Shivers’ Control Flow Analysis

Joachim Breitner

August 28, 2014

Abstract

In his dissertation [3], Olin Shivers introduces a concept of control flow graphs for functional languages, provides an algorithm to statically derive a safe approximation of the control flow graph and proves this algorithm correct. In this research project [1], Shivers’ algorithms and proofs are formalized using the HOLCF extension of the logic HOL in the theorem prover Isabelle.

Contents

I. The definitions
   1. Syntax 2
   2. Standard semantics 3
   3. Exact nonstandard semantics 7
   4. Abstract nonstandard semantics 12

II. The main results
   5. The exact call cache is a map 17
      5.1. Preparations .................................................. 17
      5.2. The proof .................................................... 20
   6. The abstract semantics is correct 24
      6.1. Abstraction functions ................................. 25
      6.2. Lemmas about abstraction functions ................ 27
      6.3. Approximation relation ................................. 27
      6.4. Lemmas about the approximation relation ......... 28
First, we define the syntax tree of a program in our toy functional language, using continuation passing style, corresponding to section 3.2 in Shivers’ dissertation.

We assume that the program to be investigated is already parsed into a syntax tree. Furthermore, we assume that distinct labels were added to distinguish different code positions and that the program has been alphatised, i.e. that each variable name is only
bound once. This binding position is, as a convenience, considered part of the variable name.

```plaintext
type-synonym label = nat
type-synonym var = label × string

definition binder :: var ⇒ label where [simp]: binder v = fst v
```

The syntax consists now of lambda abstractions, call expressions and values, which can either be lambdas, variable references, constants or primitive operations. A program is a lambda expression.

Shivers’ language has as the set of basic values integers plus a special value for false. We simplified this to just the set of integers. The conditional If considers zero as false and any other number as true.

Shivers also restricts the values in a call expression: No constant maybe be used as the called value, and no primitive operation may occur as an argument. This restriction is dropped here and just leads to runtime errors when evaluating the program.

```plaintext
datatype prim = Plus label | If label label
datatype lambda = Lambda label var list call
  and call = App label val val list
  and val = L lambda | R label var | C label int | P prim

type-synonym prog = lambda
```

Three example programs. These were generated using the Haskell implementation of Shivers’ algorithm that we wrote as a prototype[2].

```plaintext
abbreviation ex1 == (Lambda 1 [(1,"cont")]) (App 2 (R 3 (1,"cont")) [(C 4 0)])
abbreviation ex2 == (Lambda 1 [(1,"cont")]) (App 2 (P (Plus 3)) [(C 4 1), (C 5 1), (R 6 (1,"cont"))])
abbreviation ex3 == (Lambda 1 [(1,"cont")]) (Let label (var × lambda) list call
  and call = App label val val list
  and val = L lambda | R label var | C label int | P prim
```

end

2. Standard semantics

```plaintext
theory Eval
  imports HOLCF HOLCFUtils CPSScheme
```

3
We begin by giving the standard semantics for our language. Although this is not actually used to show any results, it is helpful to see that the later algorithms “look similar” to the evaluation code and the relation between calls done during evaluation and calls recorded by the control flow graph.

We follow the definition in Figure 3.1 and 3.2 of Shivers’ dissertation, with the clarifications from Section 4.1. As explained previously, our set of values encompasses just the integers, there is no separate value for false. Also, values and procedures are not distinguished by the type system.

Due to recursion, one variable can have more than one currently valid binding, and due to closures all bindings can possibly be accessed. A simple call stack is therefore not sufficient. Instead we have a contour counter, which is increased in each evaluation step. It can also be thought of as a time counter. The variable environment maps tuples of variables and contour counter to values, thus allowing a variable to have more than one active binding. A contour environment lists the currently visible binding for each binding position and is preserved when a lambda expression is turned into a closure.

\[
\begin{align*}
\text{type-synonym} & \quad \text{contour} = \text{nat} \\
\text{type-synonym} & \quad \text{benv} = \text{label} \rightarrow \text{contour} \\
\text{type-synonym} & \quad \text{closure} = \text{lambda} \times \text{benv}
\end{align*}
\]

The set of semantic values consist of the integers, closures, primitive operations and a special value Stop. This is passed as an argument to the program and represents the terminal continuation. When this value occurs in the first position of a call, the program terminates.

\[
\text{datatype} \quad d = \begin{cases} 
\text{DI} \text{ int} \\
\text{DC} \text{ closure} \\
\text{DP} \text{ prim} \\
\text{Stop} 
\end{cases}
\]

\[
\text{type-synonym} \quad \text{venv} = \text{var} \times \text{contour} \rightarrow d
\]

The function \(A\) evaluates a syntactic value into a semantic datum. Constants and primitive operations are left untouched. Variable references are resolved in two stages: First the current binding contour is fetched from the binding environment \(\beta\), then the stored value is fetched from the variable environment \(ve\). A lambda expression is bundled with the current contour environment to form a closure.

\[
\text{fun} \quad \text{evalV} :: \text{val} \Rightarrow \text{benv} \Rightarrow \text{venv} \Rightarrow d \ (A) \\
\text{where} \quad A \ (C - i) \ \beta \ ve = \text{DI} \ i \\
\quad A \ (P \ \text{prim}) \ \beta \ ve = \text{DP} \ \text{prim}
\]

\[
4
\]
A \ (R - var) \beta \ ve = 
(case \beta \ (binder \ var) \ of 
Some \ l \Rightarrow \ (case \ ve \ (var,l) \ of \ Some \ d \Rightarrow \ d))

A \ (L \ lam) \beta \ ve = DC \ (lam, \beta)

The answer domain of our semantics is the set of integers, lifted to obtain an additional element denoting bottom. Shivers distinguishes runtime errors from non-termination. Here, both are represented by $\bot$.

**Type-synonym**

ans = int lift

To be able to do case analysis on the custom datatypes `lambda`, `d`, `call` and `prim` inside a function defined with `fixrec`, we need continuity results for them. These are all of the same shape and proven by case analysis on the discriminator.

**Lemma**

`cont2cont-case-lambda` [simp, `cont2cont`]:

assumes $\forall a \ b \ c. \ cont \ (\lambda x. \ f x \ a \ b \ c)$

shows $\ cont \ (\lambda x. \ \text{case-lambda} \ (f x) \ l)$

using `assms` by `(cases l) auto`

**Lemma**

`cont2cont-case-d` [simp, `cont2cont`]:

assumes $\forall y. \ cont \ (\lambda x. \ f1 \ x \ y)$

and $\forall y. \ cont \ (\lambda x. \ f2 \ x \ y)$

and $\forall y. \ cont \ (\lambda x. \ f3 \ x \ y)$

and $\ cont \ (\lambda x. \ f4 \ x)$

shows $\ cont \ (\lambda x. \ \text{case-d} \ (f1 \ x) \ (f2 \ x) \ (f3 \ x) \ (f4 \ x) \ d)$

using `assms` by `(cases d) auto`

**Lemma**

`cont2cont-case-call` [simp, `cont2cont`]:

assumes $\forall a \ b \ c. \ cont \ (\lambda x. \ f1 \ x \ a \ b \ c)$

and $\forall a \ b \ c. \ cont \ (\lambda x. \ f2 \ x \ a \ b \ c)$

shows $\ cont \ (\lambda x. \ \text{case-call} \ (f1 \ x) \ (f2 \ x) \ c)$

using `assms` by `(cases c) auto`

**Lemma**

`cont2cont-case-prim` [simp, `cont2cont`]:

assumes $\forall y. \ cont \ (\lambda x. \ f1 \ x \ y)$

and $\forall y \ z. \ cont \ (\lambda x. \ f2 \ x \ y \ z)$

shows $\ cont \ (\lambda x. \ \text{case-prim} \ (f1 \ x) \ (f2 \ x) \ p)$

using `assms` by `(cases p) auto`

As usual, the semantics of a functional language is given as a denotational semantics. To that end, two functions are defined here: $F$ applies a procedure to a list of arguments. Here closures are unwrapped, the primitive operations are implemented and the terminal continuation $\text{Stop}$ is handled. $C$ evaluates a call expression, either by evaluating
procedure and arguments and passing them to \( F \), or by adding the bindings of a *Let* expression to the environment.

Note how the contour counter is incremented before each call to \( F \) or when a *Let* expression is evaluated.

With mutually recursive equations, such as those given here, the existence of a function satisfying these is not obvious. Therefore, the `fixrec` command from the HOLCF package is used. This takes a set of equations and builds a functional from that. It mechanically proves that this functional is continuous and thus a least fixed point exists. This is then used to define \( F \) and \( C \) and proof the equations given here. To use the HOLCF setup, the continuous function arrow \( \to \) with application operator \( \cdot \) is used and our types are wrapped in `discr` and `lift` to indicate which partial order is to be used.

**type-synonym** \( fstate = (d \times d \text{ list} \times \text{venv} \times \text{contour}) \)

**type-synonym** \( cstate = (\text{call} \times \text{benv} \times \text{venv} \times \text{contour}) \)

```plaintext
fixrec evalF :: \( fstate \) discr \( \to \) \( \text{ans} (F) \)
and evalC :: \( cstate \) discr \( \to \) \( \text{ans} (C) \)
where evalF \( \cdot \) \( fstate \) = (case undiscr \( fstate \) of
  (DC (Lambda lab vs c, \( \beta \)), as, ve, b) \Rightarrow
    (if length vs = length as
    then let \( \beta' = \beta (\lambda b' \to b) \);
         ve' = map-upds ve (map \( \lambda v. (v, b') \)) \( \text{as} \)
      in \( C \cdot \text{Discr} (c, \beta', \text{ve}', \text{b}') \))
    else \( \perp \))
  | (DP (Plus c),[DI a1, DI a2, cnt],[ve, b]) \Rightarrow
    let b' = Suc b;
    \( \beta = [c \mapsto b] \)
    in \( F \cdot \text{Discr} (\text{cnt},[\text{DI} \ (a1 + a2)],\text{ve}, \text{b}') \))
  | (DP (prim.If ct cf),[DI v, contt, contf],[ve, b]) \Rightarrow
    (if v \neq 0
     then let b' = Suc b;
     \( \beta = [\text{ct} \mapsto b] \)
     in \( F \cdot \text{Discr} (\text{contt},[],\text{ve}, \text{b}') \))
    else let b' = Suc b;
    \( \beta = [\text{cf} \mapsto b] \)
    in \( F \cdot \text{Discr} (\text{contf},[],\text{ve}, \text{b}') \))
  | (Stop,[DI i],\_,\_) \Rightarrow \text{Def i}
  | \_ \Rightarrow \perp)
where evalC \( \cdot \) \( cstate \) = (case undiscr \( cstate \) of
  (App lab f vs, \beta, ve, b) \Rightarrow
    let f' = \( \lambda \beta \text{ve} \);
    as = map \( \lambda v. \ \lambda \beta \text{ve} \) \( \text{vs} \);
    b' = Suc b
    in \( F \cdot \text{Discr} (f',\text{as},\text{ve}, \text{b}') \))
```
To evaluate a full program, it is passed to \( \mathcal{F} \) with proper initializations of the other arguments. We test our semantics function against two example programs and observe that the expected value is returned.

\[
\text{definition } \text{evalCPS} :: \text{prog} \Rightarrow \text{ans} (PR)
\]

\[
\text{where } PR \ l = (\text{let } ve = \text{empty}; \\
\beta = \text{empty}; \\
\hat{f} = A (L \ l) \beta \ \text{ve} \\
in \ F::(\text{Discr} (\hat{f}, [\text{Stop}], ve, 0)))
\]

\[
\text{lemma correct-ex1: } PR \ ex1 = \text{Def 0}
\]

\[
\text{unfolding evalCPS-def } \\
\text{by simp}
\]

\[
\text{lemma correct-ex2: } PR \ ex2 = \text{Def 2}
\]

\[
\text{unfolding evalCPS-def } \\
\text{by simp}
\]

\[
\text{end}
\]

3. Exact nonstandard semantics

\[
\text{theory ExCF} \\
\text{imports HOLCF HOLCFUtils CPSScheme Utils}
\]

\[
\text{begin}
\]

We now alter the standard semantics given in the previous section to calculate a control flow graph instead of the return value. At this point, we still “run” the program in full, so this is not yet the static analysis that we aim for. Instead, this is the reference for the correctness proof of the static analysis: If an edge is recorded here, we expect it to be found by the static analysis as well.

In preparation of the correctness proof we change the type of the contour counters. Instead of plain natural numbers as in the previous sections we use lists of labels, remembering at each step which part of the program was just evaluated.

Note that for the exact semantics, this is information is not used in any way and it would
have been possible to just use natural numbers again. This is reflected by the preorder instance for the contours which only look at the length of the list, but not the entries.

**definition** \( \text{contour} \) = \( \text{(UNIV::label list set)} \)

**typedef** \( \text{contour} = \text{contour} \)

**unfolding** \( \text{contour-def by auto} \)

**definition** \( \text{initial-contour (b}_0) \)

**where** \( b_0 = \text{Abs-contour} \) []

**definition** \( \text{nb} \)

**where** \( \text{nb} b c = \text{Abs-contour} \) (c # \text{Rep-contour} b)

**instantiation** \( \text{contour :: preorder} \)

**begin**

**definition** \( \text{le-contour-def: } b \leq b' \longleftrightarrow \text{length (Rep-contour } b) \leq \text{length (Rep-contour } b') \)

**definition** \( \text{less-contour-def: } b < b' \longleftrightarrow \text{length (Rep-contour } b) < \text{length (Rep-contour } b') \)

**instance** **proof**

**qed** (\( \text{auto simp add:le-contour-def less-contour-def Rep-contour-inverse Abs-contour-inverse contour-def} \))

**end**

Three simple lemmas helping Isabelle to automatically prove statements about contour numbers.

**lemma** \( \text{nb-le-less[iff]} \): \( \text{nb} b c \leq b' \longleftrightarrow b < b' \)

**unfolding** \( \text{nb-def} \)

**by** (\( \text{auto simp add:le-contour-def less-contour-def Rep-contour-inverse Abs-contour-inverse contour-def} \))

**lemma** \( \text{nb-less[iff]} \): \( b' < \text{nb} b c \longleftrightarrow b' \leq b \)

**unfolding** \( \text{nb-def} \)

**by** (\( \text{auto simp add:le-contour-def less-contour-def Rep-contour-inverse Abs-contour-inverse contour-def} \))

**declare** \( \text{less-imp-le[where 'a = contour, intro]} \)

The other types used in our semantics functions have not changed.

