A wp Calculus for a Preferential Computations
—
Mechanisation in Isabelle/HOL

Frank Zeyda

August 1, 2022

Abstract

This document accompanies the paper *bGSL: An Imperative Language for Specification and Refinement of Backtracking Programs* by Steve Dunne, João F. Ferreira, Alexandra Mendes, Campbell Ritchie, Bill Stoddart and Frank Zeyda, accepted for publication at Journal of Logical and Algebraic Methods in Programming (JLAMP).

The mechanisation reported here is ongoing work with additional contributions not mentioned in the JLAMP paper mentioned above. We plan to publish this development separately in the future.

Contents

1 Preliminaries

```
theory Preliminaries
imports Bounded_Set
begin
```

1.1 Bounded Sets

Sets with bounds that are large enough by constructions.

```
definition mk_bset :: "'a set ⇒ 'a set['a set]" ("⦃_⦄") where
"mk_bset = Abs_bset"

lift_definition bsubset :: "'a set['k] ⇒ 'a set['k] ⇒ bool"
is "op ⊆" .

lift_definition bimage :: "('a ⇒ 'b) ⇒ 'a set['k] ⇒ 'b set['k]"
is "image" using card_of_image ordLeq_ordLess_trans by blast

notation bmember (infix "∈_b" 50)
notation bsubset (infix "⊆_b" 50)
notation bimage (infixr "‘_b" 90)
```

1.2 Theorems

```
thm card_of_ordLeqI
thm card_of_ordLess
thm surj_imp_ordLeq
```

The following theorems seem not to be needed at the moment.

```
theorem bset_ordLeq_mono_lemma :
"|B| ≤o |C| ⟹ |A| <o natLeq +c |B| ⟹ |A| <o natLeq +c |C|"
using csum_mono2 ordLess_ordLeq_trans by blast

theorem bset_ordLess_mono_lemma :
"|B| <o |C| ⟹ |A| <o natLeq +c |B| ⟹ |A| <o natLeq +c |C|"
using csum_mono2 ordLess_imp_ordLeq ordLess_ordLeq_trans by blast

theorem mk_bset_inverses [simp] :
"set_bset (mk_bset s) = s"
"mk_bset (set_bset b) = b"
apply (unfold mk_bset_def)
apply (metis Collect_mem_eq bCollect.abs_eq bCollect.rep_eq)
apply (simp add: set_bset_inverse)
done

theorem mk_bset_inject [simp] :
"(mk_bset s) = (mk_bset t) ⟹ s = t"
apply (metis mk_bset_inverses(1))
done
end
```

2 Gödel-Dummet Logic

```
theory GD3
imports Main "~~/src/HOL/Eisbach/Eisbach"
begin
```

2.1 Logic Type

We formalised a three-valued Gödel–Dummett logic with `false` being 0, `true` being 0.5, and `super` being 1. Our intuitive interpretation is that both, `true` and `super` are notions of truth. They differ in that `super` represents vacuous truth that arises from an implication whose antecedent does not hold.

```
datatype gd3 =
  GD3_False | GD3_True | GD3_Super

notation GD3_True ("true")
notation GD3_False ("false")
notation GD3_Super ("super")
```

Order of Truth Value

```
instantiation gd3 :: linorder
begin
fun less_eq_gd3 :: "gd3 ⇒ gd3 ⇒ bool" where
"false ≤ false = True" |
"false ≤ true = True" |
"false ≤ super = True" |
"true ≤ false = False" |
"true ≤ true = True" |
"true ≤ super = True" |
"super ≤ false = False" |
"super ≤ true = False" |
"super ≤ super = True"
definition less_gd3 :: "gd3 ⇒ gd3 ⇒ bool" where
"less_gd3 P Q ⟷ (P ≤ Q) ∧ (P ≠ Q)"
instance
apply (intro_classes)
apply (unfold less_gd3_def)
— Subgoal 1
apply (induct_tac x, induct_tac[!] y) [1]
apply (simp_all) [9]
— Subgoal 2
apply (induct_tac x) [1]
apply (simp_all) [3]
— Subgoal 3
apply (atomize (full))
apply (rule allI)+
apply (induct_tac x, induct_tac[!] y, induct_tac[!] z) [1]
apply (simp_all) [27]
— Subgoal 4
apply (atomize (full))
apply (rule allI)+
apply (induct_tac x, induct_tac[!] y) [1]
apply (simp_all) [9]
— Subgoal 5
apply (atomize (full))
```

```
apply (rule allI)+
apply (induct_tac x, induct_tac[!] y) [1]
apply (simp_all) [9]
done
end
```

2.2 Meta-logical Operators

The following three operators provide different ways of interpreting a GD3 predicate as a Boolean predicate. The `GD3_Shriek` operator is the conventional interpretation in Gödel Logic — only `super` is consider to be `True`. The `GD3_Prop` operator associates both `true` and `super` with HOL truth. Lastly, the `GD3_Truth` operator interprets `true` as being true but, unlike `GD3_Prop`, not `super`.

```
definition GD3_Shriek :: "gd3 ⇒ bool" ("_!" [1000] 1000) where
"GD3_Shriek P = (P = GD3_Super)"

theorem GD3_Shriek_simps [simp] :
"GD3_Shriek GD3_False = False"
"GD3_Shriek GD3_True = False"
"GD3_Shriek GD3_Super = True"
apply (unfold GD3_Shriek_def)
apply (simp_all)
done

definition GD3_Prop :: "gd3 ⇒ bool" ("⌊_⌋") where
"GD3_Prop P = (P = GD3_True ∨ P = GD3_Super)"

theorem GD3_Prop_simps [simp] :
"GD3_Prop GD3_False = False"
"GD3_Prop GD3_True = True"
"GD3_Prop GD3_Super = True"
apply (unfold GD3_Prop_def)
apply (simp_all)
done

definition GD3_Truth :: "gd3 ⇒ bool" ("⟨_⟩") where
"GD3_Truth P = (P = GD3_True)"

theorem GD3_Truth_simps [simp] :
"GD3_Truth GD3_False = False"
"GD3_Truth GD3_True = True"
"GD3_Truth GD3_Super = False"
apply (unfold GD3_Truth_def)
apply (simp_all)
done
```

The operators `GD3_Lift` and `GD3_Elate` lift a HOL predicate into an GD3 predicate. The difference between them is in the interpretation of `True`, namely as either `true` or `super`.

```
definition GD3_Lift :: "bool ⇒ gd3" ("_↑" [1000] 1000) where
"GD3_Lift P = (if P then GD3_True else GD3_False)"

theorem GD3_Lift_simps :
"GD3_Lift False = GD3_False"
"GD3_Lift True = GD3_True"
```

```
apply (unfold GD3_Lift_def)
apply (simp_all)
done

definition GD3_Elate :: "bool ⇒ gd3" ("_⇑" [1000] 1000) where
"GD3_Elate P = (if P then GD3_Super else GD3_False)"

theorem GD3_Elate_simps :
"GD3_Elate False = GD3_False"
"GD3_Elate True = GD3_Super"
apply (unfold GD3_Elate_def)
apply (simp_all)
done
```

Congruence is a weak notion of equivalence between GD3 predicates that does not distinguish between `true` and `super`, using the interpretation `GD3_Prop` defined above.

```
definition GD3_Cong :: "gd3 ⇒ gd3 ⇒ bool" (infix "≅" 50) where
"GD3_Cong P Q = (⌊P⌋ ⟷ ⌊Q⌋)"

theorem GD3_Cong_simps [simp] :
"GD3_Cong GD3_False GD3_False = True"
"GD3_Cong GD3_False GD3_True = False"
"GD3_Cong GD3_False GD3_Super = False"
"GD3_Cong GD3_True GD3_False = False"
"GD3_Cong GD3_True GD3_True = True"
"GD3_Cong GD3_True GD3_Super = True"
"GD3_Cong GD3_Super GD3_False = False"
"GD3_Cong GD3_Super GD3_True = True"
"GD3_Cong GD3_Super GD3_Super = True"
apply (unfold GD3_Cong_def)
apply (unfold GD3_Prop_def)
apply (simp_all)
done
```

2.3 Logical Connectives

We remark that all connectives have their standard meaning as in GD3.

```
definition GD3_And :: "gd3 ⇒ gd3 ⇒ gd3" where
"GD3_And = min"

theorem GD3_And_simps [simp] :
"GD3_And GD3_False GD3_False = GD3_False"
"GD3_And GD3_False GD3_True = GD3_False"
"GD3_And GD3_False GD3_Super = GD3_False"
"GD3_And GD3_True GD3_False = GD3_False"
"GD3_And GD3_True GD3_True = GD3_True"
"GD3_And GD3_True GD3_Super = GD3_True"
"GD3_And GD3_Super GD3_False = GD3_False"
"GD3_And GD3_Super GD3_True = GD3_True"
"GD3_And GD3_Super GD3_Super = GD3_Super"
apply (unfold GD3_And_def)
apply (unfold min_def)
apply (simp_all)
done
```

```
definition GD3_Or :: "gd3 ⇒ gd3 ⇒ gd3" where
"GD3_Or = max"

theorem GD3_Or_simps [simp] :
"GD3_Or GD3_False GD3_False = GD3_False"
"GD3_Or GD3_False GD3_True = GD3_True"
"GD3_Or GD3_False GD3_Super = GD3_Super"
"GD3_Or GD3_True GD3_False = GD3_True"
"GD3_Or GD3_True GD3_True = GD3_True"
"GD3_Or GD3_True GD3_Super = GD3_Super"
"GD3_Or GD3_Super GD3_False = GD3_Super"
"GD3_Or GD3_Super GD3_True = GD3_Super"
"GD3_Or GD3_Super GD3_Super = GD3_Super"
apply (unfold GD3_Or_def)
apply (unfold max_def)
apply (simp_all)
done

definition GD3_Imp :: "gd3 ⇒ gd3 ⇒ gd3" where
"GD3_Imp P Q = (if P ≤ Q then GD3_Super else Q)"

theorem GD3_Imp_simps [simp] :
"GD3_Imp GD3_False GD3_False = GD3_Super"
"GD3_Imp GD3_False GD3_True = GD3_Super"
"GD3_Imp GD3_False GD3_Super = GD3_Super"
"GD3_Imp GD3_True GD3_False = GD3_False"
"GD3_Imp GD3_True GD3_True = GD3_Super"
"GD3_Imp GD3_True GD3_Super = GD3_Super"
"GD3_Imp GD3_Super GD3_False = GD3_False"
"GD3_Imp GD3_Super GD3_True = GD3_True"
"GD3_Imp GD3_Super GD3_Super = GD3_Super"
apply (unfold GD3_Imp_def)
apply (simp_all)
done

definition GD3_Iff :: "gd3 ⇒ gd3 ⇒ gd3" where
"GD3_Iff P Q = GD3_And (GD3_Imp P Q) (GD3_Imp Q P)"

theorem GD3_Iff_simps [simp] :
"GD3_Iff GD3_False GD3_False = GD3_Super"
"GD3_Iff GD3_False GD3_True = GD3_False"
"GD3_Iff GD3_False GD3_Super = GD3_False"
"GD3_Iff GD3_True GD3_False = GD3_False"
"GD3_Iff GD3_True GD3_True = GD3_Super"
"GD3_Iff GD3_True GD3_Super = GD3_True"
"GD3_Iff GD3_Super GD3_False = GD3_False"
"GD3_Iff GD3_Super GD3_True = GD3_True"
"GD3_Iff GD3_Super GD3_Super = GD3_Super"
apply (unfold GD3_Iff_def)
apply (simp_all)
done

theorem GD3_Iff_eq :
"(GD3_Iff P Q)! ⟷ P = Q"
apply (induct_tac P, induct_tac[!] Q)
```

```
apply (simp_all)
done

definition GD3_Not :: "gd3 ⇒ gd3" where
"GD3_Not P = (GD3_Imp P GD3_False)"

theorem GD3_Not_simps [simp] :
"GD3_Not GD3_False = GD3_Super"
"GD3_Not GD3_True = GD3_False"
"GD3_Not GD3_Super = GD3_False"
apply (unfold GD3_Not_def)
apply (simp_all)
done
```

The 'or-else' operator `P ▷ Q` is used to define the wp effect of preference. It is a novel construct of our encoding and not present in GD3. Operationally, it takes the truth value of its left-hand predicate `P` unless `P` equals to `super`. If so, it takes the truth value of its right-hand predicate `Q`. We notice that `P ▷ Q` is not monotonic in the first operand, although it is monotonic in the second one.

```
definition GD3_Orelse :: "gd3 ⇒ gd3 ⇒ gd3" where
"GD3_Orelse P Q = (if P! then Q else P)"

theorem GD3_Orelse_simps [simp] :
"GD3_Orelse GD3_False GD3_False = GD3_False"
"GD3_Orelse GD3_False GD3_True = GD3_False"
"GD3_Orelse GD3_False GD3_Super = GD3_False"
"GD3_Orelse GD3_True GD3_False = GD3_True"
"GD3_Orelse GD3_True GD3_True = GD3_True"
"GD3_Orelse GD3_True GD3_Super = GD3_True"
"GD3_Orelse GD3_Super GD3_False = GD3_False"
"GD3_Orelse GD3_Super GD3_True = GD3_True"
"GD3_Orelse GD3_Super GD3_Super = GD3_Super"
apply (unfold GD3_Orelse_def)
apply (simp_all)
done
```

