Generating Cases from Labeled Subgoals

Lars Noschinski

March 17, 2025

Contents

Abstract

Isabelle/Isar provides *named cases* to structure proofs. This article contains an implementation of a proof method `casify`, which can be used to easily extend proof tools with support for named cases. Such a proof tool must produce labeled subgoals, which are then interpreted by `casify`.

As examples, this work contains verification condition generators producing named cases for three languages: The Hoare language from `HOL/Library`, a monadic language for computations with failure (inspired by the AutoCorres tool), and a language of conditional expressions. These VCGs are demonstrated by a number of example programs.

theory *Case-Labeling*
imports *Main*
keywords *print-nested-cases* :: *diag*
begin

1 Labeling Subgoals

context begin
qualified type-synonym *prg-ctxt-var* = *unit*
qualified type-synonym *prg-ctxt* = *string* × *nat* × *prg-ctxt-var list*

Embed variables in terms

qualified definition *VAR* :: *'v* ⇒ *prg-ctxt-var* **where**
VAR - = ()

Labeling of a subgoal

qualified definition *VC* :: *prg-ctxt list* ⇒ *'a* ⇒ *'a* **where**
VC ct P ≡ *P*

Computing the statement numbers and context

qualified definition *CTXT* :: *nat* ⇒ *prg-ctxt list* ⇒ *nat* ⇒ *'a* ⇒ *'a* **where**
CTXT inp ct outp P ≡ *P*

Labeling of a term binding or assumption

qualified definition *BIND* :: *string* ⇒ *nat* ⇒ *'a* ⇒ *'a* **where**
BIND name inp P ≡ *P*

Hierarchy labeling

qualified definition *HIER* :: *prg-ctxt list* ⇒ *'a* ⇒ *'a* **where**
HIER ct P ≡ *P*

Split Labeling. This is used as an assumption

qualified definition *SPLIT* :: *'a* ⇒ *'a* ⇒ *bool* **where**
SPLIT v w ≡ *v* = *w*

Disambiguation Labeling. This is used as an assumption

qualified definition *DISAMBIG* :: *nat* ⇒ *bool* **where**
DISAMBIG n ≡ *True*

lemmas *LABEL-simps* = *BIND-def CTXT-def HIER-def SPLIT-def VC-def*

lemma *Initial-Label*: *CTXT 0* [] *outp P* ⟹ *P*
by (*simp add*: *Case-Labeling.CTXT-def*)

lemma
BIND-I: *P* ⟹ *BIND name inp P* **and**
BIND-D: *BIND name inp P* ⟹ *P* **and**

VC-I: $P \Longrightarrow VC\ ct\ P$
unfolding *Case-Labeling.BIND-def Case-Labeling.VC-def* **.**

lemma *DISAMBIG-I*: $(DISAMBIG\ n \Longrightarrow P) \Longrightarrow P$
by (*auto simp*: *DISAMBIG-def Case-Labeling.VC-def*)

lemma *DISAMBIG-E*: $(DISAMBIG\ n \Longrightarrow P) \Longrightarrow P$
by (*auto simp*: *DISAMBIG-def*)

Lemmas for the tuple postprocessing

lemma *SPLIT-reflection*: $SPLIT\ x\ y \Longrightarrow (x \equiv y)$
unfolding *SPLIT-def* **by** (*rule eq-reflection*)

lemma *rev-SPLIT-reflection*: $(x \equiv y) \Longrightarrow SPLIT\ x\ y$
unfolding *SPLIT-def* **..**

lemma *SPLIT-sym*: $SPLIT\ x\ y \Longrightarrow SPLIT\ y\ x$
unfolding *SPLIT-def* **by** (*rule sym*)

lemma *SPLIT-thin-refl*: $[\![SPLIT\ x\ x;\ PROP\ W]\!] \Longrightarrow PROP\ W$ **.**

lemma *SPLIT-subst*: $[\![SPLIT\ x\ y;\ P\ x]\!] \Longrightarrow P\ y$
unfolding *SPLIT-def* **by** *hypsubst*

lemma *SPLIT-prodE*:
assumes *SPLIT* (*x1*, *y1*) (*x2*, *y2*)
obtains *SPLIT x1 x2 SPLIT y1 y2*
using *assms* **unfolding** *SPLIT-def* **by** *auto*

end

The labeling constants were qualified to not interfere with any other theory. The following locale allows using a nice syntax in other theories

locale *Labeling-Syntax* **begin**
abbreviation *VAR* **where** *VAR* ≡ *Case-Labeling.VAR*
abbreviation *VC* (‹*V*⟨(2-,-:/ -)⟩›) **where** *VC bl ct* ≡ *Case-Labeling.VC* (*bl* # *ct*)
abbreviation *CTXT* (‹*C*⟨(2-,-,-:/ -)⟩›) **where** *CTXT* ≡ *Case-Labeling.CTXT*
abbreviation *BIND* (‹*B*⟨(2-,-:/ -)⟩›) **where** *BIND* ≡ *Case-Labeling.BIND*
abbreviation *HIER* (‹*H*⟨(2-:/ -)⟩›) **where** *HIER* ≡ *Case-Labeling.HIER*
abbreviation *SPLIT* **where** *SPLIT* ≡ *Case-Labeling.SPLIT*
end

