TE.wtPB-def TE.wtC-def*
by *auto*

lemma *wtPB-tC-Φ*[*simp*]: *TE.wtPB* (*tC ‘ Φ*)

using *Ik.wt-Φ* **unfolding** *Ik.wtPB-def TE.wtPB-def* **by** *auto*

lemma *wtPB-tPB*[*simp*]: *TE.wtPB tPB* **unfolding** *tPB-def* **by** *simp*

lemma *wt-Tag*:
assumes *TE.wt* (*Fn* (*Tag* σ) *Tl*)
shows $\exists$ *T*. *Tl* = [*T*] $\wedge$ *TE.wt T* $\wedge$ *tpOf T* = σ
using *assms*
by *simp* (*metis* (*opaque-lifting*, *no-types*) *list.inject list-all-simps*(*1*) *map-eq-Cons-conv neq-Nil-conv*)

lemma *tpOf-Tag*: *TE.tpOf* (*Fn* (*Tag* σ) *Tl*) = σ **by** *simp*

lemma *wt-Wit*:
assumes *TE.wt* (*Fn* (*Wit* σ) *Tl*)
shows *Tl* = []
using *assms* **by** *simp*

lemma *tpOf-Wit*: *TE.tpOf* (*Fn* (*Wit* σ) *Tl*) = σ **by** *simp*

end

8.2 Soundness

context *ModelIkTpart* **begin**

fun *TE-intF* **where**
TE-intF (*Oldf f*) *al* = *intF f al*
| *TE-intF* (*Tag* σ) *al* = *hd al*
| *TE-intF* (*Wit* σ) *al* = *pickT* σ

end

sublocale *ModelIkTpart* < *TE?* : *Struct*
where *wtFsym* = *TE-wtFsym* **and** *arOf* = *TE-arOf* **and** *resOf* = *TE-resOf* **and** *intF* = *TE-intF*
proof
fix *ef al* **assume** *TE-wtFsym ef* **and** *list-all2 intT* (*TE-arOf ef*) *al*
thus *intT* (*TE-resOf ef*) (*TE-intF ef al*)
using *intF* **by** (*cases ef*, *auto*)
qed *auto*

context *ModelIkTpart* **begin**

lemma *tNN-int*[*simp*]: *TE.int* ξ (*tNN T*) = *Ik.int* ξ *T*
proof(*induct T*)

case (*Fn f Tl*)
hence *0*: *map* (*TE.int ξ* ∘ *tNN*) *Tl* = *map* (*Ik.int ξ*) *Tl*
unfolding *list-eq-iff list-all-iff* **by** *auto*
show *?case* **by** (*simp add*: *0*)
qed *auto*

lemma *map-tNN-int*[*simp*]: *map* (*TE.int ξ* ∘ *tNN*) *Tl* = *map* (*Ik.int ξ*) *Tl*
unfolding *list-eq-iff list-all-iff* **by** *auto*

lemma *t-int*[*simp*]: *TE.int ξ* (*tT T*) = *Ik.int ξ T*
by (*cases T*, *auto*)

lemma *map-t-int*[*simp*]: *map* (*TE.int ξ* ∘ *tT*) *Tl* = *map* (*Ik.int ξ*) *Tl*
unfolding *list-eq-iff list-all-iff* **by** *auto*

lemma *tL-satL*[*simp*]: *TE.satL ξ* (*tL l*) ⟷ *Ik.satL ξ l*
apply(*cases l*) **apply** (*rename-tac* [!] *atm*) **apply**(*case-tac* [!] *atm*) **by** *auto*

lemma *tC-satC*[*simp*]: *TE.satC ξ* (*tC c*) ⟷ *Ik.satC ξ c*
unfolding *TE.satC-def Ik.satC-def tC-def* **by** (*induct c*, *auto*)

lemma *tC-Φ-satPB*[*simp*]: *TE.satPB ξ* (*tC* ‘ Φ) ⟷ *Ik.satPB ξ* Φ
unfolding *TE.satPB-def Ik.satPB-def* **by** *auto*

lemma *Fax-Wax-satPB*:
TE.satPB ξ (*Fax*) ∧ *TE.satPB ξ* (*Wax*)
unfolding *TE.satPB-def TE.satC-def Fax-def Wax-def*
by (*auto simp add*: *lOfFax-def rOfFax-def lOfWax-def rOfWax-def*)

lemmas *Fax-satPB*[*simp*] = *Fax-Wax-satPB*[*THEN conjunct1*]
lemmas *Wax-satPB*[*simp*] = *Fax-Wax-satPB*[*THEN conjunct2*]

lemma *soundness*: *TE.SAT tPB*
unfolding *TE.SAT-def tPB-def* **using** *SAT* **unfolding** *Ik.SAT-def* **by** *auto*

theorem *T-soundness*:
Model TE-wtFsym wtPsym TE-arOf TE-resOf parOf tPB intT TE-intF intP
apply *standard* **using** *wtPB-tPB soundness* **by** *auto*

end

sublocale *ModelIkTpart* < *TE?* : *Model*
where *wtFsym* = *TE-wtFsym* **and** *arOf* = *TE-arOf* **and** *resOf* = *TE-resOf*
and Φ = *tPB* **and** *intF* = *TE-intF*
apply *standard* **using** *wtPB-tPB soundness* **by** *auto*

8.3 Completeness

definition *iimg B f a* ≡ *if a* ∈ *f* ‘ *B then a else f a*

lemma *inImage-iimg*[*simp*]: *a* ∈ *f* ‘ *B* ⟹ *iimg B f a* = *a*
unfolding *iimg-def* **by** *auto*

lemma *not-inImage-iimg*[*simp*]: *a* ∉ *f* ‘ *B* ⟹ *iimg B f a* = *f a*
unfolding *iimg-def* **by** *auto*

lemma *iimg*[*simp*]: *a* ∈ *B* ⟹ *iimg B f* (*f a*) = *f a*
by (*cases a* ∈ *f* ‘ *B*, *auto*)

locale *ProblemIkTpart-TEModel* =
Ik? : *ProblemIkTpart wtFsym wtPsym arOf resOf parOf* Φ *infTp prot protFw* +
TE? : *Model ProblemIkTpart.TE-wtFsym wtFsym resOf wtPsym*
ProblemIkTpart.TE-arOf arOf ProblemIkTpart.TE-resOf resOf parOf
tPB eintT eintF eintP
for *wtFsym* :: *'fsym* ⇒ *bool*
and *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *infTp* **and** *prot* **and** *protFw*
and *eintT* **and** *eintF* **and** *eintP*

context *ProblemIkTpart-TEModel* **begin**

definition
ntsem σ ≡
if unprot σ ∨ *protFw σ then id*
else iimg {*b*. *eintT σ b*} (*eintF* (*Tag σ*) *o singl*)

lemma *unprot-ntsem*[*simp*]: *unprot σ* ∨ *protFw σ* ⟹ *ntsem σ a* = *a*
unfolding *ntsem-def* **by** *simp*

lemma *protFw-ntsem*[*simp*]: *protFw σ* ⟹ *ntsem σ a* = *a*
unfolding *ntsem-def* **by** *simp*

lemma *inImage-ntsem*[*simp*]:
a ∈ (*eintF* (*Tag σ*) *o singl*) ‘ {*b*. *eintT σ b*} ⟹ *ntsem σ a* = *a*
unfolding *ntsem-def* **by** *simp*

lemma *not-unprot-inImage-ntsem*[*simp*]:
assumes ¬ *unprot σ* **and** ¬ *protFw σ* **and** *a* ∉ (*eintF* (*Tag σ*) *o singl*) ‘ {*b*. *eintT σ b*}
shows *ntsem σ a* = *eintF* (*Tag σ*) [*a*]
using *assms* **unfolding** *ntsem-def* **by** (*simp add*: *singl-def*)

lemma *ntsem*[*simp*]:
eintT σ b ⟹ *ntsem σ* (*eintF* (*Tag σ*) [*b*]) = *eintF* (*Tag σ*) [*b*]
unfolding *singl-def*[*symmetric*] **by** *simp*

lemma *eintT-ntsem*:
assumes *a*: *eintT σ a* **shows** *eintT σ* (*ntsem σ a*)
proof(*cases unprot σ* ∨ *protFw σ*)
case *False* **note** *unprot* = *False* **show** *?thesis*
proof(*cases a* ∈ (*eintF* (*Tag σ*) *o singl*) ‘ {*b*. *eintT σ b*})
case *False* **hence** *1*: *ntsem σ a* = *eintF* (*Tag σ*) [*a*] **using** *unprot* **by** *simp*
show *?thesis* **unfolding** *1* **using** *a TE.intF*
by (*metis TE-arOf.simps TE-resOf.simps TE-wtFsym.simps list-all2-Cons list-all2-Nil*)
qed(*insert a*, *auto*)
qed(*insert a*, *simp*)