2.4 Quantifiers

We note that we cannot use HOL's `Min` and `Max` functions to define the semantics of GD3 quantifiers since those two operators by default only apply to finite sets.

```
definition GD3_Forall :: "('a ⇒ gd3) ⇒ gd3" where
"GD3_Forall P = (if (∀x. (P x)!) then GD3_Super else (∀x. ⌊P x⌋)↑)"

definition GD3_Exists :: "('a ⇒ gd3) ⇒ gd3" where
"GD3_Exists P = (if (∃x. (P x)!) then GD3_Super else (∃x. ⌊P x⌋)↑)"
```

2.5 Predicate Lattice

Is there any use in instantiating a `complete_lattice` too?

```
instantiation gd3 :: lattice
begin
definition bot_gd3 :: "gd3" where
"bot_gd3 = GD3_False"
```

```
definition top_gd3 :: "gd3" where
"top_gd3 = GD3_Super"
definition inf_gd3 :: "gd3 ⇒ gd3 ⇒ gd3" where
"inf_gd3 = GD3_And"
definition sup_gd3 :: "gd3 ⇒ gd3 ⇒ gd3" where
"sup_gd3 = GD3_Or"
instance
apply (intro_classes)
apply (unfold bot_gd3_def top_gd3_def inf_gd3_def sup_gd3_def)
— Subgoal 1
apply (induct_tac x, induct_tac[!] y) [1]
apply (simp_all) [9]
— Subgoal 2
apply (induct_tac x, induct_tac[!] y) [1]
apply (simp_all) [9]
— Subgoal 3
apply (atomize (full))
apply (rule allI)+
apply (induct_tac x, induct_tac[!] y, induct_tac[!] z) [1]
apply (simp_all) [27]
— Subgoal 4
apply (induct_tac x, induct_tac[!] y) [1]
apply (simp_all) [9]
— Subgoal 5
apply (induct_tac x, induct_tac[!] y) [1]
apply (simp_all) [9]
— Subgoal 6
apply (atomize (full))
apply (rule allI)+
apply (induct_tac x, induct_tac[!] y, induct_tac[!] z) [1]
apply (simp_all) [27]
done
end
```

2.6 Predicate Parser

Note that {_} act as escape quotes into the HOL parser.

```
nonterminal "gd3term"

no_notation GD3_Prop ("⌊_⌋")
no_notation GD3_Truth ("⟨_⟩")
no_notation GD3_Shriek ("_!" [1000] 1000)

syntax "_gd3_top" :: "gd3term ⇒ gd3" ("‘_‘")
syntax "_gd3_prop" :: "gd3term ⇒ bool" ("⌊_⌋")
syntax "_gd3_truth" :: "gd3term ⇒ bool" ("⟨_⟩")
syntax "_gd3_shriek" :: "gd3term ⇒ bool" ("_!" [1000] 1000)
syntax "_gd3_idt" :: "idt ⇒ gd3term" ("_")
syntax "_gd3_appl" :: "term ⇒ cargs ⇒ gd3term" ("(1_/ _)" [1000, 1000] 999)
syntax "_gd3_term" :: "gd3 ⇒ gd3term" ("{_}")
syntax "_gd3_lift" :: "bool ⇒ gd3term" ("_↑" [1000] 1000)
syntax "_gd3_elate" :: "bool ⇒ gd3term" ("_⇑" [1000] 1000)
syntax "_gd3_true" :: "gd3term" ("true")
syntax "_gd3_false" :: "gd3term" ("false")
syntax "_gd3_super" :: "gd3term" ("super")
```

```
syntax "_gd3_not" :: "gd3term ⇒ gd3term" ("¬ _" [40] 40)
syntax "_gd3_and" :: "gd3term ⇒ gd3term ⇒ gd3term" (infixr "∧" 35)
syntax "_gd3_or" :: "gd3term ⇒ gd3term ⇒ gd3term" (infixr "∨" 30)
syntax "_gd3_imp" :: "gd3term ⇒ gd3term ⇒ gd3term" (infixr "⇒" 25)
syntax "_gd3_iff" :: "gd3term ⇒ gd3term ⇒ gd3term" (infixr "⇔" 20)
syntax "_gd3_orelse" :: "gd3term ⇒ gd3term ⇒ gd3term" (infixr "▷" 27)
syntax "_gd3_forall" :: "idts ⇒ gd3term ⇒ gd3term" ("(3∀_./ _)" 10)
syntax "_gd3_exists" :: "idts ⇒ gd3term ⇒ gd3term" ("(3∃_./ _)" 10)
syntax "_gd3_equals" :: "gd3term ⇒ gd3term ⇒ gd3term" (infix "=" 50)
syntax "_gd3_braces" :: "gd3term ⇒ gd3term" ("'(_')")

translations "_gd3_top p" ⇀ "p"
translations "_gd3_prop p" ⇌ "(CONST GD3_Prop) p"
translations "_gd3_truth p" ⇌ "(CONST GD3_Truth) p"
translations "_gd3_shriek p" ⇌ "(CONST GD3_Shriek) p"
translations "_gd3_idt x" ⇀ "x"
translations "_gd3_appl f a" ⇀ "f a"
translations "_gd3_appl f (_args a b)" ⇀ "_gd3_appl (f a) b"
translations "_gd3_term t" ⇀ "t"
translations "_gd3_lift t" ⇀ "(CONST GD3_Lift) t"
translations "_gd3_elate t" ⇀ "(CONST GD3_Elate) t"
translations "_gd3_true" ⇌ "(CONST GD3_True)"
translations "_gd3_false" ⇌ "(CONST GD3_False)"
translations "_gd3_super" ⇌ "(CONST GD3_Super)"
translations "_gd3_not p" ⇌ "(CONST GD3_Not) p"
translations "_gd3_and p q" ⇌ "(CONST GD3_And) p q"
translations "_gd3_or p q" ⇌ "(CONST GD3_Or) p q"
translations "_gd3_imp p q" ⇌ "(CONST GD3_Imp) p q"
translations "_gd3_iff p q" ⇌ "(CONST GD3_Iff) p q"
translations "_gd3_orelse p q" ⇌ "(CONST GD3_Orelse) p q"
translations "_gd3_forall x p" ⇌ "(CONST GD3_Forall) (λx. p)"
translations "_gd3_exists x p" ⇌ "(CONST GD3_Exists) (λx. p)"
translations "_gd3_equals p q" ⇀ "p = q"
translations "_gd3_braces p" ⇀ "p"
```

Avoid eta-contraction when printing GD3 quantifiers.

```
print_translation {*
  [Syntax_Trans.preserve_binder_abs_tr'
    @{const_syntax "GD3_Forall"} @{syntax_const "_gd3_forall"}]
*}

print_translation {*
  [Syntax_Trans.preserve_binder_abs_tr'
    @{const_syntax "GD3_Exists"} @{syntax_const "_gd3_exists"}]
*}
```

2.7 Homomorphism

```
theorem GD3_Truth_elim [simp] :
"⟨P⟩ = (⌊P⌋ ∧ ¬ P!)"
apply (unfold GD3_Truth_def)
apply (unfold GD3_Prop_def)
apply (unfold GD3_Shriek_def)
apply (auto)
done
```

```
theorem GD3_Cong_Prop :
"P ≅ Q = (⌊P⌋ = ⌊Q⌋)"
apply (unfold GD3_Cong_def)
apply (unfold GD3_Prop_def)
apply (simp)
done

theorem GD3_Lift_Prop [simp] :
"⌊P↑⌋ = P"
apply (unfold GD3_Lift_def)
apply (unfold GD3_Prop_def)
apply (simp)
done

theorem GD3_Lift_Truth [simp] :
"⟨P↑⟩ = P"
apply (unfold GD3_Lift_def)
apply (unfold GD3_Truth_def)
apply (simp)
done

theorem GD3_Elate_Prop [simp] :
"⌊P⇑⌋ = P"
apply (unfold GD3_Elate_def)
apply (unfold GD3_Prop_def)
apply (simp)
done

theorem GD3_Elate_Truth [simp] :
"⟨P⇑⟩ = False"
apply (unfold GD3_Elate_def)
apply (unfold GD3_Truth_def)
apply (simp)
done

theorem GD3_False_Prop :
"⌊false⌋ = False"
apply (simp)
done

theorem GD3_True_Prop :
"⌊true⌋ = True"
apply (simp)
done

theorem GD3_Super_Prop :
"⌊super⌋ = True"
apply (simp)
done

theorem GD3_Not_Prop :
"⌊¬ P⌋ = (¬ ⌊P⌋)"
apply (induct_tac P)
apply (simp_all)
done
```

```
theorem GD3_And_Prop :
"⌊P ∧ Q⌋ = (⌊P⌋ ∧ ⌊Q⌋)"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all)
done

theorem GD3_Or_Prop :
"⌊P ∨ Q⌋ = (⌊P⌋ ∨ ⌊Q⌋)"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all)
done

theorem GD3_Imp_Prop :
"⌊P ⇒ Q⌋ = (⌊P⌋ ⟶ ⌊Q⌋)"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all)
done

theorem GD3_Iff_Prop :
"⌊P ⇔ Q⌋ = (⌊P⌋ ⟷ ⌊Q⌋)"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all)
done

theorem GD3_Orelse_Prop :
"⌊P ▷ Q⌋ = (if P! then ⌊Q⌋ else ⌊P⌋)"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all)
done

theorem GD3_Forall_Prop :
"⌊∀x. p x⌋ = (∀x. ⌊{p x}⌋)"
apply (unfold GD3_Forall_def)
apply (unfold GD3_Shriek_def)
apply (unfold GD3_Prop_def)
apply (unfold GD3_Lift_def)
apply (auto)
done

theorem GD3_Exists_Prop :
"⌊∃x. p x⌋ = (∃x. ⌊p x⌋)"
apply (unfold GD3_Exists_def)
apply (unfold GD3_Shriek_def)
apply (unfold GD3_Prop_def)
apply (unfold GD3_Lift_def)
apply (auto)
done

named_theorems gd3_hom_laws

declare GD3_Truth_elim [gd3_hom_laws]
declare GD3_Cong_Prop [gd3_hom_laws]
declare GD3_Lift_Prop [gd3_hom_laws]
declare GD3_Lift_Truth [gd3_hom_laws]
```

```
declare GD3_Elate_Prop [gd3_hom_laws]
declare GD3_Elate_Truth [gd3_hom_laws]
declare GD3_False_Prop [gd3_hom_laws]
declare GD3_True_Prop [gd3_hom_laws]
declare GD3_Super_Prop [gd3_hom_laws]
declare GD3_Not_Prop [gd3_hom_laws]
declare GD3_And_Prop [gd3_hom_laws]
declare GD3_Or_Prop [gd3_hom_laws]
declare GD3_Imp_Prop [gd3_hom_laws]
declare GD3_Iff_Prop [gd3_hom_laws]
declare GD3_Orelse_Prop [gd3_hom_laws]
declare GD3_Forall_Prop [gd3_hom_laws]
declare GD3_Exists_Prop [gd3_hom_laws]
```

2.8 Shriek Laws

```
theorem GD3_Lift_Shriek [simp] :
"P↑! = False"
apply (unfold GD3_Lift_def)
apply (simp)
done

theorem GD3_Elate_Shriek [simp] :
"P⇑! = P"
apply (unfold GD3_Elate_def)
apply (simp)
done

theorem GD3_False_Shriek :
"(false)! = False"
apply (simp)
done

theorem GD3_True_Shriek :
"(true)! = False"
apply (simp)
done

theorem GD3_Super_Shriek :
"(super)! = True"
apply (simp)
done

theorem GD3_Not_Shriek :
"(¬ P)! = (¬ ⌊P⌋)"
apply (induct_tac P)
apply (simp_all)
done

theorem GD3_And_Shriek :
"(P ∧ Q)! = (P! ∧ Q!)"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all)
done

theorem GD3_Or_Shriek :
```

```
"(P ∨ Q)! = (P! ∨ Q!)"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all)
done

theorem GD3_Imp_Shriek :
"(P ⇒ Q)! = ((⌊P⌋ ⟶ Q!) ∨ (¬P! ∧ ⌊Q⌋))"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all)
done

theorem GD3_Iff_Shriek :
"(P ⇔ Q)! = ((⌊P⌋ ⟷ ⌊Q⌋) ∧ (P! ⟷ Q!))"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all)
done

theorem GD3_Orelse_Shriek :
"(P ▷ Q)! = (P! ∧ Q!)"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all)
done

theorem GD3_Forall_Shriek :
"(GD3_Forall f)! = (∀x. (f x)!)"
apply (unfold GD3_Forall_def)
apply (unfold GD3_Prop_def)
apply (unfold GD3_Lift_def)
apply (simp)
done

theorem GD3_Exists_Shriek :
"(GD3_Exists f)! = (∃x. (f x)!)"
apply (unfold GD3_Exists_def)
apply (unfold GD3_Prop_def)
apply (unfold GD3_Lift_def)
apply (simp)
done

named_theorems gd3_shriek_laws

declare GD3_Lift_Shriek [gd3_shriek_laws]
declare GD3_Elate_Shriek [gd3_shriek_laws]
declare GD3_False_Shriek [gd3_shriek_laws]
declare GD3_True_Shriek [gd3_shriek_laws]
declare GD3_Super_Shriek [gd3_shriek_laws]
declare GD3_Not_Shriek [gd3_shriek_laws]
declare GD3_And_Shriek [gd3_shriek_laws]
declare GD3_Or_Shriek [gd3_shriek_laws]
declare GD3_Imp_Shriek [gd3_shriek_laws]
declare GD3_Iff_Shriek [gd3_shriek_laws]
declare GD3_Orelse_Shriek [gd3_shriek_laws]
declare GD3_Forall_Shriek [gd3_shriek_laws]
declare GD3_Exists_Shriek [gd3_shriek_laws]
```