**type-synonym** \( \text{benv} = \text{label} \rightarrow \text{contour} \)

**type-synonym** \( \text{closure} = \text{lambda} \times \text{benv} \)

**datatype** \( d = \text{DI int} \)

\| \text{DC closure} \\
\| \text{DP prim} \\
\| \text{Stop} \)

**type-synonym** \( \text{venv} = \text{var} \times \text{contour} \rightarrow d \)
As we do not use the type system to distinguish procedural from non-procedural values, we define a predicate for that.

```isar
primrec isProc
where
  isProc (DI -) = False
| isProc (DC -) = True
| isProc (DP -) = True
| isProc Stop = True
```

To please HOLCF, we declare the discrete partial order for our types:

```isar
instantiation contour :: discrete-cpo
begin
definition [simp]: (x::contour) ⊑ y ←→ x = y
instance by default simp
end
```

```isar
instantiation d :: discrete-cpo begin
definition [simp]: (x::d) ⊑ y ←→ x = y
instance by default simp
end
```

```isar
instantiation call :: discrete-cpo begin
definition [simp]: (x::call) ⊑ y ←→ x = y
instance by default simp
end
```

The evaluation function for values has only changed slightly: To avoid worrying about incorrect programs, we return zero when a variable lookup fails. If the labels in the program given are correct, this will not happen. Shivers makes this explicit in Section 4.1.3 by restricting the function domains to the valid programs. This is omitted here.

```isar
fun evalV :: val ⇒ benv ⇒ venv ⇒ d (A)
where
  A (C - i) β ve = DI i
| A (P prim) β ve = DP prim
| A (R - var) β ve =
  (case β (binder var) of
    Some l ⇒ (case ve (var,l) of Some d ⇒ d | None ⇒ DI 0)
    | None ⇒ DI 0)
| A (L lam) β ve = DC (lam, β)
```

To be able to do case analysis on the custom datatypes `lambda`, `d`, `call` and `prim` inside a function defined with `fixrec`, we need continuity results for them. These are all of the same shape and proven by case analysis on the discriminator.

```isar
lemma cont2cont-case-lambda [simp, cont2cont]:
  assumes "∀ a b c. cont (λx. f x a b c)
  shows cont (λx. case-lambda (f x) l)
using assms
by (cases l) auto
```
lemma cont2cont-case-d [simp, cont2cont]:
  assumes \( \forall y . \text{cont}\ (\lambda x. f_1 x y) \)
  and \( \forall y . \text{cont}\ (\lambda x. f_2 x y) \)
  and \( \forall y . \text{cont}\ (\lambda x. f_3 x y) \)
  and \( \text{cont}\ (\lambda x. f_4 x) \)
  shows \( \text{cont}\ (\lambda x. \text{case-d}\ (f_1 x) (f_2 x) (f_3 x) (f_4 x) d) \)
  using assms
by (cases d) auto

lemma cont2cont-case-call [simp, cont2cont]:
  assumes \( \forall a\ b\ c . \text{cont}\ (\lambda x. f_1 x a b c) \)
  and \( \forall a\ b\ c . \text{cont}\ (\lambda x. f_2 x a b c) \)
  shows \( \text{cont}\ (\lambda x. \text{case-call}\ (f_1 x) (f_2 x) c) \)
  using assms
by (cases c) auto

lemma cont2cont-case-prim [simp, cont2cont]:
  assumes \( \forall y . \text{cont}\ (\lambda x. f_1 x y) \)
  and \( \forall y\ z . \text{cont}\ (\lambda x. f_2 x y z) \)
  shows \( \text{cont}\ (\lambda x. \text{case-prim}\ (f_1 x) (f_2 x) p) \)
  using assms
by (cases p) auto

Now, our answer domain is not any more the integers, but rather call caches. These
are represented as sets containing tuples of call sites (given by their label) and binding
environments to the called value. The argument types are unaltered.

In the functions \( F \) and \( C \), upon every call, a new element is added to the resulting set.
The \textit{STOP} continuation now ignores its argument and returns the empty set instead.
This corresponds to Figure 4.2 and 4.3 in Shivers’ dissertation.

type-synonym ccache = ((label \times benv) \times d) set

type-synonym ans = ccache

type-synonym fstate = (d \times d list \times venv \times contour)

type-synonym cstate = (call \times benv \times venv \times contour)

fixrec evalF :: fstate discr \rightarrow ans (F)
  and evalC :: cstate discr \rightarrow ans (C)
where \( F\cdot fstate = \case undiscr fstate of\)
  (DC (Lambda lab vs c, \( \beta \)), as, ve, b) \Rightarrow
  (\text{if length vs = length as then let } \beta' = \beta (\text{lab} \mapsto b);\)
    \( ve' = \text{map-upds ve (map (\lambda v.(v,b)) vs) as} \)
  in \( C\cdot \text{Discr (c,} \beta',ve',b) \)
  else \( \bot \)
  | (DP (Plus c),[DI a1, DI a2, cnt],ve,b) \Rightarrow

10
(if isProc cnt
then let $b' = nb b c$;
  $\beta = [c \mapsto b]$
  in $\mathcal{F}(\text{Discr (cnt,}\mathcal{DI} (a1 + a2)],ve,b'))$
else $\bot$
$\mid (\text{DP (prim.If ct cf),}\mathcal{DI} v, \text{contt, contf},ve,b) \Rightarrow$
(if isProc contt $\land$ isProc contf
then
(if $v \neq 0$
then let $b' = nb b ct$;
  $\beta = [ct \mapsto b]$
  in $\mathcal{F}(\text{Discr (contt,}\mathcal{],ve,b'}))$
else let $b' = nb b cf$;
  $\beta = [cf \mapsto b]$
  in $\mathcal{F}(\text{Discr (contf,}\mathcal{],ve,b'}))$
else $\bot$)
$\mid (\text{Stop,}\mathcal{DI} i,\neg\neg) \Rightarrow \{}$
\text{-} $\Rightarrow \bot$
$\)}

$\mid C\cdot\text{estate} = (\text{case undiscr estate of}$
  (App lab $f \text{ vs,} \beta ve, b) \Rightarrow$
  let $f' = A f \beta ve$;
    $as = \text{map (}\lambda v, A v \beta ve\text{) vs}$;
    $b' = nb b lab$
  in if isProc $f'$
    then $\mathcal{F}(\text{Discr (f',as,ve,b')}) \cup \{((\text{lab,} \beta),f')\}$
  else $\bot$
$\mid (\text{Let lab ls c',} \beta ,\text{ve,b) \Rightarrow}$
  let $b' = nb b lab$;
  $\beta' = \beta (\text{lab} \mapsto b')$
  $ve' = ve \leftrightarrow \text{map-of (map (}\lambda (v,l), (v,b'), A (L l) \beta' ve\text{) ls)}$
  in $C\cdot(\text{Discr (c',} \beta',\text{ve',b')})$
)

In preparation of later proofs, we give the cases of the generated induction rule names and also create a large rule to deconstruct the an value of type fstate into the various cases that were used in the definition of $\mathcal{F}$.

**lemmas** evalF-evalC-induct = evalF-evalC.induct[case-names Admissibility Bottom Next]

**lemmas** cl-cases = prod.exhaust[OF lambda.exhaust, \text{of } \lambda a \text{ - } a]

**lemmas** ds-cases-plus = list.exhaust[
  \text{OF - d.exhaust, } of \text{ - } \lambda a \text{ - } a,
  \text{OF - list.exhaust, } of \text{ - } \lambda x \text{ - } x,
  \text{OF - d.exhaust, } of \text{ - } \lambda x \text{ - } a \text{ - } a,
  \text{OF - list.exhaust, } of \text{ - } \lambda x \text{ - } x \text{ - } x]
The exact semantics of a program again uses $F$ with properly initialized arguments. For the first two examples, we see that the function works as expected.

definition evalCPS :: prog ⇒ ans (PR)
where PR l = (let ve = empty;
β = empty;
\text{in } F · (Discr (f, \text{Stop}, ve, b_0)))

lemma correct-ex1: PR ex1 = \{(2, [1 \mapsto b_0]), \text{Stop}\}
unfolding evalCPS-def
by simp

lemma correct-ex2: PR ex2 = \{(2, [1 \mapsto b_0]), \text{DP (Plus 3)},
                        ((3, [3 \mapsto nb b_0 2]), \text{Stop})\}
unfolding evalCPS-def
by simp

end

4. Abstract nonstandard semantics

theory AbsCF
  imports HOLCF HOLCFUtils CPSScheme Utils SetMap
begin

default-sort type

After having defined the exact meaning of a control graph, we now alter the algorithm into a statically computable. We note that the contour pointer in the exact semantics is taken from an infinite set. This is unavoidable, as recursion depth is unbounded. But if
this were not the case and the set were finite, the function would be calculable, having finite range and domain.

Therefore, we make the set of contour counter values finite and accept that this makes our result less exact, but calculable. We also do not work with values any more but only remember, for each variable, what possible lambdas can occur there. Because we do not have exact values any more, in a conditional expression, both branches are taken.

We want to leave the exact choice of the finite contour set open for now. Therefore, we define a type class capturing the relevant definitions and the fact that the set is finite. Isabelle expects type classes to be non-empty, so we show that the unit type is in this type class.

```isabelle
class contour = finite + 
  fixes nb-a :: 'a ⇒ label ⇒ 'a (nb)
  and a-initial-contour :: 'a (b)

instantiation unit :: contour
begin
  definition nb - - = ()
  definition b0 = ()
  instance by default auto
end
```

Analogous to the previous section, we define types for binding environments, closures, procedures, semantic values (which are now sets of possible procedures) and variable environment. Their types are parametrized by the chosen set of abstract contours.

The abstract variable environment is a partial map to sets in Shivers’ dissertation. As he does not need to distinguish between a key not in the map and a key mapped to the empty set, this presentation is redundant. Therefore, I encoded this as a function from keys to sets of values. The theory `SetMap` contains functions and lemmas to work with such maps, symbolized by an appended dot (e.g. `{}, ∪).

```isabelle
type-synonym 'c a-benv = label ⇒ 'c (- benv [1000])
type-synonym 'c a-closure = lambda × 'c benv (- closure [1000])

datatype 'c proc (- prc [1000])
  = PC 'c closure
  | PP prim
  | AStop

type-synonym 'c a-d = 'c prc set (- d [1000])
type-synonym 'c a-venv = var × 'c ⇒ 'c d (- venv [1000])
```

13
The evaluation function now ignores constants and returns singletons for primitive operations and lambda expressions.

```plaintext
fun evalVa : val ⇒ 'c benv ⇒ 'c venv ⇒ 'c d (A)
where A (C - i) β ve = {}
  | A (P prim) β ve = {PP prim}
  | A (R - var) β ve =
    Some l ⇒ ve (var,l)
    | None ⇒ {}}
  | A (L lam) β ve = {PC (lam, β)}
```

The types of the calculated graph, the arguments to \( \hat{F} \) and \( \hat{C} \) resemble closely the types in the exact case, with each type replaced by its abstract counterpart.

```plaintext
type-synonym 'c a-ccache = ((label × 'c benv) × 'c proc) set (- ccache [1000])
type-synonym 'c a-ans = 'c ccache (- ans [1000])
type-synonym 'c a-fstate = ('c proc × 'c d list × 'c venv × 'c) (- fstate [1000])
type-synonym 'c a-cstate = (call × 'c benv × 'c venv × 'c) (- cstate [1000])
```

And yet again, cont2cont results need to be shown for our custom data types.

```plaintext
lemma cont2cont-case-lambda [simp, cont2cont]:
  assumes ∃ a b c. cont (λx. f x a b c)
  shows cont (λx. case-lambda (f x) l)
using assms
by (cases l) auto
```

```plaintext
lemma cont2cont-case-proc [simp, cont2cont]:
  assumes ∃ y. cont (λx. f1 x y)
  and ∃ y. cont (λx. f2 x y)
  and cont (λx. f3 x)
  shows cont (λx. case-proc (f1 x) (f2 x) (f3 x) d)
using assms
by (cases d) auto
```

```plaintext
lemma cont2cont-case-call [simp, cont2cont]:
  assumes ∃ a b c. cont (λx. f1 x a b c)
  and ∃ a b c. cont (λx. f2 x a b c)
  shows cont (λx. case-call (f1 x) (f2 x) c)
using assms
by (cases c) auto
```

```plaintext
lemma cont2cont-case-prim [simp, cont2cont]:
  assumes ∃ y. cont (λx. f1 x y)
  and ∃ y z. cont (λx. f2 x y z)
  shows cont (λx. case-prim (f1 x) (f2 x) p)
```
using assms
by (cases p) auto

We can now define the abstract nonstandard semantics, based on the equations in Figure 4.5 and 4.6 of Shivers’ dissertation. In the AStop case, {} is returned, while for wrong arguments, ⊥ is returned. Both actually represent the same value, the empty set, so this is just an aesthetic difference.

fixrec a-evalF :: 'c::contour fstate discr → 'c ans (F)
and a-evalC :: 'c::contour cstate discr → 'c ans (C)

where F·fstate = (case undiscr fstate of
  (PC (Lambda lab vs c, β), as, ve, b) ⇒
    (if length vs = length as
     then let β' = β (lab ↦ b);
          ve' = ve ∪ (⋃ (map (λ(v,a). {(v,b) := a}). (zip vs as)))
          in C.(Discr (c,β',ve',b))
     else ⊥))
| (PP (Plus c),[-,-],cnts),ve,b) ⇒
  let b' = nb b c;
  β = [c ↦ b]
  in (⋃ cnt ∈ cnts . F.(Discr ([{}],ve,b')))
  ∪ {((c, β), cont) | cont . cont ∈ cnts}
| (PP (prim.If ct cf),[-,-],cntts),ve,b) ⇒
  ((
    let b' = nb b ct;
    β = [ct ↦ b]
    in (⋃ cnt ∈ cntts . F.(Discr ([{}],ve,b')))
  ∪ {((ct, β), cont) | cont . cont ∈ cntts})
∪(
  let b' = nb b cf;
  β = [cf ↦ b]
  in (⋃ cnt ∈ cntts . F.(Discr ([{}],ve,b')))
  ∪ {((cf, β), cont) | cont . cont ∈ cntts}
))
| (AStop,[,-,-]) ⇒ {}
| - ⇒ ⊥
)

| C·cstate = (case undiscr estate of
  (App lab f vs,β,ve,b) ⇒
    let fs = A f β ve;
    as = map (λv. A v β ve) vs;
    b' = nb b lab
    in (⋃ f' ∈ fs . F.(Discr (f',as,ve,b')))
    ∪ {((lab, β),f') | f' ∈ fs}
| (Let lab ls c',β,ve,b) ⇒
    let b' = nb b lab;
    β' = β (lab ↦ b');
Again, we name the cases of the induction rule and build a nicer case analysis rule for arguments of type $\textit{fstate}$.

**lemmas** a-evalF-evalC-induct = a-evalF-a-evalC-induct[case-names Admissibility Bottom Next]

**fun** a-evalF-cases
**where** a-evalF-cases (PC (Lambda lab vs c, $\beta$)) as $ve$ $b$ = undefined
  | a-evalF-cases (PP (Plus cp)) [a1, a2, cnt] $ve$ $b$ = undefined
  | a-evalF-cases (PP (prim.If cp1 cp2)) [v, cntt, cntf] $ve$ $b$ = undefined
  | a-evalF-cases AStop [v] $ve$ $b$ = undefined

**lemmas** a-fstate-case-x = a-evalF-cases.cases[
  OF case-split, of - $\lambda$- vs - as - - , length vs = length as,
  case-names Closure Closure-inv Plus If Stop]

**lemmas** a-cl-cases = prod.exhaust[OF lambda.exhaust, of - $\lambda$ a - . a]
**lemmas** a-ds-cases = list.exhaust[
  OF - list.exhaust, of - - $\lambda$- x. x,
  OF - - list.exhaust ,of - - $\lambda$- - - x. x ,
  OF - - - list.exhaust,of - - $\lambda$- - - - x. x
]
**lemmas** a-ds-cases-stop = list.exhaust[OF - list.exhaust, of - - $\lambda$- x. x]
**lemmas** a-cl-cases = prod.cases4[OF proc.exhaust, of - $\lambda$ x - - . x,
  OF a-cl-cases prim.exhaust, of - $\lambda$ - - - a . a - $\lambda$ - - - a. a,
  OF case-split a-ds-cases a-ds-cases a-ds-cases-stop,
  of - $\lambda$- as - - - - - vs - ., length vs = length as - $\lambda$ - ds - - - - . ds $\lambda$ - ds - - - , ds $\lambda$ - ds - - . ds]

Not surprisingly, the abstract semantics of a whole program is defined using $\hat{F}$ with suitably initialized arguments. The function $\textit{the-elem}$ extracts a value from a singleton set. This works because we know that $\hat{A}$ returns such a set when given a lambda expression.

