

# 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]. Essentially, 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

<b>1</b>	<b>Preliminaries</b>	<b>3</b>
1.1	Miscelanea . . . . .	3
1.2	List combinators . . . . .	4
1.3	Variables . . . . .	9
<b>2</b>	<b>Syntax of Terms and Clauses</b>	<b>11</b>
<b>3</b>	<b>Many-Typed (Many-Sorted) First-Order Logic</b>	<b>16</b>
3.1	Signatures . . . . .	16
3.2	Well-typed (well-formed) terms, clauses, literals and problems	20
3.3	Structures . . . . .	22
3.4	Problems . . . . .	29
3.5	Models of a problem . . . . .	30
<b>4</b>	<b>Monotonicity</b>	<b>31</b>
4.1	Fullness and infiniteness . . . . .	31
4.2	Monotonicity . . . . .	32
<b>5</b>	<b>The First Monotonicity Calculus</b>	<b>37</b>
5.1	Naked variables . . . . .	37
5.2	The calculus . . . . .	38
5.3	Extension of a structure to an infinite structure by adding indistinguishable elements . . . . .	38
5.4	The soundness of the calculus . . . . .	39
<b>6</b>	<b>The Second Monotonicity Calculus</b>	<b>43</b>
6.1	Extension policies . . . . .	43
6.2	Naked variables . . . . .	44
6.3	The calculus . . . . .	45
6.4	Constant policy on types . . . . .	46
6.5	Extension of a structure to an infinite structure by adding indistinguishable elements . . . . .	47
6.6	The soundness of the calculus . . . . .	49
<b>7</b>	<b>Guard-Based Encodings</b>	<b>55</b>
7.1	The guard translation . . . . .	55
7.2	Soundness . . . . .	60
7.3	Completeness . . . . .	62
7.4	The result of the guard translation is an infiniteness-augmented problem . . . . .	68
7.5	The verification of the second monotonicity calculus criterion for the guarded problem . . . . .	68

<b>8</b>	<b>Tag-Based Encodings</b>	<b>71</b>
8.1	The tag translation . . . . .	71
8.2	Soundness . . . . .	75
8.3	Completeness . . . . .	76
8.4	The result of the tag translation is an infiniteness-augmented problem . . . . .	85
8.5	The verification of the first monotonicity calculus criterion for the tagged problem . . . . .	85
<b>9</b>	<b>Untyped (Unsorted) First-Order Logic</b>	<b>87</b>
9.1	Signatures . . . . .	87
9.2	Structures . . . . .	88
9.3	Problems . . . . .	90
9.4	Models of a problem . . . . .	90
<b>10</b>	<b>The type-erasure translation from many-typed to untyped FOL</b>	<b>91</b>
10.1	Preliminaries . . . . .	91
10.2	The translation . . . . .	92
10.3	Completeness . . . . .	93
10.4	Soundness for monotonic problems . . . . .	94
<b>11</b>	<b>End Results in Locale-Free Form</b>	<b>95</b>
11.1	Soundness . . . . .	96
11.2	Completeness . . . . .	100

## 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**  $pick\ X \equiv SOME\ x.\ x \in X$

**lemma**  $pick[simp]: x \in X \implies pick\ X \in X$

**unfolding**  $pick-def$  **by**  $(metis\ someI-ex)$

**lemma**  $pick-NE[simp]: X \neq \{\} \implies pick\ X \in X$  **by**  $auto$

**definition**  $sappend$  (**infix**  $@@$  60) **where**

$Al\ @@\ Bl = \{al\ @\ bl \mid al\ bl.\ al \in Al \wedge bl \in Bl\}$

**lemma**  $sappend-NE[simp]: A\ @@\ B \neq \{\} \longleftrightarrow A \neq \{\} \wedge B \neq \{\}$

**unfolding**  $sappend-def$  **by**  $auto$

**abbreviation**  $fst3 :: 'a * 'b * 'c \Rightarrow 'a$  **where**  $fst3\ abc \equiv fst\ abc$

**abbreviation**  $snd3\ abc \equiv fst\ (snd\ abc)$

**abbreviation**  $trd3\ abc \equiv snd\ (snd\ abc)$

**hide-const**  $int$

**abbreviation**  $any \equiv undefined$

Non-emptiness of predicates:

**abbreviation**  $(input)\ NE\ \varphi \equiv \exists\ a.\ \varphi\ a$

**lemma**  $NE-NE: NE\ NE$

**apply** $(rule\ exI[of\ -\ \lambda\ a.\ True])$  **by**  $auto$

**lemma**  $length-Suc-0:$

$length\ al = Suc\ 0 \longleftrightarrow (\exists\ a.\ al = [a])$

**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]:  $\text{length } al = \text{Suc } 0 \implies \text{singl } (\text{hd } al) = al$   
**unfolding** *singl-def* **by** (*cases al, auto*)

**lemma** *singl-inj*[simp]:  $\text{singl } a = \text{singl } b \longleftrightarrow a = b$   
**unfolding** *singl-def* **by** *simp*

**definition** *list A*  $\equiv \text{SOME } al. \text{distinct } al \wedge \text{set } al = A$

**lemma** *distinct-set-list*:  
 $\text{finite } A \implies \text{distinct } (\text{list } A) \wedge \text{set } (\text{list } A) = A$   
**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]:  $\text{set } (\text{list } (\text{set } xl)) = \text{set } xl$  **by** *auto*

**lemma** *length-list*[simp]:  $\text{finite } A \implies \text{length } (\text{list } A) = \text{card } A$   
**by** (*metis distinct-card distinct-set-list*)

**lemma** *list-all-mp*[*elim*]:  
**assumes** *list-all*  $(\lambda a. \varphi a \longrightarrow \psi a)$  *al* **and** *list-all*  $\varphi$  *al*  
**shows** *list-all*  $\psi$  *al*  
**using** *assms* **unfolding** *list-all-iff* **by** *auto*

**lemma** *list-all-map*:  
 $\text{list-all } \varphi (\text{map } f al) = \text{list-all } (\varphi \circ f) al$   
**by** (*metis (no-types) length-map list-all-length nth-map o-def*)

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

**lemma** *distinct-set-eq-Singl*[simp]:  $\text{distinct } al \implies \text{set } al = \{a\} \longleftrightarrow al = [a]$   
**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]:  $\text{list } \{b\} = [b]$   
**unfolding** *list-def* **apply**(*rule some-equality*) **by** *auto*

**lemma** *list-insert*:  
**assumes** *A*:  $\text{finite } A$  **and** *b*:  $b \notin A$   
**shows**  
 $\exists al1 al2.$   
 $A = \text{set } (al1 @ al2) \wedge \text{distinct } (al1 @ [b] @ al2) \wedge$   
 $\text{list } (\text{insert } b A) = al1 @ [b] @ al2$   
**proof**–

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

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

**lemma** *list-ex-list[*simp*]*:  
*finite*  $A \implies \text{list-ex } \varphi \ (\text{list } A) = (\exists a \in A. \varphi \ a)$   
**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 - - F = any*

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

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

**definition** *map2* **where**  
 $\text{map2 } f \ xl \ yl \equiv \text{map } (\text{case-prod } f) \ (\text{zip } xl \ yl)$

**lemma** *nth-map2[*simp*]*:  
**assumes**  $\text{length } bl = \text{length } al$  **and**  $i < \text{length } al$   
**shows**  $(\text{map2 } f \ al \ bl) ! i = f \ (al!i) \ (bl!i)$   
**using** *assms* **unfolding** *map2-def* **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*  $\longleftrightarrow$  *ys* = []  
**using** *list-all2-Nil-iff* **by** *auto*

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

**lemma** *list-all2-ConsL*:  
**assumes** *list-all2* *R* (*x* # *xs'*) *ys*  
**shows**  $\exists$  *y ys'*. *ys* = *y* # *ys'*  $\wedge$  *R* *x y*  $\wedge$  *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*:  $\bigwedge$  *y ys'*.  $\llbracket$ *ys* = *y* # *ys'*; *R* *x y*; *list-all2* *R* *xs' ys'* $\rrbracket \implies$  *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*:  $\bigwedge$  *y ys'*.  $\llbracket$ *ys* = *y* # *ys'*; *R* *x y*; *list-all2* *R* *xs' ys'* $\rrbracket \implies$  *phi*  
**shows** *phi*  
**using** *list-all2-ConsL* *assms* **by** *metis*

**lemma** *list-all2-ConsR*:  
**assumes** *list-all2* *R* *xs* (*y* # *ys'*)  
**shows**  $\exists$  *x xs'*. *xs* = *x* # *xs'*  $\wedge$  *R* *x y*  $\wedge$  *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*:  $\bigwedge$  *x xs'*.  $\llbracket$ *xs* = *x* # *xs'*; *R* *x y*; *list-all2* *R* *xs' ys'* $\rrbracket \implies$  *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*:  $\bigwedge$  *x xs'*.  $\llbracket$ *xs* = *x* # *xs'*; *R* *x y*; *list-all2* *R* *xs' ys'* $\rrbracket \implies$  *phi*  
**shows** *phi*  
**using** *list-all2-ConsR* *assms* **by** *metis*

**lemma** *ex-list-all2*:  
**assumes**  $\bigwedge$  *x*. *x*  $\in$  *set xs*  $\implies$   $\exists$  *y*. *f* *x y*  
**shows**  $\exists$  *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  $\bigwedge i. i < \text{length } xs2 \implies R (xs1!i) (xs2!i) \longleftrightarrow R' (ys1!i) (ys2!i)$

shows  $\text{list-all2 } R \ xs1 \ xs2 \longleftrightarrow \text{list-all2 } R' \ ys1 \ ys2$

by (metis assms list-all2-length)

**lemma** list-all2-o:  $\text{list-all2 } (P \circ f) \ al \ bl = \text{list-all2 } P \ (map \ f \ al) \ bl$

unfolding list-all2-map1 comp-def ..

**lemma** set-size-list:

assumes  $x \in \text{set } xs$

shows  $f \ x \leq \text{size-list } f \ xs$

by (metis assms size-list-estimation' order-eq-refl)

**lemma** nth-size-list:

assumes  $i < \text{length } xs$

shows  $f \ (xs!i) \leq \text{size-list } f \ xs$

by (metis assms nth-mem set-size-list)

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

$\text{list-all2 } (\lambda x. f) \ xs \ ys \longleftrightarrow$

$\text{length } xs = \text{length } ys \wedge \text{list-all } f \ ys$

by (metis list-all2-length list-all-length)

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

$\text{list-all2 } (\lambda x \ y. f \ x) \ xs \ ys \longleftrightarrow$

$\text{length } xs = \text{length } ys \wedge \text{list-all } f \ xs$

by (metis (lifting) list-all2-length list-all-length)

**lemma** list-all2-list-all-2[simp]:

$\text{list-all2 } f \ xs \ xs \longleftrightarrow \text{list-all } (\lambda x. f \ x \ x) \ xs$

by (auto simp add: list-all2-iff list-all-iff zip-same)

**lemma** list-all2-map-map:

$\text{list-all2 } \varphi \ (map \ f \ Tl) \ (map \ g \ Tl) =$

$\text{list-all } (\lambda T. \varphi \ (f \ T) \ (g \ T)) \ Tl$

unfolding list-all2-map1 list-all2-map2 list-all2-list-all-2 ..

**lemma** length-map2[simp]:

assumes  $\text{length } ys = \text{length } xs$

shows  $\text{length } (map2 \ f \ xs \ ys) = \text{length } xs$

using assms unfolding map2-def by auto

**lemma** listAll2-map2I[intro?]:

assumes  $\text{length } xs = \text{length } ys$

and  $\bigwedge i. i < \text{length } xs \implies R (xs!i) (f (xs!i) (ys!i))$

shows  $\text{list-all2 } R \ xs \ (map2 \ f \ xs \ ys)$



**apply**(*rule list-all2I*) **using** *assms unfolding set-zip map2-def* **by** *auto*

**lemma** *set-incl-pred*:

$A \leq B \iff (\forall a. A a \longrightarrow B a)$

**by** (*metis predicate1D predicate1I*)

**lemma** *set-incl-pred2*:

$A \leq B \iff (\forall a1 a2. A a1 a2 \longrightarrow B a1 a2)$

**by** (*metis predicate2I rev-predicate2D*)

**lemma** *set-incl-pred3*:

$A \leq B \iff (\forall a1 a2 a3. A a1 a2 a3 \longrightarrow B a1 a2 a3)$  (**is** -  $\iff$  ?*R*)

**proof** -

**have**  $A \leq B \iff (\forall a1. A a1 \leq B a1)$  **by** (*metis le-funD le-funI*)

**also have** ...  $\iff$  ?*R* **apply**(*rule iff-allI*)

**unfolding** *set-incl-pred2* ..

**finally show** ?*thesis* .

**qed**

**lemma** *set-incl-pred4*:

$A \leq B \iff (\forall a1 a2 a3 a4. A a1 a2 a3 a4 \longrightarrow B a1 a2 a3 a4)$  (**is** -  $\iff$  ?*R*)

**proof** -

**have**  $A \leq B \iff (\forall a1. A a1 \leq B a1)$  **by** (*metis le-funD le-funI*)

**also have** ...  $\iff$  ?*R* **apply**(*rule iff-allI*)

**unfolding** *set-incl-pred3* ..

**finally show** ?*thesis* .

**qed**

**lemma** *list-all-mono*:

**assumes**  $\phi \leq \chi$

**shows**  $\text{list-all } \phi \leq \text{list-all } \chi$

**using** *assms unfolding set-incl-pred list-all-iff* **by** *auto*

**lemma** *list-all2-mono*:

**assumes**  $\phi \leq \chi$

**shows**  $\text{list-all2 } \phi \leq \text{list-all2 } \chi$

**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*:  $|\text{UNIV}::\text{var set}| =_o \text{natLeq}$

**proof** -

**have**  $|\text{UNIV}::\text{var set}| =_o |\text{UNIV}::\text{nat set}|$

**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**  $0: (UNIV :: var\ set) \neq \{\}$  **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**  $|A| \leq_o |B|$   
**proof** –  
**have**  $|A| \leq_o\ natLeq$  **using** *A* **unfolding** *countable-card-le-natLeq* .  
**also have**  $natLeq \leq_o |B|$  **by** (*metis B infinite-iff-natLeq-ordLeq*)  
**finally show** *?thesis* .  
**qed**

**definition** *part12-pred*  $V\ V1\ V2 \equiv$   
 $V = fst\ V1\ V2 \cup snd\ V1\ V2 \wedge fst\ V1\ V2 \cap snd\ V1\ V2 = \{\} \wedge$   
*infinite (fst V1-V2)  $\wedge$  infinite (snd V1-V2)*

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

**lemma** *part12-pred*:  
**assumes** *infinite (V::'a set)* **shows**  $\exists\ V1\ V2.$  *part12-pred V V1-V2*  
**proof** –  
**obtain**  $f :: nat \Rightarrow 'a$  **where**  $f: inj\ f$  **and**  $r: range\ f \subseteq V$   
**using** *assms* **by** (*metis infinite-iff-countable-subset*)  
**let**  $?u = \lambda\ k::nat. 2 * k$  **let**  $?v = \lambda\ k::nat. Suc\ (2 * k)$   
**let**  $?A = ?u \text{ ' } UNIV$  **let**  $?B = ?v \text{ ' } UNIV$   
**have**  $0: inj\ ?u \wedge inj\ ?v$  **unfolding** *inj-on-def* **by** *auto*  
**hence**  $1: infinite\ ?A \wedge infinite\ ?B$  **using** *finite-imageD* **by** *auto*  
**let**  $?V1 = f \text{ ' } ?A$  **let**  $?V2 = V - ?V1$   
**have**  $V = ?V1 \cup ?V2 \wedge ?V1 \cap ?V2 = \{\}$  **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  $\subseteq$  ?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  $\implies$  part1 V  $\cup$  part2 V = V
using part12 unfolding part1-def part2-def part12-pred-def by auto

```

```

lemma part1-Int-part2: infinite V  $\implies$  part1 V  $\cap$  part2 V = {}
using part12 unfolding part1-def part2-def part12-pred-def by auto

```

```

lemma infinite-part1: infinite V  $\implies$  infinite (part1 V)
using part12 unfolding part1-def part12-pred-def by auto

```

```

lemma part1-su: infinite V  $\implies$  part1 V  $\subseteq$  V
using part1-Un-part2 by auto

```

```

lemma infinite-part2: infinite V  $\implies$  infinite (part2 V)
using part12 unfolding part2-def part12-pred-def by auto

```

```

lemma part2-su: infinite V  $\implies$  part2 V  $\subseteq$  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**  $\bigwedge x. \varphi (Var\ x)$

**and**  $\bigwedge f\ Tl. list\text{-all}\ \varphi\ Tl \implies \varphi (Fn\ f\ Tl)$

**shows**  $\varphi\ T$

**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*  $\cup$  *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*  $\Phi$  =  $\bigcup\ \{varsC\ c\ |\ c. c \in \Phi\}$

Substitution:

**fun** *subst* **where**

*subst*  $\pi$  (*Var* *x*) =  $\pi\ x$

|

*subst*  $\pi$  (*Fn* *f* *Tl*) = *Fn* *f* (*map* (*subst*  $\pi$ ) *Tl*)

**fun** *substA* **where**

$substA \ \pi \ (Eq \ T1 \ T2) = Eq \ (subst \ \pi \ T1) \ (subst \ \pi \ T2)$   
 $|$   
 $substA \ \pi \ (Pr \ p \ Tl) = Pr \ p \ (map \ (subst \ \pi) \ Tl)$

**fun** *substL* **where**  
 $substL \ \pi \ (Pos \ at) = Pos \ (substA \ \pi \ at)$   
 $|$   
 $substL \ \pi \ (Neg \ at) = Neg \ (substA \ \pi \ at)$

**definition**  $substC \ \pi \ c = map \ (substL \ \pi) \ c$

**definition**  $substPB \ \pi \ \Phi = \{substC \ \pi \ c \mid c. c \in \Phi\}$

**lemma** *subst-cong*:  
**assumes**  $\bigwedge x. x \in vars \ T \implies \pi1 \ x = \pi2 \ x$   
**shows**  $subst \ \pi1 \ T = subst \ \pi2 \ T$   
**using** *assms* **by** (*induct T, auto simp add: list-all-iff*)

**lemma** *substA-congA*:  
**assumes**  $\bigwedge x. x \in varsA \ at \implies \pi1 \ x = \pi2 \ x$   
**shows**  $substA \ \pi1 \ at = substA \ \pi2 \ at$   
**using** *assms subst-cong[of -  $\pi1 \ \pi2$ ]*  
**by** (*cases at, fastforce+*)