definition
intT σ a ≡
if unprot σ then eintT σ a
else if protFw σ then eintT σ a ∧ *eintF* (*Tag σ*) [*a*] = *a*
else eintT σ a ∧ *a* ∈ (*eintF* (*Tag σ*) *o singl*) ‘ {*b*. *eintT σ b*}
definition
intF f al ≡
if unprot (*resOf f*) ∨ *protFw* (*resOf f*)
then eintF (*Oldf f*) (*map2 ntsem* (*arOf f*) *al*)
else eintF (*Tag* (*resOf f*)) [*eintF* (*Oldf f*) (*map2 ntsem* (*arOf f*) *al*)]
definition
intP p al ≡ *eintP p* (*map2 ntsem* (*parOf p*) *al*)

lemma *TE-Tag*:
assumes *f*: *wtFsym f* **and** *al*: *list-all2 eintT* (*arOf f*) *al*
shows *eintF* (*Tag* (*resOf f*)) [*eintF* (*Oldf f*) *al*] = *eintF* (*Oldf f*) *al*
proof−
define *xl* **where** *xl* = *getVars* (*arOf f*)
have *l*[*simp*]: *length xl* = *length al length al* = *length* (*arOf f*)
unfolding *xl-def* **using** *al* **unfolding** *list-all2-iff* **by** *auto*
have *1*[*simp*]: ⋀ *i*. *i* < *length* (*arOf f*) ⟹ *tpOfV* (*xl*!*i*) = (*arOf f*)!*i*
unfolding *xl-def* **by** *auto*
have *xl*[*simp*]: *distinct xl* **unfolding** *xl-def* **using** *distinct-getVars* **by** *auto*
define ξ **where** ξ = *pickE xl al*
have ξ: *TE.wtE* ξ **unfolding** ξ-*def* **apply**(*rule wtE-pickE*)
using *al list-all2-nthD* **by** *auto*
have [*simp*]: ⋀ *i*. *i* < *length* (*arOf f*) ⟹ ξ (*xl* ! *i*) = *al* ! *i*
using *al* **unfolding** ξ-*def* **by** (*auto simp*: *list-all2-length intro*: *pickE*)
have *0*: *map* (*TE.int* ξ) (*getTvars* (*arOf f*)) = *al*
apply(*rule nth-equalityI*)

```
  using al by (auto simp: list-all2-length getTvars-def xl-def[symmetric])
  have TE.satPB ξ Fax using TE.sat-Φ[OF ξ] unfolding tPB-def by simp
  hence TE.satC ξ [Pos (Eq (lOfFax f) (rOfFax f))]
  unfolding TE.satPB-def Fax-def using f by auto
  hence TE.satA ξ (Eq (lOfFax f) (rOfFax f)) unfolding TE.satC-def by simp
  thus ?thesis using al by (simp add: lOfFax-def rOfFax-def 0)
qed

lemma TE-Wit:
assumes σ: ¬ isRes σ protFw σ
shows eintF (Tag σ) [eintF (Wit σ) []] = eintF (Wit σ) []
proof−
  define ξ where ξ = pickE [] []
  have ξ: TE.wtE ξ unfolding ξ-def apply(rule wtE-pickE) by auto
  have TE.satPB ξ Wax using TE.sat-Φ[OF ξ] unfolding tPB-def by simp
  hence TE.satC ξ [Pos (Eq (lOfWax σ) (rOfWax σ))]
  unfolding TE.satPB-def Wax-def using σ by auto
 hence TE.satA ξ (Eq (lOfWax σ) (rOfWax σ)) unfolding TE.satC-def by auto
  thus ?thesis unfolding TE.satA.simps lOfWax-def rOfWax-def by simp
qed

lemma NE-intT-forget: NE (intT σ)
proof−
  obtain b where b: eintT σ b using TE.NE-intT by blast
  show ?thesis proof(cases unprot σ)
    case True thus ?thesis using b unfolding intT-def by auto
  next
    case False note unprot = False show ?thesis
    proof(cases protFw σ)
      case True note protFw = True show ?thesis proof(cases isRes σ)
        case True then obtain f where f: wtFsym f and σ: σ = resOf f
        unfolding isRes-def by auto
        define al where al = map pickT (arOf f)
        have al: list-all2 eintT (arOf f) al
        unfolding al-def list-all2-map2 unfolding list-all2-length by auto
        define a where a = eintF (Oldf f) al
        have eintT σ a unfolding a-def σ using f al
        by (metis TE-arOf.simps TE-resOf.simps TE-wtFsym.simps TE.intF)
       moreover have eintF (Tag σ) [a] = a unfolding σ a-def using TE-Tag[OF
f al] .
       ultimately show ?thesis using unprot protFw unfolding intT-def by auto
      next
        case False
        define a where a = eintF (Wit σ) []
        have eintT σ a unfolding a-def
        by (metis False TE-arOf.simps TE-resOf.simps TE-wtFsym.simps TE.intF
list-all2-NilR)
        moreover have eintF (Tag σ) [a] = a unfolding a-def using TE-Wit[OF
False protFw] .
```

```
      ultimately show ?thesis using unprot protFw unfolding intT-def by auto
     qed
    next
     case False  hence eintT σ (eintF (Tag σ) [b])
     using b list-all2-Cons list-all2-NilL
     by (metis TE.intF TE-arOf.simps TE-resOf.simps TE-wtFsym.simps)
     hence intT σ (eintF (Tag σ) [b])
     unfolding intT-def singl-def[abs-def] using b False by auto
     thus ?thesis by blast
    qed
  qed
qed

lemma wt-intF:
assumes f: wtFsym f and al: list-all2 intT (arOf f) al
shows intT (resOf f) (intF f al)
proof−
  let ?t = eintF (Tag (resOf f))
  let ?t'al = map2 ntsem (arOf f) al
  have al: list-all2 eintT (arOf f) al
  using al unfolding list-all2-length intT-def by metis
  have 0: list-all2 eintT (arOf f) ?t'al
  proof(rule listAll2-map2I)
    show l: length (arOf f) = length al
    using al unfolding list-all2-length by simp
    fix i assume i < length (arOf f)
    hence 1: eintT (arOf f ! i) (al ! i)
    using al unfolding list-all2-length by simp
    show eintT (arOf f ! i) (ntsem (arOf f ! i) (al ! i))
    using eintT-ntsem[OF 1] .
  qed
  hence 1: eintT (resOf f) (eintF (Oldf f) ?t'al)
  by (metis TE-arOf.simps TE-resOf.simps TE-wtFsym.simps f TE.intF)
  show ?thesis  proof(cases unprot (resOf f))
    case True thus ?thesis unfolding intF-def intT-def by (simp add: 1)
  next
    case False note unprot = False show ?thesis proof(cases protFw (resOf f))
      case True thus ?thesis using unprot TE-Tag[OF f 0] 1
      unfolding intF-def intT-def by simp
    next
      case False
      have eintT (resOf f) (intF f al) using intF-def 0 1 TE-Tag f by auto
      moreover have
      intF f al ∈ (eintF (Tag (resOf f)) o singl) ‘ {b. eintT (resOf f) b}
      unfolding intF-def using 1 unprot False by (auto simp add: singl-def)
      ultimately show ?thesis using False unfolding intT-def by simp
    qed
  qed
qed
```

lemma *Struct*: *Struct wtFsym wtPsym arOf resOf intT intF intP*
apply *standard* **using** *NE-intT-forget wt-intF* **by** *auto*

end

sublocale *ProblemIkTpart-TEModel* < *Ik?* : *Struct*
where *intT* = *intT* **and** *intF* = *intF* **and** *intP* = *intP*
using *Struct* **.**

context *ProblemIkTpart-TEModel* **begin**

definition
invt σ *a* $\equiv$ *if unprot* σ $\vee$ *protFw* σ *then a else* (*SOME b. eintT* σ *b* $\wedge$ *eintF* (*Tag* σ) [*b*] = *a*)

lemma *unprot-invt*[*simp*]: *unprot* σ $\vee$ *protFw* σ $\Longrightarrow$ *invt* σ *a* = *a*
unfolding *invt-def* **by** *auto*

lemma *invt-invt-inImage*:
assumes σ: $\neg$ *unprot* σ $\neg$ *protFw* σ
and *a*: *a* $\in$ (*eintF* (*Tag* σ) *o singl*) ' {*b. eintT* σ *b*}
shows *eintT* σ (*invt* σ *a*) $\wedge$ *eintF* (*Tag* σ) [*invt* σ *a*] = *a*
proof−
obtain *b* **where** *eintT* σ *b* **and** *eintF* (*Tag* σ) [*b*] = *a*
using *a* **unfolding** *image-def singl-def*[*symmetric*] **by** *auto*
thus *?thesis* **using** σ **unfolding** *invt-def* **apply** *simp* **apply**(*rule someI-ex*) **by** *auto*
qed