**definition** evalCP$\Sigma$-a :: prog => ('c::contour) $\alpha\text{\textit{ns}}$ ($\mathcal{P}\mathcal{R}$)
**where** $\mathcal{P}\mathcal{R}$ l = (let $ve$ = {};
  $\beta$ = empty;
  $f$ = $\hat{A}$ (L l) $\beta$ $ve$
  in $\hat{F}$.Discr (the-elem $f$,[[AStop]],$ve$,$\hat{b}_0$))

end
Part II.

The main results

5. The exact call cache is a map

theory ExCFSV
imports ExCF
begin

5.1. Preparations

Before we state the main result of this section, we need to define

- the set of binding environments occurring in a semantic value (which exists only if it is a closure),
- the set of binding environments in a variable environment, using the previous definition,
- the set of contour counters occurring in a semantic value and
- the set of contour counters occurring in a variable environment.

fun benv-in-d :: d ⇒ benv set
where benv-in-d (DC (l,β)) = {β}
     | benv-in-d - = {}

definition benv-in-ve :: venv ⇒ benv set
where benv-in-ve ve = ∪{benv-in-d d | d . d ∈ ran ve}

fun contours-in-d :: d ⇒ contour set
where contours-in-d (DC (l,β)) = ran β
     | contours-in-d - = {}

definition contours-in-ve :: venv ⇒ contour set
where contours-in-ve ve = ∪{contours-in-d d | d . d ∈ ran ve}

The following 6 lemmas allow us to calculate the above definition, when applied to constructs used in our semantics function, e.g. map updates, empty maps etc.

lemma benv-in-ve-upds:
assumes eq-length: length vs = length ds
and ∀ β∈benv-in-ve ve. Q β
and ∀ d′∈set ds. ∀ β∈benv-in-d d′. Q β
shows ∀ β∈benv-in-ve (ve(map (λv. (v, b′)) vs [→] ds)). Q β
proof

fix \( \beta \)

assume ass; \( \beta \in benv-in-ve (ve(map (\lambda v. (v, b'')) vs \rightarrow ds)) \)

then obtain \( d \) where \( \beta \in benv-in-d d \) and \( d \in ran (ve(map (\lambda v. (v, b'')) vs \rightarrow ds)) \) unfolding benv-in-ve-def by auto

moreover have \( ran (ve(map (\lambda v. (v, b'')) vs \rightarrow ds)) \subseteq ran ve \cup set ds \) using eq-length by(auto intro!:ran-ups)

ultimately

have \( d \in ran ve \lor d \in set ds \) by auto

thus \( Q \beta \) using assms(2,3) \( \langle \beta \in benv-in-d d \rangle \) unfolding benv-in-ve-def by auto

qed

lemma benv-in-eval:

assumes \( \forall \beta' \in benv-in-ve ve. Q \beta' \)

and \( Q \beta \)

shows \( \forall d \in ran ve \ (A v \beta ve). Q \beta \)

proof(cases v)

case (R - var)

thus \?thesis

proof (cases \( \beta \ (fst var) \))

case None with \( R \) show \?thesis by simp

case (Some \( cnt \)) show \?thesis

proof (cases ve (\( var, cnt \)))

case None with Some \( R \) show \?thesis by simp

case (Some \( d \))

hence \( d \in ran ve \) unfolding ran-def by blast

thus \?thesis using Some \( \langle \beta \ (fst var) = Some cnt \rangle R \) assms(1)

unfolding benv-in-ve-def by auto

qed

qed next

case (L \( l \)) thus \?thesis using assms(2) by simp

next
case P thus \?thesis by simp

qed

lemma contours-in-ve-empty[simp]: contours-in-ve empty = {}

unfolding contours-in-ve-def by auto

lemma contours-in-ve-ups:

assumes eq-length: \( length vs = length ds \)

and \( \forall b' \in contours-in-ve ve. Q b' \)

and \( \forall d' \in set ds. \forall b' \in contours-in-d d'. Q b' \)

shows \( \forall b' \in contours-in-ve (ve(map (\lambda v. (v, b'')) vs \rightarrow ds)). Q b' \)

proof–

have \( ran (ve(map (\lambda v. (v, b'')) vs \rightarrow ds)) \subseteq ran ve \cup set ds \) using eq-length by(auto intro!:ran-ups)

thus \?thesis using assms(2,3) unfolding contours-in-ve-def by blast

qed
lemma contours-in-vc-upds-binds:
  assumes ∀ b′∈contours-in-vc ve. Q b′
  and ∀ b′∈ran β′. Q b′
  shows ∀ b′∈contours-in-vc (ve ++ map-of (map (λ(v,l). ((v,b′′), A (L l) β′ ve)) ls)). Q b′
proof
  fix b′ assume b′∈contours-in-vc (ve ++ map-of (map (λ(v,l). ((v,b′′), A (L l) β′ ve)) ls))
  then obtain d where d:d ∈ ran (ve ++ map-of (map (λ(v,l). ((v,b′′), A (L l) β′ ve)) ls))
  and b:b′ ∈ contours-in-d d unfolding contours-in-vc-def by auto
  have ran (ve ++ map-of (map (λ(v,l). ((v,b′′), A (L l) β′ ve)) ls)) ⊆ ran ve ∪ ran (map-of (map (λ(v,l). ((v,b′”), A (L l) β′ ve)) ls))
    by(auto intro!:ran-concat)
  also have ... ⊆ ran ve ∪ snd set (map (λ(v,l). ((v,b′’), A (L l) β′ ve)) ls)
    by (rule Un-mono[of ran ve ran ve, OF subset-refl ran-map-of])
  also have ... ⊆ ran ve ∪ set (map (λ(v,l). (A (L l) β′ ve)) ls)
    by (rule Un-mono[of ran ve ran ve, OF subset-refl])auto
finally have d ∈ ran ve ∪ set (map (λ(v,l). (A (L l) β′ ve)) ls) using d by auto
  thus Q b′ using assms b unfolding contours-in-vc-def by auto
qed

lemma contours-in-eval:
  assumes ∀ b′∈contours-in-vc ve. Q b′
  and ∀ b′∈ran β. Q b′
  shows ∀ b′∈contours-in-d (A f β ve). Q b′
unfolding contours-in-vc-def
proof( cases f )
case (R - var)
  thus ?thesis
proof ( cases β (fst var))
  case None with R show ?thesis by simp next
  case (Some cnt) show ?thesis
  proof ( cases ve (var,cnt))
    case None with Some R show ?thesis by simp next
    case (Some d)
      hence d ∈ ran ve unfolding ran-def by blast
      thus ?thesis using Some ⟨β (fst var) = Some cnt; R ∨ b′∈contours-in-vc ve. Q b′⟩
        unfolding contours-in-vc-def
        by auto
    qed
  qed next
  case (L l) thus ?thesis using ∃ b′∈ ran β. Q b′ by simp next
  case C thus ?thesis by simp next
  case P thus ?thesis by simp
qed
5.2. The proof

The set returned by $\mathcal{F}$ and $\mathcal{C}$ is actually a partial map from callsite/binding environment pairs to called values. The corresponding predicate in Isabelle is \emph{single-valued}.

We would like to show an auxiliary result about the contour counter passed to $\mathcal{F}$ and $\mathcal{C}$ (such that it is an unused counter when passed to $\mathcal{F}$ and others) first. Unfortunately, this is not possible with induction proofs over fixed points: While proving the inductive case, one does not show results for the function in question, but for an information-theoretical approximation. Thus, any previously shown results are not available. We therefore intertwine the two inductions in one large proof.

This is a proof by fixpoint induction, so we have are obliged to show that the predicate is admissible and that it holds for the base case, i.e. the empty set. For the proof of admissibility, \emph{HOLCF} provides a number of introduction lemmas that, together with some additions in \emph{HOLCFUtils} and the continuity lemmas, mechanically prove admissibility. The base case is trivial.

The remaining case is the preservation of the properties when applying the recursive equations to a function known to have the desired property. Here, we break the proof into the various cases that occur in the definitions of $\mathcal{F}$ and $\mathcal{C}$ and use the induction hypotheses.

\textbf{lemma} \texttt{cc-single-valued'}:
\[\forall b' \in \text{contours-in-ve} \land b' < b \land \forall d' \in \text{set ds} \land \forall b' \in \text{contours-in-d} \land d' \land b' < b \Rightarrow (\text{single-valued} (\mathcal{F} \cdot \text{Discr} (d, ds, ve, b))) \land (\forall (\text{lab}, \beta, t) \in \mathcal{F} \cdot \text{Discr} (d, ds, ve, b). \exists b'. b' \in \text{ran} \beta \land b \leq b')\]
\textit{and} \[\forall b' \in \text{ran} \beta' \land \forall b' \in \text{contours-in-ve} \land b' \leq b \Rightarrow (\text{single-valued} (\mathcal{C} \cdot \text{Discr} (c, \beta', ve, b))) \land (\forall (\text{lab}, \beta, t) \in \mathcal{C} \cdot \text{Discr} (c, \beta', ve, b). \exists b'. b' \in \text{ran} \beta \land b \leq b')\]
\textbf{proof} (\emph{induct arbitrary; d ds ve b c \beta' rule: evalF-evalC-induct})
\textbf{case} Admissibility \textbf{show} ?case
\textit{by} (\emph{intro adm-lemmas adm-ball' adm-prod-split adm-not-conj adm-not-mem adm-single-valued cont2cont})
\textbf{next}
\textbf{case} Bottom \{
\textbf{case} 1 \textbf{thus} ?case \textbf{by} auto
\textbf{next}
\}
case 2 thus \( \text{case by auto} \)
}
next
case (Next evalF evalC)

Nicer names for the hypotheses:

- \( \text{hyps-C-b} \)
- \( \text{hyps-F-b} \)
- \( \text{hyps-F-sv} \)
- \( \text{hyps-C-sv} \)

\[
\begin{align*}
\text{case} & \quad (\text{Next evalF evalC}) \\
\text{proof} & \quad \text{(cases (d,ds,ve,b) rule:fstate-case, auto simp del:Un-insert-left Un-insert-right)}
\end{align*}
\]

Case Closure

- \( \text{Fix lab' and vs :: var list and c and } \beta' :: \text{benv} \)
  - \( \text{Assume prem-d: } \forall b' \in \text{ran } (\beta'(lab' \mapsto b)) \)
  - \( \text{Assume eq-length: length } vs = \text{length } ds \)
  - \( \text{Have new: } b \in \text{ran } (\beta'(lab' \mapsto b)) \text{ by simp} \)
  - \( \text{Have } b \text{-dom-beta: } \forall b' \in \text{ran } \beta', b' \leq b \)
  - \( \text{Proof fix } b' \text{ assume } b' \in \text{ran } (\beta'(lab' \mapsto b)) \)
    - \( \text{Hence } b' \in \text{ran } \beta' \lor b' \leq b \text{ by (auto dest:ran-upd}[T \text{HEN subsetD})] \)
    - \( \text{Thus } b' \leq b \text{ using prem-d by auto} \)
  - \( \text{Qed} \)
  - \( \text{From contours-in-ve-upds[OF eq-length Next.prems(1) Next.prems(\beta)]} \)
  - \( \text{Have } b \text{-dom-ve: } \forall b' \in \text{contours-in-ve (ve(map (} (v, b) \text{)} vs [\mapsto] ds)). b' \leq b} \)
    - \( \text{By auto} \)
  - \( \text{Show single-valued (evalC::Discr (c, } \beta'(lab' \mapsto b), \text{ve(map (} (v, b) \text{)} vs [\mapsto] ds), b))} \)
    - \( \text{By (rule hyps-C-sv[OF new b-dom-beta b-dom-ve, of c])} \)
  - \( \text{Fix lab and } \beta \text{ and } t} \)
  - \( \text{Assume ((lab, } \beta), t) \in \text{evalC::Discr(c, } \beta'(lab' \mapsto b), \text{ve(map (} (v, b) \text{)} vs [\mapsto] ds),b))} \)
    - \( \text{Thus } \exists b', b' \in \text{ran } \beta \land b \leq b' \)
    - \( \text{By (auto dest: hyps-C-b[OF new b-dom-beta b-dom-ve])} \)
  - \( \text{Next} \)

Case Plus

- \( \text{Fix cp and i1 and i2 and cnt} \)
  - \( \text{Assume } \forall b' \in \text{contours-in-d cnt. } b' < b \)
  - \( \text{Hence } b \text{-dom-ds: } \forall b' \in \text{contours-in-d cnt. } b' < nb b \text{ cp by auto} \)
  - \( \text{Have } b \text{-dom-vs: } \forall d' \in \text{set } [DI (i1+i2)], \forall b' \in \text{contours-in-d d'. } b' < nb b \text{ cp by auto} \)
  - \( \text{Have } b \text{-dom-ve: } \forall b' \in \text{contours-in-ve ve. } b' < nb b \text{ cp using Next.prems(1) by auto} \)
  - \( \{ \)
\textbf{Case If (false branch).} Variable names swapped for easier code reuse.

\begin{verbatim}
 fix t
 assume \((\langle cp, [cp \mapsto b] \rangle), t) \in \text{evalF}\left(\text{Discr (cntt, [DI (i1 + i2)], ve, nb \ b \ cp)}\right)
 hence False by (auto dest:hyps-F-b\left(\text{OF b-dom-ve b-dom-d b-dom-ds}\right))
 \}
 with \text{hyps-F-sv(OF b-dom-ve b-dom-d b-dom-ds)}
 show \text{single-valued (\langle evalF\left(\text{Discr (cntt, [DI (i1 + i2)], ve, nb \ b \ cp)}\right)\rangle)}
 \cup \langle\langle cp, [cp \mapsto b]\rangle, cntt\rangle\rangle
 by (auto intro:single-valued-insert)

 fix lab \beta t
 assume \((\langle lab, \beta, t) \in \text{evalF}\left(\text{Discr (cntt, [DI (i1 + i2)], ve, nb \ b \ cp)}\right)\rangle
 thus \exists b'. b' \in \text{ran} \beta \land b \leq b'
 by (auto dest: hyps-F-b\left(\text{OF b-dom-ve b-dom-d b-dom-ds}\right))
\end{verbatim}

\textbf{next}

\textbf{Case If (true branch)}

\begin{verbatim}
 fix cp1 cp2 i cntt cntf
 assume \forall b\in\text{contours-in-d cntt, b' < b}
 hence \text{b-dom-d:} \forall b\in\text{contours-in-d cntt, b' < nb \ b \ cp1 by auto}
 have \text{b-dom-ds:} \forall \ d' \in \text{set} \] . \forall b\in\text{contours-in-d d', b' < nb \ b \ cp1 by auto}
 have \text{b-dom-ve:} \forall b' \in \text{contours-in-ve ve, b' < nb \ b \ cp1 using Next.prems(I) by auto}
 {  
 fix t
 assume \langle\langle cp1, [cp1 \mapsto b]\rangle, t) \in \text{evalF}\left(\text{Discr (cntt, [], ve, nb \ b \ cp1)}\right)
 hence False by (auto dest:hyps-F-b\left(\text{OF b-dom-ve b-dom-d b-dom-ds}\right))
 \}
 with \text{Next.hyps(I)(OF b-dom-ve b-dom-d b-dom-ds, THEN conjunct1)}
 show \text{single-valued (\langle evalF\left(\text{Discr (cntt, [], ve, nb \ b \ cp1)}\right)\rangle)}
 \cup \langle\langle cp1, [cp1 \mapsto b]\rangle, cntt\rangle\rangle
 by (auto intro:single-valued-insert)

 fix lab \beta t
 assume \langle\langle lab, \beta, t) \in \text{evalF}\left(\text{Discr (cntt, [], ve, nb \ b \ cp1)}\rangle
 thus \exists b'. b' \in \text{ran} \beta \land b \leq b'
 by (auto dest: hyps-F-b\left(\text{OF b-dom-ve b-dom-d b-dom-ds}\right))
\end{verbatim}