**lemma** *substL-congL*:  
**assumes**  $\bigwedge x. x \in varsL \ l \implies \pi1 \ x = \pi2 \ x$   
**shows**  $substL \ \pi1 \ l = substL \ \pi2 \ l$   
**using** *assms substA-congA[of -  $\pi1 \ \pi2$ ]* **by** (*cases l, auto*)

**lemma** *substC-congC*:  
**assumes**  $\bigwedge x. x \in varsC \ c \implies \pi1 \ x = \pi2 \ x$   
**shows**  $substC \ \pi1 \ c = substC \ \pi2 \ c$   
**using** *substL-congL[of -  $\pi1 \ \pi2$ ]* *assms* **unfolding** *substC-def varsC-def*  
**by** (*induct c, auto*)

**lemma** *substPB-congPB*:  
**assumes**  $\bigwedge x. x \in varsPB \ \Phi \implies \pi1 \ x = \pi2 \ x$   
**shows**  $substPB \ \pi1 \ \Phi = substPB \ \pi2 \ \Phi$   
**using** *substC-congC[of -  $\pi1 \ \pi2$ ]* *assms* **unfolding** *substPB-def varsPB-def*  
**by** *simp-all (metis (lifting) substC-congC)*

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

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

**lemma** *varsL-substL*:

$varsL (substL \pi l) = (\bigcup x \in varsL l. vars (\pi x))$

**using** *varsA-substA*[of  $\pi$ ] **by** (*cases l, auto*)

**lemma** *varsC-substC*:

$varsC (substC \pi c) = (\bigcup x \in varsC c. vars (\pi x))$

**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]*:  $varsPB (\Phi1 \cup \Phi2) = varsPB \Phi1 \cup varsPB \Phi2$

**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]*:

$varsPB (\Phi1 @@ \Phi2) \subseteq varsPB \Phi1 \cup varsPB \Phi2$

**by** (*unfold varsPB-def sappend-def, fastforce*)

**lemma** *varsPB-sappend[simp]*:

**assumes** *1*:  $\Phi1 \neq \{\}$  **and** *2*:  $\Phi2 \neq \{\}$

**shows**  $varsPB (\Phi1 @@ \Phi2) = varsPB \Phi1 \cup varsPB \Phi2$

**proof** *safe*

**fix** *x*

**{assume**  $x \in varsPB \Phi1$

**then obtain** *c1 c2* **where**  $x \in varsC c1$  **and**  $c1 \in \Phi1$  **and**  $c2 \in \Phi2$

**using** *2* **unfolding** *varsPB-def* **by** *auto*

**thus**  $x \in varsPB (\Phi1 @@ \Phi2)$  **unfolding** *sappend-def varsPB-def* **by** *fastforce*

**}**

**{assume**  $x \in varsPB \Phi2$

**then obtain** *c1 c2* **where**  $x \in varsC c2$  **and**  $c1 \in \Phi1$  **and**  $c2 \in \Phi2$

**using** *1* **unfolding** *varsPB-def* **by** *auto*

**thus**  $x \in varsPB (\Phi1 @@ \Phi2)$  **unfolding** *sappend-def varsPB-def* **by** *fastforce*

**}**

**qed**(*unfold varsPB-def sappend-def, fastforce*)

**lemma** *varsPB-substPB*:

$varsPB (substPB \pi \Phi) = (\bigcup x \in varsPB \Phi. vars (\pi x))$  (**is - = ?K**)

**proof** *safe*

**fix** *x* **assume**  $x \in varsPB (substPB \pi \Phi)$

**then obtain** *c* **where**  $c \in \Phi$  **and**  $x \in varsC (substC \pi c)$

**unfolding** *varsPB-def substPB-def* **by** *auto*

**thus**  $x \in ?K$  **unfolding** *varsC-substC varsPB-def* **by** *auto*

**next**

**fix** *x y* **assume**  $y \in varsPB \Phi$  **and**  $x: x \in vars (\pi y)$

**then obtain** *c* **where**  $c: c \in \Phi$  **and**  $y \in varsC c$  **unfolding** *varsPB-def* **by**

*auto*

hence  $x \in \text{vars}C$  ( $\text{subst}C \pi c$ ) **using**  $x$  **unfolding**  $\text{vars}C\text{-subst}C$  **by**  $\text{auto}$   
**thus**  $x \in \text{vars}PB$  ( $\text{subst}PB \pi \Phi$ ) **using**  $c$  **unfolding**  $\text{vars}PB\text{-def}$   $\text{subst}PB\text{-def}$  **by**  
 $\text{auto}$   
**qed**

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

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

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

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

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

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

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

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

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

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

**lemma**  $\text{finite-vars}$ :  $\text{finite} (\text{vars } T)$   
**by**( $\text{induct } T, \text{auto simp add: list-all-iff}$ )

**lemma**  $\text{finite-varsA}$ :  $\text{finite} (\text{varsA } at)$

```

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  $\Phi \implies$  finite (varsPB  $\Phi$ )
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  $\Rightarrow$  bool
and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf :: 'fsym  $\Rightarrow$  'tp
and parOf :: 'psym  $\Rightarrow$  '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  $\Rightarrow$  'tp)  $\Rightarrow$  bool where
tpOfV-pred f  $\equiv \forall \sigma$ . infinite (f -' { $\sigma$ })

```

```

definition tpOfV  $\equiv$  SOME f. tpOfV-pred f

```

```

lemma infinite-fst-vimage:
infinite ((fst :: 'a  $\times$  nat  $\Rightarrow$  '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*:  $\exists f. \text{tpOfV-pred } f$

**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 × nat) set)
  have U2: U2  $\equiv$  Ut × Unt unfolding Ut-def Unt-def U2-def UNIV-Times-UNIV
  .
  have |U2| =o |Unt × Ut| unfolding U2 by (metis card-of-Times-commute)
  also have |Unt × 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 - ' {σ})*  
**using** *tpOfV-pred-tpOfV* **unfolding** *tpOfV-pred-def* **by** *auto*

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

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

**lemma** *tinfinite-singl[simp]*:

**assumes** *tinfinite V* **shows** *tinfinite (V - {x})*

**unfolding** *tinfinite-def* **proof** *safe*

**fix**  $\sigma$  **assume**  $0: \text{finite } ((V - \{x\}) \cap \text{tpOfV} - ' \{\sigma\})$

**have** *finite*  $((V \cap \text{tpOfV} - ' \{\sigma\}) - \{x\})$  **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*  $\cup$  *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*  $\cap$  *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*  $\subseteq$  *V*  
**using** *assms* **unfolding** *tinfinite-def* *tpart1-def*  
**using** *part1-su* **by** (*auto intro: UN-least*)

**lemma** *tpart1-in*:  
**assumes** *tinfinite V* **and**  $x \in$  *tpart1 V* **shows**  $x \in$  *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**  $\sigma$  **assume**  
*finite* ( $(\bigcup \sigma'. \text{part1 } (V \cap \text{tpOfV } -' \{\sigma'\})) \cap \text{tpOfV } -' \{\sigma\}$ ) (**is** *finite* ?*A*)  
**moreover** **have** ?*A* = *part1* ( $V \cap \text{tpOfV } -' \{\sigma\}$ )  
**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**  $\sigma$  **assume**  
*finite* ( $(\bigcup \sigma'. \text{part2 } (V \cap \text{tpOfV } -' \{\sigma'\})) \cap \text{tpOfV } -' \{\sigma\}$ ) (**is** *finite* ?*A*)  
**moreover** **have** ?*A* = *part2* ( $V \cap \text{tpOfV } -' \{\sigma\}$ )  
**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*  $V \subseteq V$   
**using** *assms unfolding tinfinite-def tpart2-def*  
**using** *part2-su* **by** (*auto intro: UN-least*)

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

Typed-pick: picking a variable of a given type

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

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

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

**lemma** *tpick-in[simp]*: *tinfinite*  $V \implies \text{tpick } \sigma V \in V$   
**and** *tpOfV-tpick[simp]*: *tinfinite*  $V \implies \text{tpOfV } (\text{tpick } \sigma V) = \sigma$   
**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*  $(\sigma \# \sigma l) =$   
 $(\text{let } xl = \text{getVars } \sigma l \text{ in } (\text{tpick } \sigma (UNIV - \text{set } xl)) \# xl)$

**lemma** *distinct-getVars*: *distinct*  $(\text{getVars } \sigma l)$   
**using** *tpick-in[OF finite-tinfinite]* **by** (*induct*  $\sigma l$ , *auto simp: Let-def*)

**lemma** *length-getVars[simp]*: *length*  $(\text{getVars } \sigma l) = \text{length } \sigma l$   
**by**(*induct*  $\sigma l$ , *auto simp: Let-def*)

**lemma** *map-tpOfV-getVars[simp]*: *map* *tpOfV*  $(\text{getVars } \sigma l) = \sigma l$   
**using** *tpOfV-tpick[OF finite-tinfinite]* **by** (*induct*  $\sigma l$ , *auto simp: Let-def*)

```

lemma tpOfV-getVars-nth[simp]:
assumes  $i < \text{length } \sigma l$  shows  $\text{tpOfV } (\text{getVars } \sigma l ! i) = \sigma l ! i$ 
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
 $\text{tpOf } (\text{Var } x) = \text{tpOfV } x$ 
|
 $\text{tpOf } (\text{Fn } f \text{ Tl}) = \text{resOf } f$ 

```

```

fun wt where
 $\text{wt } (\text{Var } x) \longleftrightarrow \text{True}$ 
|
 $\text{wt } (\text{Fn } f \text{ Tl}) \longleftrightarrow$ 
 $\text{wtFsym } f \wedge \text{list-all } \text{wt } \text{Tl} \wedge \text{arOf } f = \text{map } \text{tpOf } \text{Tl}$ 

```

```

fun wtA where
 $\text{wtA } (\text{Eq } T1 \text{ T2}) \longleftrightarrow \text{wt } T1 \wedge \text{wt } T2 \wedge \text{tpOf } T1 = \text{tpOf } T2$ 
|
 $\text{wtA } (\text{Pr } p \text{ Tl}) \longleftrightarrow$ 
 $\text{wtPsym } p \wedge \text{list-all } \text{wt } \text{Tl} \wedge \text{parOf } p = \text{map } \text{tpOf } \text{Tl}$ 

```

```

fun wtL where
 $\text{wtL } (\text{Pos } a) \longleftrightarrow \text{wtA } a$ 
|
 $\text{wtL } (\text{Neg } a) \longleftrightarrow \text{wtA } a$ 

```