Lemmas for converting terms from *Suc/0* notation to numerals

lemma *Suc-numerals-conv*:
$Suc\ 0 = Numeral1$
$Suc\ (numeral\ n) = numeral\ (n + num.One)$
by *auto*

lemmas *Suc-numeral-simps* = *Suc-numerals-conv add-num-simps*

2 Casify

Introduces a command **print-nested-cases**. This is similar to **print-cases**, but shows also the nested cases.

ML-file ‹*print-nested-cases.ML*›

ML-file ‹*util.ML*›

Introduces the proof method.

ML-file ‹*casify.ML*›

```
ML ‹
  val casify-defs = Casify.Options { simp-all-cases=true, split-right-only=true, protect-subgoals=false }
›
```

```
method-setup prepare-labels = ‹
  Scan.succeed (fn ctxt => SIMPLE-METHOD (ALLGOALS (Casify.prepare-labels-tac ctxt)))
› VCG labelling: prepare labels
```

method-setup *casify* = ‹*Casify.casify-method-setup casify-defs*›
  VCG labelling: *Turn the labels into cases*

end

3 Examples

3.1 A labeling VCG for a monadic language

theory *Monadic-Language*
imports
  Complex-Main
  ../Case-Labeling
  HOL−Eisbach.Eisbach
begin

ML-file ‹*../util.ML*›

```
ML ‹
  fun vcg-tac nt-rules nt-comb ctxt =
    let
      val rules = Named-Theorems.get ctxt nt-rules
```

$val\ comb = Named\text{-}Theorems.get\ ctxt\ nt\text{-}comb$
$in\ REPEAT\text{-}ALL\text{-}NEW\text{-}FWD\ (\ resolve\text{-}tac\ ctxt\ rules\ ORELSE'\ (resolve\text{-}tac\ ctxt\ comb\ THEN'\ resolve\text{-}tac\ ctxt\ rules))\ end$
›

This language is inspired by the languages used in AutoCorres [1]

consts *bind* :: $'a\ option \Rightarrow ('a \Rightarrow 'b\ option) \Rightarrow 'b\ option$ (**infixr** ‹|>>› *4*)
consts *return* :: $'a \Rightarrow 'a\ option$
consts *while* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a\ option) \Rightarrow ('a \Rightarrow 'a\ option)$
consts *valid* :: $bool \Rightarrow 'a\ option \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$

named-theorems *vcg*
named-theorems *vcg-comb*

method-setup *vcg* = ‹
Scan.succeed (fn ctxt => SIMPLE-METHOD (FIRSTGOAL (vcg-tac @{named-theorems vcg} @{named-theorems vcg-comb} ctxt)))
›

axiomatization where
return[*vcg*]: $valid\ (Q\ x)\ (return\ x)\ Q$ **and**
bind[*vcg*]: $[\![\bigwedge x.\ valid\ (R\ x)\ (c2\ x)\ Q;\ valid\ P\ c1\ R]\!] \Longrightarrow valid\ P\ (bind\ c1\ c2)\ Q$ **and**
while[*vcg*]: $\bigwedge c.\ [\![\bigwedge x.\ valid\ (I\ x \wedge b\ x)\ (c\ x)\ I;\ \bigwedge x.\ I\ x \wedge \neg b\ x \Longrightarrow Q\ x]\!] \Longrightarrow valid\ (I\ x)\ (while\ b\ I\ c\ x)\ Q$ **and**
cond[*vcg*]: $\bigwedge b\ c1\ c2.\ valid\ P1\ c1\ Q \Longrightarrow valid\ P2\ c2\ Q \Longrightarrow valid\ (if\ b\ then\ P1\ else\ P2)\ (if\ b\ then\ c1\ else\ c2)\ Q$ **and**
case-prod[*vcg*]: $\bigwedge P.\ [\![\bigwedge x\ y.\ v = (x,y) \Longrightarrow valid\ (P\ x\ y)\ (B\ x\ y)\ Q]\!]$
$\Longrightarrow valid\ (case\ v\ of\ (x,y) \Rightarrow P\ x\ y)\ (case\ v\ of\ (x,y) \Rightarrow B\ x\ y)\ Q$ **and**
conseq[*vcg-comb*]: $[\![valid\ P'\ c\ Q;\ P \Longrightarrow P']\!] \Longrightarrow valid\ P\ c\ Q$