\textbf{next}

\textbf{Case If (false branch).} Variable names swapped for easier code reuse.

\begin{verbatim}
 fix cp2 cp1 i cntt cntf
 assume \forall b\in\text{contours-in-d cntt, b' < b}
 hence \text{b-dom-d:} \forall b\in\text{contours-in-d cntt, b' < nb \ b \ cp1 by auto}
 have \text{b-dom-ds:} \forall \ d' \in \text{set} \] . \forall b\in\text{contours-in-d d', b' < nb \ b \ cp1 by auto}
 have \text{b-dom-ve:} \forall b' \in \text{contours-in-ve ve, b' < nb \ b \ cp1 using Next.prems(I) by auto}
 {  
 fix t
 assume \langle\langle cp1, [cp1 \mapsto b]\rangle, t) \in \text{evalF}\left(\text{Discr (cntt, [], ve, nb \ b \ cp1)}\right)
 hence False by (auto dest:hyps-F-b\left(\text{OF b-dom-ve b-dom-d b-dom-ds}\right))
\end{verbatim}
with Next.hyps(1) [OF b-dom-ve b-dom-d b-dom-ds, THEN conjunct1]
show single-valued ((evalF (Discr (cntt, [], ve, nb b cp1)))
  ∪ {((cp1, [cp1 → b]), cntt)})
  by (auto intro:single-valued-insert)

fix lab β t
assume ((lab, β), t) ∈ evalF (Discr (cntt, [], ve, nb b cp1))
thus ∃ b'. b' ∈ ran β ∧ b ≤ b'
  by (auto dest: hyps-F-b [OF b-dom-ve b-dom-d b-dom-ds])
qed
next
case (2 ve b c β')
thus ?case
proof (cases c, auto simp add:HOL.Let-def simp del: Un-insert-left Un-insert-right evalV.simps)

Case App

fix lab' f vs

have prem2': ∀ b' ∈ ran β', b' < nb b lab' using Next.prems(2) by auto
have prem3': ∀ b' ∈ contours-in-ve ve. b' < nb b lab' using Next.prems(3) by auto
note c-in-e = contours-in-eval [OF prem3' prem2']

have b-dom-d: ∀ b' ∈ contours-in-d (evalV f β' ve). b' < nb b lab' by (rule c-in-e)
have b-dom-ds: ∀ d' ∈ set (map (λv. evalV v β' ve) vs). ∀ b' ∈ contours-in-d d'. b' < nb b lab'
  using c-in-e by auto
have b-dom-ve: ∀ b' ∈ contours-in-ve ve. b' < nb b lab' by (rule prem3')

have ∀ y. ((lab', β'), y) ∉ evalF (Discr (evalV f β' ve, map (λv. evalV v β' ve) vs, ve, nb b lab'))
  proof (rule allI, rule notI)
    fix y assume ((lab', β'), y) ∈ evalF (Discr (evalV f β' ve, map (λv. evalV v β' ve) vs, ve, nb b lab'))
    hence ∃ b'. b' ∈ ran β' ∧ nb b lab' ≤ b'
      by (auto dest: hyps-F-b [OF b-dom-ve b-dom-d b-dom-ds])
    thus False using prem2' by (auto iff:less-le-not-le)
  qed

with hyps-F-sv [OF b-dom-ve b-dom-d b-dom-ds]
show single-valued (evalF (Discr (evalV f β' ve, map (λv. evalV v β' ve) vs, ve, nb b lab'))
  ∪ {((lab', β'), evalV f β' ve)})
  by (auto intro:single-valued-insert)

fix lab β t
assume ((lab, β), t) ∈ evalF (Discr (evalV f β' ve, map (λv. evalV v β' ve) vs, ve, nb b lab'))
thus \( \exists b'. b' \in \text{ran } \beta \land b \leq b' \)
by (auto dest: hyps-F-b[OF \ b\-dom-ve \ b\-dom-d \ b\-dom-ds])

next

Case Let

fix \( \text{lab} \) ls c'

have prem2': \( \forall b' \in \text{ran } (\beta'([\text{lab}' \mapsto nb \ b \ \text{lab}'])). b' \leq nb \ b \ \text{lab}' \)

proof

fix b' assume b' \( \in \text{ran } (\beta'([\text{lab}' \mapsto nb \ b \ \text{lab}'])) \)

hence b' \( \in \text{ran } \beta' \lor b' = nb \ b \ \text{lab}' \) by (auto dest: ran-upd[THEN subsetD])

thus b' \( \leq nb \ b \ \text{lab}' \) using Next.prems(2) by auto

qed

have prem3': \( \forall b' \in \text{contours-in-ve} \ \text{ve}. b' \leq nb \ b \ \text{lab}' \) using Next.prems(3)

by auto

note c-in-e = contours-in-eval[OF prem3' prem2']

note c-in-ve' = contours-in-ve-upds-binds[OF prem3' prem2']

have b-dom-ve: \( \forall b' \in \text{contours-in-ve} (\text{ve} ++ \text{map-of } (\lambda(v,l). ((v,nb \ b \ \text{lab}'))), \text{eval}(L \ l) ((\beta'(\text{lab}' \mapsto nb \ b \ \text{lab}'))) \ \text{ve})) \ \text{ls}), b' \leq nb \ b \ \text{lab}' \)

by (rule c-in-ve')

have b-dom-beta: \( \forall b' \in \text{ran } (\beta'(\text{lab}' \mapsto nb \ b \ \text{lab}')). b' \leq nb \ b \ \text{lab}' \) by (rule prem2')

have new: \( nb \ b \ \text{lab}' \in \text{ran } (\beta'(\text{lab}' \mapsto nb \ b \ \text{lab}')) \) by simp

from hyps-C-so[OF new b-dom-beta b-dom-ve, of c']

show single-valued (evalC·(Discr c', \beta'(\text{lab}' \mapsto nb \ b \ \text{lab}')), \text{ve} ++ \text{map-of } (\lambda(v,l). ((v, nb \ b \ \text{lab}')), \text{eval}(L \ l) (\beta'(\text{lab}' \mapsto nb \ b \ \text{lab}')) \ \text{ve}))\ls), \ \text{nb} \ b \ \text{lab}'))).

fix \( \text{lab} \beta \ t \)

assume ((\text{lab}, \beta), t) \( \in \text{evalC·(Discr c', \beta'(\text{lab}' \mapsto nb \ b \ \text{lab}')), \text{ve} ++ \text{map-of } (\lambda(v,l). ((v, nb \ b \ \text{lab}')), \text{A}(L \ l) (\beta'(\text{lab}' \mapsto nb \ b \ \text{lab}')) \ \text{ve}))\ls), \ \text{nb} \ b \ \text{lab}' \))

thus \( \exists b'. b' \in \text{ran } \beta \land b \leq b' \)

by -(drule hyps-C-b[OF new b-dom-beta b-dom-ve], auto)

qed

}\}

qed

lemma single-valued (PR prog)

unfolding evalCPS-def

by ((subst HOL.Let-def)+, rule cc-single-valued[THEN conjunct1], auto)

end

6. The abstract semantics is correct

theory AbsCFCorrect

imports AbsCF ExCF >>/src/Tools/Adhoc-Overloading
begin

default-sort \textit{type}

The intention of the abstract semantics is to safely approximate the real control flow. This means that every call recorded by the exact semantics must occur in the result provided by the abstract semantics, which in turn is allowed to predict more calls than actually done.

6.1. Abstraction functions

This relation is expressed by abstraction functions and approximation relations. For each of our data types, there is an abstraction function \textit{abs-<type>}, mapping the a value from the exact setup to the corresponding value in the abstract view. The approximation relation then expresses the fact that one abstract value of such a type is safely approximated by another.

Because we need an abstraction function for contours, we extend the \textit{contour} type class by the abstraction functions and two equations involving the \textit{nb} and \textit{b0} symbols.

\begin{verbatim}
\textbf{class} \textit{contour-a} = \textit{contour} +
  \textbf{fixes} \textit{abs-cnt} :: \textit{contour} \Rightarrow \textquote{a}
  \textbf{assumes} \textit{abs-cnt-nb}[\textit{simp}]: \textit{abs-cnt} (\textit{nb b lab}) = \textit{\hat{nb}} (\textit{abs-cnt b}) \textit{lab}
  \textbf{and} \textit{abs-cnt-initial}[\textit{simp}]: \textit{abs-cnt}(\textit{b0}) = \textit{\hat{b0}}
\end{verbatim}

\textbf{instantiation} \textit{unit} :: \textit{contour-a}

\textbf{begin}
\textbf{definition} \textit{abs-cnt} - = ()
\textbf{instance by} \textit{default auto}
\textbf{end}

It would be unwieldly to always write out \textit{abs-<type>} \textit{x}. We would rather like to write \textit{|x|} if the type of \textit{x} is known, as Shivers does it as well. Isabelle allows one to use the same syntax for different symbols. In that case, it generates more than one parse tree and picks the (hopefully unique) tree that typechecks.

Unfortunately, this does not work well in our case: There are eight \textit{abs-<type>} functions and some expressions later have multiple occurrences of these, causing an exponential blow-up of combinations.

Therefore, we use a module by Christian Sternagel and Alexander Krauss for ad-hoc overloading, where the choice of the concrete function is done at parse time and immediately. This is used in the following to set up the the symbol \textit{|-|} for the family of abstraction functions.
consts abs :: 'a ⇒ 'b (|.|)

adhoc-overloading
abs abs-cnt

definition abs-benv :: benv ⇒ 'c::contour-a benv
where abs-benv β = map-option abs-cnt ∘ β

adhoc-overloading
abs abs-benv

primrec abs-closure :: closure ⇒ 'c::contour-a closure
where abs-closure (l,β) = (l,|β|)

adhoc-overloading
abs abs-closure

primrec abs-d :: d ⇒ 'c::contour-a d
where abs-d (DI i) = {}
     | abs-d (DP p) = { PP p }
     | abs-d (DC cl) = { PC |cl| }
     | abs-d (Stop) = { AStop }

adhoc-overloading
abs abs-d

definition abs-venv :: venv ⇒ 'c::contour-a venv
where abs-venv ve = (λ(v,b-a). ∪\{{ case ve (v,b) of Some d ⇒ |d|, None ⇒ {} | b. |b| = b-a \})

adhoc-overloading
abs abs-venv

definition abs-ccache :: ccache ⇒ 'c::contour-a ccache
where abs-ccache cc = (∪\{(c,β),d) ∈ cc . \{((c,abs-benv β), p) | p . p ∈ abs-d d\})

adhoc-overloading
abs abs-ccache

fun abs-fstate :: fstate ⇒ 'c::contour-a fstate
where abs-fstate (d,ds,ve,b) = (the-elem |d|, map abs-d ds, |ve|, |b|)

adhoc-overloading
abs abs-fstate

fun abs-cstate :: cstate ⇒ 'c::contour-a cstate
where abs-cstate (c,β,ve,b) = (c, |β|, |ve|, |b|)
6.2. Lemmas about abstraction functions

Some results about the abstractions functions.

```plaintext
lemma abs-benv-empty[simp]: |empty| = empty
unfolding abs-benv-def by simp
```

```plaintext
lemma abs-benv-upd[simp]: |β(c↦→b)| = |β| (c ↦→ |b| )
unfolding abs-benv-def by simp
```

```plaintext
lemma the-elem-is-Proc:
assumes isProc cnt
shows the-elem |cnt| ∈ |cnt|
using assms by (cases cnt)auto
```

```plaintext
lemma [simp]: |{}| = {}
unfolding abs-ccache-def by auto
```

```plaintext
lemma abs-cache-singleton [simp]: |{(c,β )| ∈ |d|}|
unfolding abs-ccache-def by simp
```

```
lemma abs-venv-empty[simp]: |empty| = {}.
apply (rule ext) by (auto simp add: abs-venv-def smap-empty-def)
```

6.3. Approximation relation

The family of relations defined here capture the notion of safe approximation.

```plaintext
consts approx :: 'a ⇒ 'a ⇒ bool (- ≪ -)
definition venv-approx :: 'c venv ⇒'c venv ⇒ bool
  where venv-approx = smap-less
```

```plaintext
approx venv-approx
```

```plaintext
definition ccache-approx :: 'c ccache ⇒'c ccache ⇒ bool
  where ccache-approx = less-eq
```

```plaintext
approx ccache-approx
```

```plaintext
definition d-approx :: 'c d ⇒'c d ⇒ bool
  where d-approx = less-eq
```

```plaintext
approx d-approx
```
definition ds-approx :: 'c d list ⇒ 'c d list ⇒ bool
  where ds-approx = list-all2 d-approx

adhoc-overloading
  approx ds-approx

inductive fstate-approx :: 'c fstate ⇒ 'c fstate ⇒ bool
  where \[ [ \langle v, e \rangle \leq _{ve} \langle v', e' \rangle; \langle d, s \rangle \leq _{ds} \langle d', s' \rangle ] \] ⇒ fstate-approx (proc, ds, ve, b) (proc', ds', ve', b)

adhoc-overloading
  approx fstate-approx

inductive cstate-approx :: 'c cstate ⇒ 'c cstate ⇒ bool
  where \[ [ \langle c, \beta, e \rangle \leq _{ve} \langle c, \beta, e' \rangle ] \] ⇒ cstate-approx (c, \beta, ve, b) (c, \beta, ve', b)

adhoc-overloading
  approx cstate-approx

6.4. Lemmas about the approximation relation

Most of the following lemmas reduce an approximation statement about larger structures, as they are occurring the semantics functions, to statements about the components.

lemma venv-approx-trans [trans]:
  fixes ve1 ve2 ve3 :: 'c venv
  shows \[ [ \langle v, e \rangle \leq _{ve} \langle v', e' \rangle; \langle v, e \rangle \leq _{ve} \langle v', e' \rangle ] \] ⇒ (ve1 \leq _{ve} ve3)
  unfolding venv-approx-def by (rule smap-less-trans)

lemma abs-venv-union: |ve1 ++ ve2| \leq _{ve} |ve1| \cup. |ve2|
  by (auto simp add: venv-approx-def smap-less-def smap-union-def, split option.split_asm)

lemma abs-venv-map-of-rev: |map-of (rev l)| \leq _{ve} \cup. (map (\lambda (v, k), \| v \mapsto k \|) \) l)

proof (induct l)
  case Nil show \?
case unfolding abs-venv-def by (auto simp: venv-approx-def smap-less-def)
next
  case (Cons a l)
    obtain v k where a=(v,k) by (rule prod.exhaust)
    hence |map-of (rev (a#l))| \leq _{ve} (\| v \mapsto k \| \cup. |map-of (rev l)|) \Rightarrow 'a venv
      by (auto intro: abs-venv-union)
    also
    have ... \leq _{ve} \cup. (map (\lambda (v, k), \| v \mapsto k \|) \) l)
      by (auto intro!: smap-union-monot OF smap-less-refl Cons[unfolded venv-approx-def] simp: venv-approx-def)
    also
    have ... = \cup. (\| v \mapsto k \| \# map (\lambda (v, k), \| v \mapsto k \|) \) l)
  28
by \((\text{rule } \text{smapping-Union-union})\)
also
have \(\ldots = \bigcup (\text{map } (\lambda (v,k). \llbracket v \mapsto k \rrbracket ) (a \# l))\)
using \((a = (v,k))\)
by \textit{auto}
finally
show \(\text{?case} \).
qed