```

definition wtC  $\equiv \text{list-all } \text{wtL}$ 

```

```

lemma wtC-append[simp]:  $\text{wtC } (c1 @ c2) \longleftrightarrow \text{wtC } c1 \wedge \text{wtC } c2$ 

```

**unfolding** *wtC-def* **by** *simp*

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

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

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

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

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

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

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

**lemma** *wt-tpOf-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-all-iff*)

**lemmas** *wt-subst[simp]* = *wt-tpOf-subst*[*THEN* *conjunct1*]  
**lemmas** *tpOf-subst[simp]* = *wt-tpOf-subst*[*THEN* *conjunct2*]

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

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

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

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

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

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

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

**end**

### 3.3 Structures

We split a structure 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*:  $\llbracket \text{wtFsym } f; \text{list-all2 } \text{intT } (\text{arOf } f) \text{ al} \rrbracket \implies \text{intT } (\text{resOf } f) (\text{intF } f \text{ al})$   
**and**  
*dummy*: *intP* = *intP*  
**begin**

Well-typed environment:

**definition** *wtE*  $\xi \equiv \forall x. \text{intT } (\text{tpOfV } x) (\xi x)$

**lemma** *wtTE-intT[simp]*:  $\text{wtE } \xi \implies \text{intT } (\text{tpOfV } x) (\xi x)$   
**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**

$pickE (xl::var list) al \equiv$

$SOME \xi. wtE \xi \wedge (\forall i < length xl. \xi (xl!i) = al!i)$

**lemma**  $ex-pickE$ :

**assumes**  $length xl = length al$

**and**  $distinct xl$  **and**  $\bigwedge i. i < length xl \implies intT (tpOfV (xl!i)) (al!i)$

**shows**  $\exists \xi. wtE \xi \wedge (\forall i < length xl. \xi (xl!i) = al!i)$

**using**  $assms$

**proof**( $induct$  rule:  $list-induct2$ )

**case**  $Nil$  **show**  $?case$  **apply**(rule  $exI[of - \lambda x. pickT (tpOfV x)]$ )

**unfolding**  $wtE-def$  **by**  $auto$

**next**

**case** ( $Cons x xl a al$ )

**then obtain**  $\xi$  **where**  $1: wtE \xi$  **and**  $2: \forall i < length xl. \xi (xl!i) = al!i$  **by**  $force$

**define**  $\xi'$  **where**  $\xi' x' = (if x = x' then a else \xi x')$  **for**  $x'$

**show**  $?case$

**proof**(rule  $exI[of - \xi']$ ,  $unfold wtE-def$ ,  $safe$ )

**fix**  $x'$  **show**  $intT (tpOfV x') (\xi' x')$

**using**  $1 Cons.premis(2)[of 0]$  **unfolding**  $\xi'-def$  **by**  $auto$

**next**

**fix**  $i$  **assume**  $i: i < length (x \# xl)$

**thus**  $\xi' ((x \# xl) ! i) = (a \# al) ! i$

**proof**( $cases i$ )

**case** ( $Suc j$ ) **hence**  $j: j < length xl$  **using**  $i$  **by**  $auto$

**have**  $\neg x = (x \# xl) ! i$  **using**  $Suc i Cons.premis(1)$  **by**  $auto$

**thus**  $?thesis$  **using**  $Suc$  **using**  $j Cons.premis(1) Cons.hyps 2$  **unfolding**  $\xi'-def$

**by**  $auto$

**qed**( $insert Cons.premis(1) Cons.hyps 2$ ,  $unfold \xi'-def$ ,  $simp$ )

**qed**

**qed**

**lemma**  $wtE-pickE-pickE$ :

**assumes**  $length xl = length al$

**and**  $distinct xl$  **and**  $\bigwedge i. i < length xl \implies intT (tpOfV (xl!i)) (al!i)$

**shows**  $wtE (pickE xl al) \wedge (\forall i. i < length xl \longrightarrow pickE xl al (xl!i) = al!i)$

**proof**–

**let**  $?phi = \lambda \xi. wtE \xi \wedge (\forall i < length xl. \xi (xl!i) = al!i)$

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

**definition**  $\text{pickAnyE} \equiv \text{pickE } [] \ []$

**lemma** *wtE-pickAnyE*[simp]:  $\text{wtE } \text{pickAnyE}$   
**unfolding** *pickAnyE-def* **by** (*rule wtE-pickE*) *auto*

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

**fun** *satA* **where**  
 $\text{satA } \xi \ (Eq \ T1 \ T2) \longleftrightarrow \text{int } \xi \ T1 = \text{int } \xi \ T2$   
 $|\$   
 $\text{satA } \xi \ (Pr \ p \ Tl) \longleftrightarrow \text{intP } p \ (\text{map } (\text{int } \xi) \ Tl)$

**fun** *satL* **where**  
 $\text{satL } \xi \ (Pos \ a) \longleftrightarrow \text{satA } \xi \ a$   
 $|\$   
 $\text{satL } \xi \ (Neg \ a) \longleftrightarrow \neg \ \text{satA } \xi \ a$

**definition**  $\text{satC } \xi \equiv \text{list-ex } (\text{satL } \xi)$

**lemma** *satC-append*[simp]:  $\text{satC } \xi \ (c1 \ @ \ c2) \longleftrightarrow \text{satC } \xi \ c1 \ \vee \ \text{satC } \xi \ c2$   
**unfolding** *satC-def* **by** *auto*

**lemma** *satC-iff-set*:  $\text{satC } \xi \ c \longleftrightarrow (\exists l \in \text{set } c. \text{satL } \xi \ l)$   
**unfolding** *satC-def Bex-set[symmetric]* **..**

**definition**  $\text{satPB } \xi \ \Phi \equiv \forall c \in \Phi. \text{satC } \xi \ c$

**lemma** *satPB-Un*[simp]:  $\text{satPB } \xi \ (\Phi1 \cup \Phi2) \longleftrightarrow \text{satPB } \xi \ \Phi1 \ \wedge \ \text{satPB } \xi \ \Phi2$   
**unfolding** *satPB-def* **by** *auto*

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



**lemma** *satPB-sappend*[simp]:  $\text{satPB } \xi (\Phi 1 \text{ @@ } \Phi 2) \longleftrightarrow \text{satPB } \xi \Phi 1 \vee \text{satPB } \xi \Phi 2$

**unfolding** *satPB-def sappend-def* **by** (*fastforce simp: satC-append*)

**definition** *SAT*  $\Phi \equiv \forall \xi. \text{wtE } \xi \longrightarrow \text{satPB } \xi \Phi$

**lemma** *SAT-UN*[simp]:  $\text{SAT } (\bigcup i \in I. \Phi i) \longleftrightarrow (\forall i \in I. \text{SAT } (\Phi i))$

**unfolding** *SAT-def* **by** *auto*

Soundness of typing w.r.t. interpretation:

**lemma** *wt-int*:

**assumes** *wtE*:  $\text{wtE } \xi$  **and** *wt*:  $\text{wt } T$

**shows**  $\text{intT } (\text{tpOf } T) (\text{int } \xi T)$

**using** *wt* **apply**(*induct T*) **using** *wtE*

**by** (*auto intro!: intF simp add: list-all2-map-map*)

**lemma** *int-cong*:

**assumes**  $\bigwedge x. x \in \text{vars } T \implies \xi 1 x = \xi 2 x$

**shows**  $\text{int } \xi 1 T = \text{int } \xi 2 T$

**using** *assms* **proof**(*induct T*)

**case** (*Fn f Tl*)

**hence**  $1: \text{map } (\text{int } \xi 1) Tl = \text{map } (\text{int } \xi 2) Tl$

**unfolding** *list-all-iff* **by** (*auto intro: map-ext*)

**show** *?case* **apply** *simp* **by** (*metis 1*)

**qed** *auto*

**lemma** *satA-cong*:

**assumes**  $\bigwedge x. x \in \text{varsA } at \implies \xi 1 x = \xi 2 x$

**shows**  $\text{satA } \xi 1 at \longleftrightarrow \text{satA } \xi 2 at$

**using** *assms int-cong[of -  $\xi 1 \xi 2$ ]*

**apply**(*cases at*) **apply**(*fastforce intro!: int-cong[of -  $\xi 1 \xi 2$ ]*)

**apply** *simp* **by** (*metis (hide-lams, mono-tags) map-eq-conv*)

**lemma** *satL-cong*:

**assumes**  $\bigwedge x. x \in \text{varsL } l \implies \xi 1 x = \xi 2 x$

**shows**  $\text{satL } \xi 1 l \longleftrightarrow \text{satL } \xi 2 l$

**using** *assms satA-cong[of -  $\xi 1 \xi 2$ ]* **by** (*cases l, auto*)

**lemma** *satC-cong*:

**assumes**  $\bigwedge x. x \in \text{varsC } c \implies \xi 1 x = \xi 2 x$

**shows**  $\text{satC } \xi 1 c \longleftrightarrow \text{satC } \xi 2 c$

**using** *assms satL-cong[of -  $\xi 1 \xi 2$ ]* **unfolding** *satC-def varsC-def*

**apply** (*induct c*) **by** (*fastforce intro!: satL-cong[of -  $\xi 1 \xi 2$ ]*)**+**

**lemma** *satPB-cong*:

**assumes**  $\bigwedge x. x \in \text{varsPB } \Phi \implies \xi 1 x = \xi 2 x$

**shows**  $\text{satPB } \xi 1 \Phi \longleftrightarrow \text{satPB } \xi 2 \Phi$

**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*:  
 $int \xi o subst \varrho = int (int \xi o \varrho)$   
**apply**(*rule ext*) **apply**(*subst comp-def*) **unfolding** *int-o[symmetric]* ..

**lemma** *satA-o*:  
 $satA (int \xi o \varrho) at = satA \xi (substA \varrho at)$   
**by** (*cases at, simp-all add: int-o-subst int-o[of \xi \varrho]*)

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

**lemma** *satA-o-subst*:  
 $satA \xi o substA \varrho = satA (int \xi o \varrho)$   
**apply**(*rule ext*) **apply**(*subst comp-def*) **unfolding** *satA-o[symmetric]* ..

**lemma** *satL-o*:  
 $satL (int \xi o \varrho) l = satL \xi (substL \varrho l)$   
**using** *satA-o[of \xi \varrho]* **by** (*cases l, simp-all*)

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

**lemma** *satL-o-subst*:  
 $satL \xi o substL \varrho = satL (int \xi o \varrho)$   
**apply**(*rule ext*) **apply**(*subst comp-def*) **unfolding** *satL-o[symmetric]* ..

**lemma** *satC-o*:  
 $satC (int \xi o \varrho) c = satC \xi (substC \varrho c)$   
**using** *satL-o[of \xi \varrho]* **unfolding** *satC-def substC-def* **by** (*induct c, auto*)

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

**lemma** *satC-o-subst*:  
 $satC \xi o substC \varrho = satC (int \xi o \varrho)$   
**apply**(*rule ext*) **apply**(*subst comp-def*) **unfolding** *satC-o[symmetric]* ..

**lemma** *satPB-o*:  
 $satPB (int \xi o \varrho) \Phi = satPB \xi (substPB \varrho \Phi)$   
**using** *satC-o[of \xi \varrho]* **unfolding** *satPB-def substPB-def* **by** *auto*

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

**lemma** *satPB-o-subst*:  
 $satPB \xi o substPB \varrho = satPB (int \xi o \varrho)$

**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 o ρ) x)* **apply**(*subst 0*) **unfolding** *comp-def*

**apply**(*rule wt-int[OF 1]*) **using** *2* **by** *auto*

**qed**

**definition** *compE ρ ξ x*  $\equiv$  *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* =  $\bigcup$  *set (map fsymsL c)*

**definition** *fsymsPB Φ* =  $\bigcup$  {*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**  $0$ :  $\bigwedge f. f \in fsyms\ T \implies intF1\ f = intF2\ f$   
**shows**  $Struct.int\ intF1\ \xi\ T = Struct.int\ intF2\ \xi\ T$   
**using**  $0$  **proof**( $induct\ T$ )  
    **case** ( $Fn\ f\ Tl$ )  
    **hence**  $1$ :  $map\ (Struct.int\ intF1\ \xi)\ Tl = map\ (Struct.int\ intF2\ \xi)\ Tl$   
    **unfolding**  $list-all-iff\ map-ext$  **by**  $auto$   
    **show**  $?case$   
    **using**  $Fn\ Struct.int.simps[OF\ S1, of\ \xi]\ Struct.int.simps[OF\ S2, of\ \xi]$  **apply**  
     $simp$   
    **using**  $1$  **by**  $metis$   
**qed** ( $auto\ simp: Struct.int.simps[OF\ S1, of\ \xi]\ Struct.int.simps[OF\ S2, of\ \xi]$ )

**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**  $0$ :  $\bigwedge f. f \in fsymsA\ at \implies intF1\ f = intF2\ f$   
**shows**  $Struct.satA\ intF1\ intP\ \xi\ at \iff Struct.satA\ intF2\ intP\ \xi\ at$   
**using**  $0$   $fsyms-int-cong[OF\ S1\ S2]$   
**apply**( $cases\ at$ )  
**apply**( $fastforce\ intro!: fsyms-int-cong[OF\ S1\ S2, of\ -\ \xi]$   
     $simp: Struct.satA.simps[OF\ S1, of\ \xi]\ Struct.satA.simps[OF\ S2, of\ \xi]$ )  
**apply** ( $simp\ add: Struct.satA.simps[OF\ S1, of\ \xi]\ Struct.satA.simps[OF\ S2, of\ \xi]$ )  
**by** ( $metis\ (hide-lams, mono-tags)\ map-eq-conv$ )

**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**  $0$ :  $\bigwedge f. f \in fsymsL\ l \implies intF1\ f = intF2\ f$   
**shows**  $Struct.satL\ intF1\ intP\ \xi\ l \iff Struct.satL\ intF2\ intP\ \xi\ l$   
**using**  $0$   $fsyms-satA-cong[OF\ S1\ S2]$   
**by** ( $cases\ l, auto\ simp: Struct.satL.simps[OF\ S1, of\ \xi]\ Struct.satL.simps[OF\ S2, of\ \xi]$ )

**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**  $0$ :  $\bigwedge f. f \in fsymsC\ c \implies intF1\ f = intF2\ f$   
**shows**  $Struct.satC\ intF1\ intP\ \xi\ c \iff Struct.satC\ intF2\ intP\ \xi\ c$   
**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**  $0$ :  $\bigwedge f. f \in fsymsPB\ \Phi \implies intF1\ f = intF2\ f$

**shows**  $Struct.satPB\ intF1\ intP\ \xi\ \Phi \longleftrightarrow Struct.satPB\ intF2\ intP\ \xi\ \Phi$   
**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 ( $\Phi1 \cup \Phi2$ ) = fsymsPB  $\Phi1 \cup$  fsymsPB  $\Phi2$*   
**unfolding** *fsymsPB-def by auto*

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

**lemma** *fsymsPB-sappend-incl[simp]:*  
*fsymsPB ( $\Phi1 @@ \Phi2$ )  $\subseteq$  fsymsPB  $\Phi1 \cup$  fsymsPB  $\Phi2$*   
**by** (*unfold fsymsPB-def sappend-def, fastforce*)

**lemma** *fsymsPB-sappend[simp]:*  
**assumes** *1:  $\Phi1 \neq \{\}$  and 2:  $\Phi2 \neq \{\}$*   
**shows** *fsymsPB ( $\Phi1 @@ \Phi2$ ) = fsymsPB  $\Phi1 \cup$  fsymsPB  $\Phi2$*   
**proof** *safe*  
**fix** *x*  
**{** **assume** *x  $\in$  fsymsPB  $\Phi1$*   
**then obtain** *c1 c2 where x  $\in$  fsymsC  $c1$  and  $c1 \in \Phi1$  and  $c2 \in \Phi2$*   
**using** *2 unfolding fsymsPB-def by auto*  
**thus** *x  $\in$  fsymsPB ( $\Phi1 @@ \Phi2$ )* **unfolding** *sappend-def fsymsPB-def by*  
*fastforce*  
**}**  
**{** **assume** *x  $\in$  fsymsPB  $\Phi2$*   
**then obtain** *c1 c2 where x  $\in$  fsymsC  $c2$  and  $c1 \in \Phi1$  and  $c2 \in \Phi2$*   
**using** *1 unfolding fsymsPB-def by auto*  
**thus** *x  $\in$  fsymsPB ( $\Phi1 @@ \Phi2$ )* **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**  $\bigwedge al. list-all2\ intT\ (arOf\ ef)\ al \implies intT\ (resOf\ ef)\ (EF\ al)$   
**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  $\Rightarrow$  'tp list*

```

and resOf :: 'fsym  $\Rightarrow$  'tp
and parOf :: 'psym  $\Rightarrow$  'tp list
+
fixes  $\Phi$  :: ('fsym, 'psym) prob
assumes wt- $\Phi$ : wtPB  $\Phi$ 

```

### 3.5 Models of a problem

Model of a problem:

```

locale Model = Problem + Struct +
assumes SAT: SAT  $\Phi$ 
begin
lemma sat- $\Phi$ : wtE  $\xi \Longrightarrow$  satPB  $\xi$   $\Phi$ 
using SAT unfolding SAT-def by auto
end

```

**end**

```

theory CM
imports M
begin

```

```

locale Tstruct = M.Tstruct intT
for intT :: 'tp  $\Rightarrow$  univ  $\Rightarrow$  bool

```

```

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

```

```

locale Model = M.Model wtFsym wtPsym arOf resOf parOf  $\Phi$  intT intF intP
for wtFsym and wtPsym
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf and parOf :: 'psym  $\Rightarrow$  'tp list and  $\Phi$ 
and intT :: 'tp  $\Rightarrow$  univ  $\Rightarrow$  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*  $\sigma \equiv \forall d. \text{intT } \sigma d$

**locale** *FullStruct* = *F?* : *Struct* +

**assumes** *full*: *full*  $\sigma$

**begin**

**lemma** *full2[simp]*: *intT*  $\sigma d$

**using** *full* **unfolding** *full-def* **by** *simp*

**lemma** *full-True*: *intT* =  $(\lambda \sigma D. \text{True})$

**apply**(*intro ext*) **by** *auto*

**end**

**locale** *FullModel* =

*F?* : *Model* *wtFsym* *wtPsym* *arOf* *resOf* *parOf*  $\Phi$  *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**  $\Phi$  **and** *intT* **and** *intF* **and** *intP*

An infinite structure is one with all carriers infinite:

**locale** *InfStruct* = *I?* : *Struct* +

**assumes** *inf*: *infinite*  $\{a. \text{intT } \sigma a\}$

**locale** *InfModel* =

*I?* : *Model* *wtFsym* *wtPsym* *arOf* *resOf* *parOf*  $\Phi$  *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**  $\Phi$  **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*  $\Phi$

**abbreviation** *FFullModel*  $\equiv$  *FullModel* *wtFsym* *wtPsym* *arOf* *resOf* *parOf*  $\Phi$

**abbreviation** *IInfModel*  $\equiv$  *InfModel* *wtFsym* *wtPsym* *arOf* *resOf* *parOf*  $\Phi$

**end**

Problem that deduces some infiniteness constraints:

```

locale ProblemIk = Ik? : Problem wtFsym wtPsym arOf resOf parOf  $\Phi$ 
for wtFsym :: 'fsym  $\Rightarrow$  bool and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf and parOf and  $\Phi$ 
+
fixes infTp :: 'tp  $\Rightarrow$  bool

assumes infTp:
 $\bigwedge \sigma \text{ intT intF intP } (a::\text{univ}). \llbracket \text{infTp } \sigma; \text{MModel intT intF intP} \rrbracket \Longrightarrow \text{infinite } \{a. \text{intT } \sigma \ a\}$ 

locale ModelIk =
Ik? : ProblemIk wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp +
Ik? : Model wtFsym wtPsym arOf resOf parOf  $\Phi$  intT intF intP
for wtFsym :: 'fsym  $\Rightarrow$  bool and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf and parOf and  $\Phi$  and infTp and intT and intF and intP
begin
lemma infTp-infinite[simp]: infTp  $\sigma \Longrightarrow \text{infinite } \{a. \text{intT } \sigma \ a\}$ 
apply(rule ProblemIk.infTp[of wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp])
apply unfold-locales by simp
end

```

## 4.2 Monotonicity

**context** *Problem* **begin**

**definition**

```

monot  $\equiv$ 
 $(\exists \text{ intT intF intP. MModel intT intF intP})$ 
 $\longrightarrow$ 
 $(\exists \text{ intTI intFI intPI. IInfModel intTI intFI intPI})$ 
end

```

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

```

locale MonotProblemIk =
MonotProblem wtFsym wtPsym arOf resOf parOf  $\Phi$  +
ProblemIk wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp
for wtFsym :: 'fsym  $\Rightarrow$  bool and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list and resOf and parOf and  $\Phi$  and infTp

```

**context** *MonotProblem*  
**begin**

**definition** *MI-pred* *K*  $\equiv$  *IInfModel (fst3 K) (snd3 K) (trd3 K)*

**definition** *MI*  $\equiv$  *SOME K. MI-pred K*



**lemma** *MI-pred*:  
**assumes** *MModel intT intF intP*  
**shows**  $\exists K. MI\text{-}pred\ K$   
**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*  $\equiv$  *fst3 MI*  
**definition** *intFI*  $\equiv$  *snd3 MI*  
**definition** *intPI*  $\equiv$  *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*  $\sigma \equiv$  *SOME F. bij-betw F {a::univ. intT  $\sigma$  a} (UNIV::univ set)*  
**definition** *fromFull*  $\sigma \equiv$  *inv-into {a::univ. intT  $\sigma$  a} (toFull  $\sigma)$*

**definition**  $\text{intTF } \sigma \ a \equiv \text{True}$   
**definition**  $\text{intFF } f \ al \equiv \text{toFull } (\text{resOf } f) (\text{intF } f (\text{map2 } \text{fromFull } (\text{arOf } f) \ al))$   
**definition**  $\text{intPF } p \ al \equiv \text{intP } p (\text{map2 } \text{fromFull } (\text{parOf } p) \ al)$

**lemma**  $\text{intTF}$ :  $\text{intTF } \sigma \ a$   
**unfolding**  $\text{intTF-def}$  **by**  $\text{auto}$

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

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

**lemma**  $\text{toFull-fromFull[simp]}$ :  $\text{toFull } \sigma \ (\text{fromFull } \sigma \ a) = a$   
**by** ( $\text{metis UNIV-I bij-betw-inv-into-right fromFull-def toFull}$ )

**lemma**  $\text{fromFull-toFull[simp]}$ :  $\text{intT } \sigma \ a \implies \text{fromFull } \sigma \ (\text{toFull } \sigma \ a) = a$   
**by** ( $\text{metis CollectI bij-betw-inv-into-left toFull fromFull-def}$ )

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

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

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

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

**lemma**  $\text{Tstruct}$ :  $\text{Tstruct intTF}$   
**apply** ( $\text{unfold-locales}$ ) **unfolding**  $\text{intTF-def}$  **by**  $\text{simp}$

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

**end**

**sublocale**  $\text{InfModel} < \text{FullStruct}$   
**where**  $\text{intT} = \text{intTF}$  **and**  $\text{intF} = \text{intFF}$  **and**  $\text{intP} = \text{intPF}$   
**using**  $\text{FullStruct}$  .

**context** *InfModel* **begin**

**definition**  $kE \xi \equiv \lambda x. \text{fromFull } (tpOfV \ x) \ (\xi \ x)$

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

**assumes**  $\xi: I.wtE \ (kE \ \xi)$  **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**  $0: \text{map } (I.int \ (kE \ \xi)) \ Tl =$

$\text{map2 fromFull } (arOf \ f) \ (\text{map } (F.int \ \xi) \ Tl)$

(**is**  $\text{map } ?F \ Tl = \text{map2 fromFull } (arOf \ f) \ (\text{map } ?H \ Tl)$ )

**is**  $?L = ?R$ )

**proof**( $rule \ nth\text{-equalityI}, \ safe$ )

**have**  $l: \text{length } Tl = \text{length } (arOf \ f)$  **using**  $Fn$  **by**  $simp$

**thus**  $\text{length } ?L = \text{length } ?R$  **by**  $simp$

**fix**  $i$  **assume**  $i < \text{length } ?L$  **hence**  $i: i < \text{length } Tl$  **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**  $list\text{-all-length}$  **by**

$auto$

**have**  $wt: wt \ (Tl!i)$  **using**  $Fn \ i$  **by** ( $auto \ simp: list\text{-all-iff}$ )

**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** ( $auto \ simp: list\text{-all-iff} \ kE\text{-def}$ )

**ultimately** **have**  $?F \ (Tl!i) = fromFull \ (arOf \ f!i) \ (?H \ (Tl!i))$

**using**  $toFull\text{-iff-fromFull}$  **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**  $\xi: I.wtE \ (kE \ \xi)$  **and**  $T: wt \ T$

**shows**  $I.int \ (kE \ \xi) \ T = fromFull \ (tpOf \ T) \ (F.int \ \xi \ T)$

**using**  $kE\text{-int-toFull}[OF \ assms]$

**unfolding**  $toFull\text{-iff-fromFull}[OF \ I.wt\text{-int}[OF \ \xi \ T]]$  .

**lemma** *map-kE-int*[simp]:  
**assumes**  $\xi: I.wtE (kE \xi)$  **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, safe*)  
**apply** (*metis (lifting) length-map length-map2*)  
**by** (*metis T \xi kE-int length-map list-all-length nth-map nth-map2*)

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

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

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

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

**lemma** *F-SAT*:  $F.SAT \Phi$   
**unfolding** *F.SAT-def* **using** *kE-satPB kE-wtE* **by** *auto*

**lemma** *FullModel*: *FullModel wtFsym wtPsym arOf resOf parOf \Phi 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**  $intTF \equiv InfModel.intTF$

**definition**  $intFF \equiv InfModel.intFF\ arOf\ resOf\ intTI\ intFI$

**definition**  $intPF \equiv InfModel.intPF\ parOf\ intTI\ intPI$

Strengthening of the infiniteness condition for monotonicity, replacing in-

fininetess 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  $\cup$  nvT T2  
|  
nvA (Pr p Tl) = {}
```

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

```
definition nvC c  $\equiv$   $\bigcup$  set (map nvL c)
```

```
definition nvPB  $\equiv$   $\bigcup$  c  $\in$   $\Phi$ . nvC c
```

```
lemma nvT-vars[simp]: x  $\in$  nvT T  $\implies$  x  $\in$  vars T  
by (induct T) (auto split: if-splits)
```