Labeled rules

named-theorems *vcg-l*
named-theorems *vcg-l-comb*
named-theorems *vcg-elim*

method-setup *vcg-l* = ‹
Scan.succeed (fn ctxt => SIMPLE-METHOD (FIRSTGOAL (vcg-tac @{named-theorems vcg-l} @{named-theorems vcg-l-comb} ctxt)))
›

method *vcg-l'* = (*vcg-l*; (*elim vcg-elim*)?)

context begin
interpretation *Labeling-Syntax* **.**

lemma *L-return*[*vcg-l*]: $CTXT\ inp\ ct\ (Suc\ inp)\ (valid\ (P\ x)\ (return\ x)\ P)$
unfolding *LABEL-simps* **by** (*rule return*)

lemma *L-bind*[*vcg-l*]:
assumes $\bigwedge x.$ *CTXT* (*Suc outp′*) ((″*bind*″,*outp′*, [*VAR x*]) # *ct*) *outp* (*valid* (*R x*) (*c2 x*) *Q*)
assumes *CTXT inp ct outp′* (*valid P c1 R*)
shows *CTXT inp ct outp* (*valid P* (*bind c1 c2*) *Q*)
using *assms* **unfolding** *LABEL-simps* **by** (*rule bind*)

lemma *L-while*[*vcg-l*]:
fixes *inp ct* **defines** *ct′* ≡ λ*x*. (″*while*″, *inp*, [*VAR x*]) # *ct*
assumes $\bigwedge x.$ *CTXT* (*Suc inp*) (*ct′ x*) *outp′*
(*valid* (*BIND* ″*inv-pre*″ *inp* (*I x*) ∧ *BIND* ″*lcond*″ *inp* (*b x*)) (*c x*) (λ*x*. *BIND* ″*inv-post*″ *inp* (*I x*)))
assumes $\bigwedge x.$ *B*⟨″*inv-pre*″,*inp*: *I x*⟩ ∧ *B*⟨″*lcond*″,*inp*: ¬*b x*⟩ ⟹ *VC* (″*post*″,*outp′* , []) (*ct′ x*) (*P x*)
shows *CTXT inp ct* (*Suc outp′*) (*valid* (*I x*) (*while b I c x*) *P*)
using *assms*(*2*−) **unfolding** *LABEL-simps* **by** (*rule while*)

lemma *L-cond*[*vcg-l*]:
fixes *inp ct* **defines** *ct′* ≡ (″*if*″,*inp*,[]) # *ct*
assumes *C*⟨*Suc inp*, (″*then*″,*inp*,[]) # *ct′*,*outp*: *valid P1 c1 Q*⟩
assumes *C*⟨*Suc outp*, (″*else*″,*outp*,[]) # *ct′*,*outp′*: *valid P2 c2 Q*⟩
shows *C*⟨*inp*,*ct*,*outp′*: *valid* (*if B*⟨″*cond*″,*inp*: *b*⟩ *then B*⟨″*then*″,*inp*: *P1*⟩ *else B*⟨″*else*″,*inp*: *P2*⟩) (*if b then c1 else c2*) *Q*⟩
using *assms*(*2*−) **unfolding** *LABEL-simps* **by** (*rule cond*)

lemma *L-case-prod*[*vcg-l*]:
assumes $\bigwedge x\ y.$ *v* = (*x*,*y*) ⟹ *CTXT inp ct outp* (*valid* (*P x y*) (*B x y*) *Q*)
shows *CTXT inp ct outp* (*valid* (*case v of* (*x*,*y*) ⇒ *P x y*) (*case v of* (*x*,*y*) ⇒ *B x y*) *Q*)
using *assms* **unfolding** *LABEL-simps* **by** (*rule case-prod*)

lemma *L-conseq*[*vcg-l-comb*]:
assumes *CTXT* (*Suc inp*) *ct outp* (*valid P′ c Q*)
assumes *P* ⟹ *VC* (″*conseq*″,*inp*,[]) *ct P′*
shows *CTXT inp ct outp* (*valid P c Q*)
using *assms* **unfolding** *LABEL-simps* **by** (*rule conseq*)