\textbf{lemma abs-venv-map-of:} \(|\text{map-of } l| \leq \bigcup (\text{map } (\lambda (v,k). \llbracket v \mapsto k \rrbracket ) ) l|\)
using \(\text{abs-venv-map-of-rev}|\text{of } \text{rev } l| \) by \textit{simp}

\textbf{lemma abs-venv-singleton:} \(|\emptyset| \leq \emptyset\)
by (rule \textit{ext}, \textit{auto} \textit{simp} add:\(\text{abs-venv-def smapping-singleton-def smapping-empty-def}\))

\textbf{lemma ccache-approx-empty[simp]:}
\begin{itemize}
\item \textit{fixes } \(x :: \text{'c ccache}\)
\item \textit{shows } \(|\emptyset| \leq x|
\end{itemize}
\textit{unfolding ccache-approx-def \text{ by simp}}

\textbf{lemmas ccache-approx-trans[trans] = subset-trans[where } \('a = ((\text{label } \times \text{'c benv}) \times \text{'c proc}), \text{folded ccache-approx-def}]\)

\textbf{lemmas Un-mono-approx = Un-mono[where } \('a = ((\text{label } \times \text{'c benv}) \times \text{'c proc}), \text{folded ccache-approx-def}]\)

\textbf{lemmas Un-upper1-approx = Un-upper1[where } \('a = ((\text{label } \times \text{'c benv}) \times \text{'c proc}), \text{folded ccache-approx-def}]\)

\textbf{lemmas Un-upper2-approx = Un-upper2[where } \('a = ((\text{label } \times \text{'c benv}) \times \text{'c proc}), \text{folded ccache-approx-def}]\)

\textbf{lemma abs-ccache-union:} \(|c1 \cup c2| \leq |c1| \cup |c2|\)
\textit{unfolding ccache-approx-def abs-ccache-def \text{ by auto}}

\textbf{lemma d-approx-empty[simp]:} \(|\emptyset| \leq (d :: \text{'c d})|
\textit{unfolding d-approx-def \text{ by simp}}

\textbf{lemma ds-approx-empty[simp]:} \(|\emptyset| \leq |\emptyset|\)
\textit{unfolding ds-approx-def \text{ by simp}}

6.5. Lemma 7

Shivers’ lemma 7 says that \(\hat{\mathcal{A}}\) safely approximates \(\mathcal{A}\).

\textbf{lemma lemma7:}
\begin{itemize}
\item \textit{assumes } \(|\text{ve::venv}| \leq \text{ve-a}\)
\item \textit{shows } \(|\mathcal{A} \ f \ \beta \ \text{ve} | \leq \hat{\mathcal{A}} \ f \ |\beta| \ \text{ve-a}\)
\end{itemize}
\textit{proof (cases } \textit{f})
\textit{case } \(R - v\)
\textit{from } \textit{assms} \textit{have } \textit{asm'm: } \forall b. \text{ case-option } \{\} \text{ abs-d } \(\text{ve } (v,b)) \leq \text{ve-a } (v,|b|)\)
\textit{by (auto } \textit{simp add: d-approx-def abs-venv-def venv-approx-def smapping-less-def elim!:allE)\)}
show \( ?\text{thesis} \)
proof (cases \( \beta \) (binder \( v \)))
case None thus \( ?\text{thesis using} \ R \) by auto next
case (Some \( b \))
  thus \( ?\text{thesis using} \ R \ \text{assm}[\text{of} \ v \ b] \)
    by (auto simp add: abs-benv-def split: option.split)
qed
qed (auto simp add: d-approx-def)

6.6. Lemmas 8 and 9

The main goal of this section is to show that \( \hat{F} \) safely approximates \( F \) and that \( \hat{C} \) safely approximates \( C \). This has to be shown at once, as the functions are mutually recursive and requires a fixed point induction. To that end, we have to augment the set of continuity lemmas.

lemma cont2cont-abs-ccache [cont2cont, simp]:
  assumes cont \( f \)
  shows cont (\( \lambda x. \) abs-ccache\( (f \ x) \))
proof
  unfolding abs-ccache-def
  using \( \text{assms} \)
  by (rule cont2cont)(rule cont-const)
qed

Shivers proves these lemmas using parallel fixed point induction over the two fixed points (the one from the exact semantics and the one from the abstract semantics). But it is simpler and equivalent to just do induction over the exact semantics and keep the abstract semantics functions fixed, so this is what I am doing.

lemma lemma89:
  fixes fstate-a :: 'c::contour-a \( \hat{\text{fstate}} \) and cstate-a :: 'c::contour-a \( \hat{\text{cstate}} \)
  shows \( |\text{fstate}| \leq |\text{fstate-a}| \Rightarrow |F\cdot(\text{Discr fstate})| \leq \hat{F}\cdot(\text{Discr fstate-a}) \)
  and \( |\text{cstate}| \leq |\text{cstate-a}| \Rightarrow |C\cdot(\text{Discr cstate})| \leq \hat{C}\cdot(\text{Discr cstate-a}) \)
proof (induct arbitrary: fstate fstate-a cstate cstate-a rule: evalF-evalC-induct)
case Admissibility show ?case
  unfolding ccache-approx-def
  by (intro adm-lemmas adm-subset adm-prod-split adm-not-conj adm-not-mem adm-single-valued cont2cont)
next
case Bottom { 
  case 1 show ?case by simp next
  case 2 show ?case by simp next
}
next
case (Next evalF evalC) {
  case 1
    obtain \( d\) ds ve \( b \) where fstate: fstate = \( (d,ds,ve,b) \)
    by (cases fstate, auto)
    moreover

moreover
obtain proc ds-a ve-a b-a where fstate-a: fstate-a = (proc,ds-a,ve-a,b-a)
  by (cases fstate-a, auto)
ultimately
have abs-d: the-elem |d| = proc
and abs-ds: map abs-d ds ⊆ ds-a
and abs-ve: |ve| ⊆ ve-a
and abs-b: |b| = b-a
using 1 by (auto elim:fstate-approx.cases)
from abs-ds have ds-length: length ds = length ds-a
  by (auto simp add:ds-approx-def dest!:list-all2-lengthD)
from fstate fstate-a abs-d abs-ds abs-ve abs-ds ds-length
  show ?case
proof(cases fstate rule:fstate-case, auto simp del:a-evalF,simps a-evalC,simps set-map)

Case Lambda

fix β and lab and vs:: var list and c
assume ds-a-length: length vs = length ds-a

have |β(lab ↦ b)| = |β| (lab ↦ b-a)
  unfolding below-fun-def using abs-b by simp

moreover
{ have |ve(map (λv. (v, b)) vs ↦ ds)|
  ⊆ |ve| ∪. |map-of (rev (map (λv. (v, b)) vs) ds))|
  unfolding map-apsds-def by (intro abs-venv-union)
also
have ... ⊆ ve-a ∪. (∪. (map (λ(v,k). [(v) ↦ k]) ) (zip (map (λv. (v, b)) vs) ds)))
  using abs-ve abs-venv-map-of-rev
  by (auto intro:smap-union-monono simp add:venv-approx-def)
also
have ... = ve-a ∪. (∪. (map (λ(v,y). {(v,b) ↦ y} ) (zip vs ds)))
  by (auto simp add: zip-map1 o-def split-def)
also
have ... ⊆ ve-a ∪. (∪. (map (λ(v,y). {(v,b-a) := y}. ) (zip vs ds-a)))

proof
  from abs-b abs-ds
  have list-all2 venv-approx (map (λ(v, y). [[(v, b) ↦ y]] ) (zip vs ds))
    (map (λ(v, y). {(v,b-a) := y}. ) (zip vs ds-a))
    by (auto simp add: ds-approx-def d-approx-def venv-approx-def abs-venv-singleton list-all2-conv-all-nth intro:smap-singleton-monono list-all2I)
  thus ?thesis
  by (auto simp add:venv-approx-def intro: smap-union-monono[OF smap-less-refl smap-Union-mono])
qed
finally
have |ve(map (λv, (v, b)) vs ↦ ds)|
  ⊆ ve-a ∪. (∪. (map (λ(v,y). {(v, b-a) := y}. ) (zip vs ds-a))).

31
ultimately
have prem: \( [(c, \beta(lab \mapsto b), ve(map (\lambda v. (v, b)) vs \mapsto ds), b)] \)
\( \subseteq (c, \beta(lab \mapsto b-a), ve-a \cup (\text{map} (\lambda v, y. \{(v, b-a) := y\},) (zip vs ds-a))), b-a) \)
using abs-b
by (auto intro:cstate-approx.intros simp add: abs-cstate.simps)

show \( [\text{evalC}.(\text{Discr} (c, \beta(lab \mapsto b), ve(map (\lambda v. (v, b)) vs \mapsto ds), b))] \)
\( \subseteq \hat{F}.(\text{Discr} (PC (\text{Lambda lab vs c}, \beta), ds-a, ve-a, b-a)) \)
using Next.hyps(2)[OF prem] ds-a-length
by (subst a-evalF.simps, simp del:a-evalF.simps a-evalC.simps)

next

Case Plus

fix lab a1 a2 cnt
assume isProc cnt
assume abs-ds': \{\{\}, \{\}, |cnt| \} \subseteq ds-a
then obtain a1-a a2-a cnt-a where ds-a: ds-a = [a1-a, a2-a, cnt-a] and abs-cnt: |cnt| \( \subseteq \)
cnt-a
unfolding ds-approx-def
by (cases ds-a rule:list.exhaust[OF - list.exhaust[OF - list.exhaust, of - - \lambda- x, x], of - - \lambda-x, x])
(auto simp add:ds-approx-def)

have new-elem: \{[([lab, [lab \mapsto b]], cnt)]\} \( \subseteq \) \{([lab, [lab \mapsto b-a]], cnt)] |cont. cont \in \text{cnt-a}\}
using abs-cnt and abs-b
by (auto simp add:ccache-approx-def d-approx-def)

have prem: \{(cnt, [DI (a1 + a2)], ve, nb b lab)] \( \subseteq \)
(the-elem |cnt|, \{\}, ve-a, \text{nb} b-a lab)
using abs-ve and abs-b
by (auto intro:fstate-approx.intros simp add:ds-approx-def)

have \{(evalF.(\text{Discr} (cnt, [DI (a1 + a2)], ve, nb b lab)))\]
\( \subseteq \hat{F}.(\text{Discr} (\text{the-elem} |cnt|, \{\}, ve-a, \text{nb} b-a lab)) \)
by (rule Next.hyps(1)[OF prem])
also have ... \( \subseteq (\bigcup \text{cnt} \in \text{cnt-a}. \hat{F}.(\text{Discr} (\text{cnt}, \{\}, ve-a, \text{nb} b-a lab))) \)
using abs-cnt
finally
have old-elems: \{(evalF.(\text{Discr} (cnt, [DI (a1 + a2)], ve, nb b lab)))\]
\( \subseteq ( \bigcup \text{cnt} \in \text{cnt-a}. \hat{F}.(\text{Discr} (\text{cnt}, \{\}, ve-a, \text{nb} b-a lab))) \).

have \{(evalF.(\text{Discr} (cnt, [DI (a1 + a2)], ve, nb b lab))) \( \cup \{([\text{lab}, [\text{lab} \mapsto b]], \text{cnt})]\}\)
\( \subseteq \text{evalF.(\text{Discr} (cnt, [DI (a1 + a2)], ve, nb b lab)))} \)
\[ \cup \{(\text{lab}, [\text{lab} \mapsto b]), \text{cnt})\} \]
by (rule abs-ccache-anion)
also
have \( \ldots \approx \)
\( (\cup \text{cnt} \in \text{cnt-a. } \mathcal{F} : (\text{Discr} (\text{cnt}, [\{\}, \text{ve-a, } \mathcal{b} \text{ a lab} '))) \)
\( \cup \{(\text{lab}, [\text{lab} \mapsto b-a]), \text{cont} | \text{cont. cont} \in \text{cnt-a}\} \)
by (rule Un-mono-approx(OF old-elems new-elem))
finally
show \((\text{insert} (\text{lab}, [\text{lab} \mapsto b]), \text{cnt}) \)
\((\text{evalF} : (\text{Discr} (\text{cnt}, [\{\}, \text{ve-a, } \mathcal{b} \text{ a lab} '])) \)
\( \approx \mathcal{F} : (\text{Discr} (\text{PP} (\text{prim.Plus lab}), \text{ds-a, } \text{ve-a, } b-a)) \)
using \(\text{ds-a by (sub} \text{t a-evalF.simps)(auto simp del:a-evalF.simps})\)
next

Case If (true branch)

fix \(ct \text{ cf v cntt cntf}\)
assume isProc cntt
assume isProc cntf
assume abs-ds': \(\{\{\}, |\text{cnttt}|, |\text{cntf}|\} \approx \text{ds-a}\)
then obtain \(v-a \text{ cntt-a cntf-a where } \text{ds-a: } \text{ds-a} = [v-a, \text{ cntt-a, cntf-a}] \)
and abs-cntt: \(|\text{cnttt}| \approx \text{cntt-a}\)
and abs-cntf: \(|\text{cntf}| \approx \text{cntf-a}\)
by (cases \text{ds-a rule:list.exhaust}(OF - list.exhaust(OF - list.exhaust, off - off x, x], of - off x, x])
(auto simp add:ds-approx-def)

let \(?c = ct::label and ?cnt = cntt and ?cnt-a = cntt-a\)

have new-elem: \(\{(\text{?c, [?c} \mapsto b]), \text{?cnt}\}\} \approx \{(\text{?c, [?c} \mapsto b-a]), \text{cont} | \text{cont. cont} \in \text{?cnt-a}\}
using abs-cntt and abs-cntf and abs-b
by (auto simp add:ccache-approx-def d-approx-def)

have prem: \(\{(\text{?cnt}, [], \text{ve, } \mathcal{b} \text{ a } ?c)\}\)
\( \approx \mathcal{F} : (\text{Discr (the-elem |?cnt[, [], ve-a, } \mathcal{b} \text{ a } ?c)})\)
by (rule Next.hyps(1)(OF prem))
also have \(\ldots \approx \cup \text{cnt} \in \text{cnt-a. } \mathcal{F} : (\text{Discr (cnt, [], ve-a, } \mathcal{b} \text{ a } ?c))\)
using abs-cntt and abs-cntf
by (auto intro: the-elem-is-Proc(OF isProc ?cnt) simp del: a-evalF.simps simp add:ccache-approx-def d-approx-def)

finally
have old-elems: \(\text{evalF} : (\text{Discr (?cnt, [], ve, } \mathcal{b} \text{ a } ?c))\)
\( \approx \cup \text{cnt} \in \text{cnt-a. } \mathcal{F} : (\text{Discr (cnt, [], ve-a, } \mathcal{b} \text{ a } ?c}))\).