```
lemma nvA-varsA[simp]: x  $\in$  nvA at  $\implies$  x  $\in$  varsA at  
by (cases at, auto)
```

**lemma** *nvL-varsL[simp]*:  $x \in nvL\ l \implies x \in varsL\ l$   
**by** (*cases l, auto*)

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

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

## 5.2 The calculus

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

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

**end**

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

**locale** *ModelIkMcalc* =  
*ModelIk wtFsym wtPsym arOf resOf parOf*  $\Phi$  *infTp intT intF intP* +  
*ProblemIkMcalc wtFsym wtPsym arOf resOf parOf*  $\Phi$  *infTp*  
**for** *wtFsym* :: '*fsym*  $\Rightarrow$  bool **and** *wtPsym* :: '*psym*  $\Rightarrow$  bool  
**and** *arOf* :: '*fsym*  $\Rightarrow$  'tp list  
**and** *resOf* **and** *parOf* **and**  $\Phi$  **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*  $\sigma\ a \implies$  *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**  $I-intT \sigma (a::univ) \equiv infTp \sigma \longrightarrow intT \sigma a$

**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]*:  $\neg infTp \sigma \Longrightarrow I-intT \sigma a$  **unfolding** *I-intT-def*  
**by** *simp*

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

**lemma** *NE-I-intT*:  $NE (I-intT \sigma)$   
**using** *NE-intT* **by** (*cases infTp \sigma, 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*:  $infinite \{a. I-intT \sigma a\}$   
**by** (*cases infTp \sigma, 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**  $transE \xi \equiv \lambda x. proj (tpOfV x) (\xi x)$

**lemma** *transE[simp]*:  $transE \xi x = proj (tpOfV x) (\xi x)$

**unfolding** *transE-def* **by** *simp*

**lemma** *wtE-transE[simp]*:  $I.wtE \xi \implies Ik.wtE (transE \xi)$   
**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**  $\xi$ :  $I.wtE \xi$

**and** *snv*:  $\bigwedge \sigma. infTp \sigma \vee (\forall x \in nvT T. tpOfV x \neq \sigma)$

**shows**  $Ik-intT (tpOf T) (I.int \xi T)$

**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  $\xi$ ]*

**by** (*metis I.wt-int I-intT-def  $\xi$  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** *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**  $Ik.int (transE \xi) T = proj (tpOf T) (I.int \xi T)$

**using** *wt* **proof** (*induct T*)

**case** (*Fn f Tl*)

**have**  $0$ :  $Ik-intT (resOf f) (I-intF f (map (int \xi) Tl))$  (*is Ik-intT ? $\sigma$  ?a*)

**unfolding** *I-intF-def* **apply**(*rule Ik.intF*)

**using** *Fn* **unfolding** *list-all2-length list-all-iff* **by** *auto*

**have**  $1$ :  $proj ?\sigma ?a = ?a$  **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, safe*)

**have**  $l$ [*simp*]:  $length (arOf f) = length Tl$  **using** *Fn* **by** *simp*

**fix**  $i$  **assume**  $i < length (map (Ik.int (transE \xi)) Tl)$

**hence**  $i$ [*simp*]:  $i < length Tl$  **by** *simp*

**have**  $0$ :  $arOf f ! i = tpOf (Tl ! i)$  **using** *Fn* **by** *simp*

**have** [*simp*]:  $Ik.int (transE \xi) (Tl ! i) = proj (arOf f ! i) (I.int \xi (Tl ! i))$



**unfolding**  $0$  **using**  $F_n$  **by** (*auto simp: list-all-length transE-def*)  
**show**  $\text{map } (Ik.int \text{ (transE } \xi)) \text{ } Tl ! i =$   
 $\text{map2 proj (arOf } f) \text{ (map (I.int } \xi) \text{ } Tl) ! i$   
**using**  $F_n$  **unfolding** *list-all-length by simp*  
**qed**(*insert  $F_n$ , simp*)  
**qed** *simp*

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

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

**lemma** *intP-transE[simp]*:  
**assumes**  $wt: \text{list-all } wt \text{ } Tl$  **and**  $p: wtPsym \text{ } p$  **and**  $ar: parOf \text{ } p = \text{map } tpOf \text{ } Tl$   
**shows**  $Ik-intP \text{ } p \text{ (map (Ik.int (transE } \xi)) \text{ } Tl) = I-intP \text{ } p \text{ (map (I.int } \xi) \text{ } Tl)$   
**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**  $wtA: Ik.wtA \text{ } at$  **and**  $\xi: I.wtE \xi$   
**and**  $pA: \bigwedge \sigma. \text{infTp } \sigma \vee (\forall x \in nvA \text{ } at. \text{tpOfV } x \neq \sigma)$   
**shows**  $Ik.satA \text{ (transE } \xi) \text{ } at \longleftrightarrow I.satA \xi \text{ } at$   
**using** *assms* **apply** (*cases at, simp-all add: ball-Un*) **by** (*metis int-transE-snv*)

**lemma** *satA-transE[simp]*:  
**assumes**  $wtA: Ik.wtA \text{ } at$  **and**  $I.satA \xi \text{ } at$   
**shows**  $Ik.satA \text{ (transE } \xi) \text{ } at$   
**using** *assms* **apply**(*cases at*) **using** *int-transE-proj by auto*

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

**lemma** *satC-snvC-transE[simp]*:  
**assumes**  $wtC: Ik.wtC \text{ } c$  **and**  $\xi: I.wtE \xi$   
**and**  $pC: \bigwedge \sigma. \sigma \vdash c$  **and**  $Ik.satC \text{ (transE } \xi) \text{ } c$   
**shows**  $I.satC \xi \text{ } c$

**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**  $\xi: I.wtE \xi$  **shows**  $I.satPB \xi \Phi$   
**using** *Ik.wt- $\Phi$  Ik.sat- $\Phi$ [OF wtE-transE[OF  $\xi$ ]]*  
**using** *mcalc  $\xi$*  **unfolding** *Ik.satPB-def satPB-def Ik.wtPB-def nvPB-def* **by auto**

**lemma** *I-SAT*:  $I.SAT \Phi$  **unfolding** *I.SAT-def* **by auto**

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

**end**

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

**context** *ProblemIkMcalc* **begin**

**abbreviation**  $MModelIkMcalc \equiv ModelIkMcalc wtFsym wtPsym arOf resOf parOf$   
 $\Phi infTp$

**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**  $\exists intTI intFI intPI. InfModel intTI intFI intPI$   
**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  $\Phi$  infTp
  for wtFsym :: 'fsym  $\Rightarrow$  bool
  and wtPsym :: 'psym  $\Rightarrow$  bool
  and arOf :: 'fsym  $\Rightarrow$  'tp list
  and resOf and parOf and  $\Phi$  and infTp +
  fixes
    prot :: 'tp  $\Rightarrow$  bool
  and protFw :: 'tp  $\Rightarrow$  bool
  assumes
    tp-disj:  $\bigwedge \sigma. \neg \text{prot } \sigma \vee \neg \text{protFw } \sigma$ 
  and tp-mcalc:  $\bigwedge \sigma. \text{prot } \sigma \vee \text{protFw } \sigma \vee (\forall c \in \Phi. \sigma \vdash c)$ 
  begin

```

**definition** *isRes* **where** *isRes*  $\sigma \equiv \exists f. \text{wtFsym } f \wedge \text{resOf } f = \sigma$

**definition** *unprot*  $\sigma \equiv \neg \text{prot } \sigma \wedge \neg \text{protFw } \sigma$

**lemma** *unprot-mcalc*[*simp*]:  $\llbracket \text{unprot } \sigma; c \in \Phi \rrbracket \Longrightarrow \sigma \vdash c$   
**using** *tp-mcalc* **unfolding** *unprot-def* **by** *auto*

**end**

```

locale ModelIkTpart =
  Ik? : ProblemIkTpart wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp prot protFw +
  Ik? : Model wtFsym wtPsym arOf resOf parOf  $\Phi$  intT intF intP
  for wtFsym :: 'fsym  $\Rightarrow$  bool
  and wtPsym :: 'psym  $\Rightarrow$  bool
  and arOf :: 'fsym  $\Rightarrow$  'tp list
  and resOf and parOf and  $\Phi$  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  $\Phi$  infTp
for wtFsym :: 'fsym  $\Rightarrow$  bool and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf and parOf and  $\Phi$  and infTp
+ fixes pol :: 'tp  $\Rightarrow$  'psym  $\Rightarrow$  epol
and
  grdOf ::
('fsym, 'psym) cls  $\Rightarrow$  ('fsym, 'psym) lit  $\Rightarrow$  var  $\Rightarrow$  ('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  $\cup$  nv2T T2
|
nv2L (Neg (Eq T1 T2)) = {}
|
nv2L (Pos (Pr p Tl)) = {x  $\in$   $\bigcup$  set (map nv2T Tl) . pol (tpOfV x) p = Fext}
|
nv2L (Neg (Pr p Tl)) = {x  $\in$   $\bigcup$  set (map nv2T Tl) . pol (tpOfV x) p = Text}

```

```

definition nv2C c  $\equiv$   $\bigcup$  set (map nv2L c)

```

```

lemma in-nv2T: x  $\in$  nv2T T  $\longleftrightarrow$  T = Var x
apply(cases T)
  apply (metis equals0D nv2T.simps singleton-iff)
  by simp

```

```

lemma nv2T-vars[simp]: x  $\in$  nv2T T  $\implies$  x  $\in$  vars T
by (induct T, auto split: if-splits)

```

```

lemma nv2L-varsL[simp]:
assumes x  $\in$  nv2L l shows x  $\in$  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]:  $x \in nv2C\ c \implies x \in varsC\ c$ 
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  $\Rightarrow$  ('fsym, 'psym) lit  $\Rightarrow$  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  $\vdash_2$  40) where
  [simp]: infTp  $\sigma \implies \sigma \vdash_2\ c$ 
  |[simp]: ( $\bigwedge\ l\ x. \llbracket l \in set\ c; x \in nv2L\ l; tpOfV\ x = \sigma \rrbracket$ 
     $\implies isGuard\ x\ (grdOf\ c\ l\ x) \implies \sigma \vdash_2\ c$ )

```

**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  $\Phi$ 
  infTp pol grdOf
for wtFsym :: 'fsym  $\Rightarrow$  bool and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf and parOf and  $\Phi$  and infTp and pol and grdOf
+ assumes

```

```

    grdOf:  $\bigwedge c l x. \llbracket c \in \Phi; l \in \text{set } c; x \in \text{nv2L } l; \neg \text{infTp } (\text{tpOfV } x) \rrbracket$ 
            $\implies \text{grdOf } c l x \in \text{set } c$ 
and mcalc2:  $\bigwedge \sigma c. c \in \Phi \implies \sigma \vdash_2 c$ 
begin
lemma wtL-grdOf[simp]:
assumes  $c \in \Phi$  and  $l \in \text{set } c$  and  $x \in \text{nv2L } l$  and  $\neg \text{infTp } (\text{tpOfV } x)$ 
shows wtL (grdOf c l x)
using grdOf[OF assms] by (metis assms list-all-iff wtC-def wtPB-def wt- $\Phi$ )
end

locale ModellkPolMcalc2 =
  ModelIk wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp intT intF intP +
  ProblemIkPolMcalc2 wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp pol grdOf
for wtFsym :: 'fsym  $\implies$  bool and wtPsym :: 'psym  $\implies$  bool
and arOf :: 'fsym  $\implies$  'tp list
and resOf and parOf and  $\Phi$  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 different 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  $\Phi$  infTp pol grdOf
for wtFsym :: 'fsym  $\implies$  bool and wtPsym :: 'psym  $\implies$  bool
and arOf :: 'fsym  $\implies$  'tp list
and resOf and parOf and  $\Phi$  and infTp and pol and grdOf
+ assumes pol-ct: pol  $\sigma 1 P = \text{pol } \sigma 2 P$ 

context ProblemIkPolMcalc2C begin

```

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

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

**lemma** nv2L-simps[simp]:  
 $\text{nv2L } (\text{Pos } (\text{Pr } p \text{ Tl})) = (\text{case } \text{polC } p \text{ of } \text{Fext} \implies \bigcup \text{set } (\text{map } \text{nv2T } \text{Tl}) \mid - \implies \{\})$   
 $\text{nv2L } (\text{Neg } (\text{Pr } p \text{ Tl})) = (\text{case } \text{polC } p \text{ of } \text{Text} \implies \bigcup \text{set } (\text{map } \text{nv2T } \text{Tl}) \mid - \implies \{\})$

**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))  $\longleftrightarrow x \in \bigcup \text{set } (\text{map } \text{nv2T } Tl) \wedge \text{polC } p = \text{Text}$*

*isGuard x (Neg (Pr p Tl))  $\longleftrightarrow x \in \bigcup \text{set } (\text{map } \text{nv2T } Tl) \wedge \text{polC } p = \text{Fext}$*

**by** (*auto simp: pol-polC*)

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

**end**

**locale** *ModelIkPolMcalc2C =*

*ModelIk wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp intT intF intP +*

*ProblemIkPolMcalc2C wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp pol grdOf*

**for** *wtFsym :: 'fsym  $\Rightarrow$  bool and wtPsym :: 'psym  $\Rightarrow$  bool*

**and** *arOf :: 'fsym  $\Rightarrow$  'tp list*

**and** *resOf and parOf and  $\Phi$  and infTp pol and grdOf and intT and intF and intP*

## 6.5 Extension of a structure to an infinite structure by adding indistinguishable elements

**context** *ModelIkPolMcalc2C begin*

**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  $\implies$  proj  $\sigma$  a = a*

**unfolding** *proj-def by auto*

**lemma** *map-proj-id[simp]:*

**assumes** *list-all2 intT  $\sigma$  l al*

**shows** *map2 proj  $\sigma$  l al = al*

**apply** (*rule nth-equalityI*)

**using** *assms unfolding list-all2-length by auto*

**lemma** *surj-proj:*

**assumes** *intT  $\sigma$  a shows  $\exists b. \text{proj } \sigma b = a$*

**using** *assms by (intro exI[of - a]) simp*

**definition** *I-intT  $\sigma$  (a::univ)  $\equiv$  infTp  $\sigma \longrightarrow$  intT  $\sigma$  a*

**definition** *I-intF f al  $\equiv$  intF f (map2 proj (arOf f) al)*

**definition**

$I\text{-intP } p \text{ al} \equiv$   
*case*  $\text{polC } p$  *of*  
 $\text{Cext} \Rightarrow \text{intP } p \text{ (map2 proj (parOf } p) \text{ al)}$   
 $|\text{Text} \Rightarrow \text{if list-all2 intT (parOf } p) \text{ al then intP } p \text{ al else True}$   
 $|\text{Fext} \Rightarrow \text{if list-all2 intT (parOf } p) \text{ al then intP } p \text{ al else False}$

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

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

**lemma**  $\text{NE-I-intT}$ :  $\text{NE } (I\text{-intT } \sigma)$   
**using**  $\text{NE-intT}$  **by** (*cases*  $\text{infTp } \sigma$ , *auto*)

**lemma**  $I\text{-intP-Cext[simp]}$ :  
 $\text{polC } p = \text{Cext} \Longrightarrow I\text{-intP } p \text{ al} = \text{intP } p \text{ (map2 proj (parOf } p) \text{ al)}$   
**unfolding**  $I\text{-intP-def}$  **by**  $\text{simp}$

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

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

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

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

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

**lemma**  $I\text{-intF}$ :  
**assumes**  $f$ :  $\text{wfFsym } f$  **and**  $\text{al}$ :  $\text{list-all2 } I\text{-intT } (\text{arOf } f) \text{ al}$   
**shows**  $I\text{-intT } (\text{resOf } f) (I\text{-intF } f \text{ al})$   
**unfolding**  $I\text{-intT-def}$   $I\text{-intF-def}$  **apply** *safe* **apply**(*rule*  $\text{intF}[OF f]$ )



**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  $\sigma$  a}*  
**by** (*cases infTp  $\sigma$ , auto*)

**lemma** *InfStruct*: *InfStruct 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*  $\xi \equiv \lambda x. \text{proj } (tpOfV x) (\xi x)$

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

**lemma** *wtE-transE[simp]*: *I.wtE  $\xi \implies Ik.wtE (transE \xi)$*   
**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**  $\xi$ : *I.wtE  $\xi$*

**and** *nv2T*: *infTp (Ik.tpOf T)  $\vee (\forall x \in nv2T T. tpOfV x \neq Ik.tpOf T)$*

**shows** *Ik-intT (Ik.tpOf T) (I.int  $\xi T)$*

**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  $\xi$ ]*

**by** (*metis I.wt-int I-intT-def  $\xi wt$* )

**next**

**case** *False* **hence**  $\forall x \in nv2T T. tpOfV x \neq tpOf T$  **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*:  $\neg (\exists x. T = Var x)$  **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  $\xi$ ) T = proj (tpOf T) (I.int  $\xi$  T)
using wt proof (induct T)
  case (Fn f Tl)
  have 0: Ik-intT (resOf f) (I-intF f (map (int  $\xi$ ) Tl)) (is Ik-intT ? $\sigma$  ?a)
  unfolding I-intF-def apply(rule Ik.intF)
  using Fn unfolding list-all2-length list-all-iff by auto
  have 1: proj ? $\sigma$  ?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, safe)
    have l[simp]: length (arOf f) = length Tl using Fn by simp
    fix i assume i < length (map (Ik.int (transE  $\xi$ )) 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  $\xi$ ) (Tl ! i) = proj (arOf f ! i) (I.int  $\xi$  (Tl ! i))
    unfolding 0 using Fn by (auto simp: list-all-length transE-def)
    show map (Ik.int (transE  $\xi$ )) Tl ! i =
      map2 proj (arOf f) (map (I.int  $\xi$ ) 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  $\xi$ : I.wtE  $\xi$ 
and nv2T: infTp (Ik.tpOf T)  $\vee$  ( $\forall$  x  $\in$  nv2T T. tpOfV x  $\neq$  Ik.tpOf T)
shows Ik.int (transE  $\xi$ ) T = I.int  $\xi$  T
unfolding int-transE-proj[OF wt] apply(rule proj-id)
using Ik-intT-int[OF wt  $\xi$  nv2T] .