lemma *L-assm-conjE*[*vcg-elim*]:
assumes *BIND name inp* (*P* ∧ *Q*) **obtains** *BIND name inp P BIND name inp Q*
using *assms* **unfolding** *LABEL-simps* **by** *auto*

declare *conjE*[*vcg-elim*]

end

lemma *dvd-div*:
  fixes *a b c* :: *int*
  assumes *a dvd b c dvd b coprime a c*
  shows *a dvd* (*b div c*)
  using *assms coprime-dvd-mult-left-iff* **by** *fastforce*

lemma *divides*:
valid
  (*0* < (*a* :: *int*))
  (
    return a
    |>> (λ*n*.
      while
        (λ*n*. *even n*)
        (λ*n*. *0* < *n* ∧ *n dvd a* ∧ (∀ *m*. *odd m* ∧ *m dvd a* ⟶ *m dvd n*))
        (λ*n*. *return* (*n div 2*))
        n
)
)
  (λ*r*. *odd r* ∧ *r dvd a* ∧ (∀ *m*. *odd m* ∧ *m dvd a* ⟶ *m* ≤ *r*))

  apply *vcg*
    apply (*auto simp add*: *zdvd-imp-le dvd-div elim*!:
      evenE intro: *dvd-mult-right*)
  done

lemma *L-divides*:
valid
  (*0* < (*a* :: *int*))
  (
    return a
    |>> (λ*n*.
      while
        (λ*n*. *even n*)
        (λ*n*. *0* < *n* ∧ *n dvd a* ∧ (∀ *m*. *odd m* ∧ *m dvd a* ⟶ *m dvd n*))
        (λ*n*. *return* (*n div 2*))
        n
)
)
  (λ*r*. *odd r* ∧ *r dvd a* ∧ (∀ *m*. *odd m* ∧ *m dvd a* ⟶ *m* ≤ *r*))

  apply (*rule Initial-Label*)
  apply *vcg-l'*
proof *casify*
print-nested-cases
  case *bind*
  { case (*while n*)
    { case *post* **then show** *?case* **by** (*auto simp*: *zdvd-imp-le*)
    next

```
      case conseq
      from ‹0 < n› ‹even n› have 0 < n div 2
        by (simp add: pos-imp-zdiv-pos-iff zdvd-imp-le)
      moreover
      from ‹n dvd a› ‹even n› have n div 2 dvd a
        by (metis dvd-div-mult-self dvd-mult-left)
      moreover
      { fix m assume odd m m dvd a
        then have m dvd n using conseq.inv-pre(3) by simp
        moreover note ‹even n›
        moreover from ‹odd m› have coprime m 2
       by (metis dvd-eq-mod-eq-0 invertible-coprime mult-cancel-left2 not-mod-2-eq-1-eq-0)
        ultimately
        have m dvd n div 2 by (rule dvd-div)
      }
      ultimately show ?case by auto
    }
  next
    case conseq then show ?case by auto
  }
qed
```

```
lemma add:
valid
  True
  (
    while
      — COND: (λ(r,j). j < (b :: nat))
      — INV: (λ(r,j). j ≤ b ∧ r = a + j)
      — BODY: (λ(r,j). return (r + 1, j + 1))
      — START: (a,0)
    |>> (λ(r,-). return r)
  )
  (λr. r = a + b)

  by vcg auto
```

```
lemma mult:
valid
  True
  (
    while
      — COND: (λ(r,i). i < (a :: nat))
      — INV: (λ(r,i). i ≤ a ∧ r = i ∗ b)
      — BODY: (λ(r,i).
        while
```

```
        — COND: (λ(r,j). j < b)
        — INV: (λ(r,j). i < a ∧ j ≤ b ∧ r = i ∗ b + j)
        — BODY: (λ(r,j). return (r + 1, j + 1))
        — START: (r,0)
      |>> (λ(r,-). return (r, i + 1))
    )
    — START: (0,0)
  |>> (λ(r,-). return r)
)
(λr. r = a ∗ b)

by vcg auto
```