33
have |evalF.(Discr (?cnt, [], ve, nb b ?c))
    ∪ {{?c, [?c ↦ b]}, ?cnt}}
\[\subseteq\]
|evalF.(Discr (?cnt, [], ve, nb b ?c))
    ∪ |||{{?c, [?c ↦ b]}, ?cnt}}|

by (rule abs-ccache-anion)
also
have ...
\[\subseteq\]
    (∪ cnt∈?cnt-a. \(F\cdot(Discr (cnt, [], ve-a, \widehat{nb} b-a ?c)))
    ∪ {{?c, [?c ↦ b-a]}, cont} |cont. cont ∈ ?cnt-a}

by (rule Un-mono-approx(OF old-elems new-elem))
also
have ...
\[\subseteq\]
    ((∪ cnt∈?cnt-a. \(F\cdot(Discr (cnt, [], ve-a, \widehat{nb} b-a ct)))
    ∪ {{ct, [ct ↦ b-a]}, cont} |cont. cont ∈ ?cnt-a})
    ∪ ((∪ cnt∈?cntf-a. \(F\cdot(Discr (cnt, [], ve-a, nb b-a cf)))
    ∪ {{cf, [cf ↦ b-a]}, cont} |cont. cont ∈ ?cntf-a})

by (rule Un-upper1-approx|rule Un-upper2-approx)
finally
show |insert ({{?c, [?c ↦ b]}, ?cnt)
    |evalF.(Discr (v-cnt, [], ve, nb b ?c)))|\[\subseteq\]
    \(F\cdot(Discr (PP (prim.If ct cf), ds-a, ve-a, b-a))

using ds-a by (subst a-evalF.simps)(auto simp del:a-evalF.simps)

next

Case If (false branch). We use schematic variable to keep this similar to the true branch.

fix ct cf v cntt cntf
assume isProc cntt
assume isProc cntf
assume abs-ds': [{} | cnttt, | cnttf |] \[\subseteq\] ds-a
then obtain \(v\)-a cntt-a cntt-f-a where ds-a: ds-a = [v-a, cnttt-a, cnttf-a]
    and abs-cnttt: |cnttt| \[\subseteq\] cnttt-a
    and abs-cnttf: |cnttf| \[\subseteq\] cnttf-a

by (cases ds-a rule:list.exhaust[OF list.exhaust[OF list.exhaust, of - - \(a\)- x. x], of - - \(\lambda\)- x. x])
(auto simp add:ds-approx-def)

let ?c = cf:\(\lambda\) label and ?cntt = cntt and ?cntt-a = cnttt-a

have new-elem: {{(\(\lambda\) ?c. [?c ↦ b]), ?cntt}} \[\subseteq\] {{(\(\lambda\) ?c. [?c ↦ b-a]), cont} |cont. cont ∈ ?cntt-a}
using abs-cnttt and abs-cntf and abs-b
by (auto simp add:ccache-approx-def d-approx-def)

have prem: |(\(\lambda\) ?cnt, [], ve, nb b ?c)| \[\subseteq\]
    (the-elem |?cnt|, [], ve-a, nb b-a ?c)
using abs-ve and abs-b
by (auto intro:state-approx.intros)
have \( \text{evalF}(\text{Discr} (?\text{cnt}, [], ve, nb\ b\ ?c)) \)
\( \leq \) \( \hat{\text{F}}(\text{Discr} \ (\text{the-elem} \ [?\text{cnt}], [], ve-a, \widehat{nb}\ b-a\ ?c)) \)
by (rule \text{Next}\hyp\text{hyps}(1)\![OF\ \text{prem}])
also have \( \ldots \leq (\bigcup\ \text{cnt}\in?\text{cnt-a}. \ \hat{\text{F}}(\text{Discr} (\text{cnt}, [], ve-a, \widehat{nb}\ b-a\ ?c))) \)
using \text{abs-cnff} and \text{abs-cnft}
by (auto intro: the-elem-is-Proc[OF isProc ?\text{cnt}] simp del: a\-evalF.simps simp add: ccache-approx-def d-approx-def)

finally have old-elems: \( \text{evalF}(\text{Discr} (?\text{cnt}, [], ve, nb\ b\ ?c)) \)
\( \leq (\bigcup\ \text{cnt}\in?\text{cnt-a}. \ \hat{\text{F}}(\text{Discr} (\text{cnt}, [], ve-a, \widehat{nb}\ b-a\ ?c))) \).

have \( \text{evalF}(\text{Discr} (?\text{cnt}, [], ve, nb\ b\ ?c)) \)
\( \cup \{(?(\text{\text{cf}}, [?\text{ve} \mapsto \text{b}\text{-a}]), ?\text{cnt})\} \)
\( \leq \) \( \text{evalF}(\text{Discr} (?\text{cnt}, [], ve, nb\ b\ ?c)) \)
\( \cup \{\{(?!\text{ve}, [?\text{cf} \mapsto \text{b}\text{-a}]), ?\text{cnt})\} \)
by (rule \text{abs-cnff-union})
also have \( \ldots \leq \) \( (\bigcup\ \text{cnt}\in?\text{cnt-a}. \ \hat{\text{F}}(\text{Discr} (\text{cnt}, [], ve-a, \widehat{nb}\ b-a\ ?c))) \)
\( \cup \{(?!\text{ve}, [?\text{cf} \mapsto \text{b}\text{-a}]), \text{cont} | \text{cont. cont}\in?\text{cnt-a}\} \)
by (rule \text{Un-mono-approx}[OF old-elems new-elem])
also have \( \ldots \leq \) \( (\bigcup\ \text{cnt}\in?\text{cnt-a}. \ \hat{\text{F}}(\text{Discr} (\text{cnt}, [], ve-a, \widehat{nb}\ b-a\ cf))) \)
\( \cup \{(?!\text{ve}, [?\text{cf} \mapsto \text{b}\text{-a}]), \text{cont} | \text{cont. cont}\in?\text{cntf-a}\} \)
by (rule \text{Un-upper1-approx}[rule \text{Un-upper2-approx}])
finally show \( \text{insert} ((?!\text{ve}, [?\text{cf} \mapsto \text{b}\text{-a}]), ?\text{cnt}) \)
\( \text{evalF}(\text{Discr} (?\text{cnt}, [], ve, nb\ b\ ?c))) \leq \)
\( \hat{\text{F}}(\text{Discr} (\text{PP} \ (\text{prim.}\text{-If}\ ct\ cf), \text{ds-a}, ve-a, (b-a))) \)
using \text{ds-a} by (subst a\-evalF.simps)(auto simp del:a\-evalF.simps)
qed
next
case 2
obtain \( c\ \beta\ ve\ b\ \text{where}\ c\text{-state}:\ c\text{-state} = (c,\beta,ve,b) \)
by (cases c\text{-state}, auto)
moreover obtain \( c\text{-a}\ \beta\text{-a}\ ds-a\ ve-a\ b-a\ \text{where}\ c\text{-state-a}:\ c\text{-state-a} = (c\text{-a},\beta\text{-a},ve-a,b-a) \)
by (cases c\text{-state-a}, auto)
ultimately have \text{abs-c}: \( c\ =\ c\text{-a} \)
and \text{abs-\beta}: \( \beta\ =\ \beta\text{-a} \)
and \text{abs-ve}: \( ve\ \leq\ ve-a \)
and \text{abs-b}: \( b\ =\ b-a \)
using 2 by (auto elim:cstate-approx.cases)
from estate case-a abs-c abs-β abs-b
show ?case
proof (cases c, auto simp add:HOL.let_def simp del:a-evalF.simps a-evalC.simps set-map evalV.simps)

Case App

fix lab f vs
let ?d = A f β ve
assume isProc ?d

have map (abs-d o (λv. A v β ve)) vs \subseteq map (λv. \hat{A} v β-a ve-a) vs
using abs-β and lemma7[OF abs-ve, of - β]
by (auto intro!: list-all2I simp add:ass-cache-approx-def)

hence |evalF (Discr (?d, map (λv. A v β ve) vs, ve, nb b lab))|
\subseteq \hat{F} (Discr (the-elem ?d), map (λv. \hat{A} v β-a ve-a) vs, ve-a, nb \{b| lab\})
using abs-ve and abs-ctn-nb and abs-b
by -(rule Next.hyps(1),auto intro!:estate-approx.intros)
also have \ldots \subseteq (\bigcup f'∈\hat{A} f β-a ve-a.
\hat{F} (Discr (f', map (λv. \hat{A} v β-a ve-a) vs, ve-a, \hat{nb} \{b| lab\}))
using lemma7[OF abs-ve] the-elem-is-Proc[OF isProc ?d] abs-β
by (auto simp del: a-evalF.simps simp add:d-approx-def ccache-approx-def)
finally
have old-elems:
|evalF (Discr (A f β ve, map (λv. A v β ve) vs, ve, nb b lab))|
\subseteq (\bigcup f'∈\hat{A} f β-a ve-a.
\hat{F} (Discr (f', map (λv. \hat{A} v β-a ve-a) vs, ve-a, \hat{nb} \{b| lab\}))
by auto

have new-elem: |{(lab, β), A f β ve}| \subseteq |{(lab, β-a), f'}| f'∈\hat{A} f β-a ve-a
using abs-β and lemma7[OF abs-ve]
by(auto simp add:ccache-approx-def d-approx-def)

have |evalF (Discr (A f β ve, map (λv. A v β ve) vs, ve, nb b lab))|
\bigcup |{(lab, β), A f β ve}|
\subseteq |evalF (Discr (A f β ve, map (λv. A v β ve) vs, ve, nb b lab))|
\bigcup |{(lab, β), A f β ve}|
by (rule abs-ccache-union)
also have \ldots
\subseteq (\bigcup f'∈\hat{A} f β-a ve-a.
\hat{F} (Discr (f', map (λv. \hat{A} v β-a ve-a) vs, ve-a, \hat{nb} \{b| lab\}))
\bigcup |{(lab, β-a), f'}| f'∈\hat{A} f β-a ve-a
by (rule Un-monono-approx[OF old-elems new-elem])
finally
show |insert ((lab, β), A f β ve)
\[
\text{(evalF \cdot (Discr (A f \beta \ vs, map (\lambda v. A v \beta \ vs, ve, nb b lab))]) \leq C \cdot (Discr (App lab f vs, |\beta|, ve-a, |b|))}
\]

using abs-\beta 

by (subst a-evalC.simps)(auto simp add: HOL.\Let-def simp del:a-evalF.simps)

next

Case Let

fix lab binds c'

have |\beta| \{lab \mapsto \to nb b lab\} = 
\beta-a \{lab \mapsto \to nb |b| lab\}

using abs-\beta and abs-b

by simp

moreover

have |map-of (map (\lambda (v, l). ((v, nb b lab),
DC (l, \beta|lab \mapsto nb b lab\}))

binds)| \leq \bigcup \{map (\lambda (v, l).
\{(v, nb |b| lab) := \{PC (l, \beta-a \{lab \mapsto \to nb |b| lab\})\}\})

binds\}

using abs-b and abs-\beta

apply –

apply (rule venv-approx-trans[OF abs-venv-map-of])

apply (auto intro:smap-union-mono list-all2I

\simps add:venv-approx-def o-def set-zip abs-venv-singleton split-def smap-less-refl)

done

hence |ve ++ map-of 
(map (\lambda (v, l).
\{(v, nb b lab),
DC (l, \beta|lab \mapsto nb b lab\}))

binds)| \leq 

ve-a \cup.

(\bigcup \{map (\lambda (v, l).
\{(v, nb |b| lab) := \{PC (l, \beta-a \{lab \mapsto \to nb |b| lab\})\}\})

binds\})

by (rule venv-approx-trans[OF abs-venv-union
smap-union-mono[OF abs-ve[unfolded venv-approx-def], folded venv-approx-def]])

ultimately

have |evalC \cdot (Discr(c', \beta|lab \mapsto nb b lab),
ve ++ map-of 
(map (\lambda (v, l). ((v, nb b lab), DC (l, \beta|lab \mapsto nb b lab\})) bounds),

nb b lab))| \leq C \cdot (Discr (c', \beta-a \{lab \mapsto \to nb |b| lab\),
ve-a \cup.

(\bigcup \{map (\lambda (v, l).
\{(v, nb |b| lab) := \{PC (l, \beta-a \{lab \mapsto \to nb |b| lab\})\}\})

binds),

37
\[ \alpha \beta ] = \beta
\]
using abs-cnt-nb and abs-b
by \(-\) (rule Next.hyps(2), auto intro: estate-approx.intros)
thus \[ \alpha x = \alpha x \cup \alpha (r x) \]
binds),
\[ \alpha x \leq \alpha x \]
using abs-\beta
by (subst a-evlC.simps)(auto simp add: HOL.Let-def simp del:a-evlC.simps)
qed

And finally, we lift this result to \( \widehat{\mathcal{P}R} \) and \( \mathcal{P}R \).

\begin{enumerate}
\item \textbf{lemma lemma6:} \( \mathcal{P}R \leq \widehat{\mathcal{P}R} \leq \mathcal{P}R \)
\end{enumerate}

\textbf{unfolding} evlCPS-def evlCPS-a-def
by (auto intro!:lemma89 fstate-approx.intros simp del:evalF.simps a-evlF.simps simp add: ds-approx-def d-approx-def venv-approx-def)
end

\section{Generic Computability}

\begin{enumerate}
\item \textbf{theory} Computability
\item \textbf{imports} HOLCF HOLCFUtils
\end{enumerate}

begin

Shivers proves the computability of the abstract semantics functions only by generic and slightly simplified example. This theory contains the abstract treatment in Section 4.4.3. Later, we will work out the details apply this to \( \widehat{\mathcal{P}R} \).

\subsection{Non-branching case}

After the following lemma (which could go into \textit{Set-Interval}), we show Shivers’ Theorem 10. This says that the least fixed point of the equation
\[ f x = g x \cup f (r x) \]
is given by
\[ f x = \bigcup_{i \geq 0} g (r^i x). \]

The proof follows the standard proof of showing an equality involving a fixed point:

38
First we show that the right hand side fulfills the above equation and then show that our solution is less than any other solution to that equation.

**lemma** `insert_greaterThan`:

```
insert (n::nat) {n<..} = {n..}
```

by `auto`

**lemma** `theorem10`:

```
fixed g :: 'a::po ⇒ 'b::type set and r :: 'a ⇒ 'a
shows fix (Λ f x. g x ∪ f (r x)) = (Λ x. (∪ i. g (r i x)))
proof (induct rule: fix-eqI[OF cfun-eqI cfun_belowI, case-names fp least])
case (fp x)
  have g x ∪ (∪ i. g (r i x)) = g (r 0 x) ∪ (∪ i. g (Suc i x))
    by (simp add: iterate-Suc2 del: iterate-Suc)
also have ... = g (r 0 x) ∪ (∪ i∈{0<..}. g (r i x))
    by `auto`
also have ... = (∪ i∈insert 0 {0<..}. g (r i x))
    by `simp`
also have ... = (∪ i. g (r i x))
    by (simp only: insert_greaterThan atLeast-0 )
finally show ?case by `auto`
next
case (least f x)
hence expand: `∀ x. f x = (g x ∪ f (r x))` by (auto simp: cfun-eq-iff)
{ fix n
  have f x = (∪ i∈{..n}. g (r i x)) ∪ f (Suc n x)
    proof (induct n)
      case 0 thus ?case by (auto simp add: expand[of i x])
      case (Suc n)
      then have f x = (∪ i∈{..n}. g (r i x)) ∪ f (Suc n x) by `simp`
      also have ... = (∪ i∈{..n}. g (r i x))
        ∪ g (Suc n x) ∪ f (Suc (Suc n) x)
        by (subst expand[of r Suc n x], `auto`)
      also have ... = (∪ i∈insert (Suc n) {..n}. g (r i x)) ∪ f (Suc (Suc n) x)
        by `auto`
      also have ... = (∪ i∈{..Suc n}. g (r i x)) ∪ f (Suc (Suc n) x)
        by (simp add: atMost-Suc)
      finally show ?case .
    qed
  } note `fin = this`
  have (∪ i. g (r i x)) ⊆ f x
    proof (rule UN-least)
      fix i
      show g (r i x) ⊆ f x
        using `fin[of i]` by `auto`
    qed
  thus ?case
    apply (subst sqsubset_is_subset) by `auto`
```

39
7.2. Branching case

Actually, our functions are more complicated than the one above: The abstract semantics functions recurse with multiple arguments. So we have to handle a recursive equation of the kind

\[ f \ x = g \ x \cup \bigcup_{a \in R} f \ r. \]

By moving to the power-set relatives of our function, e.g.

\[ g^Y = \bigcup_{a \in A} g \ a \quad \text{and} \quad R^Y = \bigcup_{a \in R} R \ a \]

the equation becomes

\[ f^Y = g^Y \cup f (R^Y) \]

(which is shown in Lemma 11) and we can apply Theorem 10 to obtain Theorem 12.