```

```

lemma isGuard-not-satL-intT:
assumes wtL: Ik.wtL l
and ns:  $\neg$  I.satL  $\xi$  l
and g: isGuard x l and  $\xi$ : I.wtE  $\xi$ 
shows Ik-intT (tpOfV x) ( $\xi$  x) (is Ik-intT ? $\sigma$  ( $\xi$  x))

```

```

proof(cases l)
  case (Pos at) show ?thesis proof(cases at)
    case (Pr p Tl)
    then obtain T where Tin: T  $\in$  set Tl and x: x  $\in$  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**  $list\text{-}all2\ Ik\text{-}intT\ (parOf\ p)\ (map\ (I.int\ \xi)\ Tl)\ (is\ ?phi)$   
**using**  $ns\ unfolding\ Neg\ Pr\ using\ pol\ by\ (cases\ ?phi,\ auto)$   
**hence**  $Ik\text{-}intT\ ?\sigma\ (I.int\ \xi\ T)$   
**using**  $ns\ 0\ i\ Ti\ unfolding\ Neg\ Pr\ by\ (auto\ simp\ add:\ list\text{-}all2\ \text{-}length\ nth\ \text{-}map)$   
**thus**  $?thesis\ unfolding\ T\ by\ simp$   
**qed**  
**qed**

**lemma**  $int\text{-}transE[simp]$ :  
**assumes**  $wt: Ik.wt\ T$  **and**  $\xi: I.wtE\ \xi$  **and**  
 $nv2T: \bigwedge x. [\neg\ infTp\ (tpOfV\ x); x \in nv2T\ T] \implies$   
 $\exists l. Ik.wtL\ l \wedge \neg I.satL\ \xi\ l \wedge isGuard\ x\ l$   
**shows**  $Ik.int\ (transE\ \xi)\ T = I.int\ \xi\ T$   
**proof**( $cases\ infTp\ (Ik.tpOf\ T) \vee (\forall x \in nv2T\ T. tpOfV\ x \neq Ik.tpOf\ T)$ )  
**case**  $True$   
**thus**  $?thesis\ using\ int\text{-}transE\text{-}nv2T[OF\ wt\ \xi]$  **by**  $auto$   
**next**  
**define**  $\sigma$  **where**  $\sigma = Ik.tpOf\ T$   
**case**  $False$  **then** **obtain**  $x$  **where**  $i: \neg\ infTp\ \sigma$  **and**  $x: x \in nv2T\ T$   
**unfolding**  $\sigma\text{-}def$  **by**  $auto$   
**hence**  $T: T = Var\ x$  **by** ( $simp\ add:\ in\text{-}nv2T$ )  
**hence**  $\sigma: \sigma = tpOfV\ x$  **unfolding**  $\sigma\text{-}def$  **by**  $simp$   
**obtain**  $l$  **where**  $0: Ik.wtL\ l \wedge \neg I.satL\ \xi\ l \wedge isGuard\ x\ l$   
**using**  $nv2T[OF\ i[unfolded\ \sigma]\ x]$  **by**  $auto$   
**show**  $?thesis\ unfolding\ T\ using\ isGuard\text{-}not\text{-}satL\text{-}intT[OF\ 0\ \xi]$  **by**  $simp$   
**qed**

**lemma**  $intT\text{-}int\text{-}transE[simp]$ :  
**assumes**  $wt: Ik.wt\ T$  **and**  $\xi: I.wtE\ \xi$  **and**  
 $nv2T: \bigwedge x. [\neg\ infTp\ (tpOfV\ x); x \in nv2T\ T] \implies$   
 $\exists l. Ik.wtL\ l \wedge \neg I.satL\ \xi\ l \wedge isGuard\ x\ l$   
**shows**  $Ik\text{-}intT\ (Ik.tpOf\ T)\ (I.int\ \xi\ T)$   
**proof**–  
**have**  $0: I.int\ \xi\ T = Ik.int\ (transE\ \xi)\ T$  **using**  $int\text{-}transE[OF\ assms]$  **by**  $simp$   
**show**  $?thesis\ unfolding\ 0\ using\ Ik.wt\text{-}int[OF\ wtE\text{-}transE[OF\ \xi]\ wt]$ .  
**qed**

**lemma**  $map\text{-}int\text{-}transE\text{-}nv2T[simp]$ :  
**assumes**  $wt: list\text{-}all\ Ik.wt\ Tl$  **and**  $\xi: I.wtE\ \xi$  **and**  
 $nv2T: \bigwedge x. [\neg\ infTp\ (tpOfV\ x); \exists T \in set\ Tl. x \in nv2T\ T] \implies$   
 $\exists l. Ik.wtL\ l \wedge \neg I.satL\ \xi\ l \wedge isGuard\ x\ l$   
**shows**  $map\ (Ik.int\ (transE\ \xi))\ Tl = map\ (I.int\ \xi)\ Tl$   
**apply**( $rule\ nth\ \text{-}equalityI$ ) **using**  $assms$  **by** ( $force\ simp:\ list\text{-}all\ \text{-}iff\ intro:\ int\text{-}transE$ )+

**lemma**  $list\text{-}all2\text{-}intT\text{-}int\text{-}transE\text{-}nv2T[simp]$ :  
**assumes**  $wt: list\text{-}all\ Ik.wt\ Tl$  **and**  $\xi: I.wtE\ \xi$  **and**  
 $nv2T: \bigwedge x. [\neg\ infTp\ (tpOfV\ x); \exists T \in set\ Tl. x \in nv2T\ T] \implies$   
 $\exists l. Ik.wtL\ l \wedge \neg I.satL\ \xi\ l \wedge isGuard\ x\ l$   
**shows**  $list\text{-}all2\ Ik\text{-}intT\ (map\ Ik.tpOf\ Tl)\ (map\ (I.int\ \xi)\ Tl)$

**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**  $\text{map } (Ik.int \ (transE \ \xi)) \ Tl =$   
 $\text{map2 } proj \ (\text{map } tpOf \ Tl) \ (\text{map } (I.int \ \xi) \ Tl)$   
**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**  $\xi$ : *I.wtE \ \xi* **and**  
 $nv2T$ :  $\bigwedge x. \llbracket \neg \ infTp \ (tpOfV \ x); \ x \in \ nv2L \ ll \rrbracket \implies$   
 $\exists l'. \ Ik.wtL \ l' \wedge \neg \ I.satL \ \xi \ l' \wedge \ isGuard \ x \ l'$   
**and** *Ik.satL (transE \ \xi) l*  
**shows** *I.satL \ \xi 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 \ \xi) 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 \ \xi) Tl), force, force*)  
**qed**(*insert assms int-transE-proj, unfold Neg, auto*)  
**qed**

**lemma** *satPB-transE[simp]*:  
**assumes**  $\xi$ : *I.wtE \ \xi* **shows** *I.satPB \ \xi \ \Phi*  
**unfolding** *I.satPB-def* **proof** *safe*  
**fix** *c* **assume** *cin*:  $c \in \Phi$  **let** *?thesis* = *I.satC \ \xi \ c*  
**have** *mc*:  $\bigwedge \sigma. \ \sigma \vdash_2 \ c$  **using** *mcalc2[OF cin]* .  
**have** *c*: *Ik.satC (transE \ \xi) c*  
**using** *sat-\Phi wtE-transE[OF \ \xi] cin* **unfolding** *Ik.satPB-def* **by** *auto*  
**have** *wtC*: *Ik.wtC c* **using** *wt-\Phi cin* **unfolding** *wtPB-def* **by** *auto*  
**obtain** *l* **where** *lin*:  $l \in \text{set } c$  **and** *l*: *Ik.satL (transE \ \xi) 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*:  $\bigwedge l'. \ l' \in \text{set } c \implies \neg \ I.satL \ \xi \ l'$  **unfolding** *I.satC-iff-set* **by** *auto*  
**have** *I.satL \ \xi l*  
**proof** (*rule satL-transE[OF wtL \ \xi - 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 < 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  $\Phi$  infTp prot protFw
for wtFsym :: 'fsym  $\Rightarrow$  bool
and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf and parOf and  $\Phi$  and infTp and prot and protFw +
fixes
  protCl :: 'tp  $\Rightarrow$  bool
assumes
  protCl-prot[simp]:  $\bigwedge \sigma. \text{protCl } \sigma \Longrightarrow \text{prot } \sigma$ 

and protCl-fsym:  $\bigwedge f. \text{protCl } (\text{resPf } f) \Longrightarrow \text{list-all protCl } (\text{arOf } f)$ 

locale ModellkTpartG =
Ik? : ProblemIkTpartG wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp prot protFw
protCl +
Ik? : ModellkTpart wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp prot protFw intT
intF intP
for wtFsym :: 'fsym  $\Rightarrow$  bool
and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf and parOf and  $\Phi$  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  $i < \text{length } (\text{arOf } f)$ 
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  $\sigma$ )  $\longleftrightarrow$   $\neg$  isRes  $\sigma \vee$  protCl  $\sigma$ 
```

```
fun GE-arOf where
  GE-arOf (Oldf f) = arOf f
| GE-arOf (Wit  $\sigma$ ) = []
```

```
fun GE-resOf where
  GE-resOf (Oldf f) = resOf f
| GE-resOf (Wit  $\sigma$ ) =  $\sigma$ 
```

```
fun GE-wtPsym where
  GE-wtPsym (Oldp p)  $\longleftrightarrow$  wtPsym p
| GE-wtPsym (Guard  $\sigma$ )  $\longleftrightarrow$   $\neg$  unprot  $\sigma$ 
```

```
fun GE-parOf where
  GE-parOf (Oldp p) = parOf p
| GE-parOf (Guard  $\sigma$ ) = [ $\sigma$ ]
```

**lemma** countable-GE-wtFsym: countable (Collect GE-wtFsym) (is countable ?K)

**proof**–

```
let ?F =  $\lambda$  ef. case ef of Oldf f  $\Rightarrow$  Inl f | Wit  $\sigma \Rightarrow$  Inr  $\sigma$ 
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  $\subseteq$  ?L apply clarify by (case-tac ef, auto)
ultimately have |?K|  $\leq_o$  |?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 =  $\lambda$  ep. case ep of Oldp p  $\Rightarrow$  Inl p | Guard  $\sigma \Rightarrow$  Inr  $\sigma$ 
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  $\subseteq$  ?L apply clarify by (case-tac ep, auto)
ultimately have |?K|  $\leq_o$  |?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*  $\Rightarrow$  (*'fsym*, *'tp*) *efsym*, (*'psym*, *'tp*) *epsym*) *lit*  
**where** *grdLit* *x*  $\equiv$  *Neg* (*Pr* (*Guard* (*tpOfV* *x*)) [*Var* *x*])

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

**fun** *glitOfL* ::  
(*'fsym*, *'psym*) *lit*  $\Rightarrow$  (*'fsym*, *'tp*) *efsym*, (*'psym*, *'tp*) *epsym*) *lit* *set*  
**where**  
*glitOfL* (*Pos* *at*) =  
{*grdLit* *x* | *x*. *x*  $\in$  *varsA* *at*  $\wedge$  (*prot* (*tpOfV* *x*)  $\vee$  (*protFw* (*tpOfV* *x*)  $\wedge$  *x*  $\in$  *nvA* *at*))}  
|  
*glitOfL* (*Neg* *at*) = {*grdLit* *x* | *x*. *x*  $\in$  *varsA* *at*  $\wedge$  *prot* (*tpOfV* *x*)}

**definition** *glitOfC* *c*  $\equiv$   $\bigcup$  *set* (*map* *glitOfL* *c*)

**lemma** *finite-glitOfL[simp]*: *finite* (*glitOfL* *l*)

**proof** –

**have** *glitOfL* *l*  $\subseteq$  *grdLit* ‘ {*x* . *x*  $\in$  *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**  $gC\ c \equiv (map\ gL\ c) \ @\ (list\ (glitOfC\ c))$

**lemma**  $set\text{-}gC[simp]$ :  $set\ (gC\ c) = gL\ ' (set\ c) \cup\ glitOfC\ c$   
**unfolding**  $gC\text{-}def$  **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**  $fax\ f \equiv [Pos\ (cOfFax\ f)]$

**definition**  $faxCD\ f \equiv map\ Neg\ (hOfFax\ f) \ @\ fax\ f$

**definition**

$Fax \equiv \{fax\ f \mid f.\ wtFsym\ f \wedge \neg\ unprot\ (resOf\ f) \wedge \neg\ protCl\ (resOf\ f)\} \cup$   
 $\{faxCD\ f \mid f.\ wtFsym\ f \wedge protCl\ (resOf\ f)\}$

The witness axioms:

**definition**  $wax\ \sigma \equiv [Pos\ (Pr\ (Guard\ \sigma)\ [Fn\ (Wit\ \sigma)\ []])]$

**definition**  $Wax \equiv \{wax\ \sigma \mid \sigma.\ \neg\ unprot\ \sigma \wedge (\neg\ isRes\ \sigma \vee protCl\ \sigma)\}$

**definition**  $gPB = gC\ ' \Phi \cup Fax \cup Wax$

Well-typedness of the translation:

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

**lemma**  $wt\text{-}g[simp]$ :  $Ik.\ wt\ T \implies GE.\ wt\ (gT\ T)$   
**by**  $(induct\ T,\ auto\ simp\ add:\ list\text{-}all\text{-}iff)$

**lemma**  $wtA\text{-}gA[simp]$ :  $Ik.\ wtA\ at \implies GE.\ wtA\ (gA\ at)$   
**by**  $(cases\ at,\ auto\ simp\ add:\ list\text{-}all\text{-}iff)$

**lemma**  $wtL\text{-}gL[simp]$ :  $Ik.\ wtL\ l \implies GE.\ wtL\ (gL\ l)$   
**by**  $(cases\ l,\ auto)$

**lemma**  $wtC\text{-}map\text{-}gL[simp]$ :  $Ik.\ wtC\ c \implies GE.\ wtC\ (map\ gL\ c)$   
**unfolding**  $Ik.\ wtC\text{-}def\ GE.\ wtC\text{-}def$  **by**  $(induct\ c,\ auto)$

**lemma**  $wtL\text{-}grdLit\text{-}unprot[simp]$ :  $\neg\ unprot\ (tpOfV\ x) \implies GE.\ wtL\ (grdLit\ x)$   
**unfolding**  $grdLit\text{-}def$  **by**  $auto$

**lemma** *wtL-grdLit[simp]*:  $\text{prot } (tpOfV x) \vee \text{protFw } (tpOfV x) \implies GE.wtL (grdLit x)$   
**apply**(rule *wtL-grdLit-unprot*) **unfolding** *unprot-def* **by** *auto*

**lemma** *wtL-glitOfL[simp]*:  $l' \in \text{glitOfL } l \implies GE.wtL l'$   
**by** (*cases l, auto*)

**lemma** *wtL-glitOfC[simp]*:  $l' \in \text{glitOfC } c \implies GE.wtL l'$   
**unfolding** *glitOfC-def* *GE.wtC-def* **by** (*induct c, auto*)

**lemma** *wtC-list-glitOfC[simp]*:  $GE.wtC (\text{list } (\text{glitOfC } c))$   
**unfolding** *glitOfC-def* *GE.wtC-def* **by** *auto*

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

**lemma** *wtA-cOfFax-unprot[simp]*:  $\llbracket wtFsym f; \neg \text{unprot } (\text{resOf } f) \rrbracket \implies GE.wtA (cOfFax f)$   
**unfolding** *cOfFax-def* **by** *simp*

**lemma** *wtA-cOfFax[simp]*:  
 $\llbracket wtFsym f; \text{prot } (\text{resOf } f) \vee \text{protFw } (\text{resOf } f) \rrbracket \implies GE.wtA (cOfFax f)$   
**apply**(rule *wtA-cOfFax-unprot*) **unfolding** *unprot-def* **by** *auto*

**lemma** *wtA-hOfFax[simp]*:  
 $\llbracket wtFsym f; \text{protCl } (\text{resOf } f) \rrbracket \implies \text{list-all } GE.wtA (hOfFax f)$   
**unfolding** *hOfFax-def* **unfolding** *list-all-length*  
**by** (*auto simp add: singl-def unprot-def*)

**lemma** *wtC-fax-unprot[simp]*:  $\llbracket wtFsym f; \neg \text{unprot } (\text{resOf } f) \rrbracket \implies GE.wtC (fax f)$   
**unfolding** *fax-def* *GE.wtC-def* **by** *auto*

**lemma** *wtC-fax[simp]*:  $\llbracket wtFsym f; \text{prot } (\text{resOf } f) \vee \text{protFw } (\text{resOf } f) \rrbracket \implies GE.wtC (fax f)$   
**apply**(rule *wtC-fax-unprot*) **unfolding** *unprot-def* **by** *auto*

**lemma** *wtC-faxCD[simp]*:  $\llbracket wtFsym f; \text{protCl } (\text{resOf } f) \rrbracket \implies GE.wtC (faxCD f)$   
**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]*:  $\llbracket \neg \text{unprot } \sigma; \neg \text{isRes } \sigma \vee \text{protCl } \sigma \rrbracket \implies GE.wtC (wax \sigma)$   
**unfolding** *wax-def* *GE.wtC-def* **by** *simp*

**lemma** *wtC-wax*[simp]:  $\llbracket \text{prot } \sigma \vee \text{protFw } \sigma; \neg \text{isRes } \sigma \vee \text{protCl } \sigma \rrbracket \implies \text{GE.wtC}$   
*(wax  $\sigma$ )*

**apply**(*rule wtC-wax-unprot*) **unfolding** *unprot-def* **by** *auto*

**lemma** *wtPB-Wax*[simp]:  $\text{GE.wtPB Wax}$

**unfolding** *Wax-def* *GE.wtPB-def* **by** *auto*

**lemma** *wtPB-gC- $\Phi$* [simp]:  $\text{GE.wtPB (gC '  $\Phi$ )}$

**using** *Ik.wt- $\Phi$*  **unfolding** *Ik.wtPB-def* *GE.wtPB-def* **by** *auto*

**lemma** *wtPB-tPB*[simp]:  $\text{GE.wtPB gPB}$  **unfolding** *gPB-def* **by** *simp*

**lemma** *wtA-Guard*:

**assumes**  $\text{GE.wtA (Pr (Guard } \sigma) Tl)$

**shows**  $\exists T. Tl = [T] \wedge \text{GE.wt } T \wedge \text{tpOf } T = \sigma$

**using** *assms* **by** *simp* (*metis (hide-lams, no-types) list.inject map-eq-Cons-conv*  
*list-all-simps map-is-Nil-conv neq-Nil-conv*)

**lemma** *wt-Wit*:

**assumes**  $\text{GE.wt (Fn (Wit } \sigma) Tl)$  **shows**  $Tl = []$

**using** *assms* **by** *simp*

**lemma** *tpOf-Wit*:  $\text{GE.tpOf (Fn (Wit } \sigma) Tl) = \sigma$  **by** *simp*

**end**

## 7.2 Soundness

**context** *ModelIkTpartG* **begin**

**fun** *GE-intF* **where**

$\text{GE-intF (Oldf } f) al = \text{intF } f al$   
 $|\text{GE-intF (Wit } \sigma) al = \text{pickT } \sigma$

**fun** *GE-intP* **where**

$\text{GE-intP (Oldp } p) al = \text{intP } p al$   
 $|\text{GE-intP (Guard } \sigma) al = \text{True}$

**end**

**sublocale** *ModelIkTpartG* < *GE?* : *Struct*

**where**  $\text{wtFsym} = \text{GE-wtFsym}$  **and**  $\text{wtPsym} = \text{GE-wtPsym}$  **and**  
 $\text{arOf} = \text{GE-arOf}$  **and**  $\text{resOf} = \text{GE-resOf}$  **and**  $\text{parOf} = \text{GE-parOf}$   
**and**  $\text{intF} = \text{GE-intF}$  **and**  $\text{intP} = \text{GE-intP}$

**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]*:  $GE.int\ \xi\ (gT\ T) = Ik.int\ \xi\ T$

**proof** (*induct T*)

**case** (*Fn f Tl*)

**hence** *0*:  $map\ (GE.int\ \xi\ \circ\ gT)\ Tl = map\ (Ik.int\ \xi)\ Tl$

**unfolding** *list-eq-iff list-all-iff* **by** *auto*

**show** *?case* **by** (*simp add: 0*)

**qed** *auto*

**lemma** *map-g-int[simp]*:  $map\ (GE.int\ \xi\ \circ\ gT)\ Tl = map\ (Ik.int\ \xi)\ Tl$

**unfolding** *list-eq-iff list-all-iff* **by** *auto*

**lemma** *gA-satA[simp]*:  $GE.satA\ \xi\ (gA\ at) \longleftrightarrow Ik.satA\ \xi\ at$

**apply**(*cases at*) **by** *auto*

**lemma** *gL-satL[simp]*:  $GE.satL\ \xi\ (gL\ l) \longleftrightarrow Ik.satL\ \xi\ l$

**apply**(*cases l*) **by** *auto*

**lemma** *map-gL-satC[simp]*:  $GE.satC\ \xi\ (map\ gL\ c) \longleftrightarrow Ik.satC\ \xi\ c$

**unfolding** *GE.satC-def Ik.satC-def* **by** (*induct c, auto*)

**lemma** *gC-satC[simp]*:

**assumes**  $Ik.satC\ \xi\ c$  **shows**  $GE.satC\ \xi\ (gC\ c)$

**using** *assms unfolding gC-def* **by** *simp*

**lemma** *gC-Φ-satPB[simp]*:

**assumes**  $Ik.satPB\ \xi\ \Phi$  **shows**  $GE.satPB\ \xi\ (gC\ \Phi)$

**using** *assms unfolding GE.satPB-def Ik.satPB-def* **by** *auto*

**lemma** *Fax-Wax-satPB*:

$GE.satPB\ \xi\ (Fax) \wedge GE.satPB\ \xi\ (Wax)$

**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**  $\Phi = gPB$  **and** *intF = GE-intF and intP = GE-intP*  
**using** *G-soundness .*

### 7.3 Completeness

**locale** *ProblemIkTpartG-GEModel =*  
*Ik? : ProblemIkTpartG wtFsym wtPsym arOf resOf parOf  $\Phi$  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  $\Rightarrow$  bool*  
**and** *wtPsym :: 'psym  $\Rightarrow$  bool*  
**and** *arOf :: 'fsym  $\Rightarrow$  'tp list*  
**and** *resOf and parOf and  $\Phi$  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  $\sigma$  a  $\equiv$*   
*if unprot  $\sigma$  then eintT  $\sigma$  a*  
*else eintT  $\sigma$  a  $\wedge$  eintP (Guard  $\sigma$ ) [a]*

**definition**

*intF f al  $\equiv$  eintF (Oldf f) al*

**definition**

*intP p al  $\equiv$  eintP (Oldp p) al*

**lemma** *GE-Guard-all:*

**assumes** *f: wtFsym f*

**and** *al: list-all2 eintT (arOf f) al*

**shows**

$(\neg \text{unprot } (\text{resOf } f) \wedge \neg \text{protCl } (\text{resOf } f)$   
 $\longrightarrow \text{eintP } (\text{Guard } (\text{resOf } f)) [\text{eintF } (\text{Oldf } f) \text{ al}]$   
 $\wedge$   
 $(\text{protCl } (\text{resOf } f) \longrightarrow$