2.9 Proof Tactic

2.9.1 Transfer Laws

```
theorem GD3_transfer :
"(P = Q) ⟷ (⌊P⌋ ⟷ ⌊Q⌋) ∧ (P! ⟷ Q!)"
apply (induct_tac P, induct_tac[!] Q)
apply (simp_all add: GD3_Prop_def)
done
```

2.9.2 Utility Laws

```
theorem GD3_Shriek_Prop [intro, simp] :
"P! ⟹ ⌊P⌋"
apply (unfold GD3_Shriek_def)
apply (unfold GD3_Prop_def)
apply (simp)
done
```

```
theorem GD3_Prop_neq_False :
"⌊P⌋ ⟷ (P ≠ GD3_False)"
apply (induct_tac P)
apply (unfold GD3_Prop_def)
apply (simp_all)
done
```

Use the below for customising the tactic's default simplifications.

```
named_theorems gd3_simp_laws
```

2.9.3 Proof Method

The GD3 proof tactic may still be subject to improvements.

```
method gd3_prove_tac = (
  (simp_all)?,
  (unfold GD3_transfer)?,
    (((unfold gd3_hom_laws gd3_shriek_laws gd3_simp_laws)
      | (clarsimp) | (safe))+)?,
  (auto)?)
```

2.10 Representation

Representation of a GD3 predicate by a pair of HOL predicates.

```
theorem gd3_rep_lemma :
"P = ‘P!⇑ ∨ ⌊P⌋↑‘"
apply (gd3_prove_tac)
done
```

```
theorem gd3_rep_exists :
"⋀P. ∃p q. P = ‘p⇑ ∨ q↑‘"
apply (metis gd3_rep_lemma)
done
```

```
theorem gd3_forall_transfer :
"(∀R. f R) = (∀p q. f ‘p⇑ ∨ q↑‘)"
apply (metis gd3_rep_lemma)
```

```
done

theorem gd3_exists_transfer :
"(∃R. p R) = (∃P Q. p ‘P⇑ ∨ Q↑‘)"
apply (metis gd3_rep_lemma)
done
```

2.11 Property Validation

2.11.1 Idempotence

```
theorem GD3_And_idem :
"‘P ∧ P‘ = ‘P‘"
apply (gd3_prove_tac)
done

theorem GD3_Or_idem :
"‘P ∨ P‘ = ‘P‘"
apply (gd3_prove_tac)
done

theorem GD3_Orelse_idem :
"‘P ▷ P‘ = ‘P‘"
apply (gd3_prove_tac)
done
```

2.11.2 Commutativity

```
theorem GD3_And_commute :
"‘P ∧ Q‘ = ‘Q ∧ P‘"
apply (gd3_prove_tac)
done

theorem GD3_Or_commute :
"‘P ∨ Q‘ = ‘Q ∨ P‘"
apply (gd3_prove_tac)
done

theorem GD3_Iff_commute :
"‘P ⇔ Q‘ = ‘Q ⇔ P‘"
apply (gd3_prove_tac)
done
```

2.11.3 Associativity

```
theorem GD3_And_assoc :
"‘(P ∧ Q) ∧ R‘ = ‘P ∧ (Q ∧ R)‘"
apply (gd3_prove_tac)
done

theorem GD3_Or_assoc :
"‘(P ∨ Q) ∨ R‘ = ‘P ∨ (Q ∨ R)‘"
apply (gd3_prove_tac)
done
```

The following law from classical logic does not hold in GD3.

```
theorem GD3_Iff_assoc :
"‘(P ⇔ Q) ⇔ R‘ = ‘P ⇔ (Q ⇔ R)‘"
apply (gd3_prove_tac)
oops

theorem GD3_Orelse_assoc :
"‘(P ▷ Q) ▷ R‘ = ‘P ▷ (Q ▷ R)‘"
apply (gd3_prove_tac)
oops
```

2.11.4 Distributivity

```
theorem GD3_And_Or_distr :
"‘P ∧ (Q ∨ R)‘ = ‘(P ∧ Q) ∨ (P ∧ R)‘"
"‘(P ∨ Q) ∧ R‘ = ‘(P ∧ R) ∨ (Q ∧ R)‘"
apply (gd3_prove_tac)
done

theorem GD3_Or_And_distr :
"‘P ∨ (Q ∧ R)‘ = ‘((P ∨ Q) ∧ (P ∨ R))‘"
"‘(P ∧ Q) ∨ R‘ = ‘(P ∨ R) ∧ (Q ∨ R)‘"
apply (gd3_prove_tac)
done
```

2.11.5 De Morgan Laws

```
theorem GD3_de_Morgan :
"‘¬ (P ∧ Q)‘ = ‘¬ P ∨ ¬ Q‘"
"‘¬ (P ∨ Q)‘ = ‘¬ P ∧ ¬ Q‘"
apply (gd3_prove_tac)
done
```

2.11.6 Zero Laws

```
theorem GD3_And_zero :
"‘false ∧ P‘ = ‘false‘"
"‘P ∧ false‘ = ‘false‘"
apply (gd3_prove_tac)
done

theorem GD3_Or_zero :
"‘super ∨ P‘ = ‘super‘"
"‘P ∨ super‘ = ‘super‘"
apply (gd3_prove_tac)
done

theorem GD3_Imp_zero :
"‘false ⇒ P‘ = ‘super‘"
"‘P ⇒ super‘ = ‘super‘"
apply (gd3_prove_tac)
done

theorem GD3_Orelse_zero :
"‘(false ▷ P)‘ = ‘false‘"
"‘(true ▷ P)‘ = ‘true‘"
apply (gd3_prove_tac)
```

done

2.11.7 Unit Laws

```
theorem GD3_And_Unit :
"‘super ∧ P‘ = ‘P‘"
"‘P ∧ super‘ = ‘P‘"
apply (gd3_prove_tac)
done

theorem GD3_Or_Unit :
"‘false ∨ P‘ = ‘P‘"
"‘P ∨ false‘ = ‘P‘"
apply (gd3_prove_tac)
done

theorem GD3_Imp_Unit :
"‘super ⇒ P‘ = ‘P‘"
apply (gd3_prove_tac)
done

theorem GD3_Imp_False :
"‘P ⇒ false‘ = ‘¬ P‘"
apply (gd3_prove_tac)
done

theorem GD3_Orelse_unit :
"‘(super ▷ P)‘ = ‘P‘"
apply (gd3_prove_tac)
done
```

2.11.8 Or-else Laws

TODO: Proof further useful laws for the or-else operator.

```
theorem GD3_Orelse_chain_elim :
"‘(P ▷ Q) ▷ P‘ = ‘P ▷ Q‘"
apply (gd3_prove_tac)
done

theorem GD3_Orelse_absorb :
"P! ⟶ Q! ⟹ ‘(P ▷ Q)‘ = ‘P‘"
apply (gd3_prove_tac)
done
```

2.12 Violated Properties

The most notable property from classical logic that does not persist to hold in any instance of Gödel-Dummet logic is cancellation of double negation.

```
theorem "‘¬ ¬ P‘ = ‘P‘"
apply (gd3_prove_tac)
oops
```

We do however have the following weaker law for triple negation.

```
theorem GD3_Triple_Neg :
```

```
"‘¬ ¬ ¬ P‘ = ‘¬ P‘"
apply (gd3_prove_tac)
done
```

Congruence can be proved even in the case of double negation.

```
theorem GD3_Neg_Neg_Cong :
"⌊¬ ¬ P⌋ = ⌊P⌋"
apply (gd3_prove_tac)
done
```

The law below is provable in LMC but not so in GD3.

```
theorem "‘true ⇒ P‘ = ‘P‘"
apply (gd3_prove_tac)
oops
end
```

3 State Predicates

```
theory State_Pred
imports GD3
begin
```

We encode a simple model of predicates as state functions.

3.1 Predicate Types

```
type_synonym ('state, 'logic) pred = "'state ⇒ 'logic"

type_synonym 'state hol_pred = "('state, bool) pred"
type_synonym 'state gd3_pred = "('state, gd3) pred"

translations (type) "'state hol_pred" ↽ (type) "('state, bool) pred"
translations (type) "'state gd3_pred" ↽ (type) "('state, gd3) pred"
```

3.2 Meta-logical Operators

```
definition Prop_Pred :: "'state gd3_pred ⇒ 'state hol_pred" where
"Prop_Pred p = (λs. ⌊p s⌋)"

definition Truth_Pred :: "'state gd3_pred ⇒ 'state hol_pred" where
"Truth_Pred p = (λs. ⟨p s⟩)"

definition Shriek_Pred :: "'state gd3_pred ⇒ 'state hol_pred" where
"Shriek_Pred p = (λs. (p s)!)"

definition Lift_Pred :: "'state hol_pred ⇒ 'state gd3_pred" where
"Lift_Pred p = (λs. (p s)↑)"

definition Elate_Pred :: "'state hol_pred ⇒ 'state gd3_pred" where
"Elate_Pred p = (λs. (p s)⇑)"
```

3.3 Logical Constants

```
definition True_Pred :: "'state hol_pred" where
"True_Pred = (λ_. True)"

definition False_Pred :: "'state hol_pred" where
"False_Pred = (λ_. False)"

definition GD3_True_Pred :: "'state gd3_pred" where
"GD3_True_Pred = (λ_. GD3_True)"

definition GD3_False_Pred :: "'state gd3_pred" where
"GD3_False_Pred = (λ_. GD3_False)"

definition GD3_Super_Pred :: "'state gd3_pred" where
"GD3_Super_Pred = (λ_. GD3_Super)"
```

3.4 Connectives

```
definition Not_Pred :: "'state hol_pred ⇒ 'state hol_pred" where
"Not_Pred P = (λs. ¬ (P s))"
```

definition And_Pred ::
  "'state hol_pred ⇒ 'state hol_pred ⇒ 'state hol_pred" **where**
"And_Pred P Q = (λs. (P s) ∧ (Q s))"

definition Or_Pred ::
  "'state hol_pred ⇒ 'state hol_pred ⇒ 'state hol_pred" **where**
"Or_Pred P Q = (λs. (P s) ∨ (Q s))"

definition Imp_Pred ::
  "'state hol_pred ⇒ 'state hol_pred ⇒ 'state hol_pred" **where**
"Imp_Pred P Q = (λs. (P s) ⟶ (Q s))"

definition Iff_Pred ::
  "'state hol_pred ⇒ 'state hol_pred ⇒ 'state hol_pred" **where**
"Iff_Pred P Q = (λs. (P s) ⟷ (Q s))"

definition GD3_Not_Pred :: "'state gd3_pred ⇒ 'state gd3_pred" **where**
"GD3_Not_Pred P = (λs. '¬ (P s)')"

definition GD3_And_Pred ::
  "'state gd3_pred ⇒ 'state gd3_pred ⇒ 'state gd3_pred" **where**
"GD3_And_Pred P Q = (λs. '(P s) ∧ (Q s)')"

definition GD3_Or_Pred ::
  "'state gd3_pred ⇒ 'state gd3_pred ⇒ 'state gd3_pred" **where**
"GD3_Or_Pred P Q = (λs. '(P s) ∨ (Q s)')"

definition GD3_Imp_Pred ::
  "'state gd3_pred ⇒ 'state gd3_pred ⇒ 'state gd3_pred" **where**
"GD3_Imp_Pred P Q = (λs. '(P s) ⇒ (Q s)')"

definition GD3_Iff_Pred ::
  "'state gd3_pred ⇒ 'state gd3_pred ⇒ 'state gd3_pred" **where**
"GD3_Iff_Pred P Q = (λs. '(P s) ⇔ (Q s)')"

definition GD3_Orelse_Pred ::
  "'state gd3_pred ⇒ 'state gd3_pred ⇒ 'state gd3_pred" **where**
"GD3_Orelse_Pred P Q = (λs. '(P s) ▷ (Q s)')"