lemmas *invt*[*simp*] = *invt-invt-inImage*[*THEN conjunct1*]
lemmas *invt-inImage*[*simp*] = *invt-invt-inImage*[*THEN conjunct2*]

term *invt*
definition *eenv* ξ *x* $\equiv$ *invt* (*tpOfV x*) (ξ *x*)

lemma *wt-eenv*:
assumes ξ: *Ik.wtE* ξ **shows** *TE.wtE* (*eenv* ξ)
unfolding *TE.wtE-def* **proof** *safe*
fix *x* **let** *?σ* = *TE.tpOfV x*
show *eintT ?σ* (*eenv* ξ *x*) **proof**(*cases unprot ?σ*)
case *True* **note** *unprot* = *True*
thus *?thesis* **unfolding** *eenv-def* **by** (*metis* ξ *Ik.wtTE-intT intT-def unprot-invt*)
next
case *False* **note** *unprot* = *False* **show** *?thesis* **proof**(*cases protFw ?σ*)

```
    case True thus ?thesis unfolding eenv-def using unprot ξ
    by (metis Ik.wtTE-intT intT-def unprot-invt)
   next
    case False thus ?thesis unfolding eenv-def using unprot ξ
    by (metis (no-types) ξ Ik.wtE-def intT-def invt)
   qed
  qed
qed

lemma int-tNN[simp]:
assumes T: Ik.Ik.wt T and ξ: Ik.wtE ξ
shows TE.int (eenv ξ) (tNN T) = Ik.int ξ T
using T proof(induct T)
  case (Var x) let ?σ = TE.tpOfV x  show ?case
  proof(cases unprot ?σ)
   case False note unprot = False
   show ?thesis proof(cases protFw ?σ)
    case True thus ?thesis using unprot ξ
    unfolding eenv-def Ik.wtE-def intT-def by simp
   next
    case False  hence ξ x ∈ (eintF (Tag ?σ) ∘ singl) ‘ {b. eintT ?σ b}
     using ξ unprot unfolding wtE-def intT-def singl-def[abs-def] by (simp cong
del: image-cong-simp)
    thus ?thesis using unprot unfolding eenv-def using False by simp
   qed
  qed(unfold eenv-def, simp)
next
  case (Fn f Tl)
  let ?eξ = eenv ξ let ?ar = arOf f let ?r = resOf f
  have l: length ?ar = length Tl using Fn by simp
  have 0: map2 ntsem ?ar (map (Ik.int ξ) Tl) =
          map (TE.int ?eξ ∘ tNN) Tl  (is ?L = ?R)
  proof(rule nth-equalityI)
   fix i assume i < length ?L
   hence i: i < length ?ar using l by simp
   hence 1: TE.int (eenv ξ) (tNN (Tl!i)) = Ik.int ξ (Tl!i)
    using Fn by (auto simp: list-all-length)
   have 2: ?ar ! i = Ik.Ik.tpOf (Tl!i) using Fn i by simp
   have 3: intT (?ar ! i) (Ik.int ξ (Tl ! i))
    unfolding 2 apply(rule wt-int)
    using Fn ξ i by (auto simp: list-all-length)
   show ?L!i = ?R!i apply (cases unprot (?ar ! i) ∨ protFw (?ar ! i))
    using i 1 l 3 unfolding intT-def by auto
  qed(insert l, auto)
  show ?case apply(cases unprot ?r ∨ protFw ?r)
   using [[unfold-abs-def = false]]
   unfolding Ik.int.simps TE.int.simps tT.simps unfolding intF-def using Fn
0 by auto
qed
```

lemma *map-int-tNN*[*simp*]:
assumes *Tl*: *list-all Ik.Ik.wt Tl* **and** *ξ*: *Ik.wtE ξ*
shows
map2 ntsem (*map Ik.Ik.tpOf Tl*) (*map* (*Ik.int ξ*) *Tl*) =
map (*TE.int* (*eenv ξ*) ∘ *tNN*) *Tl*
proof−
{fix *i* **assume** *i*: *i* < *length Tl*
hence *wt*: *Ik.Ik.wt* (*Tl*!*i*) **using** *Tl* **unfolding** *list-all-length* **by** *simp*
have *intT* (*Ik.Ik.tpOf* (*Tl*!*i*)) (*Ik.int ξ* (*Tl*!*i*)) **using** *Ik.wt-int*[*OF ξ wt*] **.**
}
thus *?thesis*
using [[*unfold-abs-def* = *false*]]
using *assms* **unfolding** *intT-def list-all-length*
unfolding *list-eq-iff* **apply** *clarsimp* **by** (*metis inImage-ntsem unprot-ntsem*)
qed

lemma *int-t*[*simp*]:
assumes *T*: *Ik.Ik.wt T* **and** *ξ*: *Ik.wtE ξ*
shows *TE.int* (*eenv ξ*) (*tT T*) = *Ik.int ξ T*
using *T* **proof**(*induct T*)
case (*Var x*) **let** *?σ* = *tpOfV x* **show** *?case*
proof(*cases unprot ?σ*)
case *False* **note** *unprot* = *False*
show *?thesis* **proof**(*cases protFw ?σ*)
case *True* **thus** *?thesis* **using** *unprot ξ*
unfolding *eenv-def Ik.wtE-def intT-def* **by** *simp*
next
case *False* **hence** *ξ x* ∈ (*eintF* (*Tag ?σ*) ∘ *singl*) ‘ {*b*. *eintT ?σ b*}
using *ξ unprot* **unfolding** *wtE-def intT-def singl-def*[*abs-def*] **by** (*simp cong del*: *image-cong-simp*)
thus *?thesis* **using** *unprot* **unfolding** *eenv-def* **using** *False* **by** *simp*
qed
qed(*unfold eenv-def*, *simp*)
next
case (*Fn f Tl*)
let *?eξ* = *eenv ξ* **let** *?ar* = *arOf f* **let** *?r* = *resOf f*
have *l*: *length ?ar* = *length Tl* **using** *Fn* **by** *simp*
have *ar*: *?ar* = *map Ik.Ik.tpOf Tl* **using** *Fn* **by** *simp*
have *0*: *map2 ntsem ?ar* (*map* (*Ik.int ξ*) *Tl*) = *map* (*TE.int ?eξ* ∘ *tNN*) *Tl*
unfolding *ar* **apply**(*rule map-int-tNN*[*OF* - *ξ*]) **using** *Fn* **by** *simp*
show *?case* **apply**(*cases unprot ?r* ∨ *protFw ?r*)
using [[*unfold-abs-def* = *false*]]
unfolding *Ik.int.simps TE.int.simps tT.simps* **unfolding** *intF-def* **using** *Fn 0*
by *auto*
qed

lemma *map-int-t*[*simp*]:
assumes *Tl*: *list-all Ik.Ik.wt Tl* **and** *ξ*: *Ik.wtE ξ*

shows
map2 ntsem (*map Ik.Ik.tpOf Tl*) (*map* (*Ik.int* ξ) *Tl*) =
map (*TE.int* (*eenv* ξ) $\circ$ *tT*) *Tl*
proof−
{fix *i* **assume** *i*: *i* < *length Tl*
hence *wt*: *Ik.Ik.wt* (*Tl*!*i*) **using** *Tl* **unfolding** *list-all-length* **by** *simp*
have *intT* (*Ik.Ik.tpOf* (*Tl*!*i*)) (*Ik.int* ξ (*Tl*!*i*)) **using** *wt-int*[*OF* ξ *wt*] **.**
}
thus *?thesis*
using [[*unfold-abs-def* = *false*]]
using *assms* **unfolding** *intT-def list-all-length*
unfolding *list-eq-iff* **apply** *clarsimp* **by** (*metis inImage-ntsem unprot-ntsem*)
qed

lemma *satL-tL*[*simp*]:
assumes *l*: *Ik.Ik.wtL l* **and** ξ: *Ik.wtE* ξ
shows *TE.satL* (*eenv* ξ) (*tL l*) $\longleftrightarrow$ *Ik.satL* ξ *l*
using *assms* **apply**(*cases l*) **apply** (*rename-tac* [!] *atm*) **by** (*case-tac* [!] *atm*) (*auto simp add*: *intP-def*)