```

    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]:  $\bigwedge i. i < \text{length } (arOf f) \implies tpOfV (xl!i) = (arOf f)!i$ 
  unfolding xl-def by auto
  have xl[simp]: distinct xl unfolding xl-def using distinct-getVars by auto
  define  $\xi$  where  $\xi = \text{pickE } xl$ 
  have  $\xi$ : GE.wtE  $\xi$  unfolding  $\xi$ -def apply(rule wtE-pickE) using al list-all2-nthD
by auto
  have [simp]:  $\bigwedge i. i < \text{length } (arOf f) \implies \xi (xl ! i) = al ! i$ 
  using al unfolding  $\xi$ -def by (auto simp: list-all2-length intro: pickE)
  have 0: map (GE.int  $\xi$ ) (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  $\xi$  (map Neg (map2 (Pr ◦ Guard) (arOf f) (map singl (getTvars (arOf
f))))))  $\longleftrightarrow$ 
     $\neg$  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: map2-def map-zip-map2 singl-def getTvars-def xl-def[symmetric])
  have Fax: GE.satPB  $\xi$  Fax using GE.sat- $\Phi$ [OF  $\xi$ ] unfolding gPB-def by simp
  {assume ?A1
    hence GE.satC  $\xi$  (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  $\xi$  (faxCD f) using f Fax unfolding GE.satPB-def Fax-def by
auto
    thus ?C using  $\langle ?H2 \rangle$ 
    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:  $\neg$  unprot (resOf f)  $\neg$  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**  $H$ :  $list-all2\ (eintP\ \circ\ Guard)\ (arOf\ f)\ (map\ singl\ al)$   
**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**  $f2$ :  $\neg\ unprot\ (resOf\ f)$  **and**  $al$ :  $list-all2\ eintT\ (arOf\ f)\ al$   
**and**  $H$ :  $list-all2\ (eintP\ \circ\ Guard)\ (arOf\ f)\ (map\ singl\ al)$   
**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**  $\sigma$ :  $\neg\ unprot\ \sigma \neg\ isRes\ \sigma \vee\ protCl\ \sigma$   
**shows**  $eintP\ (Guard\ \sigma)\ [eintF\ (Wit\ \sigma)\ \square]$   
**proof** –  
**define**  $\xi$  **where**  $\xi = pickE\ \square\ \square$   
**have**  $\xi$ :  $GE.wtE\ \xi$  **unfolding**  $\xi-def$  **apply**( $rule\ wtE-pickE$ ) **by**  $auto$   
**have**  $GE.satPB\ \xi\ Wax$  **using**  $GE.sat-\Phi[OF\ \xi]$  **unfolding**  $gPB-def$  **by**  $simp$   
**hence**  $GE.satC\ \xi\ (wax\ \sigma)$  **unfolding**  $Wax-def\ GE.satPB-def$  **using**  $\sigma$  **by**  $auto$   
**thus**  $?thesis$  **unfolding**  $satC-def\ wax-def$  **by**  $simp$   
**qed**

**lemma**  $NE-intT-forget$ :  $NE\ (intT\ \sigma)$

**proof** –

**obtain**  $a$  **where**  $a$ :  $eintT\ \sigma\ a$  **using**  $GE.NE-intT$  **by**  $blast$   
**show**  $?thesis$   
**proof**( $cases\ unprot\ \sigma$ )  
**case**  $True$   
**thus**  $?thesis$  **using**  $a$  **unfolding**  $intT-def$  **by**  $auto$   
**next**  
**case**  $False$  **note**  $unprot = False$   
**show**  $?thesis$  **proof**( $cases\ isRes\ \sigma$ )  
**case**  $True$  **then** **obtain**  $f$  **where**  $f$ :  $wtFsym\ f$  **and**  $\sigma$ :  $\sigma = resOf\ f$   
**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**  $a$ :  $eintT\ \sigma\ a$  **unfolding**  $a-def\ \sigma\ intF-def$  **using**  $f\ al$   
**by** ( $metis\ GE-arOf.simps\ GE-resOf.simps\ GE-wtFsym.simps\ GE.intF$ )  
**show**  $?thesis$  **proof** ( $cases\ protCl\ \sigma$ )  
**case**  $True$   
**define**  $a$  **where**  $a = eintF\ (Wit\ \sigma)\ \square$   
**have**  $eintT\ \sigma\ a$  **unfolding**  $a-def$   
**by** ( $metis\ True\ GE-arOf.simps\ GE-resOf.simps\ GE-wtFsym.simps\ intF\ list-all2-Nil$ )



```

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

unfolding  $list$ -all-length unprot-def by auto

hence 2:  $list$ -all2 ( $eintP \circ$  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*  $\xi \implies$  *GE.wtE*  $\xi$   
**unfolding** *Ik.wtE-def GE.wtE-def intT-def* **by** *metis*

**lemma** *int-g[simp]*: *GE.int*  $\xi$  (*gT T*) = *Ik.int*  $\xi$  *T*  
**proof** (*induct T*)  
**case** (*Fn f Tl*)  
**let** *?ar* = *arOf f* **let** *?r* = *resOf f*  
**have** *0*: *map* (*Ik.int*  $\xi$ ) *Tl* = *map* (*GE.int*  $\xi \circ$  *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*  $\xi$ ) *Tl* = *map* (*GE.int*  $\xi \circ$  *gT*) *Tl*  
**apply**(*rule nth-equalityI*) **unfolding** *list-all-length* **by** *auto*

**lemma** *satA-gA[simp]*: *GE.satA*  $\xi$  (*gA at*)  $\longleftrightarrow$  *Ik.satA*  $\xi$  *at*  
**by** (*cases at*) (*auto simp add: intP-def*)

**lemma** *satL-gL[simp]*: *GE.satL*  $\xi$  (*gL l*)  $\longleftrightarrow$  *Ik.satL*  $\xi$  *l*  
**by** (*cases l*) *auto*

**lemma** *satC-map-gL[simp]*: *GE.satC*  $\xi$  (*map gL c*)  $\longleftrightarrow$  *Ik.satC*  $\xi$  *c*  
**unfolding** *GE.satC-def Ik.satC-def* **by** (*induct c*) *auto*

**lemma** *wtE-not-grdLit-unprot[simp]*:  
**assumes** *Ik.wtE*  $\xi$  **and**  $\neg$  *unprot* (*tpOfV x*)  
**shows**  $\neg$  *GE.satL*  $\xi$  (*grdLit x*)  
**using** *assms* **unfolding** *Ik.wtE-def intT-def grdLit-def* **by** *simp*

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

**lemma** *wtE-not-glitOfL[simp]*:  
**assumes** *Ik.wtE*  $\xi$   
**shows**  $\neg$  *GE.satC*  $\xi$  (*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*  $\xi$   
**shows**  $\neg$  *GE.satC*  $\xi$  (*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*  $\xi$  **and** *GE.satC*  $\xi$  (*gC*  $c$ )  
**shows** *Ik.satC*  $\xi$   $c$   
**using** *assms unfolding gC-def* **by** *simp*

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

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

**lemma** *G-completeness: Model wtFsym wtPsym arOf resOf parOf*  $\Phi$  *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  $\Phi$  = 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  $\Phi$  = gPB

```

**proof**

```

fix  $\sigma$  eintT eintF eintP a assume  $\sigma$ : infTp  $\sigma$ 
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
       $\Phi$  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  $\sigma$  a} using infTp[OF  $\sigma$  MM] .
moreover have {a. ?GE-intT  $\sigma$  a}  $\subseteq$  {a. eintT  $\sigma$  a}
using ProblemIkTpartG-GEModel.intT-def[OF 0] by auto
ultimately show infinite {a. eintT  $\sigma$  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  $\sigma$ ) = Fext

```

```

lemma pol-ct: pol  $\sigma$ 1 p = pol  $\sigma$ 2 p

```

```

by(cases p, auto)

```

```

definition grdOf c l x = grdLit x

```

```

end

```

```

sublocale ProblemIkTpartG < GE?: ProblemIkPol

```

**where**  $wtFsym = GE\text{-}wtFsym$  **and**  $wtPsym = GE\text{-}wtPsym$   
**and**  $arOf = GE\text{-}arOf$  **and**  $resOf = GE\text{-}resOf$  **and**  $parOf = GE\text{-}parOf$   
**and**  $\Phi = gPB$  **and**  $pol = pol$  **and**  $grdOf = grdOf$  ..

**context** *ProblemIkTpartG* **begin**

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

**lemma**  $nv2L\text{-}nvL[simp]$ :  $GE.nv2L (gL l) = GE.nvL l$   
**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**  $l \in set\ c$  **and**  $mc: GE.mcalc\ \sigma\ c$   
**shows**  $infTp\ \sigma \vee (\forall x \in GE.nv2L (gL\ l). tpOfV\ x \neq \sigma)$   
**using** *assms mc nv2L-nvL unfolding GE.mcalc-iff GE.nvC-def* **apply** *simp*  
**using**  $nv2L\text{-}nvL[of\ l]$   
**by** (*metis empty-subsetI equalityI nv2L-nvL*)

**lemma**  $isGuard\text{-}grdLit[simp]$ :  $GE.isGuard\ x (grdLit\ x)$   
**unfolding** *grdLit-def* **by** *auto*

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

**lemma**  $mcalc\text{-}mcalc2$ :  $GE.mcalc\ \sigma\ c \implies GE.mcalc2\ \sigma (gC\ c)$   
**using**  $nv2L$  **unfolding** *GE.mcalc2-iff gC-def glitOfC-def grdOf-def* **by** *auto*

**lemma**  $nv2L\text{-}wax[simp]$ :  $l' \in set (wax\ \sigma) \implies GE.nv2L\ l' = \{\}$   
**unfolding** *wax-def* **by** *auto*

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

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

**lemma**  $nv2L\text{-}hOfFax[simp]$ :  
**assumes**  $at \in set (hOfFax\ \sigma)$   
**shows**  $GE.nv2L (Neg\ at) = \{\}$   
**using** *assms* **unfolding** *hOfFax-def map2-def* **by** *auto*