4 Labeled

```
lemma L-mult:
valid
  True
  (
    while
      — COND: (λ(r,i). i < (a :: nat))
      — INV: (λ(r,i). i ≤ a ∧ r = i ∗ b)
      — BODY: (λ(r,i).
        while
          — COND: (λ(r,j). j < b)
          — INV: (λ(r,j). i < a ∧ j ≤ b ∧ r = i ∗ b + j)
          — BODY: (λ(r,j). return (r + 1, j + 1))
          — START: (r,0)
        |>> (λ(r,-). return (r, i + 1))
      )
      — START: (0,0)
    |>> (λ(r,-). return r)
  )
  (λr. r = a ∗ b)

  apply (rule Initial-Label)
  apply vcg-l′
proof casify
  case while
  { case while
    { case post then show ?case by auto
    next
      case conseq then show ?case by auto
    }
  next
    case post then show ?case by auto
  next
    case conseq then show ?case by auto
  }
```

```
next
  case conseq then show ?case by auto
qed

lemma L-paths:
valid
  (path ≠ [])
  ( while
      — COND: (λ(p,r). p ≠ [])
      — INV: (λ(p,r). distinct r ∧ hd (r @ p) = hd path ∧ last (r @ p) = last path)
      — BODY: (λ(p,r).
        return (hd p)
        |>> (λx.
          if (r ≠ [] ∧ x = hd r)
          then return []
          else (if x ∈ set r
            then return (takeWhile (λy. y ≠ x) r)
            else return (r))
        |>> (λr'. return (tl p, r' @ [x])
        )
        )
        )
      — START: (path, [])
    |>> (λ(-,r). return r)
  )
  (λr. distinct r ∧ hd r = hd path ∧ last r = last path)

  apply (rule Initial-Label)
  apply vcg-l'
proof casify
  case conseq then show ?case by auto
next
  case (while p r)
  { case conseq
    from conseq have r = [] ⟹ ?case by (cases p) auto
    moreover
    from conseq have r ≠ [] ⟹ hd p = hd r ⟹ ?case by (cases p) auto
    moreover
    { assume A: r ≠ [] hd p ≠ hd r
      have hd (takeWhile (λy. y ≠ hd p) r @ hd p # tl p) = hd r
        using A by (cases r) auto
      moreover
      have last (takeWhile (λy. y ≠ hd p) r @ hd p # tl p) = last (r @ p)
        using A ‹p ≠ []› by auto
      moreover
      have distinct (takeWhile (λy. y ≠ hd p) r @ [hd p])
        using conseq by (auto dest: set-takeWhileD)
      ultimately
      have ?case using A conseq by auto
```

```
    }
    ultimately show ?case by blast
  next
    case post then show ?case by auto
  }
qed

end
```

4.1 Decomposing Conditionals

```
theory Conditionals
imports
  Complex-Main
  ../Case-Labeling
  HOL−Eisbach.Eisbach
begin

context begin
  interpretation Labeling-Syntax .

  lemma DC-conj:
    assumes C⟨inp,ct,outp′: a⟩ C⟨outp′,ct,outp: b⟩
    shows C⟨inp,ct,outp: a ∧ b⟩
    using assms unfolding LABEL-simps by auto

  lemma DC-if:
    fixes ct defines ct′ ≡ λpos name. (name, pos,[]) # ct
    assumes H⟨ct′ inp ′′then′′: a⟩ ⟹ C⟨Suc inp,ct′ inp ′′then′′, outp′: b⟩
    assumes H⟨ct′ outp′ ′′else′′: ¬a⟩ ⟹ C⟨Suc outp′,ct′ outp′ ′′else′′, outp: c⟩
    shows C⟨inp,ct,outp: if a then b else c⟩
    using assms(2−) unfolding LABEL-simps by auto

  lemma DC-final:
    assumes V⟨(′′g′′,inp,[]), ct: a⟩
    shows C⟨inp,ct,Suc inp: a⟩
    using assms unfolding LABEL-simps by auto

end

method vcg-dc = (intro DC-conj DC-if; rule DC-final)

lemma
  assumes a: a
    and d: b ⟹ c ⟹ d
    and d′: b ⟹ c ⟹ d′
    and e: b ⟹ ¬c ⟹ e
    and f: ¬b ⟹ f
  shows a ∧ (if b then (if c then d ∧ d′ else e) else f)
```

apply (*rule Initial-Label*)
apply *vcg-dc*
proof *casify*
case *g* **show** *?case* **by** (*fact a*)
next
case *then* **note** $b = \langle b \rangle$
{ case *then* **note** $c = \langle c \rangle$
{ case *g* **show** *?case* **using** *b c* **by** (*fact d*)
next
case *ga* **show** *?case* **using** *b c* **by** (*fact d′*)
}
next
case *else*
{ case *g* **show** *?case* **using** *then else* **by** (*fact e*) **}**
}
next
case *else*
{ case *g* **show** *?case* **using** *else* **by** (*fact f*) **}**
qed

4.2 Protecting similar subgoals

The proof below fails if the `disambig_subgoals` option is omitted: all three subgoals have the same conclusion and can be discharged without using their assumptions. If the case **g** is solved first, it discharges instead the subgoal $a \Longrightarrow b$, making the case **then** fail afterwards.