3.5 Universal Closure

syntax "_Pred_Closure" ::"'state hol_pred ⇒ bool" ("⟨_⟩")

translations "_Pred_Closure P" ⇌ "(CONST All) P"

3.6 Notations

notation Prop_Pred ("$\lfloor\_\rfloor_p$")
notation Truth_Pred ("$\langle\_\rangle_p$")
notation Shriek_Pred ("_$!_p$" [1000] 1000)
notation Lift_Pred ("_$\uparrow_p$" [1000] 1000)
notation Elate_Pred ("_$\Uparrow_p$" [1000] 1000)

notation True_Pred ("$\mathtt{True}_p$")
notation False_Pred ("$\mathtt{False}_p$")

notation Not_Pred ("$\neg_p$ _" [240] 240)
notation And_Pred (**infixr** "$\wedge_p$" 235)
notation Or_Pred (**infixr** "$\vee_p$" 230)
notation Imp_Pred (**infixr** "$\Rightarrow_p$" 225)
notation Iff_Pred (**infixr** "$\Leftrightarrow_p$" 220)

notation GD3_True_Pred ("true$_g$")
notation GD3_False_Pred ("false$_g$")
notation GD3_Super_Pred ("super$_g$")
notation GD3_Not_Pred ("$\neg_g$ _" [240] 240)
notation GD3_And_Pred (**infixr** "$\wedge_g$" 235)
notation GD3_Or_Pred (**infixr** "$\vee_g$" 230)
notation GD3_Imp_Pred (**infixr** "$\Rightarrow_g$" 225)
notation GD3_Iff_Pred (**infixr** "$\Leftrightarrow_g$" 220)
— Review the precedence of the following operator.
notation GD3_Orelse_Pred (**infixr** "$\triangleright_g$" 227)

3.7 Simplifications

named_theorems pred_defs

declare Prop_Pred_def [pred_defs]
declare Truth_Pred_def [pred_defs]
declare Shriek_Pred_def [pred_defs]
declare Lift_Pred_def [pred_defs]
declare Elate_Pred_def [pred_defs]

declare True_Pred_def [pred_defs]
declare False_Pred_def [pred_defs]
declare Not_Pred_def [pred_defs]
declare And_Pred_def [pred_defs]
declare Or_Pred_def [pred_defs]
declare Imp_Pred_def [pred_defs]
declare Iff_Pred_def [pred_defs]

named_theorems gd3_pred_defs

declare GD3_True_Pred_def [gd3_pred_defs]
declare GD3_False_Pred_def [gd3_pred_defs]
declare GD3_Super_Pred_def [gd3_pred_defs]
declare GD3_Not_Pred_def [gd3_pred_defs]
declare GD3_And_Pred_def [gd3_pred_defs]
declare GD3_Or_Pred_def [gd3_pred_defs]
declare GD3_Imp_Pred_def [gd3_pred_defs]
declare GD3_Iff_Pred_def [gd3_pred_defs]
declare GD3_Orelse_Pred_def [gd3_pred_defs]

declare gd3_pred_defs [pred_defs]
declare pred_defs [gd3_simp_laws]

3.8 Dynamic Attribute

ML {*
```
  fun get_pred_defs ctx =
    Named_Theorems.get ctx @{named_theorems pred_defs};
```

```
  fun unfold_pred_defs ctx =
    Simplifier.full_simplify (ctx
      addsimps [@{thm fun_eq_iff}]
      addsimps (get_pred_defs ctx));

  fun unfold_preds_rule ctx =
    (Object_Logic.rulify ctx) o (unfold_pred_defs ctx);

  val unfold_preds_attr =
    Thm.rule_attribute (unfold_preds_rule o Context.proof_of);
*}
```

```
attribute_setup unfold_preds =
  ‹Scan.succeed unfold_preds_attr› "unfold state predicates"
```

3.9 Proof Support

```
theorem pointwise :
"P!_p = Q!_p ⟹ (P s)! = (Q s)!"
"⌊P⌋_p = ⌊Q⌋_p ⟹ ⌊P s⌋ = ⌊Q s⌋"
"P!_p = ⌊Q⌋_p ⟹ (P s)! = ⌊Q s⌋"
"⌊P⌋_p = Q!_p ⟹ ⌊P s⌋ = (Q s)!"
apply (unfold fun_eq_iff)
apply (simp_all add: pred_defs)
done
```

3.10 Theorems

3.10.1 Meta-logical Laws

TODO: There may be a few inverse laws missing below. Review!

```
theorem Prop_Elate_Pred_inverse :
"(⋀s. P s ∈ {false, super}) ⟹ ⌊P⌋_p⇑_p = P"
apply (unfold fun_eq_iff)
apply (gd3_prove_tac)
done

theorem Shriek_Elate_Pred_inverse :
"(⋀s. P s ∈ {false, super}) ⟹ P!_p⇑_p = P"
apply (unfold fun_eq_iff)
apply (gd3_prove_tac)
done

theorem Elate_Prop_Pred_inverse [simp] :
"⌊P⇑_p⌋_p = P"
apply (unfold fun_eq_iff)
apply (gd3_prove_tac)
done

theorem Elate_Truth_Pred_False [simp] :
"⟨P⇑_p⟩_p = False_p"
apply (unfold fun_eq_iff)
apply (gd3_prove_tac)
done
```

```
theorem Elate_Prop_Shriek_inverse [simp] :
"P⇑p!p = P"
apply (unfold fun_eq_iff)
apply (gd3_prove_tac)
done

theorem Lift_Prop_Pred_inverse [simp] :
"⌊P↑p⌋p = P"
apply (unfold fun_eq_iff)
apply (gd3_prove_tac)
done

theorem Lift_Truth_Pred_inverse [simp] :
"⟨P↑p⟩p = P"
apply (unfold fun_eq_iff)
apply (gd3_prove_tac)
done

theorem Lift_Shriek_Pred_false [simp] :
"P↑p!p = Falsep"
apply (unfold fun_eq_iff)
apply (gd3_prove_tac)
done

theorem Prop_equals_Shriek_Pred :
"(⋀s. P s ∈ {false, super}) ⟹ ⌊P⌋p = P!p"
apply (unfold fun_eq_iff)
apply (gd3_prove_tac)
done

theorem Shriek_equals_Prop_Pred :
"(⋀s. P s ∈ {false, super}) ⟹ P!p = ⌊P⌋p"
apply (unfold fun_eq_iff)
apply (gd3_prove_tac)
done
```

3.10.2 HOL Predicate Laws

```
theorem Neg_Neg_Pred :
"¬p ¬p P = P"
apply (simp add: pred_defs)
done

theorem Equal_Pred_iff :
"P = Q ⟷ ⟨P ⇒p Q⟩ ∧ ⟨Q ⇒p P⟩"
apply (unfold pred_defs)
apply (auto)
done

theorem Not_Pred_inject :
"(¬p P = ¬p Q) ⟷ (P = Q)"
apply (unfold pred_defs)
apply (simp add: fun_eq_iff)
done
```

3.10.3 GD3 Predicate Laws

theorem GD3_And_Shriek_Pred_distr :
"$(P \wedge_g Q)!_p = P!_p \wedge_p Q!_p$"
apply (gd3_prove_tac)
done

theorem GD3_Imp_Shriek_Pred_distr :
"$(P \Rightarrow_g Q)!_p = (\lfloor P \rfloor_p \Rightarrow_p Q!_p) \vee_p (\neg_p P!_p \wedge_p \lfloor Q \rfloor_p)$"
apply (gd3_prove_tac)
done

theorem GD3_Orelse_Shriek_Pred_distr :
"$(P \triangleright_g Q)!_p = P!_p \wedge_p Q!_p$"
apply (gd3_prove_tac)
done

theorem GD3_Imp_Prop_Pred_distr :
"$\lfloor P \Rightarrow_g Q \rfloor_p = \lfloor P \rfloor_p \Rightarrow_p \lfloor Q \rfloor_p$"
apply (gd3_prove_tac)
done

3.10.4 Transfer Theorems

theorem GD3_Forall_Pred_transfer :
"$(\forall Q.\ P\ \lfloor Q \rfloor_p) = (\forall Q.\ P\ Q)$"
"$(\forall Q.\ P\ Q!_p) = (\forall Q.\ P\ Q)$"
apply (unfold pred_defs)
— Subgoal 1
apply (rule iffI)
— Subgoal 1.1
apply (clarify)
apply (drule_tac x = "$(\lambda s.\ (Q\ s)\uparrow)$" **in** spec)
apply (gd3_prove_tac) [1]
— Subgoal 1.2
apply (clarsimp)
— Subgoal 2
apply (rule iffI)
— Subgoal 2.1
apply (clarify)
apply (drule_tac x = "$(\lambda s.\ (Q\ s)\Uparrow)$" **in** spec)
apply (gd3_prove_tac) [1]
— Subgoal 2.2
apply (clarsimp)
done

theorem GD3_Exists_Pred_transfer :
"$(\exists Q.\ P\ \lfloor Q \rfloor_p) = (\exists Q.\ P\ Q)$"
"$(\exists Q.\ P\ Q!_p) = (\exists Q.\ P\ Q)$"
apply (unfold pred_defs)
— Subgoal 1
apply (rule iffI)
— Subgoal 1.1
apply (clarify)
apply (rule_tac x = "$(\lambda s.\ \lfloor Q\ s \rfloor)$" **in** exI)
apply (assumption)

— Subgoal 1.2
apply (clarify)
apply (rule_tac x = "(λs. (Q s)↑)" **in** exI)
apply (gd3_prove_tac) [1]
— Subgoal 2
apply (rule iffI)
— Subgoal 2.1
apply (clarify)
apply (rule_tac x = "(λs. (Q s)!)" **in** exI)
apply (simp)
— Subgoal 1.2
apply (clarify)
apply (rule_tac x = "(λs. (Q s)⇑)" **in** exI)
apply (gd3_prove_tac) [1]
done
end

4 GSL Syntax

```
theory GSL
imports Preliminaries State_Pred
begin
```

4.1 Type Synonyms

```
type_synonym 'state update = "'state ⇒ 'state"
```

4.2 Expression Model

```
type_synonym ('a, 'state) expr = "'state ⇒ 'a"

definition ConstE :: "'a ⇒ ('a, 'state) expr" where
"ConstE x ≡ (λ_. x)"

notation ConstE ("¢'(_')")

named_theorems expr_defs

declare ConstE_def [expr_defs]
```

4.3 Variable Model

```
record ('a, 'state) var =
  get :: "'state ⇒ 'a"
  set :: "('a, 'state) expr ⇒ 'state update"

notation get ("(_·/_)" [1000, 1000] 1000)
notation set ("(_↩/_ in /_)" [1000, 1000, 1000] 0)
```

4.4 Extended GSL

We use the more abstract notion of state update in the syntax in place of assignment. This facilitates defining different kinds of assignments, such as parallel assignment. Note that both unbounded choice operators impose a cardinality bound on their set argument; this bound is captured by the type parameter `'bound`. We later defined operators that automatically raise this bound so that by construction, we ensure that the bound is large enough to accommodate any set of `gsl_ext` elements in a parsed term.

```
datatype ('state, 'bound) gsl_ext =
  Skip |
  Update "'state update" |
  Seq "('state, 'bound) gsl_ext" "('state, 'bound) gsl_ext" |
  Pre "'state hol_pred" "('state, 'bound) gsl_ext" |
  Guard "'state hol_pred" "('state, 'bound) gsl_ext" |
  Choice "('state, 'bound) gsl_ext" "('state, 'bound) gsl_ext" |
  Angelic "('state, 'bound) gsl_ext" "('state, 'bound) gsl_ext" |
  Pref "('state, 'bound) gsl_ext" "('state, 'bound) gsl_ext" |
  UChoice "('state, 'bound) gsl_ext set['bound]" |
  AChoice "('state, 'bound) gsl_ext set['bound]"
```

4.5 Monotonic GSL

We define monotonic GSL as an inductive subset of 'extended' GSL.

```
inductive_set GSL :: "('state, 'bound) gsl_ext set" where
"Skip ∈ GSL" |
"Update u ∈ GSL" |
"⟦S ∈ GSL; T ∈ GSL⟧ ⟹ Seq S T ∈ GSL" |
"⟦S ∈ GSL⟧ ⟹ Pre p S ∈ GSL" |
"⟦S ∈ GSL⟧ ⟹ Guard g S ∈ GSL" |
"⟦S ∈ GSL; T ∈ GSL⟧ ⟹ Choice S T ∈ GSL" |
"⟦S ∈ GSL; T ∈ GSL⟧ ⟹ Angelic S T ∈ GSL" |
"⟦∀S. S ∈b SS ⟶ S ∈ GSL⟧ ⟹ UChoice SS ∈ GSL" |
"⟦∀S. S ∈b SS ⟶ S ∈ GSL⟧ ⟹ AChoice SS ∈ GSL"
```

4.6 Magic and Abort

```
abbreviation Abort :: "('state, 'bound) gsl_ext" where
"Abort ≡ Pre Falsep Skip"

abbreviation Magic :: "('state, 'bound) gsl_ext" where
"Magic ≡ Guard Falsep Skip"
```