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

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

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

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

**lemma** *mcalc2*:  
**assumes**  $c': c' \in \text{gPB}$   
**shows**  $\text{GE.mcalc2 } \sigma c'$   
**proof**(*cases*  $c' \in \text{Fax} \cup \text{Wax}$ )  
  **case** *True* **thus** *?thesis* **using** *nv2L-Wax nv2L-Fax*

```

  unfolding GE.mcalc2-iff by fastforce
next
  case False hence c': c' ∈ 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  $\Phi = gPB$  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  $\Phi = gPB$ 
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*)  $\longleftrightarrow$  *wtFsym f*  
| *TE-wtFsym* (*Tag*  $\sigma$ )  $\longleftrightarrow$  *True*  
| *TE-wtFsym* (*Wit*  $\sigma$ )  $\longleftrightarrow$   $\neg$  *isRes*  $\sigma$

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

**fun** *TE-resOf* **where**  
*TE-resOf* (*Oldf f*) = *resOf f*  
| *TE-resOf* (*Tag*  $\sigma$ ) =  $\sigma$   
| *TE-resOf* (*Wit*  $\sigma$ ) =  $\sigma$

**lemma** *countable-TE-wtFsym*: *countable* (*Collect TE-wtFsym*) (**is** *countable ?K*)

**proof**–

**let** *?F* =  $\lambda$  *ef*. *case ef of Oldf f*  $\Rightarrow$  *Inl f* | *Tag*  $\sigma$   $\Rightarrow$  *Inr (Inl*  $\sigma$ ) | *Wit*  $\sigma$   $\Rightarrow$  *Inr (Inr*  $\sigma$ )

**let** *?U* = (*UNIV::'tp set*)  $\langle + \rangle$  (*UNIV::'tp set*)

**let** *?L* = (*Collect wtFsym*)  $\langle + \rangle$  *?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*  $\subseteq$  *?L* **apply** *clarify* **by** (*case-tac ef, auto*)

**ultimately** **have**  $|?K| \leq_o |?L|$  **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-wtFsym* **by** *auto*

**context** *ProblemIkTpart*  
**begin**



**fun** *tNN* **where**

*tNN* (*Var* *x*) =  
  (*if unprot* (*tpOfV* *x*)  $\vee$  *protFw* (*tpOfV* *x*) then *Var* *x* else *Fn* (*Tag* (*tpOfV* *x*))  
  [*Var* *x*])  
|  
*tNN* (*Fn* *f* *Tl*) = (*if unprot* (*resOf* *f*)  $\vee$  *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*  $\equiv$  *map tL*

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

**definition** *lOfFax* *f* = *Fn* (*Tag* (*resOf* *f*)) [*rOfFax* *f*]

**definition** *Fax*  $\equiv$  {[*Pos* (*Eq* (*lOfFax* *f*) (*rOfFax* *f*))] | *f*. *wtFsym* *f*}

**definition** *rOfWax*  $\sigma$  = *Fn* (*Wit*  $\sigma$ ) []

**definition** *lOfWax*  $\sigma$  = *Fn* (*Tag*  $\sigma$ ) [*rOfWax*  $\sigma$ ]

**definition** *Wax*  $\equiv$  {[*Pos* (*Eq* (*lOfWax*  $\sigma$ ) (*rOfWax*  $\sigma$ ))] |  $\sigma$ .  $\neg$  *isRes*  $\sigma$   $\wedge$  *protFw*  $\sigma$ }

**definition** *tPB* = *tC* ‘  $\Phi \cup Fax \cup Wax$

**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]*:  $Ik.wt\ T \implies TE.wt\ (tNN\ T)$   
**by** (*induct T*, *auto simp add: list-all-iff*)

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

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

**lemma** *wtC-tC[simp]*:  $Ik.wtC\ c \implies TE.wtC\ (tC\ c)$   
**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]*:  $TE.tpOf\ (rOfWax\ \sigma) = \sigma$   
**unfolding** *rOfWax-def* **by** *simp*

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

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

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

**lemma** *wt-rOfWax[simp]*:  $\neg\ isRes\ \sigma \implies TE.wt\ (rOfWax\ \sigma)$   
**unfolding** *rOfWax-def* **by** *simp*

**lemma** *wt-lOfWax[simp]*:  $\neg\ isRes\ \sigma \implies TE.wt\ (lOfWax\ \sigma)$   
**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-Phi[simp]*:  $TE.wtPB\ (tC\ \Phi)$   
**using** *Ik.wt-Phi* **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**  $TE.wt (Fn (Tag \sigma) Tl)$   
**shows**  $\exists T. Tl = [T] \wedge TE.wt T \wedge tpOf T = \sigma$   
**using** *assms*  
**by** *simp (metis (hide-lams, no-types) list.inject list-all-simps(1) map-eq-Cons-conv neq-Nil-conv)*

**lemma** *tpOf-Tag*:  $TE.tpOf (Fn (Tag \sigma) Tl) = \sigma$  **by** *simp*

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

**lemma** *tpOf-Wit*:  $TE.tpOf (Fn (Wit \sigma) Tl) = \sigma$  **by** *simp*

**end**

## 8.2 Soundness

**context** *ModelIkTpart* **begin**

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

**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]*:  $TE.int \xi (tNN T) = Ik.int \xi T$   
**proof**(*induct T*)  
**case** ( $Fn f Tl$ )  
**hence**  $0$ :  $map (TE.int \xi \circ tNN) Tl = map (Ik.int \xi) Tl$   
**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  $\xi$   $\circ$  tNN) Tl = map (Ik.int  $\xi$ ) Tl*  
**unfolding** *list-eq-iff list-all-iff* **by** *auto*

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

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

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

**lemma** *tC-satC[simp]*: *TE.satC  $\xi$  (tC c)  $\longleftrightarrow$  Ik.satC  $\xi$  c*  
**unfolding** *TE.satC-def Ik.satC-def tC-def* **by** (*induct c, auto*)

**lemma** *tC- $\Phi$ -satPB[simp]*: *TE.satPB  $\xi$  (tC '  $\Phi$ )  $\longleftrightarrow$  Ik.satPB  $\xi$   $\Phi$*   
**unfolding** *TE.satPB-def Ik.satPB-def* **by** *auto*

**lemma** *Fax-Wax-satPB*:  
*TE.satPB  $\xi$  (Fax)  $\wedge$  TE.satPB  $\xi$  (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**  *$\Phi = tPB$*  **and** *intF = TE-intF*  
**apply** *standard* **using** *wtPB-tPB soundness* **by** *auto*

### 8.3 Completeness

**definition** *img B f a  $\equiv$  if  $a \in f ' B$  then  $a$  else  $f a$*

**lemma** *inImage-iimg[simp]*:  $a \in f \text{ ` } B \implies \text{iimg } B f a = a$   
**unfolding** *iimg-def* **by** *auto*

**lemma** *not-inImage-iimg[simp]*:  $a \notin f \text{ ` } B \implies \text{iimg } B f a = f a$   
**unfolding** *iimg-def* **by** *auto*

**lemma** *iimg[simp]*:  $a \in B \implies \text{iimg } B f (f a) = f a$   
**by** (*cases a \in f ` B, auto*)

**locale** *ProblemIkTpart-TEModel* =  
*Ik?* : *ProblemIkTpart wtFsym wtPsym arOf resOf parOf \Phi 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 \implies bool*  
**and** *wtPsym* :: *'psym \implies bool*  
**and** *arOf* :: *'fsym \implies 'tp list*  
**and** *resOf and parOf and \Phi and infTp and prot and protFw*  
**and** *eintT and eintF and eintP*

**context** *ProblemIkTpart-TEModel* **begin**

**definition**

*ntsem \sigma \equiv*  
*if unprot \sigma \vee protFw \sigma then id*  
*else iimg \{b. eintT \sigma b\} (eintF (Tag \sigma) o singl)*

**lemma** *unprot-ntsem[simp]*:  $\text{unprot } \sigma \vee \text{protFw } \sigma \implies \text{ntsem } \sigma a = a$   
**unfolding** *ntsem-def* **by** *simp*

**lemma** *protFw-ntsem[simp]*:  $\text{protFw } \sigma \implies \text{ntsem } \sigma a = a$   
**unfolding** *ntsem-def* **by** *simp*

**lemma** *inImage-ntsem[simp]*:  
 $a \in (\text{eintF } (\text{Tag } \sigma) \text{ o singl}) \text{ ` } \{b. \text{eintT } \sigma b\} \implies \text{ntsem } \sigma a = a$   
**unfolding** *ntsem-def* **by** *simp*

**lemma** *not-unprot-inImage-ntsem[simp]*:  
**assumes**  $\neg \text{unprot } \sigma$  **and**  $\neg \text{protFw } \sigma$  **and**  $a \notin (\text{eintF } (\text{Tag } \sigma) \text{ o singl}) \text{ ` } \{b. \text{eintT } \sigma b\}$   
**shows**  $\text{ntsem } \sigma a = \text{eintF } (\text{Tag } \sigma) [a]$   
**using** *assms unfolding ntsem-def by (simp add: singl-def)*

**lemma** *ntsem[simp]*:

$eintT \sigma b \implies ntsem \sigma (eintF (Tag \sigma) [b]) = eintF (Tag \sigma) [b]$   
**unfolding** *singl-def[symmetric]* **by** *simp*

**lemma** *eintT-ntsem*:

**assumes**  $a: eintT \sigma a$  **shows**  $eintT \sigma (ntsem \sigma a)$

**proof**(*cases unprot  $\sigma \vee protFw \sigma$* )

**case** *False* **note**  $unprot = False$  **show** *?thesis*

**proof**(*cases  $a \in (eintF (Tag \sigma) o singl) \text{ ' } \{b. eintT \sigma b\}$* )

**case** *False* **hence**  $1: ntsem \sigma a = eintF (Tag \sigma) [a]$  **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**

$intT \sigma a \equiv$

*if unprot  $\sigma$  then  $eintT \sigma a$*

*else if  $protFw \sigma$  then  $eintT \sigma a \wedge eintF (Tag \sigma) [a] = a$*

*else  $eintT \sigma a \wedge a \in (eintF (Tag \sigma) o singl) \text{ ' } \{b. eintT \sigma b\}$*

**definition**

$intF f al \equiv$

*if unprot ( $resOf f$ )  $\vee protFw (resOf f)$*

*then  $eintF (Oldf f) (map2 ntsem (arOf f) al)$*

*else  $eintF (Tag (resOf f)) [eintF (Oldf f) (map2 ntsem (arOf f) al)]$*

**definition**

$intP p al \equiv eintP p (map2 ntsem (parOf p) al)$

**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**  $1[simp]: \bigwedge i. i < length (arOf f) \implies tpOfV (xl!i) = (arOf f)!i$

**unfolding**  $xl-def$  **by** *auto*

**have**  $xl[simp]: distinct xl$  **unfolding**  $xl-def$  **using** *distinct-getVars* **by** *auto*

**define**  $\xi$  **where**  $\xi = pickE xl al$

**have**  $\xi: TE.wtE \xi$  **unfolding**  $\xi-def$  **apply**(*rule wtE-pickE*)

**using**  $al$  *list-all2-nthD* **by** *auto*

**have**  $[simp]: \bigwedge i. i < length (arOf f) \implies \xi (xl ! i) = al ! i$

**using**  $al$  **unfolding**  $\xi-def$  **by** (*auto simp: list-all2-length intro: pickE*)

**have**  $0: map (TE.int \xi) (getTvars (arOf f)) = al$

**apply**(*rule nth-equalityI*)

**using**  $al$  **by** (*auto simp: list-all2-length getTvars-def xl-def[symmetric]*)

**have**  $TE.satPB \xi Fax$  **using**  $TE.sat-\Phi[OF \xi]$  **unfolding** *tPB-def* **by** *simp*

hence  $TE.satC \xi [Pos (Eq (lOfFax f) (rOfFax f))]$   
 unfolding  $TE.satPB-def Fax-def$  using  $f$  by *auto*  
 hence  $TE.satA \xi (Eq (lOfFax f) (rOfFax f))$  unfolding  $TE.satC-def$  by *simp*  
 thus *?thesis* using *al* by (*simp add: lOfFax-def rOfFax-def 0*)  
 qed

**lemma** *TE-Wit*:

**assumes**  $\sigma: \neg isRes \sigma protFw \sigma$

**shows**  $eintF (Tag \sigma) [eintF (Wit \sigma) []] = eintF (Wit \sigma) []$

**proof**–

**define**  $\xi$  where  $\xi = pickE [] []$

**have**  $\xi: TE.wtE \xi$  unfolding  $\xi-def$  **apply**(*rule wtE-pickE*) by *auto*

**have**  $TE.satPB \xi Wax$  using  $TE.sat-\Phi[OF \xi]$  unfolding *tPB-def* by *simp*

  hence  $TE.satC \xi [Pos (Eq (lOfWax \sigma) (rOfWax \sigma))]$

  unfolding  $TE.satPB-def Wax-def$  using  $\sigma$  by *auto*

  hence  $TE.satA \xi (Eq (lOfWax \sigma) (rOfWax \sigma))$  unfolding  $TE.satC-def$  by *auto*

  thus *?thesis* unfolding  $TE.satA.simps lOfWax-def rOfWax-def$  by *simp*

qed

**lemma** *NE-intT-forget*:  $NE (intT \sigma)$

**proof**–

**obtain**  $b$  where  $b: eintT \sigma b$  using  $TE.NE-intT$  by *blast*

**show** *?thesis* **proof**(*cases unprot*  $\sigma$ )

**case** *True* **thus** *?thesis* using  $b$  unfolding *intT-def* by *auto*

**next**

**case** *False* **note**  $unprot = False$  **show** *?thesis*

**proof**(*cases protFw*  $\sigma$ )

**case** *True* **note**  $protFw = True$  **show** *?thesis* **proof**(*cases isRes*  $\sigma$ )

**case** *True* **then obtain**  $f$  where  $f: wtFsym f$  and  $\sigma: \sigma = resOf f$

      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**  $eintT \sigma a$  unfolding  $a-def \sigma$  using  $f al$

      by (*metis TE-arOf.simps TE-resOf.simps TE-wtFsym.simps TE.intF*)

**moreover have**  $eintF (Tag \sigma) [a] = a$  unfolding  $\sigma a-def$  using  $TE-Tag[OF f al]$ .

**ultimately show** *?thesis* using  $unprot protFw$  unfolding *intT-def* by *auto*

**next**

**case** *False*

**define**  $a$  where  $a = eintF (Wit \sigma) []$

**have**  $eintT \sigma a$  unfolding  $a-def$

      by (*metis False TE-arOf.simps TE-resOf.simps TE-wtFsym.simps TE.intF list-all2-NilR*)

**moreover have**  $eintF (Tag \sigma) [a] = a$  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  $\text{eintT } \sigma (\text{eintF } (\text{Tag } \sigma) [b])$ 
using  $b \text{ list-all2-Cons list-all2-NilL}$ 
by ( $\text{metis TE.intF TE-arOf.simps TE-resOf.simps TE-wtFsym.simps}$ )
hence  $\text{intT } \sigma (\text{eintF } (\text{Tag } \sigma) [b])$ 
unfolding  $\text{intT-def singl-def[abs-def]}$  using  $b \text{ False}$  by auto
thus ?thesis by blast
qed
qed
qed

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

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

**lemma** *unprot-inv*[*simp*]: *unprot*  $\sigma \vee$  *protFw*  $\sigma \implies$  *inv*  $\sigma$  *a* = *a*  
**unfolding** *inv-def* **by auto**

**lemma** *inv-inv-inImage*:

**assumes**  $\sigma$ :  $\neg$  *unprot*  $\sigma \neg$  *protFw*  $\sigma$

**and** *a*:  $a \in$  (*eintF* (*Tag*  $\sigma$ ) *o* *singl*) ‘ {*b*. *eintT*  $\sigma$  *b*}

**shows** *eintT*  $\sigma$  (*inv*  $\sigma$  *a*)  $\wedge$  *eintF* (*Tag*  $\sigma$ ) [*inv*  $\sigma$  *a*] = *a*

**proof**–

**obtain** *b* **where** *eintT*  $\sigma$  *b* **and** *eintF* (*Tag*  $\sigma$ ) [*b*] = *a*

**using** *a* **unfolding** *image-def singl-def*[*symmetric*] **by auto**

**thus** *?thesis* **using**  $\sigma$  **unfolding** *inv-def* **apply** *simp* **apply**(*rule someI-ex*) **by**

*auto*

**qed**

**lemmas** *inv*[*simp*] = *inv-inv-inImage*[*THEN conjunct1*]

**lemmas** *inv-inImage*[*simp*] = *inv-inv-inImage*[*THEN conjunct2*]

**term** *inv*

**definition** *eenv*  $\xi$  *x*  $\equiv$  *inv* (*tpOfV* *x*) ( $\xi$  *x*)

**lemma** *wt-eenv*:

**assumes**  $\xi$ : *Ik.wtE*  $\xi$  **shows** *TE.wtE* (*eenv*  $\xi$ )

**unfolding** *TE.wtE-def* **proof** *safe*

**fix** *x* **let**  $\sigma$  = *TE.tpOfV* *x*

**show** *eintT*  $\sigma$  (*eenv*  $\xi$  *x*) **proof**(*cases unprot*  $\sigma$ )

**case** *True* **note** *unprot* = *True*

**thus** *?thesis* **unfolding** *eenv-def* **by** (*metis*  $\xi$  *Ik.wtTE-intT intT-def unprot-inv*)

**next**

**case** *False* **note** *unprot* = *False* **show** *?thesis* **proof**(*cases protFw*  $\sigma$ )

**case** *True* **thus** *?thesis* **unfolding** *eenv-def* **using** *unprot*  $\xi$

**by** (*metis* *Ik.wtTE-intT intT-def unprot-inv*)

```

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
    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, safe)
  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**  
 $map2\ ntsem\ (map\ Ik.Ik.tpOf\ Tl)\ (map\ (Ik.int\ \xi)\ Tl) =$   
 $map\ (TE.int\ (eenv\ \xi)\ \circ\ tNN)\ Tl$   
**proof** –  
 {**fix**  $i$  **assume**  $i: i < length\ Tl$   
**hence**  $wt: Ik.Ik.wt\ (Tl!i)$  **using**  $Tl$  **unfolding**  $list-all-length$  **by**  $simp$   
**have**  $intT\ (Ik.Ik.tpOf\ (Tl!i))\ (Ik.int\ \xi\ (Tl!i))$  **using**  $Ik.wt-int[OF\ \xi\ wt]$ .  
 }  
**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**  $\xi: Ik.wtE\ \xi$   
**shows**  $TE.int\ (eenv\ \xi)\ (tT\ T) = Ik.int\ \xi\ T$   
**using**  $T$  **proof**( $induct\ T$ )  
**case**  $(Var\ x)$  **let**  $?\sigma = tpOfV\ x$  **show**  $?case$   
**proof**( $cases\ unprot\ ?\sigma$ )  
**case**  $False$  **note**  $unprot = False$   
**show**  $?thesis$  **proof**( $cases\ protFw\ ?\sigma$ )  
**case**  $True$  **thus**  $?thesis$  **using**  $unprot\ \xi$   
**unfolding**  $eenv-def\ Ik.wtE-def\ intT-def$  **by**  $simp$   
**next**  
**case**  $False$  **hence**  $\xi\ x \in (eintF\ (Tag\ ?\sigma)\ \circ\ singl)\ \{b.\ eintT\ ?\sigma\ b\}$   
**using**  $\xi\ unprot$  **unfolding**  $wtE-def\ intT-def\ singl-def[abs-def]$  **by**  $simp$   
**thus**  $?thesis$  **using**  $unprot$  **unfolding**  $eenv-def$  **using**  $False$  **by**  $simp$   
**qed**  
**qed**( $unfold\ eenv-def,\ simp$ )  
**next**  
**case**  $(Fn\ f\ Tl)$   
**let**  $?e\xi = eenv\ \xi$  **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**  $0: map2\ ntsem\ ?ar\ (map\ (Ik.int\ \xi)\ Tl) = map\ (TE.int\ ?e\xi\ \circ\ tNN)\ Tl$   
**unfolding**  $ar$  **apply**( $rule\ map-int-tNN[OF\ -\ \xi]$ ) **using**  $Fn$  **by**  $simp$   
**show**  $?case$  **apply**( $cases\ unprot\ ?r\ \vee\ protFw\ ?r$ )  
**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**  $\xi: Ik.wtE\ \xi$   
**shows**  
 $map2\ ntsem\ (map\ Ik.Ik.tpOf\ Tl)\ (map\ (Ik.int\ \xi)\ Tl) =$   
 $map\ (TE.int\ (eenv\ \xi)\ \circ\ tT)\ Tl$   
**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) ↔ 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) ↔ 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) ↔ 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**  $\Phi = tPB$   
**apply** *standard* **by** *auto*

**sublocale** *ProblemIkTpart* < *TE?* : *ProblemIk*  
**where** *wtFsym* = *TE-wtFsym* **and** *arOf* = *TE-arOf* **and** *resOf* = *TE-resOf*  
**and**  $\Phi = tPB$   
**proof**  
**fix**  $\sigma$  *eintT eintF eintP a* **assume**  $\sigma$ : *infTp*  $\sigma$   
**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*  
 $\Phi$  *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*  $\{a. ?TE-intT \sigma a\}$  **using** *infTp[OF*  $\sigma$  *MM* **]** .  
**moreover** **have**  $\{a. ?TE-intT \sigma a\} \subseteq \{a. eintT \sigma a\}$   
**using** *ProblemIkTpart-TEModel.intT-def*[*OF* *0*] **by** *auto*  
**ultimately show** *infinite*  $\{a. eintT \sigma a\}$  **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]*:  $\neg unprot \sigma \implies (\forall x \in TE.nvT (tT T). tpOfV x \neq \sigma)$   
**apply**(*induct T*) **by** *auto*

**lemma** *nvL-tL[simp]*:  $\neg unprot \sigma \implies (\forall x \in TE.nvL (tL l). tpOfV x \neq \sigma)$   
**apply**(*cases l*) **apply**(*rename-tac* [!] *atm*) **apply**(*case-tac* [!] *atm*) **by** *auto* (*metis nvT-t*)<sup>+</sup>

**lemma** *nvC-tC[simp]*:  $\neg unprot \sigma \implies (\forall x \in TE.nvC (tC c). tpOfV x \neq \sigma)$   
**unfolding** *tC-def* *TE.nvC-def* **apply** (*induct c*)  
**by** *auto* (*metis* (*full-types*) *nvL-tL*)<sup>+</sup>

**lemma** *unprot-nvT-t[simp]*:  
 $unprot (tpOfV x) \implies x \in TE.nvT (tT T) \longleftrightarrow x \in TE.nvT T$

**by** (*induct T, auto*)

**lemma** *tpL-nvT-tL[simp]*:

*unprot (tpOfV x)  $\implies x \in TE.nvL (tL l) \longleftrightarrow x \in TE.nvL l$*

**by** (*cases l, rename-tac [!] atm, case-tac [!] atm, auto*)

**lemma** *unprot-nvC-tC[simp]*:

*unprot (tpOfV x)  $\implies x \in TE.nvC (tC c) \longleftrightarrow x \in TE.nvC c$*

**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  $\notin TE.nvT (lOfWax \sigma')$   $x \notin TE.nvT (rOfWax \sigma')$*

**unfolding** *lOfFax-def rOfFax-def lOfWax-def rOfWax-def* **by** *auto*

**lemma** *nvC-Fax*: *c  $\in Fax \implies TE.nvC c = \{\}$*  **unfolding** *Fax-def TE.nvC-def*  
**by** *auto*

**lemma** *mcalc-Fax*: *c  $\in Fax \implies TE.mcalc \sigma c$*  **using** *nvC-Fax* **unfolding** *TE.mcalc-iff*  
**by** *auto*