The `disambig_subgoals` options prevents this by inserting vacuous assumptions.

lemma
assumes *b*
shows (*if a then b else b*) $\wedge$ *b*
apply (*rule Initial-Label*)
apply *vcg-dc*
proof (*casify* (*disambig-subgoals*))
case *g* **show** *?case* **by** (*fact* ‹*b*›)
next
case *then* **case** *g* **show** *?case* **by** (*fact* ‹*b*›)
next
case *else* **case** *g* **show** *?case* **by** (*fact* ‹*b*›)
qed

4.3 Unnamed Cases

lemma
assumes $a \Longrightarrow b \; \neg a \Longrightarrow c \; d$
shows (*if a then b else c*) $\wedge$ *d*
apply (*rule Initial-Label*)
apply *vcg-dc*

```
  apply (simp-all only: LABEL-simps)[2]
proof (casify (disambig-subgoals))
  case unnamed from ‹a› show ?case by fact
next
  case unnameda from ‹¬a› show ?case by fact
next
  case g show ?case by fact
qed

end
theory Labeled-Hoare
imports
  ../../Case-Labeling
  HOL−Hoare.Hoare-Logic
begin
```

4.4 A labeling VCG for HOL/Hoare

```
context begin
  interpretation Labeling-Syntax .

  lemma LSeqRule:
    assumes C⟨IC,CT,OC1: Valid P c1 a1 Q⟩
      and C⟨Suc OC1,CT,OC: Valid Q c2 a2 R⟩
    shows C⟨IC,CT,OC: Valid P (Seq c1 c2) (Aseq a1 a2) R⟩
    using assms unfolding LABEL-simps by (rule SeqRule)

  lemma LSkipRule:
    assumes V⟨(''weaken'', IC, []),CT: p ⊆ q⟩
    shows C⟨IC,CT,IC: Valid p SKIP a q⟩
    using assms unfolding LABEL-simps  by (rule SkipRule)

  lemmas LAbortRule = LSkipRule  — dummy version

  lemma LBasicRule:
    assumes V⟨(''basic'', IC, []),CT: p ⊆ {s. f s ∈ q}⟩
    shows C⟨IC,CT,IC: Valid p (Basic f) a q⟩
    using assms unfolding LABEL-simps  by (rule BasicRule)

  lemma LCondRule:
    fixes IC CT defines CT' ≡ (''cond'', IC, []) # CT
    assumes V⟨(''vc'', IC, []),(''cond'', IC, []) # CT: p ⊆ {s. (s ∈ b ⟶ s ∈ w)
∧ (s ∉ b ⟶ s ∈ w')}⟩
     and C⟨Suc IC,(''then'', IC, []) # (''cond'', IC, []) # CT,OC1: Valid w c1 a1
q⟩
      and C⟨Suc OC1,(''else'', Suc OC1, []) # (''cond'', IC, []) # CT,OC: Valid
w' c2 a2 q⟩
    shows C⟨IC,CT,OC: Valid p (Cond b c1 c2) (Acond a1 a2) q⟩
    using assms(2−) unfolding LABEL-simps by (rule CondRule)
```

lemma *LWhileRule*:
fixes *IC CT* **defines** $CT' \equiv (''while'', IC, []) \# CT$
assumes $V\langle(''precondition'', IC, []),(''while'', IC, []) \# CT: p \subseteq i\rangle$
and $C\langle Suc\ IC,(''invariant'', Suc\ IC, []) \# (''while'', IC, []) \# CT,OC: Valid\ (i \cap b)\ c\ (A\ 0)\ i\rangle$
and $V\langle(''postcondition'', IC, []),(''while'', IC, []) \# CT: i \cap - b \subseteq q\rangle$
shows $C\langle IC,CT,OC: Valid\ p\ (While\ b\ c)\ (Awhile\ i\ v\ A)\ q\rangle$
using *assms*(*2*−) **unfolding** *LABEL-simps* **by** (*rule WhileRule*)

lemma *LABELs-to-prems*:
$(C\langle IC, CT, OC: True\rangle \Longrightarrow P) \Longrightarrow C\langle IC, CT, OC: P\rangle$
$(V\langle x, ct: True\rangle \Longrightarrow P) \Longrightarrow V\langle x, ct: P\rangle$
unfolding *LABEL-simps* **by** *simp-all*

lemma *LABELs-to-concl*:
$C\langle IC, CT, OC: True\rangle \Longrightarrow C\langle IC, CT, OC: P\rangle \Longrightarrow P$
$V\langle x, ct: True\rangle \Longrightarrow V\langle x, ct: P\rangle \Longrightarrow P$
unfolding *LABEL-simps* **.**