We define the power-set relative for a function together with some properties.

**definition** powerset-lift :: (\'a::cpo \to \'b::type set) \Rightarrow \'a set \to \'b set ()

\[ \text{where } f = (\Lambda \ S. \ (\bigcup_{y \in S} \ f \ y)) \]

**lemma** powerset-lift-singleton[simp]:
\[ f\{x\} = f \cdot x \]
unfolding powerset-lift-def by simp

**lemma** powerset-lift-union[simp]:
\[ f\ (A \cup B) = f \cdot A \cup f \cdot B \]
unfolding powerset-lift-def by auto

**lemma** UNION-commute: (\bigcup_{x \in A. \bigcup_{y \in B} . \ P \ x \ y} = (\bigcup_{y \in B} . \bigcup_{x \in A . \ P \ x \ y})
by auto

**lemma** powerset-lift-UNION:
\[ (\bigcup_{x \in S} . \ g\ (A \ x)) = g\ (\bigcup_{x \in S} . \ A \ x) \]
unfolding powerset-lift-def by auto

**lemma** powerset-lift-iterate-UNION:
\[ (\bigcup_{x \in S} . \ (g)\ ^{i} (A \ x)) = (g)\ ^{i} (\bigcup_{x \in S} . \ A \ x) \]
by (induct \ i, auto simp add: powerset-lift-UNION)

**lemmas** powerset-distr = powerset-lift-UNION powerset-lift-iterate-UNION

Lemma 11 shows that if a function satisfies the relation with the branching \( R \), its powerset function satisfies the powerset variant of the equation.
lemma lemma11:
fixes $f :: 'a \rightarrow 'b$ set and $g :: 'a \rightarrow 'b$ set and $R :: 'a \rightarrow 'a$ set
assumes $\bigwedge x. f x = g x \cup (\bigcup y \in R x. f y)$
shows $f S = g S \cup f(R S)$
proof
  have $f S = (\bigcup x \in S . f x)$ unfolding powerset-lift-def by auto
  also have $\ldots = (\bigcup x \in S . g x \cup (\bigcup y \in R x. f y))$ apply (subst assms) by simp
  also have $\ldots = g S \cup f(R S)$ by (auto simp add:powerset-lift-def)
  finally show $\textit{thesis}$.
qed

Theorem 10 as it will be used in Theorem 12.
lemmas theorem10ps = theorem10[of $g \in$] for $g r$

Now we can show Lemma 12: If $F$ is the least solution to the recursive power-set equation, then $x \mapsto F x$ is the least solution to the equation with branching $R$.

We fix the type variable $'a$ to be a discrete cpo, as otherwise $x \mapsto \{x\}$ is not continuous.

lemma theorem12' :
fixes $g :: 'a::discrete-cpo \rightarrow 'b::type$ set and $R :: 'a \rightarrow 'a$ set
assumes $F\cdot fix :: (\Lambda x. g x \cup F\cdot (R x))$
shows $fix\cdot (\Lambda x. g x \cup (\bigcup y \in R x. f y)) = (\Lambda x. F\cdot x)$
proof (induct rule:fix-eqI cfun-belowI, case-names fp least)
  have $F\cdot union :: (\Lambda x. g ((R)^i x))$
    using $F\cdot fix$ by (simp)(rule theorem10ps)
  case (fp $x$)
  have $g x \cup (\bigcup x' \in R\cdot x. F\cdot x'\cdot) = g\cdot \{x\} \cup F\cdot (R\cdot \{x\})$
    unfolding powerset-lift-singleton
    by (auto simp add: powerset-distr UNION-commute F-union)
  also have $\ldots = F\cdot \{x\}$
    by (subst (2) fix-eq4[OF $F\cdot fix$], auto)
  finally show $\textit{case}$ by simp
next
  case (least $f'\cdot x$)
  hence expand: $f' = (\Lambda x. g x \cup (\bigcup y \in R x. f x\cdot y))$ by simp
  have $f' = (\Lambda x. g S \cup f\cdot (R S))$ by (subst expandd, rule cfun-eqI, auto simp add:powerset-lift-def)
  hence $(\Lambda F. F\cdot x. g x \cup F\cdot (R x))\cdot (f') = f'\cdot x$ by simp
  from fix-least[OF this] and $F\cdot fix$
  have $F \subseteq f'\cdot x$ by simp
  hence $F\cdot \{x\} \subseteq f'\cdot \{x\}$
    by (subst (asm) cfun-below_iff, auto simp del:powerset-lift-singleton)
  thus $\textit{case}$ by (auto simp add:sqsubset_is-subset)
qed

lemma theorem12:
The point of the abstract semantics is that it is computable. To show this, we exploit the special structure of $\hat{F}$ and $\hat{C}$. Each call adds some elements to the result set and joins this with the results from a number of recursive calls. So we separate these two actions into separate functions. These take as arguments the direct sum of $\hat{f}\text{state}$ and $\hat{c}\text{state}$, i.e. we treat the two mutually recursive functions now as one.

\texttt{abs-g} gives the local result for the given argument.

\texttt{fixrec abs-g : ('c::contour $\hat{f}\text{state}$ + 'c $\hat{c}\text{state}$) discr $\rightarrow$ 'c $\hat{a}\text{ans}$}

\texttt{where abs-g x = (case undiscr x of}

\hspace{1cm} (\texttt{Inl (PC (Lambda lab vs c, \beta), as, ve, b)}) $\Rightarrow$ \{\}

\hspace{1cm} $|$ (\texttt{Inl (PP (Plus c),[\_,conts],ve,b)}) $\Rightarrow$

\hspace{2cm} \texttt{let b' = nb b c;}

\hspace{2cm} $\beta = [c \mapsto b]$

\hspace{2cm} in \{((c, \beta), cont) | cont . cont $\in$ cnts\}

\hspace{1cm} $|$ (\texttt{Inl (PP (prim.If ct cf),[\_,cntts],cntfs,ve,b)}) $\Rightarrow$

\hspace{2cm} ((

\hspace{3cm} \texttt{let b' = nb b ct;}

\hspace{3cm} $\beta = [ct \mapsto b]$

\hspace{3cm} in \{((ct, \beta), cnt) | cnt . cnt $\in$ cntts\}

\hspace{2cm} ) $\cup$

\hspace{3cm} \texttt{let b' = nb b cf;}

\hspace{3cm} $\beta = [cf \mapsto b]$

\hspace{3cm} in \{((cf, \beta), cnt) | cnt . cnt $\in$ cntfs\}

\hspace{1cm} )

\hspace{1cm} $|$ (\texttt{Inl (AStop,[\_,\_,\_])}) $\Rightarrow$ \{\}

\hspace{1cm} $|$ (\texttt{Inl _}) $\Rightarrow$ \$

\hspace{1cm} $|$ (\texttt{Inr (App lab f vs,\beta,ve,b)}) $\Rightarrow$

\hspace{2cm} \texttt{let fs = \hat{A} f \beta ve;}

\hspace{2cm} \texttt{as = map (\lambda v. \hat{A} v \beta ve) vs;}

\hspace{2cm} \texttt{b' = nb b lab}

\hspace{2cm} in \{((lab, \beta),f') | f'. f' $\in$ fs\}

\hspace{1cm} $|$ (\texttt{Inr (Let lab ls c',\beta,ve,b)}) $\Rightarrow$ \{\}

\)

end
abs-\(R\) gives the set of arguments passed to the recursive calls.

\textbf{fixrec} abs-\(R\) \(\mathrel{::} (\textsuperscript{c::contour} \text{\fstate} + \text{'c \text{estate}}) \text{\discr} \rightarrow (\textsuperscript{c::contour} \text{\fstate} + \text{'c \text{estate}}) \text{\discr}\)

\textbf{where} abs-\(R\) \(x = (\case\ \text{undisc} \ x\ of\)

\((\text{Inl} \ (\text{PC} \ (\text{Lambda} \ \text{lab} \ v c, \ \beta, \ \text{as}, \ \text{ve}, \ b))) \Rightarrow\)
\((\text{if length} \ \text{vs} = \text{length} \ \text{as})\)
\(\text{then let} \ \beta' = \beta \ (\text{lab} \mapsto b);\)
\(\text{ve'} = \text{ve} \cup (\bigcup \ (\text{map} \ (\lambda(v,a), \ \{(v,b) := a,\}) \ (\text{zip} \ \text{vs})))\)
\(\text{in} \ \{\text{Discr} \ (\text{Inr} \ (c,\beta',\text{ve'},b))\}\)
\(\text{else} \ \bot\)

\((\text{Inl} \ (\text{PP} \ \text{(Plus} c),\text{[]},\text{cnts},\text{ve},b)) \Rightarrow\)
\(\text{let} \ b' = \overline{nb} \ b \ c;\)
\(\beta = [c \mapsto b]\)
\(\text{in} \ \{(\bigcup \ \text{cnt} \in \text{cnts} . \ \{\text{Discr} \ (\text{Inl} \ [\text{cnt},\text{ve},b])\})\}\)

\((\text{Inl} \ (\text{PP} \ \text{(prim\_If} \ c \text{cf},\text{[]}, \text{cntts}, \text{cntfs},\text{ve},b)) \Rightarrow\)
\((\text{let} \ b' = \overline{nb} \ b \ c;\)
\(\beta = [ct \mapsto b]\)
\(\text{in} \ \{(\bigcup \ \text{cnt} \in \text{cntfs} . \ \{\text{Discr} \ (\text{Inl} \ [\text{cnt},\text{ve},b])\})\}\)

\(\bigcup\)

\((\text{Inl} \ (\text{AStop},\text{[]},\text{-},\text{-})) \Rightarrow\)

\((\text{Inl} \ \cdot) \Rightarrow \bot\)

\((\text{Inr} \ \text{App} \ \text{lab} \ f \ vs,\beta,\text{ve},b)) \Rightarrow\)
\(\text{let} \ \text{fs} = \text{\widehat{A}} \ f \ \beta \ \text{ve};\)
\(\text{as} = \text{map} \ (\lambda v, \ \text{\widehat{A}} \ v \ \beta \ \text{ve}) \ \text{vs};\)
\(b' = \overline{nb} \ b \ \text{lab}\)
\(\text{in} \ \{(\bigcup \ f' \in \text{fs} . \ \{\text{Discr} \ (\text{Inl} \ (f',\text{as},\text{ve},b'))\})\}\)

\((\text{Inr} \ \text{Let} \ \text{lab} \ l s \ e',\beta,\text{ve},b)) \Rightarrow\)
\(\text{let} \ b' = \overline{nb} \ b \ \text{lab};\)
\(\beta' = \beta \ (\text{lab} \mapsto b');\)
\(\text{ve'} = \text{ve} \cup (\bigcup \ (\text{map} \ (\lambda(v,l), \ \{(v,b') := (\text{\widehat{A}} \ (\text{L} \ l) \ \beta' \ \text{ve})\} \ \text{ls}))\)
\(\text{in} \ \{\text{Discr} \ (\text{Inr} \ (e',\beta',\text{ve'},b'))\}\)

)\)

The initial argument vector, as created by \(\widehat{PR}\).

\textbf{definition} initial-r \(\mathrel{::} \text{prog} \Rightarrow (\textsuperscript{c::contour} \text{\fstate} + \text{'c \text{estate}}) \text{\discr}\)

\textbf{where} initial-r \ \text{prog} = \text{Discr} \ (\text{Inl} \ (\text{the-elem} \ (\text{\widehat{A}} \ (\text{L} \ \text{prog} \ \text{empty} \ \{\}, \ \{\text{\AS}, \ \text{empty} \ \{\}, \ \text{empty} \ \{\ tenth\})\}))\)

43
8.1. Towards finiteness

We need to show that the set of possible arguments for a given program \( p \) is finite. Therefore, we define the set of possible procedures, of possible arguments to \( \hat{F} \), or possible arguments to \( \hat{\mathcal{C}} \) and of possible arguments.

**definition** proc-poss :: prog \( \Rightarrow \) 'c::contour proc set
  where proc-poss \( p = PC' (\lambda p \times \text{maps-over} (\text{labels} \ p) \ \text{UNIV}) \cup PP' \ \text{prims} \ p \cup \{ \text{AStop} \} \)

**definition** fstate-poss :: prog \( \Rightarrow \) 'c::contour a-fstate set
  where fstate-poss \( p = (\text{proc-poss} \ p \times \text{NList} (\text{Pow} (\text{proc-poss} \ p)) (\text{call-list-lengths} \ p) \times \text{smaps-over} (\text{vars} \ p \times \text{UNIV}) (\text{proc-poss} \ p) \times \text{UNIV}) \)

**definition** cstate-poss :: prog \( \Rightarrow \) 'c::contour a-cstate set
  where cstate-poss \( p = (\text{calls} \ p \times \text{maps-over} (\text{labels} \ p) \ \text{UNIV} \times \text{smaps-over} (\text{vars} \ p \times \text{UNIV}) (\text{proc-poss} \ p) \times \text{UNIV}) \)

**definition** arg-poss :: prog \( \Rightarrow \) ('c::contour a-fstate + 'c a-cstate) discr set
  where arg-poss \( p = \text{Discr'} (\text{fstate-poss} \ p <+> \text{cstate-poss} \ p) \)

Using the auxiliary results from CPSUtils, we see that the argument space as defined here is finite.

**lemma** finite-arg-space: finite \( (\text{arg-poss} \ p) \)
  unfolding arg-poss-def and cstate-poss-def and fstate-poss-def and proc-poss-def
  by (auto intro!: finite-cartesian-product finite-imageI maps-over-finite smaps-over-finite finite-UNIV finite-Nlist)

But is it closed? I.e. if we pass a member of \( \text{arg-poss} \) to \( \text{abs-R} \), are the generated recursive call arguments also in \( \text{arg-poss} \)? This is shown in \( \text{arg-space-complete} \), after proving an auxiliary result about the possible outcome of a call to \( \hat{A} \) and an admissibility lemma.