lemma *satC-tC*[*simp*]:
assumes *l*: *Ik.Ik.wtC c* **and** ξ: *Ik.wtE* ξ
shows *TE.satC* (*eenv* ξ) (*tC c*) $\longleftrightarrow$ *Ik.satC* ξ *c*
unfolding *TE.satC-def Ik.satC-def*
using *assms* **by** (*induct c*, *auto simp add*: *Ik.Ik.wtC-def tC-def*)

lemma *satPB-tPB*[*simp*]:
assumes ξ: *Ik.wtE* ξ
shows *TE.satPB* (*eenv* ξ) (*tC* ‘ Φ) $\longleftrightarrow$ *Ik.satPB* ξ Φ
using *Ik.wt-*Φ *assms* **unfolding** *TE.satPB-def Ik.satPB-def* **by** (*auto simp add*: *Ik.Ik.wtPB-def*)

lemma *completeness*: *Ik.SAT* Φ
unfolding *Ik.SAT-def* **proof** *safe*
fix ξ **assume** ξ: *Ik.wtE* ξ **hence** *TE.wtE* (*eenv* ξ) **by**(*rule wt-eenv*)
hence *TE.satPB* (*eenv* ξ) *tPB* **by** (*rule TE.sat-*Φ)
hence *TE.satPB* (*eenv* ξ) (*tC* ‘ Φ) **unfolding** *tPB-def* **by** *simp*
thus *Ik.satPB* ξ Φ **using** ξ **by** *simp*
qed

lemma *T-completeness*: *Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP*
by *standard* (*rule completeness*)

end

sublocale *ProblemIkTpart-TEModel* < *O?* : *Model*
where *intT* = *intT* **and** *intF* = *intF* **and** *intP* = *intP*

using *T-completeness* **.**

8.4 The result of the tag translation is an infiniteness-augmented problem

sublocale *ProblemIkTpart* < *TE?* : *Problem*
where *wtFsym* = *TE-wtFsym* **and** *arOf* = *TE-arOf* **and** *resOf* = *TE-resOf*
and Φ = *tPB*
apply *standard* **by** *auto*

sublocale *ProblemIkTpart* < *TE?* : *ProblemIk*
where *wtFsym* = *TE-wtFsym* **and** *arOf* = *TE-arOf* **and** *resOf* = *TE-resOf*
and Φ = *tPB*
proof
fix σ *eintT eintF eintP a* **assume** σ: *infTp* σ
assume *M*: *Model TE-wtFsym wtPsym TE-arOf TE-resOf parOf tPB eintT eintF eintP*
let *?TE-intT* = *ProblemIkTpart-TEModel.intT prot protFw eintT eintF*
let *?TE-intF* = *ProblemIkTpart-TEModel.intF arOf resOf prot protFw eintT eintF*
let *?TE-intP* = *ProblemIkTpart-TEModel.intP parOf prot protFw eintT eintF eintP*
have *0*: *ProblemIkTpart-TEModel wtFsym wtPsym arOf resOf parOf*
Φ *infTp prot protFw eintT eintF eintP*
using *M* **unfolding** *ProblemIkTpart-TEModel-def* **apply** *safe* **apply** *standard*
done
hence *MM*: *Ik.MModel ?TE-intT ?TE-intF ?TE-intP*
by (*rule ProblemIkTpart-TEModel.T-completeness*)
have *infinite* {*a*. *?TE-intT* σ *a*} **using** *infTp*[*OF* σ *MM*] **.**
moreover have {*a*. *?TE-intT* σ *a*} ⊆ {*a*. *eintT* σ *a*}
using *ProblemIkTpart-TEModel.intT-def*[*OF 0*] **by** *auto*
ultimately show *infinite* {*a*. *eintT* σ *a*} **using** *infinite-super* **by** *blast*
qed

8.5 The verification of the first monotonicity calculus criterion for the tagged problem

context *ProblemIkTpart* **begin**

lemma *nvT-t*[*simp*]: ¬ *unprot* σ ⟹ (∀ *x* ∈ *TE.nvT* (*tT T*). *tpOfV x* ≠ σ)
apply(*induct T*) **by** *auto*

lemma *nvL-tL*[*simp*]: ¬ *unprot* σ ⟹ (∀ *x* ∈ *TE.nvL* (*tL l*). *tpOfV x* ≠ σ)
apply(*cases l*) **apply**(*rename-tac* [!] *atm*) **apply**(*case-tac* [!] *atm*) **by** *auto* (*metis nvT-t*)+

lemma *nvC-tC*[*simp*]: ¬ *unprot* σ ⟹ (∀ *x* ∈ *TE.nvC* (*tC c*). *tpOfV x* ≠ σ)
unfolding *tC-def TE.nvC-def* **apply** (*induct c*)
by *auto* (*metis* (*full-types*) *nvL-tL*)+

lemma *unprot-nvT-t*[*simp*]:
unprot (*tpOfV x*) $\Longrightarrow$ $x \in$ *TE.nvT* (*tT T*) $\longleftrightarrow$ $x \in$ *TE.nvT T*
by (*induct T*, *auto*)

lemma *tpL-nvT-tL*[*simp*]:
unprot (*tpOfV x*) $\Longrightarrow$ $x \in$ *TE.nvL* (*tL l*) $\longleftrightarrow$ $x \in$ *TE.nvL l*
by (*cases l*, *rename-tac* [!] *atm*, *case-tac* [!] *atm*, *auto*)

lemma *unprot-nvC-tC*[*simp*]:
unprot (*tpOfV x*) $\Longrightarrow$ $x \in$ *TE.nvC* (*tC c*) $\longleftrightarrow$ $x \in$ *TE.nvC c*
unfolding *tC-def TE.nvC-def* **apply** (*induct c*) **by** *auto*

lemma *nv-OfFax*[*simp*]:
$x \notin$ *TE.nvT* (*lOfFax f*) $x \notin$ *TE.nvT* (*rOfFax f*)
unfolding *lOfFax-def rOfFax-def lOfWax-def rOfWax-def* **by** *auto*

lemma *nv-OfWax*[*simp*]:
$x \notin$ *TE.nvT* (*lOfWax* σ') $x \notin$ *TE.nvT* (*rOfWax* σ')
unfolding *lOfFax-def rOfFax-def lOfWax-def rOfWax-def* **by** *auto*

lemma *nvC-Fax*: $c \in$ *Fax* $\Longrightarrow$ *TE.nvC c* $= \{\}$ **unfolding** *Fax-def TE.nvC-def* **by** *auto*
lemma *mcalc-Fax*: $c \in$ *Fax* $\Longrightarrow$ *TE.mcalc* σ *c* **using** *nvC-Fax* **unfolding** *TE.mcalc-iff* **by** *auto*
lemma *nvC-Wax*: $c \in$ *Wax* $\Longrightarrow$ *TE.nvC c* $= \{\}$ **unfolding** *Wax-def TE.nvC-def* **by** *auto*
lemma *mcalc-Wax*: $c \in$ *Wax* $\Longrightarrow$ *TE.mcalc* σ *c* **using** *nvC-Wax*[*of c*] **by** *simp*

end

sublocale *ProblemIkTpart* $<$ *TE?*: *ProblemIkMcalc*
where *wtFsym* $=$ *TE-wtFsym* **and** *arOf* $=$ *TE-arOf* **and** *resOf* $=$ *TE-resOf*
and Φ $=$ *tPB*
proof
  fix σ *c* **assume** $c \in$ *tPB*
  thus *TE.mcalc* σ *c* **unfolding** *tPB-def* **proof** *safe*
    fix *d* **assume** *d*: $d \in \Phi$ **thus** *TE.mcalc* σ (*tC d*)
    using *unprot-mcalc*[*OF* - *d*] **unfolding** *TE.mcalc-iff* **by** (*cases unprot* σ, *auto*, *force*)
  qed(*insert mcalc-Fax mcalc-Wax*, *blast*+)
qed

context *ProblemIkTpart* **begin**

theorem *T-monotonic*:

MonotProblem TE-wtFsym wtPsym TE-arOf TE-resOf parOf tPB **..**

end

sublocale *ProblemIkTpart* < *TE?*: *MonotProblem*
where *wtFsym* = *TE-wtFsym* **and** *arOf* = *TE-arOf* **and** *resOf* = *TE-resOf* **and**
Φ = *tPB*
using *T-monotonic* **.**

end

9 Untyped (Unsorted) First-Order Logic

theory *U*
imports *TermsAndClauses*
begin

Even though untyped FOL is a particular case of many-typed FOL, we find it convenient to represent it separately.