4.7 Assignment

```
abbreviation Assign ::
  "('a, 'state) var ⇒ ('a, 'state) expr ⇒ ('state, 'bound) gsl_ext" where
"Assign v e ≡ Update (set v e)"
```

4.8 Unbounded Choice

The objective here is to define versions of the functions `UChoice` and `AChoice` that apply to general HOL (rather than bounded) sets. While doing so, they have to raise the bound of the resulting GSL program to accommodate all possible (HOL) sets that may be given as an argument. For this, we first inductively define a recasting operator that alters the bound of an encoded GSL program.

```
function (domintros) GSLRecast ::
  "('state, 'bound1) gsl_ext ⇒ ('state, 'bound2) gsl_ext" where
"GSLRecast (Skip) = Skip" |
"GSLRecast (Update u) = (Update u)" |
"GSLRecast (Seq S T) = (Seq (GSLRecast S) (GSLRecast T))" |
"GSLRecast (Pre p S) = (Pre p (GSLRecast S))" |
"GSLRecast (Guard g S) = (Guard g (GSLRecast S))" |
"GSLRecast (Choice S T) = (Choice (GSLRecast S) (GSLRecast T))" |
"GSLRecast (Angelic S T) = (Angelic (GSLRecast S) (GSLRecast T))" |
"GSLRecast (Pref S T) = (Pref (GSLRecast S) (GSLRecast T))" |
"GSLRecast (UChoice SS) = UChoice (Abs_bset (GSLRecast ` (set_bset SS)))" |
"GSLRecast (AChoice SS) = AChoice (Abs_bset (GSLRecast ` (set_bset SS)))"
by pat_completeness auto
termination
apply (rule allI)
apply (induct_tac x)
apply (simp_all add: GSLRecast.domintros)
done
```

The following type instance of `GSLRecast` upcasts the bound.

```
abbreviation GSLUpcast :: "('state, 'bound) gsl_ext ⇒
  ('state, ('state, 'bound) gsl_ext set) gsl_ext" where
```

```
"GSLUpcast ≡ GSLRecast"
```

The law below is key to automate proofs about unbounded choice.

```
theorem Abs_bset_GSLUpcast_image [simp] :
"Abs_bset (GSLUpcast ‘ SS) = GSLUpcast ‘b (mk_bset SS)"
apply (metis bimage.rep_eq mk_bset_inverses(1) set_bset_inverse)
done
```

Using up-casting, we defined new versions of the unbounded choice operators `UChoice` and `UChoice` that can be applied to `sets`. Note that these operators ensure that the bound is large enough so that the set can hold the entire universe of GSL programs at the respective model.

```
abbreviation UChoice_Set ::
  "('state, 'bound) gsl_ext set ⇒
   ('state, ('state, 'bound) gsl_ext set) gsl_ext" where
"UChoice_Set SS ≡ UChoice (GSLUpcast ‘b {|SS|})"

abbreviation AChoice_Set ::
  "('state, 'bound) gsl_ext set ⇒
   ('state, ('state, 'bound) gsl_ext set) gsl_ext" where
"AChoice_Set SS ≡ AChoice (GSLUpcast ‘b {|SS|})"
```

4.9 Binders

```
abbreviation UChoice_Binder ::
  "('a ⇒ ('state, 'bound) gsl_ext) ⇒
   ('state, ('state, 'bound) gsl_ext set) gsl_ext" where
"UChoice_Binder f ≡ UChoice_Set (range f)"

abbreviation AChoice_Binder ::
  "('a ⇒ ('state, 'bound) gsl_ext) ⇒
    ('state, ('state, 'bound) gsl_ext set) gsl_ext" where
"AChoice_Binder f ≡ AChoice_Set (range f)"
```

The next two operators correspond to Unbounded Choice in B.

```
definition UChoice_Variable ::
  "('a, 'state) var ⇒ ('state, 'bound) gsl_ext ⇒
    ('state, ('state, 'bound) gsl_ext set) gsl_ext" where
"UChoice_Variable v S = UChoice_Binder (λx. (Seq (Assign v x) S))"

definition AChoice_Variable ::
  "('a, 'state) var ⇒ ('state, 'bound) gsl_ext ⇒
    ('state, ('state, 'bound) gsl_ext set) gsl_ext" where
"AChoice_Variable v S = AChoice_Binder (λx. (Seq (Assign v x) S))"
```

4.10 Notations

```
no_syntax "_Eps" :: "[pttrn, bool] => 'a" ("(3@ _./ _)" [0, 10] 10)
no_syntax "_INF1":: "pttrns ⇒ 'b ⇒ 'b" ("(3⨅_./ _)" [0, 10] 10)
no_syntax "_SUP1":: "pttrns ⇒ 'b ⇒ 'b" ("(3⨆_./ _)" [0, 10] 10)

no_notation disj (infixr "|" 30)
no_notation inf (infixl "⊓" 70)
no_notation sup (infixl "⊔" 65)
no_notation Inf ("⨅_" [900] 900)
no_notation Sup ("⨆_" [900] 900)
```

```
notation Skip ("skip")
notation Abort ("abort")
notation Magic ("magic")
notation Update ("⊙'(_')")
notation Seq (infixl ";" 100)
notation Assign (infix ":=" 140)
notation Pre (infix "|" 130)
notation Guard (infix "→" 130)
notation Choice (infixl "⊓" 110)
notation Angelic (infixl "⊔" 110)
notation Pref (infixl "≫" 120)
notation UChoice ("⨅b")
notation AChoice ("⨆b")
notation UChoice_Set ("⨅")
notation AChoice_Set ("⨆")
notation UChoice_Binder (binder "⨅" 10)
notation AChoice_Binder (binder "⨆" 10)
notation UChoice_Variable ("(3@_./ _)" [0, 10] 10)
notation AChoice_Variable ("(3#_./ _)" [0, 10] 10)
```

4.11 Theorems

```
theorem Unfold_Choice_Set_lemma [simp] :
"S ∈b GSLUpcast `b {|SS|} ⟷ (∃S'. S' ∈ SS ∧ S = GSLUpcast S')"
apply (simp add: bimage.rep_eq bmember.rep_eq)
apply (blast)
done
end
```

5 GSL Instantiation

```
theory GSL_Inst
imports GSL wp
begin
```

5.1 State Space

Our state space contains two distinct variables x and y of type nat. We introduce the concrete state space as a record type.

```
record state =
  x :: "nat"
  y :: "nat"
```

5.2 Concrete Variables

```
definition x_var :: "(nat, state) var" where
"x_var = (|get = x, set = (λe . (λs. x_update (λ_. e s) s))|)"

definition y_var :: "(nat, state) var" where
"y_var = (|get = y, set = (λe . (λs. y_update (λ_. e s) s))|)"

notation x_var ("x")
notation y_var ("y")
```

5.3 Simplifications

```
named_theorems var_defs

declare x_var_def [var_defs]
declare y_var_def [var_defs]
```

5.4 Proof Experiments

```
declare One_nat_def [simp del]

theorem Choice_backtracks_wp :
"x := ¢(1) ⊓ x := ¢(2); (λs. x·s = 2) → skip ≡wp x := ¢(2)"
apply (unfold wp_equiv_def wp_ref_def)
apply (simp add: pred_defs expr_defs var_defs)
done

theorem UChoice_lemma :
"(@x. skip) ⊑wp x := ¢(1)"
apply (unfold UChoice_Variable_def)
apply (unfold wp_equiv_def wp_ref_def)
apply (simp add: pred_defs)
apply (clarify)
apply (drule_tac x = "x := ¢(1) ; skip" in spec)
apply (clarsimp)
apply (erule contrapos_pp, simp)
apply (rule_tac x = "x := ¢(1) ; skip" in exI)
apply (simp)
done
```

```
declare One_nat_def [simp]
end
```

6 Nelson's Operator

```
theory Nelson
imports GSL wp
begin
```

6.1 Biased Choice

```
definition Biased_Choice ::
  "('state, 'bound) gsl_ext ⇒
   ('state, 'bound) gsl_ext ⇒
   ('state, 'bound) gsl_ext" where
"Biased_Choice S T = S ⊓ ¬p fis(S) → T"

notation Biased_Choice (infixl "[+]" 120)
end
```

7 GSL Semantics (wp)

theory wp
imports GSL
begin

7.1 Type Synonyms

```
type_synonym 'state hol_trans = "'state hol_pred ⇒ 'state hol_pred"
```

7.2 Predicate Transformer

```
function (domintros) wp ::
  "('state, 'bound) gsl_ext ⇒ 'state hol_trans" where
"wp (Skip) Q = Q" |
"wp (Update u) Q = (λs. Q (u s))" |
"wp (Seq S T) Q = (wp S (wp T Q))" |
"wp (Pre p S) Q = p ∧p (wp S Q)" |
"wp (Guard g S) Q = g ⇒p (wp S Q)" |
"wp (Choice S T) Q = (wp S Q) ∧p (wp T Q)" |
"wp (Angelic S T) Q = (wp S Q) ∨p (wp T Q)" |
"wp (Pref S T) Q = undefined" |
"wp (UChoice SS) Q = (λs. (∀S. S ∈b SS ⟶ (wp S Q) s))" |
"wp (AChoice SS) Q = (λs. ¬ (∀S. S ∈b SS ⟶ ¬ (wp S Q) s))"
by pat_completeness auto
termination
apply (simp)
apply (rule allI)
apply (induct_tac a)
apply (simp_all add: wp.domintros)
apply (safe)
— Subgoal 1
apply (rule wp.domintros)
apply (transfer', simp)
— Subgoal 2
apply (rule wp.domintros)
apply (transfer', simp)
done
```

— The version below is cleaner but causes issues with the domintro law.

```
theorem wp_AChoice_lemma  :
"wp (AChoice SS) Q = (λs. (∃S. S ∈b SS ∧ (wp S Q) s))"
apply (simp)
done
```

```
syntax "_wp_syntax" ::
  "('state, 'bound) gsl_ext ⇒ 'state hol_trans" ("(3wp'(_,/ _'))")
```

```
translations "wp(S, Q)" ⇌ "(CONST wp) S Q"
```

7.3 Conjugate Transformer

```
definition cwp :: "('state, 'bound) gsl_ext ⇒ 'state hol_trans" where
"cwp S Q = ¬p wp(S, ¬p Q)"
```

```
syntax "_cwp_syntax" ::
  "('state, 'bound) gsl_ext ⇒ 'state hol_trans" ("(3cwp'(_,/ _'))")

translations "cwp(S, Q)" ⇌ "(CONST cwp) S Q"

declare cwp_def [simp]
```

7.4 Feasibility and Termination

```
definition fis_wp :: "('state, 'bound) gsl_ext ⇒ 'state hol_pred" where
"fis_wp S = ¬p wp(S, Falsep)"

definition trm_wp :: "('state, 'bound) gsl_ext ⇒ 'state hol_pred" where
"trm_wp S = wp(S, Truep)"

notation fis_wp ("fis'(_')")
notation trm_wp ("trm'(_')")
```

7.5 Refinement and Equivalence

```
definition wp_ref ::
  "('state, 'bound) gsl_ext ⇒
   ('state, 'bound) gsl_ext ⇒ bool" (infix "⊑wp" 50) where
"S ⊑wp T ⟷ (∀Q. ⟨wp(S, Q) ⇒p wp(T, Q)⟩)"

definition wp_equiv ::
  "('state, 'bound) gsl_ext ⇒
   ('state, 'bound) gsl_ext ⇒ bool" (infix "≡wp" 50) where
"S ≡wp T ⟷ (S ⊑wp T) ∧ (T ⊑wp S)"
```

7.6 Monotonicity

```
theorem wp_mono :
"(⋀P Q. ⟨P ⇒p Q⟩ ⟹ ⟨wp(S, P) ⇒p wp(S, Q)⟩)"
apply (atomize (full))
apply (induct_tac S)
apply (simp_all add: pred_defs)
— Subgoal 1
apply (meson)
— Subgoal 2
apply (meson bmember.rep_eq)
— Subgoal 3
apply (meson bmember.rep_eq)
done

theorem wp_mono_elim :
"wp(S, P) s ⟹ ⟨P ⇒p Q⟩ ⟹ wp(S, Q) s"
apply (metis Imp_Pred_def wp_mono)
done

theorems wp_mono_elim' = wp_mono_elim [unfold_preds]
```