end

ML-file ‹*labeled-hoare-tac.ML*›

method-setup *labeled-vcg* = ‹
Scan.succeed (*fn ctxt* => *SIMPLE-METHOD′* (*Labeled-Hoare.hoare-tac ctxt* (*K all-tac*)))›
verification condition generator

method-setup *labeled-vcg-simp* = ‹
Scan.succeed (*fn ctxt* => *SIMPLE-METHOD′* (*Labeled-Hoare.hoare-tac ctxt* (*asm-full-simp-tac ctxt*)))›
verification condition generator

method-setup *casified-vcg* = ‹
Scan.lift (*Casify.casify-options casify-defs*) >>
(*fn opt* => *fn ctxt* => *Util.SIMPLE-METHOD-CASES* (
HEADGOAL (*Labeled-Hoare.hoare-tac ctxt* (*K all-tac*))
THEN-CONTEXT Casify.casify-tac opt))
›

method-setup *casified-vcg-simp* = ‹
Scan.lift (*Casify.casify-options casify-defs*) >>
(*fn opt* => *fn ctxt* => *Util.SIMPLE-METHOD-CASES* (
HEADGOAL (*Labeled-Hoare.hoare-tac ctxt* (*asm-full-simp-tac ctxt*))
THEN-CONTEXT Casify.casify-tac opt))
›

end

theory *Labeled-Hoare-Examples*
imports
  Labeled-Hoare
  HOL−Hoare.Arith2
begin

4.4.1 Multiplication by successive addition

lemma *multiply-by-add*: $VARS\ m\ s\ a\ b$
  $\{a{=}A \wedge b{=}B\}$
  $m := 0;\ s := 0;$
  $WHILE\ m{\neq}a$
  $INV\ \{s{=}m{*}b \wedge a{=}A \wedge b{=}B\}$
  $DO\ s := s{+}b;\ m := m{+}(1{::}nat)\ OD$
  $\{s = A{*}B\}$
by *vcg-simp*

lemma $VARS\ M\ N\ P :: int$
 $\{m{=}M \wedge n{=}N\}$
 $IF\ M < 0\ THEN\ M := -M;\ N := -N\ ELSE\ SKIP\ FI;$
 $P := 0;$
 $WHILE\ 0 < M$
 $INV\ \{0 \le M \wedge (\exists p.\ p = (if\ m{<}0\ then\ -m\ else\ m) \wedge p{*}N = m{*}n \wedge P = (p{-}M){*}N)\}$
 $DO\ P := P{+}N;\ M := M - 1\ OD$
 $\{P = m{*}n\}$
proof *casified-vcg-simp*
  case *while*
  { case *postcondition*
    then show *?case* **by** *auto*
  next
    case *invariant*
    { case *basic*
      then show *?case* **by** (*auto simp*: *int-distrib*)
    }
  }
qed

4.4.2 Euclid's algorithm for GCD

lemma *Euclid-GCD*: $VARS\ a\ b$
 $\{0{<}A \wedge 0{<}B\}$
 $a := A;\ b := B;$
 $WHILE\ \ a \neq b$
 $INV\ \{0{<}a \wedge 0{<}b \wedge gcd\ A\ B = gcd\ a\ b\}$
 $DO\ IF\ a{<}b\ THEN\ b := b{-}a\ ELSE\ a := a{-}b\ FI\ OD$
 $\{a = gcd\ A\ B\}$
proof *casified-vcg-simp*

case *while*
{ case *postcondition*
then show *?case* **by** (*auto elim*: *gcd-nnn*)
next
case *invariant*
{ case *cond*
{ case *vc*
then show *?case*
by (*simp-all add*: *linorder-not-less gcd-diff-l gcd-diff-r less-imp-le*)
}
}
}
qed

4.4.3 Dijkstra's extension of Euclid's algorithm for simultaneous GCD and SCM

From E.W. Disjkstra. Selected Writings on Computing, p 98 (EWD474), where it is given without the invariant. Instead of defining scm explicitly we have used the theorem scm x y = x*y/gcd x y and avoided division by mupltiplying with gcd x y.