9.1 Signatures

locale *Signature* =
fixes
wtFsym :: *'fsym* ⇒ *bool*
and *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *nat*
and *parOf* :: *'psym* ⇒ *nat*
assumes *countable-wtFsym*: *countable* {*f*::*'fsym*. *wtFsym f*}
and *countable-wtPsym*: *countable* {*p*::*'psym*. *wtPsym p*}
begin

fun *wt* **where**
wt (*Var x*) ⟷ *True*
|
wt (*Fn f Tl*) ⟷ *wtFsym f* ∧ *list-all wt Tl* ∧ *arOf f* = *length Tl*

lemma *wt-induct*[*elim*, *consumes 1*, *case-names Var Fn*, *induct pred*: *wt*]:
assumes *T*: *wt T*
and *Var*: ⋀ *x*. *φ* (*Var x*)
and *Fn*:
⋀ *f Tl*.
⟦*wtFsym f*; *list-all wt Tl*; *arOf f* = *length Tl*; *list-all φ Tl*⟧
⟹ *φ* (*Fn f Tl*)
shows *φ T*
proof−

have *wt T* $\longrightarrow$ φ *T*
apply (*induct T*) **using** *Var Fn* **by** (*auto simp*: *list-all-iff*)
thus *?thesis* **using** *T* **by** *auto*
qed

definition *wtSB* $\pi \equiv \forall$ *x*. *wt* (π *x*)

lemma *wtSB-wt*[*simp*]: *wtSB* $\pi \Longrightarrow$ *wt* (π *x*)
unfolding *wtSB-def* **by** *auto*

lemma *wt-subst*[*simp*]:
assumes *wtSB* π **and** *wt T*
shows *wt* (*subst* π *T*)
using *assms* **by** (*induct T*) (*auto simp*: *list-all-length*)

lemma *wtSB-o*:
assumes *1*: *wtSB* π*1* **and** *2*: *wtSB* π*2*
shows *wtSB* (*subst* π*1 o* π*2*)
using *2* **unfolding** *wtSB-def* **using** *1* **by** *auto*

end

9.2 Structures

type-synonym *'univ env* = *var* $\Rightarrow$ *'univ*

locale *Struct* = *Signature wtFsym wtPsym arOf parOf*
for *wtFsym* **and** *wtPsym*
and *arOf* :: *'fsym* $\Rightarrow$ *nat*
and *parOf* :: *'psym* $\Rightarrow$ *nat*
+
fixes
D :: *'univ* $\Rightarrow$ *bool*
and *intF* :: *'fsym* $\Rightarrow$ *'univ list* $\Rightarrow$ *'univ*
and *intP* :: *'psym* $\Rightarrow$ *'univ list* $\Rightarrow$ *bool*
assumes
NE-D: *NE D* **and**
intF: $[\![$*wtFsym f*; *length al* = *arOf f*; *list-all D al*$]\!] \Longrightarrow$ *D* (*intF f al*)
and
dummy: *parOf* = *parOf* $\wedge$ *intF* = *intF* $\wedge$ *intP* = *intP*
begin

definition *wtE* $\xi \equiv \forall$ *x*. *D* (ξ *x*)

lemma *wtTE-D*[*simp*]: *wtE* $\xi \Longrightarrow D\ (\xi\ x)$
unfolding *wtE-def* **by** *simp*

fun *int* **where**
int ξ (*Var* x) = $\xi\ x$
|
int ξ (*Fn* f *Tl*) = *intF* f (*map* (*int* ξ) *Tl*)

lemma *wt-int*:
assumes *wtE*: *wtE* ξ **and** *wt-T*: *wt T*
shows *D* (*int* ξ *T*)
using *wt-T* **apply**(*induct T*)
apply (*metis int.simps*(*1*) *wtE wtE-def*)
unfolding *int.simps* **apply**(*rule intF*)
unfolding *list-all-map comp-def* **by** *auto*

lemma *int-cong*:
assumes $\bigwedge x.\ x \in vars\ T \Longrightarrow \xi 1\ x = \xi 2\ x$
shows *int* $\xi 1$ *T* = *int* $\xi 2$ *T*
using *assms* **apply**(*induct T*) **apply** *simp-all* **unfolding** *list-all-iff*
using *map-ext* **by** *metis*

lemma *int-o*:
int (*int* ξ *o* ϱ) *T* = *int* ξ (*subst* ϱ *T*)
apply(*induct T*) **apply** *simp-all* **unfolding** *list-all-iff o-def*
using *map-ext* **by** (*metis* (*lifting*, *no-types*))

lemmas *int-subst* = *int-o*[*symmetric*]

lemma *int-o-subst*:
int ξ *o subst* ϱ = *int* (*int* ξ *o* ϱ)
apply(*rule ext*) **apply**(*subst comp-def*) **unfolding** *int-o*[*symmetric*] **..**

lemma *wtE-o*:
assumes *1*: *wtE* ξ **and** *2*: *wtSB* ϱ
shows *wtE* (*int* ξ *o* ϱ)
unfolding *wtE-def* **apply** *safe*
unfolding *comp-def* **apply**(*rule wt-int*[*OF 1*]) **using** *2* **by** *auto*

end

context *Signature* **begin**

Well-typed (i.e., well-formed) atoms, literals, caluses and problems:

fun *wtA* **where**
wtA (*Eq T1 T2*) $\longleftrightarrow$ *wt T1* $\wedge$ *wt T2*

|
wtA (Pr p Tl) ⟷ *wtPsym p* ∧ *list-all wt Tl* ∧ *parOf p* = *length Tl*

fun *wtL* **where**
wtL (Pos a) ⟷ *wtA a*
|
wtL (Neg a) ⟷ *wtA a*

definition *wtC* ≡ *list-all wtL*

definition *wtPB* Φ ≡ ∀ *c* ∈ Φ. *wtC c*

end

context *Struct* **begin**

fun *satA* **where**
satA ξ *(Eq T1 T2)* ⟷ *int* ξ *T1* = *int* ξ *T2*
|
satA ξ *(Pr r Tl)* ⟷ *intP r (map (int* ξ*) Tl)*

fun *satL* **where**
satL ξ *(Pos a)* ⟷ *satA* ξ *a*
|
satL ξ *(Neg a)* ⟷ ¬ *satA* ξ *a*

definition *satC* ξ ≡ *list-ex (satL* ξ*)*

definition *satPB* ξ Φ ≡ ∀ *c* ∈ Φ. *satC* ξ *c*

definition *SAT* Φ ≡ ∀ ξ. *wtE* ξ ⟶ *satPB* ξ Φ

end

9.3 Problems

locale *Problem* = *Signature wtFsym wtPsym arOf parOf*
for *wtFsym* **and** *wtPsym*
and *arOf* :: *'fsym* ⇒ *nat*
and *parOf* :: *'psym* ⇒ *nat*
+
fixes Φ :: (*'fsym*, *'psym*) *prob*
assumes *wt*-Φ: *wtPB* Φ

9.4 Models of a problem

locale *Model* =
  Problem wtFsym wtPsym arOf parOf Φ +
  Struct wtFsym wtPsym arOf parOf D intF intP
for *wtFsym* **and** *wtPsym*
and *arOf* :: *'fsym* ⇒ *nat*
and *parOf* :: *'psym* ⇒ *nat*
and Φ :: (*'fsym*, *'psym*) *prob*
and *D* :: *'univ* ⇒ *bool*
and *intF* :: *'fsym* ⇒ *'univ list* ⇒ *'univ*
and *intP* :: *'psym* ⇒ *'univ list* ⇒ *bool*
+
assumes *SAT*: *SAT* Φ

end

theory *CU*
imports *U*
begin

locale *Struct* = *U.Struct wtFsym wtPsym arOf parOf D intF intP*
for *wtFsym* **and** *wtPsym*
and *arOf* :: *'fsym* ⇒ *nat*
and *parOf* :: *'psym* ⇒ *nat*
and *D* :: *univ* ⇒ *bool*
and *intF* :: *'fsym* ⇒ *univ list* ⇒ *univ*
and *intP* :: *'psym* ⇒ *univ list* ⇒ *bool*

locale *Model* = *U.Model wtFsym wtPsym arOf parOf* Φ *D intF intP*
for *wtFsym* **and** *wtPsym*
and *arOf* :: *'fsym* ⇒ *nat*
and *parOf* :: *'psym* ⇒ *nat*
and Φ
and *D* :: *univ* ⇒ *bool*
and *intF* :: *'fsym* ⇒ *univ list* ⇒ *univ*
and *intP* :: *'psym* ⇒ *univ list* ⇒ *bool*

end

10 The type-erasure translation from many-typed to untyped FOL

theory *E* **imports** *Mono CU*
begin

10.1 Preliminaries

locale *M-Signature* = *M?* : *Sig.Signature*
locale *M-Problem* = *M?* : *M.Problem*
locale *M-MonotModel* = *M?* : *MonotModel wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP*
for *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** *intT* **and** *intF* **and** *intP* **and** Φ
locale *M-FullStruct* = *M?* : *FullStruct*
locale *M-FullModel* = *M?* : *FullModel*