7.7 Theorems

```
theorem wp_equiv_equals :
"S ≡wp T ⟷ (wp S) = (wp T)"
```

```
apply (unfold wp_equiv_def wp_ref_def)
apply (simp add: pred_defs)
apply (safe)
— Subgoal 1
apply (rule ext)+
apply (rename_tac Q s)
apply (auto) [1]
— Subgoal 2
apply (auto) [1]
— Subgoal 3
apply (auto) [1]
done

theorem wp_GSLUpcase_elim [simp] :
"wp (GSLUpcast S) = (wp S)"
apply (induct_tac S)
apply (simp_all)
— Subgoal 1
apply (transfer')
apply (fastforce)
— Subgoal 2
apply (transfer')
apply (fastforce)
done

theorem UChoice_wp_ref :
"S ∈ SS ⟹ (⨅ SS) ⊑wp (GSLUpcast S)"
"(∀S' ∈ SS'. S ⊑wp S') ⟹ (GSLUpcast S) ⊑wp (⨅ SS')"
apply (unfold wp_ref_def)
apply (auto simp: pred_defs)
done

theorem AChoice_wp_ref :
"S ∈ SS ⟹ (GSLUpcast S) ⊑wp (⨆ SS)"
"(∀S ∈ SS. S ⊑wp S') ⟹ (⨆ SS) ⊑wp (GSLUpcast S')"
apply (unfold wp_ref_def)
apply (auto simp: pred_defs)
done
```

7.7.1 Refinement Laws

```
theorem wp_ref_refl :
"S ⊑wp S"
apply (unfold wp_ref_def)
apply (simp add: pred_defs)
done

theorem wp_ref_trans :
"S ⊑wp T ⟹ T ⊑wp U ⟹ S ⊑wp U"
apply (unfold wp_ref_def)
apply (simp add: pred_defs)
done

theorem wp_ref_connect :
"S ⊑wp T ⟷ S ≡wp S ⊓ T"
apply (unfold wp_equiv_def wp_ref_def)
```

```
apply (simp add: pred_defs)
done
```

7.7.2 Feasibility Laws

```
named_theorems fis_laws

theorem fis_wp_Abort [fis_laws] :
"fis(abort) = True_p"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_Magic [fis_laws] :
"fis(magic) = False_p"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_Skip [fis_laws] :
"fis(skip) = True_p"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_Update [fis_laws] :
"fis(⊙(u)) = True_p"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_Seq [fis_laws] :
"fis(S ; T) = cwp(S, fis(T))"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_Pre [fis_laws] :
"fis(p | S) = p ⇒_p fis(S)"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_Guard [fis_laws] :
"fis(g → S) = g ∧_p fis(S)"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_Choice [fis_laws] :
"fis(S ⊓ T) = (fis(S) ∨_p fis(T))"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_Angelic [fis_laws] :
```

```
"fis(S ⊔ T) = (fis(S) ∧p fis(T))"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_UChoice [fis_laws] :
"fis(⨅b SS) s = (∃S. S ∈b SS ∧ fis(S) s)"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_AChoice [fis_laws] :
"fis(⨆b SS) s = (∀S. S ∈b SS ⟶ fis(S) s)"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
done

theorem fis_wp_UChoice_Set [fis_laws] :
"fis(⨅ SS) s = (∃S∈SS. fis(S) s)"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
apply (auto)
done

theorem fis_wp_AChoice_Set [fis_laws] :
"fis(⨆ SS) s = (∀S∈SS. fis(S) s)"
apply (unfold fis_wp_def)
apply (simp_all add: pred_defs)
apply (auto)
done
```

7.7.3 Termination Laws

```
named_theorems trm_laws

theorem trm_wp_Abort [trm_laws] :
"trm(abort) = Falsep"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
done

theorem trm_wp_Magic [trm_laws] :
"trm(magic) = Truep"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
done

theorem trm_wp_Skip [trm_laws] :
"trm(skip) = Truep"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
done

theorem trm_wp_Update [trm_laws] :
"trm(⊙(u)) = Truep"
apply (unfold trm_wp_def)
```

```
apply (simp_all add: pred_defs)
done

theorem trm_wp_Seq [trm_laws] :
"trm(S ; T) = wp(S, trm(T))"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
done

theorem trm_wp_Pre [trm_laws] :
"trm(p | S) = p ∧p trm(S)"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
done

theorem trm_wp_Guard [trm_laws] :
"trm(g → S) = g ⇒p trm(S)"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
done

theorem trm_wp_Choice [trm_laws] :
"trm(S ⊓ T) = (trm(S) ∧p trm(T))"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
done

theorem trm_wp_Angelic [trm_laws] :
"trm(S ⊔ T) = (trm(S) ∨p trm(T))"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
done

theorem trm_wp_UChoice [trm_laws] :
"trm(⨅b SS) s = (∀S. S ∈b SS ⟶ trm(S) s)"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
done

theorem trm_wp_AChoice [trm_laws] :
"trm(⨆b SS) s = (∃S. S ∈b SS ∧ trm(S) s)"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
done

theorem trm_wp_UChoice_Set [trm_laws] :
"trm(⨅ SS) s = (∀S∈SS. trm(S) s)"
apply (unfold trm_wp_def)
apply (simp_all add: pred_defs)
apply (auto)
done

theorem trm_wp_AChoice_Set [trm_laws] :
"trm(⨆ SS) s = (∃S∈SS. trm(S) s)"
apply (unfold trm_wp_def)
```

```
apply (simp_all add: pred_defs)
apply (auto)
done
```

7.7.4 Monotonicity Laws

```
theorem Seq_wp_mono :
"S ⊑wp S' ⟹ T ⊑wp T' ⟹ S ; T ⊑wp S' ; T'"
apply (unfold wp_ref_def)
apply (simp add: pred_defs)
using wp_mono_elim' by force

theorem Pre_wp_mono :
"S ⊑wp S' ⟹ (P | S) ⊑wp (P | S')"
apply (unfold wp_ref_def)
apply (simp add: pred_defs)
done

theorem Guard_wp_mono :
"S ⊑wp S' ⟹ (G → S) ⊑wp (G → S')"
apply (unfold wp_ref_def)
apply (simp add: pred_defs)
done

theorem Choice_wp_mono :
"S ⊑wp S' ⟹ T ⊑wp T' ⟹ S ⊓ T ⊑wp S' ⊓ T'"
apply (unfold wp_ref_def)
apply (simp add: pred_defs)
done

theorem Angelic_wp_mono :
"S ⊑wp S' ⟹ T ⊑wp T' ⟹ S ⊔ T ⊑wp S' ⊔ T'"
apply (unfold wp_ref_def)
apply (simp add: pred_defs)
done

theorem Pref_wp_mono :
"S ⊑wp S' ⟹ S ≫ T ⊑wp S' ≫ T"
apply (unfold wp_ref_def)
apply (simp add: pred_defs)
done

theorem UChoice_wp_subset_mono :
"SS' ⊆ SS ⟹ (⨅ SS) ⊑wp (⨅ SS')"
apply (unfold wp_ref_def)
apply (auto simp: pred_defs)
done

theorem AChoice_wp_subset_mono :
"SS ⊆ SS' ⟹ (⨆ SS) ⊑wp (⨆ SS')"
apply (unfold wp_ref_def)
apply (auto simp: pred_defs)
done

theorem UChoice_wp_mono :
"(∀S'∈SS'. ∃S∈SS. S ⊑wp S') ⟹ (⨅ SS) ⊑wp (⨅ SS')"
```

```
apply (unfold wp_ref_def)
apply (simp add: pred_defs)
apply (fastforce)
done

theorem AChoice_wp_mono :
"(∀S∈SS. ∃S'∈SS'. S ⊑wp S') ⟹ (⨆ SS) ⊑wp (⨆ SS')"
apply (unfold wp_ref_def)
apply (simp add: pred_defs)
apply (fastforce)
done
end
```

8 GSL Semantics (wpe)

```
theory wpe
imports GD3 GSL  wp
begin
```

8.1 Type Synonyms

```
type_synonym 'state gd3_trans = "'state gd3_pred ⇒ 'state gd3_pred"
```

8.2 Predicate Transformer

TODO: Try and fix the sorry below by reformulating the definition.

```
function (domintros) wpe ::
  "('state, 'bound) gsl_ext ⇒ 'state gd3_trans" where
"wpe (Skip) Q = Q" |
"wpe (Update u) Q = (λs. Q (u s))" |
"wpe (Seq S T) Q = wpe S (wpe T Q)" |
"wpe (Pre p S) Q = p⇑p ∧g (wpe S Q)" |
"wpe (Guard g S) Q = g(*↑p*)⇑p ⇒g (wpe S Q)" |
"wpe (Choice S T) Q = (wpe S Q) ∧g (wpe T Q)" |
"wpe (Angelic S T) Q = (wpe S Q) ∨g (wpe T Q)" |
"wpe (Pref S T) Q = (wpe S Q) ▷g (wpe T Q)" |
"wpe (UChoice SS) Q = (λs. '∀S. {if S ∈b SS then (wpe S Q) s else super}')" |
"wpe (AChoice SS) Q = (λs. '∃S. {if S ∈b SS then (wpe S Q) s else false}')"
by pat_completeness auto
termination
apply (simp)
apply (rule allI)
apply (induct_tac a)
apply (simp_all add: wpe.domintros)
apply (safe)
— Subgoal 1
apply (rule wpe.domintros)
apply (transfer')
apply (simp)
— Subgoal 2
apply (rule wpe.domintros)
apply (transfer')
apply (simp)
done
```

We cannot use the formulations below directly in the above definition of wpe due to issues with the domintro laws and termination proof.

```
theorem wpe_UChoice [simp] :
"wpe (UChoice SS) Q = (λs. '∀S. (S ∈b SS)(*↑*)⇑ ⇒ (wpe S Q) s')"
apply (unfold wpe.simps)
apply (simp add: fun_eq_iff)
apply (gd3_prove_tac)
done

theorem wpe_AChoice [simp] :
"wpe (AChoice SS) Q = (λs. '∃S. (S ∈b SS)⇑ ∧ (wpe S Q) s')"
apply (unfold wpe.simps)
apply (simp add: fun_eq_iff)
```

```
apply (gd3_prove_tac)
done

declare wpe.simps(9) [simp del]
declare wpe.simps(10) [simp del]

syntax "_wpe_syntax" ::
  "('state, 'bound) gsl_ext ⇒ 'state gd3_trans" ("(3wpe'(_,/ _'))")

translations "wpe(S, Q)" ⇌ "(CONST wpe) S Q"
```

8.3 Conjugate Transformer

TODO: Define the corresponding conjugate GD3 transformer.

8.4 Feasibility and Termination

```
definition fis_wpe :: "('state, 'bound) gsl_ext ⇒ 'state hol_pred" where
"fis_wpe S = ¬p wpe(S, falseg)!p"

definition trm_wpe :: "('state, 'bound) gsl_ext ⇒ 'state hol_pred" where
"trm_wpe S = wpe(S, superg)!p"

notation fis_wpe ("fis#'(_')")
notation trm_wpe ("trm#'(_')")
```

8.5 Refinement and Equivalence

```
definition wpe_ref ::
  "('state, 'bound) gsl_ext ⇒
   ('state, 'bound) gsl_ext ⇒ bool" (infix "⊑wpe" 50) where
"S ⊑wpe T ⟷ (∀Q s. ⌊wpe(S, Q) ⇒g wpe(T, Q)⌋p s)"
— The following definition is equivalent, as we prove below.
— (S ⊑wpe T) = (∀Q s. (wpe(S, Q) ⇒g wpe(T, Q))!p s)

definition wpe_equiv ::
  "('state, 'bound) gsl_ext ⇒
   ('state, 'bound) gsl_ext ⇒ bool" (infix "≡wpe" 50) where
"S ≡wpe T ⟷ (S ⊑wpe T) ∧ (T ⊑wpe S)"
```

8.6 Biased Choice

```
definition Biased_Choice ::
  "('state, 'bound) gsl_ext ⇒
   ('state, 'bound) gsl_ext ⇒
   ('state, 'bound) gsl_ext" where
"Biased_Choice S T = S ⊓ ¬p fis#(S) → T"

notation Biased_Choice (infixl "[+]" 120)
```

8.7 Theorems

```
theorem wpe_GSLUpcase_elim [simp] :
"wpe (GSLUpcast S) = (wpe S)"
apply (induct_tac S)
```

```
apply (simp_all)
— Subgoal 1
apply (transfer')
apply (simp add: fun_eq_iff)
apply (gd3_prove_tac)
— Subgoal 2
apply (transfer')
apply (simp add: fun_eq_iff)
apply (gd3_prove_tac)
done

theorem UChoice_wpe_ref :
"S ∈ SS ⟹ (⨅ SS) ⊑wpe (GSLUpcast S)"
"(∀S' ∈ SS'. S ⊑wpe S') ⟹ (GSLUpcast S) ⊑wpe (⨅ SS')"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem AChoice_wpe_ref :
"S ∈ SS ⟹ (GSLUpcast S) ⊑wpe (⨆ SS)"
"(∀S ∈ SS. S ⊑wpe S') ⟹ (⨆ SS) ⊑wpe (GSLUpcast S')"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done
```