**lemma** *nvC-Wax*: *c  $\in Wax \implies TE.nvC c = \{\}$*  **unfolding** *Wax-def TE.nvC-def*  
**by** *auto*

**lemma** *mcalc-Wax*: *c  $\in Wax \implies TE.mcalc \sigma c$*  **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**  $\Phi = tPB$

**proof**

**fix**  $\sigma c$  **assume** *c  $\in tPB$*

**thus** *TE.mcalc  $\sigma c$*  **unfolding** *tPB-def* **proof** *safe*

**fix** *d* **assume** *d: d  $\in \Phi$*  **thus** *TE.mcalc  $\sigma (tC d)$*

**using** *unprot-mcalc[OF - d]* **unfolding** *TE.mcalc-iff* **by** (*cases unprot  $\sigma$ , 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
 $\Phi$  = 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  $\Rightarrow$  bool
and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  nat
and parOf :: 'psym  $\Rightarrow$  nat
assumes countable-wtFsym: countable {f::'fsym. wtFsym f}
and countable-wtPsym: countable {p::'psym. wtPsym p}
begin

```

```

fun wt where
  wt (Var x)  $\longleftrightarrow$  True
  |
  wt (Fn f Tl)  $\longleftrightarrow$  wtFsym f  $\wedge$  list-all wt Tl  $\wedge$  arOf f = length Tl

```

```

lemma wt-induct[elim, consumes 1, case-names Var Fn, induct pred: wt]:
assumes T: wt T
and Var:  $\bigwedge x. \varphi$  (Var x)
and Fn:
 $\bigwedge f$  Tl.
  [[wtFsym f; list-all wt Tl; arOf f = length Tl; list-all  $\varphi$  Tl]
   $\Longrightarrow$   $\varphi$  (Fn f Tl)
shows  $\varphi$  T
proof –
  have wt T  $\longrightarrow$   $\varphi$  T
  apply (induct T) using Var Fn by (auto simp: list-all-iff)
  thus ?thesis using T by auto

```

**qed**

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

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

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

**lemma**  $wtSB-o$ :  
**assumes**  $1: wtSB \pi1$  **and**  $2: wtSB \pi2$   
**shows**  $wtSB (subst \pi1 o \pi2)$   
**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**  
 $intF: \llbracket wtFsym f; length \ al = arOf f; list-all D \ al \rrbracket \implies D (intF f \ al)$   
**and**  
 $dummy: parOf = parOf \wedge intF = intF \wedge intP = intP$   
**begin**

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

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



```

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

```

```

lemma wt-int:
assumes wtE: wtE  $\xi$  and wt-T: wt T
shows D (int  $\xi$  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  $\bigwedge x. x \in \text{vars } T \implies \xi 1 x = \xi 2 x$ 
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  $\xi$  o  $\rho$ ) T = int  $\xi$  (subst  $\rho$  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  $\xi$  o subst  $\rho$  = int (int  $\xi$  o  $\rho$ )
apply(rule ext) apply(subst comp-def) unfolding int-o[symmetric] ..

```

```

lemma wtE-o:
assumes 1: wtE  $\xi$  and 2: wtSB  $\rho$ 
shows wtE (int  $\xi$  o  $\rho$ )
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)  $\longleftrightarrow$  wtPsym p  $\wedge$  list-all wt Tl  $\wedge$  parOf p = length Tl

```

```

fun wtL where
wtL (Pos a)  $\longleftrightarrow$  wtA a
|
wtL (Neg a)  $\longleftrightarrow$  wtA a

```

**definition** wtC  $\equiv$  list-all wtL

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

**end**

**context** Struct **begin**

```

fun satA where
satA  $\xi$  (Eq T1 T2)  $\longleftrightarrow$  int  $\xi$  T1 = int  $\xi$  T2
|
satA  $\xi$  (Pr r Tl)  $\longleftrightarrow$  intP r (map (int  $\xi$ ) Tl)

```

```

fun satL where
satL  $\xi$  (Pos a)  $\longleftrightarrow$  satA  $\xi$  a
|
satL  $\xi$  (Neg a)  $\longleftrightarrow$   $\neg$  satA  $\xi$  a

```

**definition** satC  $\xi \equiv$  list-ex (satL  $\xi$ )

**definition** satPB  $\xi \Phi \equiv \forall c \in \Phi. satC \xi c$

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

**end**

### 9.3 Problems

```

locale Problem = Signature wtFsym wtPsym arOf parOf
for wtFsym and wtPsym
and arOf :: 'fsym  $\Rightarrow$  nat
and parOf :: 'psym  $\Rightarrow$  nat
+
fixes  $\Phi$  :: ('fsym, 'psym) prob
assumes wt- $\Phi$ : wtPB  $\Phi$ 

```

### 9.4 Models of a problem

```

locale Model =
  Problem wtFsym wtPsym arOf parOf  $\Phi$  +

```

```

    Struct wtFsym wtPsym arOf parOf D intF intP
  for wtFsym and wtPsym
  and arOf :: 'fsym  $\Rightarrow$  nat
  and parOf :: 'psym  $\Rightarrow$  nat
  and  $\Phi$  :: ('fsym, 'psym) prob
  and D :: 'univ  $\Rightarrow$  bool
  and intF :: 'fsym  $\Rightarrow$  'univ list  $\Rightarrow$  'univ
  and intP :: 'psym  $\Rightarrow$  'univ list  $\Rightarrow$  bool
  +
  assumes SAT: SAT  $\Phi$ 

end

theory CU
imports U
begin

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

locale Model = U.Model wtFsym wtPsym arOf parOf  $\Phi$  D intF intP
for wtFsym and wtPsym
and arOf :: 'fsym  $\Rightarrow$  nat
and parOf :: 'psym  $\Rightarrow$  nat
and  $\Phi$ 
and D :: univ  $\Rightarrow$  bool
and intF :: 'fsym  $\Rightarrow$  univ list  $\Rightarrow$  univ
and intP :: 'psym  $\Rightarrow$  univ list  $\Rightarrow$  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  $\Phi$ 
intT intF intP
for wtFsym :: 'fsym  $\Rightarrow$  bool and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf and parOf and intT and intF and intP and  $\Phi$ 
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  $\Longrightarrow$  wt T
by (induct T, auto simp add: list-all-iff)

```

```

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

```

```

lemma wtL[simp]: M.wtL l  $\Longrightarrow$  wtL l
apply(cases l) by auto

```

```

lemma wtC[simp]: M.wtC c  $\Longrightarrow$  wtC c
unfolding M.wtC-def wtC-def by (induct c, auto)

```

```

lemma wtPB[simp]: M.wtPB  $\Phi \Longrightarrow$  wtPB  $\Phi$ 
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  $\Rightarrow$  bool and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf and parOf and D and intF and intP
```

```
sublocale UM-Struct < M? : M.Struct where intT =  $\lambda \sigma. D$ 
  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  $\xi \Longrightarrow U.wtE$   $\xi$ 
unfolding U.wtE-def M.wtE-def by auto
```

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

```
lemma int-o-e[simp]: U.int  $\xi = M.int$   $\xi$ 
unfolding int-e fun-eq-iff by simp
```

```
lemma satA-e[simp]: U.satA  $\xi at \longleftrightarrow M.satA$   $\xi at$ 
by (cases at) auto
```

```
lemma satL-e[simp]: U.satL  $\xi l \longleftrightarrow M.satL$   $\xi l$ 
apply(cases l) by auto
```

```
lemma satC-e[simp]: U.satC  $\xi c \longleftrightarrow M.satC$   $\xi c$ 
unfolding M.satC-def U.satC-def by (induct c, simp-all)
```

```
lemma satPB-e[simp]: U.satPB  $\xi \Phi \longleftrightarrow M.satPB$   $\xi \Phi$ 
unfolding M.satPB-def U.satPB-def by auto
```

**theorem** *completeness*:

**assumes** *U.SAT*  $\Phi$  **shows** *M.SAT*  $\Phi$

**using** *assms* **unfolding** *M.SAT-def satPB-e U.SAT-def* **by** *auto*

**end**

```
locale UM-Model =
  M-Problem wtFsym wtPsym arOf resOf parOf  $\Phi$  +
  UM-Struct wtFsym wtPsym arOf resOf parOf D intF intP +
  CU.Model wtFsym wtPsym length o arOf length o parOf  $\Phi$ 
```

```

      D intF intP
for wtFsym :: 'fsym  $\Rightarrow$  bool and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list
and resOf and parOf and  $\Phi$  and D and intF and intP
begin

```

```

theorem M-U-completeness: MModel ( $\lambda\sigma::'tp.$  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 =  $\lambda \sigma.$  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  $\xi \Longrightarrow$  F.wtE  $\xi$ 
unfolding U.wtE-def F.wtE-def by auto

```

```

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

```

```

lemma int-o-e[simp]: U.int  $\xi$  = F.int  $\xi$ 
unfolding fun-eq-iff by auto

```

```

lemma satA-e[simp]: U.satA  $\xi$  at  $\longleftrightarrow$  F.satA  $\xi$  at
by (cases at) auto

```

```

lemma satL-e[simp]: U.satL  $\xi$  l  $\longleftrightarrow$  F.satL  $\xi$  l
by (cases l) auto

```

```

lemma satC-e[simp]: U.satC  $\xi$  c  $\longleftrightarrow$  F.satC  $\xi$  c
unfolding F.satC-def U.satC-def by (induct c, simp-all)

```

```

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

```

```

theorem soundness: U.SAT  $\Phi$ 
unfolding U.SAT-def using sat- $\Phi$  satPB-e by auto

lemma U-Model:
CU.Model wtFsym wtPsym (length o arOf) (length o parOf)  $\Phi$  (intT any) intF
intP
by standard (rule wtPB[OF wt- $\Phi$ ], 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 o arOf) (length o parOf)  $\Phi$ 
(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  $\Phi = \Phi$  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,  $\Phi$ , 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*  $wtFsym$   $wtPsym$   $arOf$   $resOf$   $parOf$   $\Phi$   $infTp$   $tpD$   $tpFD$ , stating that:

—  $(wtFsym, wtPsym, arOf, resOf, parOf)$  form a countable signature

—  $\Phi$  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)  $CM.Model\ wtFsym\ wtPsym\ arOf\ resOf\ parOf\ \Phi\ intT\ intF\ intP$  says that  $(intT, intF, intP)$  is a model for  $\Phi$  (hence  $\Phi$  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 taken 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**  $wtFsym :: 'fsym \Rightarrow bool$  **and**  $wtPsym :: 'psym \Rightarrow bool$   
**and**  $arOf :: 'fsym \Rightarrow 'tp\ list$  **and**  $resOf :: 'fsym \Rightarrow 'tp$  **and**  $parOf :: 'psym \Rightarrow 'tp\ list$   
**and**  $\Phi :: ('fsym, 'psym)\ prob$  **and**  $infTp :: 'tp \Rightarrow bool$   
**and**  $tpD :: 'tp \Rightarrow bool$  **and**  $tpFD :: 'tp \Rightarrow bool$   
**and**  $intT :: 'tp \Rightarrow univ \Rightarrow bool$   
**and**  $intF :: 'fsym \Rightarrow univ\ list \Rightarrow univ$  **and**  $intP :: 'psym \Rightarrow univ\ list \Rightarrow bool$

— The problem translation:

— First the addition of tags (“TE” stands for “tag encoding”):

**defines**  $TE-wtFsym \equiv ProblemIkTpart.TE-wtFsym\ wtFsym\ resOf$   
**and**  $TE-arOf \equiv ProblemIkTpart.TE-arOf\ arOf$   
**and**  $TE-resOf \equiv ProblemIkTpart.TE-resOf\ resOf$   
**defines**  $TE-\Phi \equiv ProblemIkTpart.tPB\ wtFsym\ arOf\ resOf\ \Phi\ tpD\ tpFD$

— Then the deletion of types (“U” stands for “untyped”):

**and**  $U-arOf \equiv length \circ TE-arOf$   
**and**  $U-parOf \equiv length \circ parOf$   
**defines**  $U-\Phi \equiv TE-\Phi$

— The forward model translation:

— First, using monotonicity, we build an infinite model of  $\Phi$  (“I” stands for “infinite”):

**defines**  $intTI \equiv MonotProblem.intTI\ TE-wtFsym\ wtPsym\ TE-arOf\ TE-resOf\ parOf\ TE-\Phi$   
**and**  $intFI \equiv MonotProblem.intFI\ TE-wtFsym\ wtPsym\ TE-arOf\ TE-resOf\ parOf\ TE-\Phi$   
**and**  $intPI \equiv MonotProblem.intPI\ TE-wtFsym\ wtPsym\ TE-arOf\ TE-resOf\ parOf\ TE-\Phi$

— Then, by isomorphic transfer of the latter model, we build a model of  $\Phi$  that has all types interpreted as  $univ$  (“F” stands for “full”):

**defines**  $intFF \equiv InfModel.intFF\ TE-arOf\ TE-resOf\ intTI\ intFI$   
**and**  $intPF \equiv InfModel.intPF\ parOf\ intTI\ intPI$

— Then we build a model of  $U-\Phi$ :

**defines**  $U\text{-int}T \equiv \text{InfModel.int}TF \text{ (any::'tp)}$

— Assumptions of the theorem:

**assumes**

$P$ :  $\text{ProblemIkTpart wtFsym wtPsym arOf resOf parOf } \Phi \text{ infTp tpD tpFD}$

**and**  $M$ :  $\text{CM.Model wtFsym wtPsym arOf resOf parOf } \Phi \text{ intT intF intP}$

— Conclusion of the theorem:

**shows**  $CU.Model TE\text{-wtFsym wtPsym } U\text{-arOf } U\text{-parOf } U\text{-}\Phi \text{ } U\text{-int}T \text{ intFF intPF}$

**unfolding**  $U\text{-arOf-def } U\text{-parOf-def } U\text{-}\Phi\text{-def}$

**unfolding**  $U\text{-int}T\text{-def intTI-def intFI-def intPI-def intFF-def intPF-def}$

**apply**(rule  $M\text{-MonotModel.M-U-soundness}$ )

**unfolding**  $M\text{-MonotModel-def MonotModel-def}$  **apply safe**

**unfolding**  $TE\text{-wtFsym-def TE-arOf-def TE-resOf-def TE-}\Phi\text{-def}$

**apply**(rule  $\text{ProblemIkTpart.T-monotonic}$ )

**apply**(rule  $P$ )

**apply**(rule  $\text{ModelIkTpart.T-soundness}$ ) **unfolding**  $\text{ModelIkTpart-def}$  **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,  $\text{ProblemIkTpartG wtFsym wtPsym arOf resOf parOf } \Phi \text{ infTp tpD tpFD tpCD}$ , which consists of  $\text{ProblemIkTpart wtFsym wtPsym arOf resOf parOf } \Phi \text{ infTp tpD tpFD}$  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 taken 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**  $\text{guards-soundness}$ :

**fixes**  $\text{wtFsym} :: \text{'fsym} \Rightarrow \text{bool}$  **and**  $\text{wtPsym} :: \text{'psym} \Rightarrow \text{bool}$

**and**  $\text{arOf} :: \text{'fsym} \Rightarrow \text{'tp list}$  **and**  $\text{resOf} :: \text{'fsym} \Rightarrow \text{'tp}$  **and**  $\text{parOf} :: \text{'psym} \Rightarrow \text{'tp list}$

**and**  $\Phi :: (\text{'fsym}, \text{'psym}) \text{ prob}$  **and**  $\text{infTp} :: \text{'tp} \Rightarrow \text{bool}$

**and**  $\text{tpD} :: \text{'tp} \Rightarrow \text{bool}$  **and**  $\text{tpFD} :: \text{'tp} \Rightarrow \text{bool}$  **and**  $\text{tpCD} :: \text{'tp} \Rightarrow \text{bool}$

```

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:  $(D, \text{eint}F, \text{eint}P)$

### Completeness of the tag encodings

**theorem** *tags-completeness*:

```
fixes wtFsym :: 'fsym  $\Rightarrow$  bool and wtPsym :: 'psym  $\Rightarrow$  bool
and arOf :: 'fsym  $\Rightarrow$  'tp list and resOf :: 'fsym  $\Rightarrow$  'tp and parOf :: 'psym  $\Rightarrow$  'tp
list
and  $\Phi$  :: ('fsym, 'psym) prob and infTp :: 'tp  $\Rightarrow$  bool
and tpD :: 'tp  $\Rightarrow$  bool and tpFD :: 'tp  $\Rightarrow$  bool

and D :: univ  $\Rightarrow$  bool
and eintF :: ('fsym, 'tp) T.efsym  $\Rightarrow$  univ list  $\Rightarrow$  univ
and eintP :: 'psym  $\Rightarrow$  univ list  $\Rightarrow$  bool
```

— The problem translation (the same as in the case of soundness):

```
defines TE-wtFsym  $\equiv$  ProblemIkTpart.TE-wtFsym wtFsym resOf
and TE-arOf  $\equiv$  ProblemIkTpart.TE-arOf arOf
and TE-resOf  $\equiv$  ProblemIkTpart.TE-resOf resOf
defines TE- $\Phi$   $\equiv$  ProblemIkTpart.tPB wtFsym arOf resOf  $\Phi$  tpD tpFD
and U-arOf  $\equiv$  length  $\circ$  TE-arOf
and U-parOf  $\equiv$  length  $\circ$  parOf
defines U- $\Phi$   $\equiv$  TE- $\Phi$ 
```

— The backward model translation:

```
defines intT  $\equiv$  ProblemIkTpart-TEModel.intT tpD tpFD ( $\lambda\sigma::$ 'tp. D) eintF
and intF  $\equiv$  ProblemIkTpart-TEModel.intF arOf resOf tpD tpFD ( $\lambda\sigma::$ 'tp. D) eintF
and intP  $\equiv$  ProblemIkTpart-TEModel.intP parOf tpD tpFD ( $\lambda\sigma::$ 'tp. D) eintF
eintP
```

**assumes**

```
P: ProblemIkTpart wtFsym wtPsym arOf resOf parOf  $\Phi$  infTp tpD tpFD and
M: CU.Model TE-wtFsym wtPsym (length  $\circ$  TE-arOf)
(length  $\circ$  parOf) TE- $\Phi$  D eintF eintP
```

**shows** CM.Model wtFsym wtPsym arOf resOf parOf  $\Phi$  intT intF intP

**proof** –

```
have UM: UM-Model TE-wtFsym wtPsym TE-arOf TE-resOf parOf TE- $\Phi$  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

```