sublocale *M-FullStruct* < *M-Signature* **..**
sublocale *M-Problem* < *M-Signature* **..**
sublocale *M-FullModel* < *M-FullStruct* **..**
sublocale *M-MonotModel* < *M-FullStruct* **where**
intT = *intTF* **and** *intF* = *intFF* **and** *intP* = *intPF* **..**
sublocale *M-MonotModel* < *M-FullModel* **where**
intT = *intTF* **and** *intF* = *intFF* **and** *intP* = *intPF* **..**

context *Sig.Signature* **begin**

end

10.2 The translation

sublocale *M-Signature* < *U.Signature*
where *arOf* = *length o arOf* **and** *parOf* = *length o parOf*
by *standard* (*auto simp*: *countable-wtFsym countable-wtPsym*)

context *M-Signature* **begin**

lemma *wt*[*simp*]: *M.wt T* ⟹ *wt T*
by (*induct T*, *auto simp add*: *list-all-iff*)

lemma *wtA*[*simp*]: *M.wtA at* ⟹ *wtA at*
apply(*cases at*) **by** (*auto simp add*: *list-all-iff*)

lemma *wtL*[*simp*]: *M.wtL l* ⟹ *wtL l*
apply(*cases l*) **by** *auto*

lemma *wtC*[*simp*]: *M.wtC c* ⟹ *wtC c*
unfolding *M.wtC-def wtC-def* **by** (*induct c*, *auto*)

lemma *wtPB*[*simp*]: *M.wtPB* Φ ⟹ *wtPB* Φ

unfolding *M.wtPB-def wtPB-def* **by** *auto*

end

10.3 Completeness

The next puts together an M_signature with a structure for its U.flattened signature:

locale *UM-Struct* =
M? : *M-Signature wtFsym wtPsym arOf resOf parOf* +
U? : *CU.Struct wtFsym wtPsym length o arOf length o parOf D intF intP*
for *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** *D* **and** *intF* **and** *intP*

sublocale *UM-Struct* < *M?* : *M.Struct* **where** *intT* = λ *σ*. *D*
apply *standard*
apply(*rule NE-D*)
unfolding *list-all2-list-all* **apply**(*rule intF*) **by** *auto*

context *UM-Struct* **begin**

lemma *wtE*[*simp*]: *M.wtE ξ* ⟹ *U.wtE ξ*
unfolding *U.wtE-def M.wtE-def* **by** *auto*

lemma *int-e*[*simp*]: *U.int ξ T* = *M.int ξ T*
by (*induct T*, *simp-all add*: *list-all-iff*) (*metis map-eq-conv*)

lemma *int-o-e*[*simp*]: *U.int ξ* = *M.int ξ*
unfolding *int-e fun-eq-iff* **by** *simp*

lemma *satA-e*[*simp*]: *U.satA ξ at* ⟷ *M.satA ξ at*
by (*cases at*) *auto*

lemma *satL-e*[*simp*]: *U.satL ξ l* ⟷ *M.satL ξ l*
apply(*cases l*) **by** *auto*

lemma *satC-e*[*simp*]: *U.satC ξ c* ⟷ *M.satC ξ c*
unfolding *M.satC-def U.satC-def* **by** (*induct c*, *simp-all*)

lemma *satPB-e*[*simp*]: *U.satPB ξ* Φ ⟷ *M.satPB ξ* Φ
unfolding *M.satPB-def U.satPB-def* **by** *auto*

theorem *completeness*:
assumes *U.SAT* Φ **shows** *M.SAT* Φ
using *assms* **unfolding** *M.SAT-def satPB-e U.SAT-def* **by** *auto*

end

locale *UM-Model* =
  M-Problem wtFsym wtPsym arOf resOf parOf Φ +
  UM-Struct wtFsym wtPsym arOf resOf parOf D intF intP +
  CU.Model wtFsym wtPsym length o arOf length o parOf Φ
      D intF intP
for *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list*
and *resOf* **and** *parOf* **and** Φ **and** *D* **and** *intF* **and** *intP*
begin

theorem *M-U-completeness*: *MModel* (λσ::*'tp. D*) *intF intP*
apply *standard* **apply**(*rule completeness*[*OF SAT*]) **.**

end

Global statement of completeness : UM_Model consists of an M.problem and an U.model satisfying the U.translation of this problem. It is stated that it yields a model for the M.problem.

sublocale *UM-Model* < *CM.Model* **where** *intT* = λ σ. *D*
using *M-U-completeness* **.**

10.4 Soundness for monotonic problems

sublocale *M-FullStruct* < *U?* : *CU.Struct*
where *arOf* = *length o arOf* **and** *parOf* = *length o parOf* **and** *D* = *intT any*
apply *standard*
  apply(*rule NE-intT*)
  apply (*rule full2*)
  unfolding *full-True list-all2-list-all* **by** *auto*

context *M-FullModel* **begin**

lemma *wtE*[*simp*]: *U.wtE* ξ ⟹ *F.wtE* ξ
unfolding *U.wtE-def F.wtE-def* **by** *auto*

lemma *int-e*[*simp*]: *U.int* ξ *T* = *F.int* ξ *T*
by (*induct T*, *simp-all add*: *list-all-iff*) (*metis map-eq-conv*)

lemma *int-o-e*[*simp*]: *U.int* ξ = *F.int* ξ
unfolding *fun-eq-iff* **by** *auto*

lemma *satA-e*[*simp*]: *U.satA* ξ *at* ⟷ *F.satA* ξ *at*
by (*cases at*) *auto*

lemma *satL-e*[*simp*]: *U.satL* ξ *l* ⟷ *F.satL* ξ *l*
by (*cases l*) *auto*

lemma *satC-e*[*simp*]: *U.satC* ξ *c* ⟷ *F.satC* ξ *c*

unfolding *F.satC-def U.satC-def* **by** (*induct c*, *simp-all*)

lemma *satPB-e*[*simp*]: *U.satPB* ξ $\Phi \longleftrightarrow$ *F.satPB* ξ Φ
unfolding *F.satPB-def U.satPB-def* **by** *auto*

theorem *soundness*: *U.SAT* Φ
unfolding *U.SAT-def* **using** *sat-*Φ *satPB-e* **by** *auto*

lemma *U-Model*:
CU.Model wtFsym wtPsym (*length* $\circ$ *arOf*) (*length* $\circ$ *parOf*) Φ (*intT any*) *intF intP*
by *standard* (*rule wtPB*[*OF wt-*Φ], *rule soundness*)

end

sublocale *M-FullModel* < *CU.Model*
where *arOf* = *length o arOf* **and** *parOf* = *length o parOf* **and** *D* = *intT any*
using *U-Model* **.**

context *M-MonotModel* **begin**

theorem *M-U-soundness*:
CU.Model wtFsym wtPsym (*length* $\circ$ *arOf*) (*length* $\circ$ *parOf*) Φ
(*InfModel.intTF* (*any*::*'tp*))
(*InfModel.intFF arOf resOf intTI intFI*) (*InfModel.intPF parOf intTI intPI*)
apply(*rule M-FullModel.U-Model*)
unfolding *M-FullModel-def* **apply**(*rule InfModel.FullModel*)
apply(*rule MonotModel.InfModelI*) **..**

end

Global statement of the soundness theorem: M_MonotModel consists of a monotonic F.problem satisfied by an F.model. It is stated that this yields an U.Model for the translated problem.

sublocale *M-MonotModel* < *CU.Model*
where *arOf* = *length o arOf* **and** *parOf* = *length o parOf*
and Φ = Φ **and** *D* = *intTF* (*any*::*'tp*) **and** *intF* = *intFF* **and** *intP* = *intPF*
using *M-U-soundness* **.**

end

11 End Results in Locale-Free Form

theory *Encodings*
imports *G T E*
begin

This section contains the outcome of our type-encoding results, presented in a locale-free fashion. It is not very useful from an Isabelle development point of view, where the locale theorems are fine.

Rather, this is aimed as a quasi-self-contained formal documentation of the overall results for the non-Isabelle-experts.