8.7.1 Essential Lemmas

```
lemma wpe_false_super_closure [rule_format] :
"∀Q. (∀s. Q s ∈ {false, super}) ⟶ (∀s. wpe(S, Q) s ∈ {false, super})"
apply (induct_tac S)
— Subgoal 1
apply (clarsimp)
— Subgoal 2
apply (clarsimp)
— Subgoal 3
apply (clarsimp)
— Subgoal 4
apply (gd3_prove_tac) [1]
— Subgoal 5
apply (gd3_prove_tac) [1]
— Subgoal 6
apply (gd3_prove_tac) [1]
— Subgoal 7
apply (gd3_prove_tac) [1]
— Subgoal 8
apply (gd3_prove_tac) [1]
— Subgoal 9
apply (clarsimp)
apply (transfer')
apply (gd3_prove_tac) [1]
— Subgoal 10
apply (clarsimp)
apply (transfer')
apply (gd3_prove_tac) [1]
apply (blast)
done
```

```
theorem wpe_Prop_to_Shriek [rule_format] :
"∀Q. ⌊wpe(S, Q⇑p)⌋p = wpe(S, Q⇑p)!p"
apply (clarify)
apply (rule Prop_equals_Shriek_Pred)
apply (rule wpe_false_super_closure)
apply (gd3_prove_tac)
done
```

This may be the closest we get to monotonicity...

```
theorem wpe_Shriek_mono :
"⋀P Q. ⟨P!p ⇒p Q!p⟩ ⟹ ⟨wpe(S, P)!p ⇒p wpe(S, Q)!p⟩"
apply (atomize (full))
apply (induct_tac S)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
— Subgoal 1
apply (blast)
— Subgoal 2
apply (transfer')
apply (blast)
— Subgoal 3
apply (transfer')
apply (blast)
— Subgoal 4
apply (transfer')
apply (blast)
done

theorem wpe_Shriek_equal [rule_format] :
"∀P Q. P!p = Q!p ⟶ wpe(S, P)!p = wpe(S, Q)!p"
apply (unfold Equal_Pred_iff)
apply (meson wpe_Shriek_mono)
done

theorem wpe_Elate_externalise [rule_format] :
"∀Q. wpe(S, Q⇑p) = ⌊wpe(S, Q⇑p)⌋p⇑p"
"∀Q. wpe(S, Q⇑p) = wpe(S, Q⇑p)!p⇑p"
apply (clarify)
— Subgoal 1
apply (subst Prop_Elate_Pred_inverse)
— Subgoal 1.1
apply (rule wpe_false_super_closure)
apply (gd3_prove_tac) [1]
— Subgoal 2.1
apply (clarify)
— Subgoal 2
apply (subst Shriek_Elate_Pred_inverse)
— Subgoal 1.1
apply (rule wpe_false_super_closure)
apply (gd3_prove_tac) [1]
— Subgoal 2.1
apply (clarify)
done
```

```
theorem wpe_Shriek_internalise [rule_format] :
"∀Q. wpe(S, Q)!p = wpe(S, Q!p⇑p)!p"
apply (clarify)
apply (rule wpe_Shriek_equal)
apply (gd3_prove_tac) [1]
done
```

8.7.2 Refinement Semantics

```
theorem wpe_ref_refl :
"S ⊑wpe S"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem wpe_ref_trans :
"S ⊑wpe T ⟹ T ⊑wpe U ⟹ S ⊑wpe U"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem wpe_ref_connect :
"S ⊑wpe T ⟷ S ≡wpe S ⊓ T"
"S ⊑wpe T ⟷ T ≡wpe S ⊔ T"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
done

theorem wpe_alt_def_lemma :
"(∀Q. ⟨⌊wpe(S, Q) ⇒g wpe(T, Q)⌋p⟩) =
 (∀Q. ⟨(wpe(S, Q) ⇒g wpe(T, Q))!p⟩)"
apply (gd3_prove_tac)
apply (erule_tac Q = "(wpe(T, Q) s)!" in contrapos_np, simp)
apply (metis pointwise(3) wpe_Prop_to_Shriek wpe_Shriek_internalise)
done

theorem wpe_ref_alt_def :
"S ⊑wpe T = (∀Q. ⟨(wpe(S, Q) ⇒g wpe(T, Q))!p⟩)"
apply (unfold wpe_ref_def)
apply (simp add: wpe_alt_def_lemma)
done

theorem wpe_equiv_equal :
"S ≡wpe T ⟷ (wpe S) = (wpe T)"
apply (unfold wpe_equiv_def)
apply (unfold wpe_ref_alt_def)
apply (safe)
— Subgoal 1
apply (rule ext)+
apply (rename_tac Q s)
apply (gd3_prove_tac) [1]
— Subgoal 1.1
apply (blast)
— Subgoal 1.2
apply (blast)
— Subgoal 2
```

```
apply (gd3_prove_tac) [1]
— Subgoal 3
apply (gd3_prove_tac) [1]
done
```

8.7.3 Feasibility Laws

```
theorem fis_wpe_Abort [fis_laws] :
"fis#(abort) = True_p"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem fis_wpe_Magic [fis_laws] :
"fis#(magic) = False_p"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem fis_wpe_Skip [fis_laws] :
"fis#(skip) = True_p"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
done

theorem fis_wpe_Update [fis_laws] :
"fis#(⊙(u)) = True_p"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
done

theorem fis_wpe_Seq [fis_laws] :
"fis#(S ; T) = ¬_p ⌊wpe(S, (¬_p fis#(T))⇑_p)⌋_p"
apply (unfold fis_wpe_def)
apply (simp add: Neg_Neg_Pred)
apply (subst wpe_Shriek_internalise)
apply (simp add: wpe_Prop_to_Shriek)
done

theorem fis_wpe_Pre [fis_laws] :
"fis#(p | S) = p ⇒_p fis#(S)"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem fis_wpe_Guard [fis_laws] :
"fis#(g → S) = g ∧_p fis#(S)"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem fis_wpe_Choice [fis_laws] :
```

```
"fis#(S ⊓ T) = fis#(S) ∨p fis#(T)"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem fis_wpe_Angelic [fis_laws] :
"fis#(S ⊔ T) = fis#(S) ∧p fis#(T)"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem fis_wpe_Pref [fis_laws] :
"fis#(S ≫ T) = fis#(S) ∨p fis#(T)"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem fis_wpe_UChoice [fis_laws] :
"fis#(⨅b SS) s = (∃S. S ∈b SS ∧ fis#(S) s)"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem fis_wpe_AChoice [fis_laws] :
"fis#(⨆b SS) s = (∀S. S ∈b SS ⟶ fis#(S) s)"
apply (unfold fis_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done
```

8.7.4 Termination Laws

```
theorem trm_wpe_Abort [trm_laws] :
"trm#(abort) = Falsep"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem trm_wpe_Magic [trm_laws] :
"trm#(magic) = Truep"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem trm_wpe_Skip [trm_laws] :
"trm#(skip) = Truep"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
done
```

```
theorem trm_wpe_Update [trm_laws] :
"trm#(⊙(u)) = True_p"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
done

theorem trm_wpe_Seq [trm_laws] :
"trm#(S ; T) = ⌊wpe(S, trm#(T)⇑_p)⌋_p"
apply (unfold trm_wpe_def)
apply (simp add: Neg_Neg_Pred)
apply (subst Shriek_Elate_Pred_inverse)
— Subgoal 1
apply (rule wpe_false_super_closure)
apply (simp add: pred_defs)
— Subgoal 2
apply (rule Shriek_equals_Prop_Pred)
apply (rule wpe_false_super_closure)
apply (rule wpe_false_super_closure)
apply (simp add: pred_defs)
done

theorem trm_wpe_Pre [trm_laws] :
"trm#(p | S) = p ∧_p trm#(S)"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem trm_wpe_Guard [trm_laws] :
"trm#(g → S) = g ⇒_p trm#(S)"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem trm_wpe_Choice [trm_laws] :
"trm#(S ⊓ T) = (trm#(S) ∧_p trm#(T))"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem trm_wpe_Angelic [trm_laws] :
"trm#(S ⊔ T) = (trm#(S) ∨_p trm#(T))"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem fis_wpe_Biased_Choice [fis_laws] :
"fis#(S [+] T) = fis#(S ⊓ T)"
apply (unfold Biased_Choice_def)
apply (unfold fis_laws)
apply (simp add: pred_defs)
apply (auto)
```

```
done

theorem trm_wpe_Pref [trm_laws] :
"trm#(S) s ⟹ trm#(T) s ⟹ trm#(S ≫ T) s"
"trm#(S ≫ T) s ⟹ (trm#(S) ∨p trm#(T)) s"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem trm_wpe_UChoice [trm_laws] :
"trm#(⨅b SS) s = (∀S. S ∈b SS ⟶ trm#(S) s)"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done

theorem trm_wpe_AChoice [trm_laws] :
"trm#(⨆b SS) s = (∃S. S ∈b SS ∧ trm#(S) s)"
apply (unfold trm_wpe_def)
apply (simp_all add: pred_defs)
apply (gd3_prove_tac)
done
```

8.7.5 Isomorphism Laws

```
theorem wpe_Prop_hom [rule_format] :
"S ∈ GSL ⟹ ∀Q. ⌊wpe(S, Q)⌋p = wp(S, ⌊Q⌋p)"
apply (erule GSL.induct)
apply (simp_all add: fun_eq_iff)
apply (gd3_prove_tac)
done

theorem wpe_Shriek_hom [rule_format] :
"S ∈ GSL ⟹ ∀Q. wpe(S, Q)!p = wp(S, Q!p)"
apply (erule GSL.induct)
apply (simp_all add: fun_eq_iff)
apply (gd3_prove_tac)
done

theorem wp_wpe_link [rule_format] :
"S ∈ GSL ⟹ ∀Q. wp(S, Q) = ⌊wpe(S, Q↑p)⌋p"
"S ∈ GSL ⟹ ∀Q. wp(S, Q) = ⌊wpe(S, Q⇑p)⌋p"
"S ∈ GSL ⟹ ∀Q. wp(S, Q) = wpe(S, Q⇑p)!p"
apply (simp add: wpe_Prop_hom)
apply (simp add: wpe_Prop_hom)
apply (subst wpe_Elate_externalise)
apply (simp add: wpe_Prop_hom)
done

theorem fis_iso :
"S ∈ GSL ⟹ fis(S) = fis#(S)"
apply (erule GSL.induct)
apply (unfold fis_laws)
apply (simp_all)
— Subgoal 1
```

apply (simp add: wp_wpe_link(2))
— Subgoal 2
apply (meson)
done

theorem trm_iso :
"S ∈ GSL ⟹ trm(S) = trm#(S)"
apply (erule GSL.induct)
apply (unfold trm_laws)
apply (simp_all)
— Subgoal 1
apply (simp add: wp_wpe_link(2))
— Subgoal 2
apply (meson)
done

theorem wp_wpe_iso :
"S ∈ GSL ⟹
 T ∈ GSL ⟹ (S ⊑wp T) ⟷ (S ⊑wpe T)"
apply (unfold wp_equiv_def wp_ref_def)
apply (unfold wpe_equiv_def wpe_ref_def)
apply (unfold GD3_Imp_Prop_Pred_distr)
apply (subst wpe_Prop_hom, assumption)
apply (subst wpe_Prop_hom, assumption)
apply (subst GD3_Forall_Pred_transfer)
apply (unfold pred_defs)
apply (rule refl)
done

8.7.6 Miscellaneous Laws

theorem wpe_false_implies_Q :
"⟨wpe(S, false$_g$)!$_p$ ⇒$_p$ wpe(S, Q)!$_p$⟩"
apply (rule wpe_Shriek_mono)
apply (gd3_prove_tac)
done

theorems wpe_false_implies_Q' =
  wpe_false_implies_Q [unfold_preds]

theorem not_fis_sharp_imp_wpe_Shriek [rule_format] :
"∀Q s. ¬ fis#(S) s ⟶ (wpe(S, Q) s)!"
apply (unfold fis_wpe_def)
apply (clarsimp)
apply (rule wpe_false_implies_Q')
apply (simp add: pred_defs)
done

theorem not_fis_imp_wpe_Shriek [rule_format] :
"S ∈ GSL ⟹ ∀Q s. ¬ fis(S) s ⟶ (wpe(S, Q) s)!"
apply (erule GSL.induct)
apply (unfold fis_laws)
apply (simp_all)
apply (gd3_prove_tac)
apply (subgoal_tac "wp(S, wpe(T, Q)!$_p$) s")
— Subgoal 1

```
apply (metis Shriek_Pred_def wpe_Shriek_hom)
— Subgoal 2
apply (erule wp_mono_elim)
apply (simp add: Imp_Pred_def Shriek_Pred_def)
done

theorem fis_is_not_wpe_Shriek_false :
"S ∈ GSL ⟹ fis(S) = ¬p wpe(S, falseg)!p"
apply (erule GSL.induct)
apply (unfold fis_laws)
apply (simp_all) prefer 3
apply (simp add: Neg_Neg_Pred wpe_Shriek_hom)
apply (gd3_prove_tac)
apply (transfer')
apply (meson)
done

theorem fis_is_not_wpe_Shriek_true :
"S ∈ GSL ⟹ fis(S) = ¬p wpe(S, trueg)!p"
apply (erule GSL.induct)
apply (unfold fis_laws)
apply (simp_all) prefer 3
apply (simp add: Neg_Neg_Pred wpe_Shriek_hom)
apply (gd3_prove_tac)
apply (transfer')
apply (meson)
done
```