lemmas *distribs* =
diff-mult-distrib diff-mult-distrib2 add-mult-distrib add-mult-distrib2

lemma *gcd-scm*: *VARS a b x y*
$\{0<A \wedge 0<B \wedge a=A \wedge b=B \wedge x=B \wedge y=A\}$
$WHILE\ \ a \neq b$
$INV\ \{0<a \wedge 0<b \wedge gcd\ A\ B = gcd\ a\ b \wedge 2*A*B = a*x + b*y\}$
$DO\ IF\ a<b\ THEN\ (b := b-a;\ x := x+y)\ ELSE\ (a := a-b;\ y := y+x)\ FI\ OD$
$\{a = gcd\ A\ B \wedge 2*A*B = a*(x+y)\}$
proof *casified-vcg*
case *while* **{**
case *precondition* **then show** *?case* **by** *simp*
next
case *invariant*
case *cond*
case *vc*
then show *?case*
by (*simp add*: *distribs gcd-diff-r linorder-not-less gcd-diff-l*)
next
case *postcondition* **then show** *?case*
by (*simp add*: *distribs gcd-nnn*)
}
qed

4.4.4 Power by iterated squaring and multiplication

lemma *power-by-mult*: *VARS a b c*

$\{a{=}A \wedge b{=}B\}$
$c := (1{::}nat);$
$WHILE\ b \neq 0$
$INV\ \{A\hat{}B = c * a\hat{}b\}$
$DO\ \ WHILE\ b\ mod\ 2 = 0$
    $INV\ \{A\hat{}B = c * a\hat{}b\}$
    $DO\ \ a := a{*}a;\ b := b\ div\ 2\ OD;$
    $c := c{*}a;\ b := b - 1$
OD
$\{c = A\hat{}B\}$
proof *casified-vcg-simp*
  case *while*
  case *invariant*
  case *while*
  case *postcondition*
  then show *?case* **by** (*cases b*) *simp-all*
qed

4.4.5 Factorial

lemma *factorial*: $VARS\ a\ b$
$\{a{=}A\}$
$b := 1;$
$WHILE\ a \neq 0$
$INV\ \{fac\ A = b * fac\ a\}$
$DO\ b := b{*}a;\ a := a - 1\ OD$
$\{b = fac\ A\}$
  apply *vcg-simp*
  apply(*clarsimp split*: *nat-diff-split*)
  done

lemma $VARS\ i\ f$
$\{True\}$
$i := (1{::}nat);\ f := 1;$
$WHILE\ i \leq n\ INV\ \{f = fac(i - 1) \wedge 1 \leq i \wedge i \leq n{+}1\}$
$DO\ f := f{*}i;\ i := i{+}1\ OD$
$\{f = fac\ n\}$
proof *casified-vcg-simp*
  case *while*
  { case *invariant*
    { case *basic*
      then show *?case* **by** (*induct i*) *simp-all*
    }
  next
    case *postcondition*
    then have $i = Suc\ n$ **by** *simp*
    then show *?case* **by** *simp*
  }
qed

4.4.6 Quicksort

The ‘partition’ procedure for quicksort. ‘A’ is the array to be sorted (modelled as a list). Elements of A must be of class order to infer at the end that the elements between u and l are equal to pivot.

Ambiguity warnings of parser are due to := being used both for assignment and list update.

```
lemma Partition:
  fixes pivot
  defines leq ≡ λA i. ∀k. k<i ⟶ A!k ≤ pivot
  defines geq ≡ λA i. ∀k. i<k ∧ k<length A ⟶ pivot ≤ A!k
  shows
   VARS A u l
   {0 < length(A::('a::order)list)}
   l := 0; u := length A − Suc 0;
   WHILE l ≤ u
    INV {leq A l ∧ geq A u ∧ u<length A ∧ l≤length A}
    DO WHILE l < length A ∧ A!l ≤ pivot
        INV {leq A l ∧ geq A u ∧ u<length A ∧ l≤length A}
        DO l := l+1 OD;
       WHILE 0 < u ∧ pivot ≤ A!u
        INV {leq A l ∧ geq A u  ∧ u<length A ∧ l≤length A}
        DO u := u − 1 OD;
       IF l ≤ u THEN A := A[l := A!u, u := A!l] ELSE SKIP FI
    OD
   {leq A u ∧ (∀k. u<k ∧ k<l ⟶ A!k = pivot) ∧ geq A l}
  unfolding leq-def geq-def
proof casified-vcg-simp
  case basic
  then show ?case by auto
next
  case while
  { case postcondition
    then show ?case by (force simp: nth-list-update)
  next
    case invariant
    { case while
      { case invariant
        { case basic
          then show ?case by (blast elim!: less-SucE intro: Suc-leI)
        }
      }
    next
      case whilea
      { case invariant
        { case basic
          have lem: ⋀m n. m − Suc 0 < n ⟹ m < Suc n by linarith
          from basic show ?case by (blast elim!: less-SucE intro: less-imp-diff-less
```

dest: *lem*)
        }
      }
    }
  }
qed

end

References

[1] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 99–115. Springer, Jan. 2012.