11.1 Soundness

In the soundness theorems below, we employ the following Isabelle types:
- type variables (parameters):
— *'tp*, of types
— *'fsym*, of function symbols
— *'psym*, of predicate symbols
- a fixed countable universe *univ* for the carrier of the models
and various operators on these types:

(1) the constitutive parts of FOL signatures:
—- the boolean predicates *wtFsym* and *wtPsym*, indicating the "well-typed" function and predicate symbols; these are just means to select only subsets of these symbols for consideration in the signature
—- the operators arOf and resOf, giving the arity and the result type of function symbols
—- the operator parOf, giving the arity of predicate symbols

(2) the problem, Φ, which is a set of clauses over the considered signature

(3) a partition of the types in:
— *tpD*, the types that should be decorated in any case
— *tpFD*, the types that should be decorated in a featherweight fashion
— for guards only, a further refinement of *tpD*, indicating, as *tpCD*, the types that should be classically (i.e., traditionally) decorated (these partitionings are meant to provide a uniform treatment of the three styles of encodings: traditional, lightweight and featherweight)

(4) the constitutive parts of a structure over the considered signature:
—- *intT*, the interpretation of each type as a unary predicate (essentially, a set) over an arbitrary type 'univ
—- *intF* and *intP*, the interpretations of the function and predicate symbols as actual functions and predicates over *univ*.

Soundness of the tag encodings:

The assumptions of the tag soundness theorems are the following:

(a) *ProblemIkTpart wtFsym wtPsym arOf resOf parOf* Φ *infTp tpD tpFD*, stating that:

— (*wtFsym*, *wtPsym*, *arOf*, *resOf*, *parOf*) form a countable signature
— Φ is a well-typed problem over this signature
— *infTp* is an indication of the types that the problem guarantees as infinite in all models
— *tpD* and *tpFD* are disjoint and all types that are not marked as *tpD* or *tpFD* are deemed safe by the monotonicity calculus from *Mcalc*

(b) *CM.Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP* says that (*intT*, *intF*, *intP*) is a model for Φ (hence Φ is satisfiable)

The conclusion says that we obtain a model of the untyped version of the problem (after suitable tags and axioms have been added):

Because of the assumptions on tpD and tpFD, we have the following particular cases of our parameterized tag encoding:
– if *tpD* is taked to be everywhere true (hence *tpFD* becomes everywhere false), we obtain the traditional tag encoding
– if *tpD* is taken to be true precisely when the monotonicity calculus fails, we obtain the lightweight tag encoding
– if *tpFD* is taken to be true precisely when the monotonicity calculus fails, we obtain the featherweight tag encoding

theorem *tags-soundness*:
fixes *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list* **and** *resOf* :: *'fsym* ⇒ *'tp* **and** *parOf* :: *'psym* ⇒ *'tp list*
and Φ :: (*'fsym*, *'psym*) *prob* **and** *infTp* :: *'tp* ⇒ *bool*
and *tpD* :: *'tp* ⇒ *bool* **and** *tpFD* :: *'tp* ⇒ *bool*
and *intT* :: *'tp* ⇒ *univ* ⇒ *bool*
and *intF* :: *'fsym* ⇒ *univ list* ⇒ *univ* **and** *intP* :: *'psym* ⇒ *univ list* ⇒ *bool*
— The problem translation:
— First the addition of tags (“TE” stands for “tag encoding”):
defines *TE-wtFsym* ≡ *ProblemIkTpart.TE-wtFsym wtFsym resOf*
and *TE-arOf* ≡ *ProblemIkTpart.TE-arOf arOf*
and *TE-resOf* ≡ *ProblemIkTpart.TE-resOf resOf*
defines *TE-*Φ ≡ *ProblemIkTpart.tPB wtFsym arOf resOf* Φ *tpD tpFD*
— Then the deletion of types (“U” stands for “untyped”):
and *U-arOf* ≡ *length* ∘ *TE-arOf*
and *U-parOf* ≡ *length* ∘ *parOf*
defines *U-*Φ ≡ *TE-*Φ
— The forward model translation:
— First, using monotonicity, we build an infinite model of Φ (“I” stands for “infinite”):
defines *intTI* ≡ *MonotProblem.intTI TE-wtFsym wtPsym TE-arOf TE-resOf parOf TE-*Φ
and *intFI* ≡ *MonotProblem.intFI TE-wtFsym wtPsym TE-arOf TE-resOf parOf TE-*Φ
and *intPI* ≡ *MonotProblem.intPI TE-wtFsym wtPsym TE-arOf TE-resOf parOf TE-*Φ
— Then, by isomorphic transfer of the lattter model, we build a model of Φ that

has all types interpeted as *univ* (“F” stands for “full"):
defines *intFF* ≡ *InfModel.intFF TE-arOf TE-resOf intTI intFI*
and *intPF* ≡ *InfModel.intPF parOf intTI intPI*
— Then we build a model of *U*-Φ:
defines *U-intT* ≡ *InfModel.intTF* (*any*::*'tp*)

— Assumptions of the theorem:
assumes
P: *ProblemIkTpart wtFsym wtPsym arOf resOf parOf* Φ *infTp tpD tpFD*
and *M*: *CM.Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP*

— Conclusion of the theorem:
shows *CU.Model TE-wtFsym wtPsym U-arOf U-parOf U-*Φ *U-intT intFF intPF*

unfolding *U-arOf-def U-parOf-def U-*Φ*-def*
unfolding *U-intT-def intTI-def intFI-def intPI-def intFF-def intPF-def*
apply(*rule M-MonotModel.M-U-soundness*)
unfolding *M-MonotModel-def MonotModel-def* **apply** *safe*
unfolding *TE-wtFsym-def TE-arOf-def TE-resOf-def TE-*Φ*-def*
apply(*rule ProblemIkTpart.T-monotonic*)
apply(*rule P*)
apply(*rule ModelIkTpart.T-soundness*) **unfolding** *ModelIkTpart-def* **apply** *safe*
apply(*rule P*)
apply(*rule M*)
done

Soundness of the guard encodings:

Here the assumptions and conclusion have a similar shapes as those for the tag encodings. The difference is in the first assumption, *ProblemIkTpartG wtFsym wtPsym arOf resOf parOf* Φ *infTp tpD tpFD tpCD*, which consists of *ProblemIkTpart wtFsym wtPsym arOf resOf parOf* Φ *infTp tpD tpFD* together with the following assumptions about *tpCD*:
– *tpCD* is included in *tpD*
– if a result type of an operation symbol is in *tpD*, then so are all the types in its arity

We have the following particular cases of our parameterized guard encoding:
– if *tpD* and *tpCD* are taked to be everywhere true (hence *tpFD* becomes everywhere false), we obtain the traditional guard encoding
– if *tpCD* is taken to be false and *tpD* is taken to be true precisely when the monotonicity calculus fails, we obtain the lightweight tag encoding
– if *tpFD* is taken to be true precisely when the monotonicity calculus fails, we obtain the featherweight tag encoding

theorem *guards-soundness*:
fixes *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*

and *arOf* :: *'fsym* ⇒ *'tp list* **and** *resOf* :: *'fsym* ⇒ *'tp* **and** *parOf* :: *'psym* ⇒ *'tp list*
and Φ :: (*'fsym*, *'psym*) *prob* **and** *infTp* :: *'tp* ⇒ *bool*
and *tpD* :: *'tp* ⇒ *bool* **and** *tpFD* :: *'tp* ⇒ *bool* **and** *tpCD* :: *'tp* ⇒ *bool*
and *intT* :: *'tp* ⇒ *univ* ⇒ *bool*
and *intF* :: *'fsym* ⇒ *univ list* ⇒ *univ*
and *intP* :: *'psym* ⇒ *univ list* ⇒ *bool*
— The problem translation:
defines *GE-wtFsym* ≡ *ProblemIkTpartG.GE-wtFsym wtFsym resOf tpCD*
and *GE-wtPsym* ≡ *ProblemIkTpartG.GE-wtPsym wtPsym tpD tpFD*
and *GE-arOf* ≡ *ProblemIkTpartG.GE-arOf arOf*
and *GE-resOf* ≡ *ProblemIkTpartG.GE-resOf resOf*
and *GE-parOf* ≡ *ProblemIkTpartG.GE-parOf parOf*

defines *GE-*Φ ≡ *ProblemIkTpartG.gPB wtFsym arOf resOf* Φ *tpD tpFD tpCD*
and *U-arOf* ≡ *length* ∘ *GE-arOf*
and *U-parOf* ≡ *length* ∘ *GE-parOf*

defines *U-*Φ ≡ *GE-*Φ

— The model forward translation:
defines *intTI* ≡ *MonotProblem.intTI GE-wtFsym GE-wtPsym GE-arOf GE-resOf GE-parOf GE-*Φ
and *intFI* ≡ *MonotProblem.intFI GE-wtFsym GE-wtPsym GE-arOf GE-resOf GE-parOf GE-*Φ
and *intPI* ≡ *MonotProblem.intPI GE-wtFsym GE-wtPsym GE-arOf GE-resOf GE-parOf GE-*Φ