8.7.7 Monotonicity Laws

```
theorem Seq_wpe_left_mono :
"S ⊑wpe S' ⟹ S ; T ⊑wpe S' ; T"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem Seq_wpe_right_mono :
"T ⊑wpe T' ⟹ S ; T ⊑wpe S ; T'"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
oops

theorem Pre_wpe_mono :
"S ⊑wpe S' ⟹ (P | S) ⊑wpe (P | S')"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem Guard_wpe_mono :
"S ⊑wpe S' ⟹ (G → S) ⊑wpe (G → S')"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem Choice_wpe_mono :
"S ⊑wpe S' ⟹ T ⊑wpe T' ⟹ S ⊓ T ⊑wpe S' ⊓ T'"
```

```
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem Angelic_wpe_mono :
"S ⊑wpe S' ⟹ T ⊑wpe T' ⟹ S ⊔ T ⊑wpe S' ⊔ T'"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem Pref_wpe_left_mono :
"S ⊑wpe S' ⟹ S ≫ T ⊑wpe S' ≫ T"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
oops

theorem Pref_wpe_right_mono :
"T ⊑wpe T' ⟹ S ≫ T ⊑wpe S ≫ T'"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem UChoice_wpe_approx :
"S ∈ SS ⟹ (⨅ SS) ⊑wpe (GSLUpcast S)"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem AChoice_wpe_refines :
"S ∈ SS ⟹ (GSLUpcast S) ⊑wpe (⨆ SS)"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
done

theorem UChoice_wpe_mono :
"SS' ⊆ SS ⟹ (⨅ SS) ⊑wpe (⨅ SS')"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
apply (force)
done

theorem AChoice_wpe_mono :
"SS ⊆ SS' ⟹ (⨆ SS) ⊑wpe (⨆ SS')"
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
apply (force)
done
```

8.7.8 Preference Laws

```
theorem Pref_idem :
"P ≫ P ≡wpe P"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
done
```

```
theorem Pref_unit_zero_laws :
"magic ≫ P ≡wpe P"
"P ≫ magic ≡wpe P"
"abort ≫ P ≡wpe abort"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
done
```

But notice that we do not have the following property.

```
theorem "P ≫ abort ≡wpe abort"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
apply (erule_tac Q = "(wpe(P, Q) s)!" in contrapos_np)
apply (simp)
oops

theorem Pref_assoc :
"(P ≫ Q) ≫ R ≡wpe P ≫ (Q ≫ R)"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
done

theorem Pref_Seq_distr :
"(S ≫ T); R ≡wpe (S; R) ≫ (T; R)"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
done

theorem Pref_Choice_distr :
"(S ≫ T) ⊓ R ⊑wpe (S ⊓ R) ≫ (T ⊓ R)"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
done
```

But note that equivalence does not appear to hold.

```
theorem Pref_Choice_distr' :
"(S ≫ T) ⊓ R ≡wpe (S ⊓ R) ≫ (T ⊓ R)"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
oops

theorem Choice_refby_Pref :
"(S ⊓ T) ⊑wpe (S ≫ T)"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
done

theorem Pref_refby_Biased :
"(S ≫ T) ⊑wpe (S [+] T)"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (unfold Biased_Choice_def)
apply (gd3_prove_tac)
apply (erule_tac Q = "(wpe(S, Q) s)!" in contrapos_np)
apply (erule not_fis_sharp_imp_wpe_Shriek)
done
```

8.8 Proof Experiments

The next law necessitates the change in the semantics of guards when moving from LVT to GD3.

```
theorem Guard_logic_test :
"wpe(True_p → skip ≫ abort, true_g) = true_g"
apply (rule ext)
apply (gd3_prove_tac)
done

theorem Choice_backtracks_wp :
"(∀s. ¬(g (u1 s))) ⟹
 (∀s. (g (u2 s))) ⟹
  (⊙(u1) ⊓ ⊙(u2); g → skip) ≡wp ⊙(u2)"
apply (unfold wp_equiv_def wp_ref_def)
apply (gd3_prove_tac)
done

theorem Choice_backtracks_wpe :
"(∀s. ¬(g (u1 s))) ⟹
 (∀s. (g (u2 s))) ⟹
  (⊙(u1) ⊓ ⊙(u2); g → skip) ≡wpe ⊙(u2)"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
done

theorem Pref_backtracks_wpe :
"(∀s. ¬(g (u1 s))) ⟹
 (∀s. (g (u2 s))) ⟹
  (⊙(u1) ≫ ⊙(u2); g → skip) ≡wpe ⊙(u2)"
apply (unfold wpe_equiv_def wpe_ref_def)
apply (gd3_prove_tac)
done
end
```

9 Galois Connections

```
theory Galois
imports GSL wp wpe
begin
```

9.1 Adjoint Functions

```
function (domintros) L ::
  "('state, 'bound) gsl_ext ⇒
   ('state, 'bound) gsl_ext" ("L'(_')") where
"L (Skip) = (Skip)" |
"L (Update u) = (Update u)" |
"L (Seq S T) = (Seq (L S) (L T))" |
"L (Pre p S) = (Pre p (L S))" |
"L (Guard g S) = (Guard g (L S))" |
"L (Choice S T) = (Choice (L S) (L T))" |
"L (Angelic S T) = (Angelic (L S) (L T))" |
"L (Pref S T) = (Choice (L S) (L T))" |
"L (UChoice SS) = (UChoice (Abs_bset (L ` (set_bset SS))))" |
"L (AChoice SS) = (AChoice (Abs_bset (L ` (set_bset SS))))"
by pat_completeness auto
termination
apply (rule allI)
apply (induct_tac x)
apply (simp_all add: L.domintros)
done

theorem L_UChoice [simp] :
"L (UChoice SS) = (UChoice (L `b SS))"
apply (unfold L.simps)
apply (metis bimage.rep_eq set_bset_inverse)
done

theorem L_AChoice [simp] :
"L (AChoice SS) = (AChoice (L `b SS))"
apply (unfold L.simps)
apply (metis bimage.rep_eq set_bset_inverse)
done

declare L.simps(9) [simp del]
declare L.simps(10) [simp del]

abbreviation (input) R ::
  "('state, 'bound) gsl_ext ⇒
   ('state, 'bound) gsl_ext" where
"R y ≡ y"

notation L ("L'(_')")
notation R ("R'(_')")
```

9.2 Link Properties

```
theorem L_in_GSL [simp] :
"L(S) ∈ GSL"
apply (induct S)
```

```
apply (simp_all add: GSL.intros)
— Subgoal 1
apply (rule GSL.intros)
apply (transfer')
apply (clarsimp)
— Subgoal 2
apply (rule GSL.intros)
apply (transfer')
apply (clarsimp)
done

theorem L_idem [simp] :
"S ∈ GSL ⟹ L(S) = S"
apply (erule GSL.induct)
apply (simp_all)
— Subgoal 1
apply (transfer', safe)
apply (simp_all add: image_iff) [2]
— Subgoal 2
apply (transfer', safe)
apply (simp_all add: image_iff) [2]
done

theorem L_approx :
"L(S) ⊑wpe S"
apply (induct_tac S)
apply (simp_all)
apply (unfold wpe_ref_def)
apply (gd3_prove_tac)
— Subgoal 1
apply (rename_tac S T Q s)
apply (simp add: wpe_Prop_hom [unfold_preds])
apply (simp add: wp_mono_elim [unfold_preds])
— Subgoal 2
apply (rename_tac SS Q s S)
apply (transfer')
apply (blast)
— Subgoal 3
apply (rename_tac SS Q s S)
apply (transfer')
apply (blast)
done

lemma wpe_Elate_Shriek_wp_L [rule_format] :
"∀Q s. (wpe(S, Q⇑p) s)! ⟶ wp(L(S), Q) s"
— Do we really need induction here?
apply (induct_tac S)
apply (simp_all)
prefer 3
— Subgoal 3
apply (rename_tac S T)
apply (clarsimp)
apply (metis (no_types, lifting)
  Shriek_Pred_def wp_mono_elim' wpe_Elate_externalise(1) wpe_Prop_to_Shriek)
apply (gd3_prove_tac)
```

— Subgoal 1
apply (rename_tac SS Q s S)
apply (transfer')
apply (blast)
— Subgoal 2
apply (rename_tac SS Q s S)
apply (transfer')
apply (blast)
done

theorem L_strongest :
"S ∈ GSL ⟹ S ⊑wpe T ⟹ S ⊑wpe L(T)"
apply (subst sym [OF wp_wpe_iso])
apply (simp_all) [2]
apply (unfold wp_ref_def wpe_ref_alt_def)
apply (unfold pred_defs)
apply (clarsimp)
apply (rule wpe_Elate_Shriek_wp_L)
apply (drule_tac x = "Q⇑$_p$" **in** spec)
apply (drule_tac x = "s" **in** spec)
apply (simp add: GD3_Imp_Shriek)
apply (metis Prop_Pred_def pointwise(4) wp_wpe_link(2) wpe_Prop_to_Shriek)
done

9.3 Monotonicity

theorem L_mono :
"S ⊑wpe T ⟹ L(S) ⊑wpe L(T)"
using L_in_GSL L_approx L_strongest wpe_ref_trans **by** blast

theorem L_resp_equiv :
"S ≡wpe T ⟶ L(S) ≡wpe L(T)"
apply (simp add: L_mono wpe_equiv_def)
done

9.4 Galois Theorem

theorem Galois_LR :
"Y ∈ GSL ⟹ Y ⊑wp L(X) ⟷ R(Y) ⊑wpe X"
using L_in_GSL L_approx L_strongest wp_wpe_iso wpe_ref_trans **by** blast

9.5 Proof Tactic

method Pref_intro_tac = (
  (subst sym [OF Galois_LR]),
  (simp add: GSL.intros; fail),
  (simp add: wp_ref_refl)?)

9.6 Proof Experiments

theorem "{S, T, U} ⊆ GSL ⟹ (S ⊓ T) ; U ⊑wpe (S ≫ T) ; U"
apply (subst sym [OF Galois_LR])
— Subgoal 1
apply (simp add: GSL.intros)
— Subgoal 2
apply (simp add: wp_ref_refl)?

done

theorem "{S, T, U} $\subseteq$ GSL $\Longrightarrow$ (S $\sqcap$ T) ; U $\sqsubseteq$wpe (S $\gg$ T) ; U"
apply (Pref_intro_tac)
done

Attempt at providing a semantic characterisation of L.

theorem L_wpe [rule_format] :

"$\forall$Q. wpe(L(S), Q) = wpe(S, Q) $\wedge_g$ wpe(S, $\lfloor$Q$\rfloor_p\Uparrow_p$)"
apply (induct_tac S)
apply (simp_all)
— Subgoal 1
apply (clarify)
apply (rule ext)
apply (gd3_prove_tac) [1]
— Subgoal 2
apply (clarify)
apply (rule ext)
apply (gd3_prove_tac) [1]
— Subgoal 3
apply (clarify)
apply (rename_tac S T Q)
apply (thin_tac "$\forall$Q. wpe(L S, Q) = wpe(S, Q) $\wedge_g$ wpe(S, $\lfloor$Q$\rfloor_p\Uparrow_p$)")
apply (thin_tac "$\forall$Q. wpe(L T, Q) = wpe(T, Q) $\wedge_g$ wpe(T, $\lfloor$Q$\rfloor_p\Uparrow_p$)")
defer
— Subgoal 4
apply (clarify)
apply (rule ext)
apply (gd3_prove_tac) [1]
— Subgoal 5
apply (clarify)
apply (rule ext)
apply (gd3_prove_tac) [1]
— Subgoal 6
apply (clarify)
apply (rule ext)
apply (gd3_prove_tac) [1]
— Subgoal 7
apply (clarify)
apply (rename_tac S T Q)
apply (thin_tac "$\forall$Q. wpe(L S, Q) = wpe(S, Q) $\wedge_g$ wpe(S, $\lfloor$Q$\rfloor_p\Uparrow_p$)")
apply (thin_tac "$\forall$Q. wpe(L T, Q) = wpe(T, Q) $\wedge_g$ wpe(T, $\lfloor$Q$\rfloor_p\Uparrow_p$)")
defer
— Subgoal 8
apply (clarify)
apply (rename_tac S T Q)
apply (thin_tac "$\forall$Q. wpe(L S, Q) = wpe(S, Q) $\wedge_g$ wpe(S, $\lfloor$Q$\rfloor_p\Uparrow_p$)")
apply (thin_tac "$\forall$Q. wpe(L T, Q) = wpe(T, Q) $\wedge_g$ wpe(T, $\lfloor$Q$\rfloor_p\Uparrow_p$)")
defer
— Subgoal 9
apply (clarify)
apply (transfer')

apply (simp add: fun_eq_iff)
apply (gd3_prove_tac) [1]
— Subgoal 10
apply (clarify)
apply (transfer')
apply (simp add: fun_eq_iff)
apply (gd3_prove_tac) [1]
oops
end