defines *intFF* ≡ *InfModel.intFF GE-arOf GE-resOf intTI intFI*
and *intPF* ≡ *InfModel.intPF GE-parOf intTI intPI*

defines *U-intT* ≡ *InfModel.intTF* (*any*::*'tp*)

assumes
P: *ProblemIkTpartG wtFsym wtPsym arOf resOf parOf* Φ *infTp tpD tpFD tpCD*
and *M*: *CM.Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP*

shows *CU.Model GE-wtFsym GE-wtPsym U-arOf U-parOf U-*Φ *U-intT intFF intPF*

unfolding *U-arOf-def U-parOf-def U-*Φ*-def*
unfolding *U-intT-def intTI-def intFI-def intPI-def intFF-def intPF-def*
apply(*rule M-MonotModel.M-U-soundness*)
unfolding *M-MonotModel-def MonotModel-def* **apply** *safe*
 unfolding *GE-wtFsym-def GE-wtPsym-def GE-arOf-def GE-resOf-def GE-parOf-def GE-*Φ*-def*
  apply(*rule ProblemIkTpartG.G-monotonic*)
   apply(*rule P*)
   apply(*rule ModelIkTpartG.G-soundness*)

unfolding *ModelIkTpartG-def ModelIkTpart-def* **apply** *safe*
apply(*rule P*)
using *P* **unfolding** *ProblemIkTpartG-def* **apply** *fastforce*
apply(*rule M*)
done

11.2 Completeness

The setting is similar to the one for completeness, except for the following point:

(3) The constitutive parts of a structure over the untyped signature resulted from the addition of the tags or guards followed by the deletion of the types: (*D*, *eintF*, *eintP*)

Completeness of the tag encodings

theorem *tags-completeness*:
fixes *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list* **and** *resOf* :: *'fsym* ⇒ *'tp* **and** *parOf* :: *'psym* ⇒ *'tp list*
and Φ :: (*'fsym*, *'psym*) *prob* **and** *infTp* :: *'tp* ⇒ *bool*
and *tpD* :: *'tp* ⇒ *bool* **and** *tpFD* :: *'tp* ⇒ *bool*

and *D* :: *univ* ⇒ *bool*
and *eintF* :: (*'fsym*,*'tp*) *T.efsym* ⇒ *univ list* ⇒ *univ*
and *eintP* :: *'psym* ⇒ *univ list* ⇒ *bool*

— The problem translation (the same as in the case of soundness):
defines *TE-wtFsym* ≡ *ProblemIkTpart.TE-wtFsym wtFsym resOf*
and *TE-arOf* ≡ *ProblemIkTpart.TE-arOf arOf*
and *TE-resOf* ≡ *ProblemIkTpart.TE-resOf resOf*
defines *TE-*Φ ≡ *ProblemIkTpart.tPB wtFsym arOf resOf* Φ *tpD tpFD*
and *U-arOf* ≡ *length* ∘ *TE-arOf*
and *U-parOf* ≡ *length* ∘ *parOf*
defines *U-*Φ ≡ *TE-*Φ

— The backward model translation:
defines *intT* ≡ *ProblemIkTpart-TEModel.intT tpD tpFD* (λσ::*'tp. D*) *eintF*
and *intF* ≡ *ProblemIkTpart-TEModel.intF arOf resOf tpD tpFD* (λσ::*'tp. D*) *eintF*
and *intP* ≡ *ProblemIkTpart-TEModel.intP parOf tpD tpFD* (λσ::*'tp. D*) *eintF eintP*

assumes
P: *ProblemIkTpart wtFsym wtPsym arOf resOf parOf* Φ *infTp tpD tpFD* **and**
M: *CU.Model TE-wtFsym wtPsym* (*length o TE-arOf*)
(*length o parOf*) *TE-*Φ *D eintF eintP*

shows *CM.Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP*
proof−
have *UM*: *UM-Model TE-wtFsym wtPsym TE-arOf TE-resOf parOf TE-*Φ *D eintF eintP*
unfolding *UM-Model-def UM-Struct-def*
using *M* **unfolding** *CU.Model-def CU.Struct-def U.Model-def*
using *ProblemIkTpart.T-monotonic*[*OF P*,
unfolded TE-wtFsym-def[*symmetric*] *TE-arOf-def*[*symmetric*]
TE-resOf-def[*symmetric*] *TE-*Φ*-def*[*symmetric*]]
by (*auto simp*: *MonotProblem-def M-Problem-def M-Signature-def M.Problem-def*)
show *?thesis*
unfolding *intT-def intF-def intP-def*
apply(*rule ProblemIkTpart-TEModel.T-completeness*)
unfolding *ProblemIkTpart-TEModel-def* **apply** *safe*
apply(*rule P*)
apply(*rule UM-Model.M-U-completeness*)
apply(*rule UM*[*unfolded TE-wtFsym-def TE-arOf-def TE-resOf-def TE-*Φ*-def*])
done
qed

Completeness of the guard encodings

theorem *guards-completeness*:
fixes *wtFsym* :: *'fsym* ⇒ *bool* **and** *wtPsym* :: *'psym* ⇒ *bool*
and *arOf* :: *'fsym* ⇒ *'tp list* **and** *resOf* :: *'fsym* ⇒ *'tp* **and** *parOf* :: *'psym* ⇒ *'tp list*
and Φ :: (*'fsym*, *'psym*) *prob* **and** *infTp* :: *'tp* ⇒ *bool*
and *tpD* :: *'tp* ⇒ *bool* **and** *tpFD* :: *'tp* ⇒ *bool* **and** *tpCD* :: *'tp* ⇒ *bool*

and *D* :: *univ* ⇒ *bool*
and *eintF* :: (*'fsym*,*'tp*) *G.efsym* ⇒ *univ list* ⇒ *univ*
and *eintP* :: (*'psym*,*'tp*) *G.epsym* ⇒ *univ list* ⇒ *bool*

— The problem translation (the same as in the case of soundness):
defines *GE-wtFsym* ≡ *ProblemIkTpartG.GE-wtFsym wtFsym resOf tpCD*
and *GE-wtPsym* ≡ *ProblemIkTpartG.GE-wtPsym wtPsym tpD tpFD*
and *GE-arOf* ≡ *ProblemIkTpartG.GE-arOf arOf*
and *GE-resOf* ≡ *ProblemIkTpartG.GE-resOf resOf*
and *GE-parOf* ≡ *ProblemIkTpartG.GE-parOf parOf*
defines *GE-*Φ ≡ *ProblemIkTpartG.gPB wtFsym arOf resOf* Φ *tpD tpFD tpCD*
and *U-arOf* ≡ *length* ∘ *GE-arOf*
and *U-parOf* ≡ *length* ∘ *GE-parOf*
defines *U-*Φ ≡ *GE-*Φ

— The backward model translation:
defines *intT* ≡ *ProblemIkTpartG-GEModel.intT tpD tpFD* (λσ::*'tp. D*) *eintP*
and *intF* ≡ *ProblemIkTpartG-GEModel.intF eintF*
and *intP* ≡ *ProblemIkTpartG-GEModel.intP eintP*

assumes
P: *ProblemIkTpartG wtFsym wtPsym arOf resOf parOf* Φ *infTp tpD tpFD tpCD*
and
M: *CU.Model GE-wtFsym GE-wtPsym* (*length o GE-arOf*)
(*length o GE-parOf*) *GE-*Φ *D eintF eintP*

shows *CM.Model wtFsym wtPsym arOf resOf parOf* Φ *intT intF intP*
proof−
have *UM*: *UM-Model GE-wtFsym GE-wtPsym GE-arOf GE-resOf GE-parOf GE-*Φ *D eintF eintP*
unfolding *UM-Model-def UM-Struct-def*
using *M* **unfolding** *CU.Model-def CU.Struct-def U.Model-def*
using *ProblemIkTpartG.G-monotonic*[*OF P*,
unfolded GE-wtFsym-def[*symmetric*] *GE-arOf-def*[*symmetric*]
GE-wtPsym-def[*symmetric*] *GE-parOf-def*[*symmetric*]
GE-resOf-def[*symmetric*] *GE-*Φ*-def*[*symmetric*]]
by (*auto simp*: *MonotProblem-def M-Problem-def M-Signature-def M.Problem-def*)
show *?thesis*
unfolding *intT-def intF-def intP-def*
apply(*rule ProblemIkTpartG-GEModel.G-completeness*)
unfolding *ProblemIkTpartG-GEModel-def* **apply** *safe*
apply(*rule P*)
apply(*rule UM-Model.M-U-completeness*)
apply(*rule UM*[*unfolded GE-wtFsym-def GE-wtPsym-def GE-parOf-def GE-arOf-def GE-resOf-def GE-*Φ*-def*])
done
qed

end