**lemma** evalV-possible:
  assumes \( f: f \in \hat{A} \ d \ \beta \ \text{ve} \)
  and \( d: d \in \text{vals} \ p \)
  and \( \text{ve: ve} \in \text{smaps-over} (\text{vars} \ p \times \text{UNIV}) (\text{proc-poss} \ p) \)
  and \( \beta: \beta \in \text{maps-over} (\text{labels} \ p) \ \text{UNIV} \)
  shows \( f \in \text{proc-poss} \ p \)
  proof (cases \((d,\beta,\text{ve})\) rule: evalV-a.cases)
    case \((1 \ \text{cl} \ \beta' \ \text{ve}')\)
      thus \?thesis using \( f \) by auto
    next
    case \((2 \ \text{prim} \ \beta' \ \text{ve}')\)
      thus \?thesis using \( d \ f \)
        by (auto dest: vals1 simp add:proc-poss-def)
    next
    case \((3 \ \text{l var} \ \beta' \ \text{ve}')\)
      thus \?thesis using \( f \ d \ \text{smaps-over-im}[\text{OF - ve}] \)

44
by (auto split:option.split-asm dest: vals2)

next

case (4 l β ve)

thus ?thesis using f d β

by (auto dest!: vals3 simp add:proc-poss-def)

qed

lemma adm-subset: cont (λx. f x) ⇒ adm (λx. f x ⊆ S)

by (subst sqsubset-is-subset[THEN sym], introadm-lemmas cont2cont)

lemma arg-space-complete:

state ∈ arg-poss p ⇒ abs-R state ⊆ arg-poss p

proof (induct rule: abs-R.induct [case-names Admissibility Bot Step])

case Admissibility show ?case

by (intro adm-lemmas adm-subset cont2cont)

next

case Bot show ?case by simp next

case (Step abs-R)

note state = Step(2)

show ?case

proof (cases state)

case (Discr state') show ?thesis

proof (cases state')

case (Inl fstate) show ?thesis

using Inl Discr state

proof (cases fstate rule: a-fstate-case, auto)

Case Lambda

fix l vs c β as ve b

assume Discr (Inl (PC (Lambda l vs c, β), as, ve, b)) ∈ arg-poss p

hence lam: Lambda l vs c ∈ lambdas p

and beta: β ∈ maps-over (labels p) UNIV

and ve: ve ∈ smaps-over (vars p × UNIV) (proc-poss p)

and as: as ∈ NList (Pow (proc-poss p)) (call-list-lengths p)

unfolding arg-poss-def fstate-poss-def proc-poss-def by auto

from lam have c ∈ calls p

by (rule lambdas1)

moreover

from lam have l ∈ labels p

by (rule lambdas2)

with beta have β(l → b) ∈ maps-over (labels p) UNIV

by (rule maps-over-upd, auto)

moreover

from lam have vs: set vs ⊆ vars p by (rule lambdas3)
from as have \( \forall x \in \text{set as}, x \in \text{Pow (proc-poss p)} \)

unfolding NList-def nList-def by auto

with vs have ve \( \cup \bigcup . \text{map} (\lambda (v, y). \{ (v, b) := y \}) (\text{zip vs as}) \in \text{smaps-over vars p \times UNIV (proc-poss p) (is ve' \in -)} \)

by (auto intro!: smaps-over-un[OF ve] smaps-over-Union smaps-over-singleton)

ultimately

have (c, \( \beta (l \mapsto b), ?, ve', b \)) \( \in \text{cstate-poss p (is ?cstate \in -)} \)

unfolding cstate-poss-def by simp

thus Discr (Inr ?cstate) \( \in \text{arg-poss p} \)

unfolding arg-poss-def by auto

next

Case Plus

fix ve b l v1 v2 cnts cnt

assume Discr (Inl (PP (prim.Plus l), [v1, v2, cnts], ve, b)) \( \in \text{arg-poss p} \)

and cnt \( \in \text{cnts} \)

hence cnt \( \in \text{proc-poss p} \)

and ve \( \in \text{smaps-over vars p \times UNIV (proc-poss p)} \)

unfolding arg-poss-def fstate-poss-def NList-def nList-def by auto

moreover

have [[]] \( \in \text{NList (Pow (proc-poss p)) (call-list-lengths p)} \)

unfolding call-list-lengths-def NList-def nList-def by auto

ultimately

have (cnt, [[]], ve, \( \tilde{\overline{\text{nb b l}}}) \( \in \text{fstate-poss p} \)

unfolding fstate-poss-def by auto

thus Discr (Inl (cnt, [[]], ve, \( \tilde{\overline{\text{nb b l}}})) \( \in \text{arg-poss p} \)

unfolding arg-poss-def by auto

next

Case If (true case)

fix ve b l1 l2 v cntst cntsf cnt

assume Discr (Inl (PP (prim.If l1 l2), [v, cntst, cntsf], ve, b)) \( \in \text{arg-poss p} \)

and cnt \( \in \text{cntst} \)

hence cnt \( \in \text{proc-poss p} \)

and ve \( \in \text{smaps-over vars p \times UNIV (proc-poss p)} \)

unfolding arg-poss-def fstate-poss-def NList-def nList-def by auto

moreover

have [] \( \in \text{NList (Pow (proc-poss p)) (call-list-lengths p)} \)

unfolding call-list-lengths-def NList-def nList-def by auto

ultimately

have (cnt, [], ve, \( \tilde{\overline{\text{nb b l1}}}) \( \in \text{fstate-poss p} \)

unfolding fstate-poss-def by auto

thus Discr (Inl (cnt, [], ve, \( \tilde{\overline{\text{nb b l1}}})) \( \in \text{arg-poss p} \)
unfolding arg-poss-def by auto

next

Case If (false case)

fix ve b l1 l2 v cntst cntsf cnt
assume Discr (Inl (PP (prim.If l1 l2), [v, cntst, cntsf], ve, b)) ∈ arg-poss p
  and cnt ∈ cntsf
hence ve ∈ snaps-over (vars p × UNIV) (proc-poss p)
unfolding arg-poss-def fstate-poss-def NList-def nList-def
by auto
moreover
have [] ∈ NList (Pow (proc-poss p)) (call-list-lengths p)
unfolding call-list-lengths-def NList-def nList-def by auto
ultimately
have (cnt, [], ve, ñb b l2) ∈ fstate-poss p
  using Discr Inl fields state
  proof
    case fields c β ve b
      have len: length ds ∈ call-list-lengths p
        using f d ve beta by (rule evalV-possible)
        moreover
        have map (λv. À v β ve) ds ∈ NList (Pow (proc-poss p)) (call-list-lengths p)
        using ds len
      unfolding NList-def by (auto simp add:NList-def introl: evalV-possible[OF - - ve beta])
    qed
next
case (Inr cstate)
show ?thesis proof(cases cstate rule: prod-cases4)
case (fields c β ve b)
  have [] ∈ NList (Pow (proc-poss p)) (call-list-lengths p)
  unfolding call-list-lengths-def NList-def nList-def by auto
ultimately
  have (cnt, [], ve, ñb b l2) ∈ fstate-poss p
  unfolding fstate-poss-def by auto
thus Discr (Inl (cnt, [], ve, ñb b l2)) ∈ arg-poss p
unfolding arg-poss-def by auto
qed

next

case App

fix l d ds f
assume arg: Discr (Inr (App l d ds, β, ve, b)) ∈ arg-poss p
  and f ∈ À d β ve
hence c: App l d ds ∈ calls p
  and d: d ∈ vals p
  and ds: set ds ⊆ vals p
  and beta: β ∈ maps-over (labels p) UNIV
  and ve: ve ∈ snaps-over (vars p × UNIV) (proc-poss p)
by (auto simp add: arg-poss-def estate-poss-def call-list-lengths-def dest: app1 app2)

have len: length ds ∈ call-list-lengths p
  by (auto intro: rev-image-eqI[OF c] simp add: call-list-lengths-def)

have f ∈ proc-poss p
  using f d ve beta by (rule evalV-possible)
moreover
have map (λv. À v β ve) ds ∈ NList (Pow (proc-poss p)) (call-list-lengths p)
  using ds len
unfolding NList-def by (auto simp add:nList-def introl: evalV-possible[OF - - ve beta])

47
ultimately

have \((f, \text{map}\ (\lambda v. \hat{A} v \beta ve) ds, ve, \hat{nb} b l) \in \text{fstate-poss } p)\ (\is \ ?\text{fstate} \in -)

using ve

unfolding fstate-poss-def by simp

thus \text{Discr} (\text{Inl} ?\text{fstate}) \in \text{arg-poss } p

unfolding arg-poss-def by auto

next

Case Let

fix \ l \ binds \ c'

assume arg: \text{Discr} (\text{Inr} (\text{Let} \ l \ binds \ c', \beta, ve, b)) \in \text{arg-poss } p

hence \ l: l \in \text{labels } p

and \ c': c' \in \text{calls } p

and vars: \text{fst ' set bounds} \subseteq \text{vars } p

and is: \text{snd ' set bounds} \subseteq \text{lambdas } p

and beta: \beta \in \text{maps-over} (\text{labels } p) \ \text{UNIV}

and ve: ve \in \text{smaps-over} (\text{vars } p \times \text{UNIV}) (\text{proc-poss } p)

by (auto simp add:\ \text{arg-poss-def cstate-poss-def} call-list-lengths-def

dest:let1 let2 let3 let4)

have beta': \beta (l \mapsto \hat{nb} b l) \in \text{maps-over} (\text{labels } p) \ \text{UNIV} (\is \ ?\beta' \in -)

by (auto intro: \text{maps-over-upd}[OF beta l])

moreover

have ve \cup \bigcup.\text{map} (\lambda (v, lam). \{ (v, \hat{nb} b l) := \hat{A} (L lam) (\beta(l \mapsto \hat{nb} b l)) ve \}).

\in \text{smaps-over} (\text{vars } p \times \text{UNIV}) (\text{proc-poss } p) (\is \ ?\text{ve'} \in -)

using vars is beta'

by (auto intro!: \text{smaps-over-un}[OF ve] \text{smaps-over-Union})

(auto intro!:\text{smaps-over-singleton simp add: proc-poss-def})

ultimately

have \( (c', \ ?\beta', \ ?\text{ve'}, \hat{nb} b l) \in \text{cstate-poss } p)\ (\is \ ?\text{cstate} \in -)

using c' unfolding cstate-poss-def by simp

thus \text{Discr} (\text{Inr} ?\text{cstate}) \in \text{arg-poss } p

unfolding arg-poss-def by auto

qed

This result is now lifted to the powerset of abs-R.

lemma \text{arg-space-complete-ps}: \text{states} \subseteq \text{arg-poss } p \Rightarrow (\text{abs-R})\text{-states} \subseteq \text{arg-poss } p

using \text{arg-space-complete unfolding} \text{powerset-lift-def by auto}
We are not so much interested in the finiteness of the set of possible arguments but rather of the set of occurring arguments, when we start with the initial argument. But as this is of course a subset of the set of possible arguments, this is not hard to show.

**lemma** UN-iterate-less:

assumes start: \( x \in S \)

and step: \( \forall y. y \subseteq S \implies (f \cdot y) \subseteq S \)

shows \( (\bigcup i. \text{iterate } i \cdot f \cdot \{x\}) \subseteq S \)

**proof** – {

fix \( i \)

have \( \text{iterate } i \cdot f \cdot \{x\} \subseteq S \)

**proof** (induct \( i \))

\begin{cases}
\text{case } 0 & \text{show } ?\text{case using } (x \in S) \text{ by simp next}
\text{case } (\text{Suc } i) & \text{thus } ?\text{case using step[of iterate } i \cdot f \cdot \{x\}] \text{ by simp}
\end{cases}

qed

} thus \( \text{thesis} \) by auto

qed

**lemma** args-finite:

finite \( (\bigcup i. \text{iterate } i \cdot (\text{abs-R}) \cdot \{\text{initial-r } p\}) \) (is finite ?\( S \))

**proof** (rule finite-subset[OF -finite-arg-space])

have \( \text{simp}: p \in \text{lambda}s \) \( p \) by (cases \( p \), simp)

show \( ?S \subseteq \text{arg-poss } p \)

**unfolding** initial-r-def

by (rule UN-iterate-less[OF - arg-space-complete-ps])

(auto simp add:arg-poss-def fstate-poss-def proc-poss-def call-list-lengths-def NLList-def nList-def

 intro!: imageI)

qed

8.2. A decomposition

The functions \( \text{abs-g} \) and \( \text{abs-R} \) are derived from \( \hat{F} \) and \( \hat{C} \). This connection has yet to expressed explicitly.

**lemma** Un-commute-helper: \((a \cup b) \cup (c \cup d) = (a \cup c) \cup (b \cup d)\)

by auto

**lemma** a-evalF-decomp:

\( \hat{F} = \text{fst } (\text{sum-to-tup}(\text{fix}(\Lambda f x. (\bigcup y \in \text{abs-R} \cdot x. f \cdot y) \cup \text{abs-g} \cdot x))) \)

apply (subst a-evalF-def)

apply (subst fix-transform-pair-sum)

apply (rule arg-cong[of - \( \lambda x. \text{fst } (\text{sum-to-tup}(\text{fix} \cdot x))\)]

apply (simp)

apply (simp only: discr-app undiscr-Discr)

apply (rule cfun-eqI, rule cfun-eqI)

apply (case-tac a, case-tac a, simp)

apply (case-tac aa rule: a-fstate-case, simp-all add: Un-commute-helper)

apply (case-tac b rule: prod-cases4)
apply \texttt{(case-tac aa)}
apply \texttt{(simp-all add: HOL.Let-def)}
done

8.3. The iterative equation

Because of the special form of $\hat{F}$ (and thus $\hat{PR}$) derived in the previous lemma, we can apply our generic results from \textit{Computability} and express the abstract semantics as the image of a finite set under a computable function.

\textbf{lemma a-evalF-iterative:}
$\hat{F}(\text{Discr } x) = \text{abs-g}(\bigcup i. \text{iterate } i \cdot (\text{abs-R}) \cdot \{\text{Discr } (\text{Inl } x)\})$
\textbf{by (simp del: abs-R. simps abs-g. simps add: theorem12 Un-commute a-evalF-decomp)}

\textbf{lemma a-evalCPS-iterative:}
$\hat{PR} \text{ prog} = \text{abs-g}(\bigcup i. \text{iterate } i \cdot (\text{abs-R}) \cdot \{\text{initial-r prog}\})$
\textbf{unfolding evalCPS-a-def and initial-r-def}
\textbf{by (subst a-evalF-iterative, simp del: abs-R. simps abs-g. simps evalV-a. simps)}

end

Part III.
The auxiliary theories

9. Syntax tree helpers

\textbf{theory CPSUtils}
\textbf{imports CPSScheme}
\textbf{begin}

This theory defines the sets \texttt{lambdas p}, \texttt{calls p}, \texttt{calls p}, \texttt{vars p}, \texttt{labels p} and \texttt{prims p} as the subexpressions of the program \texttt{p}. Finiteness is shown for each of these sets, and some rules about how these sets relate. All these rules are proven more or less the same ways, which is very inelegant due to the nesting of the type and the shape of the derived induction rule.

It would be much nicer to start with these rules and define the set inductively. Unfortunately, that approach would make it very hard to show the finiteness of the sets in question.

\textbf{fun lambdas :: lambda \Rightarrow lambda set}
\textbf{and lambdasC :: call \Rightarrow lambda set}
\textbf{and lambdasV :: val \Rightarrow lambda set}
\textbf{where lambdas (Lambda l vs c) = (\{Lambda l vs c\} \cup lambdasC c)}
\[
\text{lambdasC} (\text{App } l \ d \ ds) = \text{lambdasV} d \cup (\text{UNION} (\text{set } ds) \text{lambdasV}) \\
\text{lambdasC} (\text{Let } l \ binds \ c') = (\text{UNION} (\text{set binds}) (\lambda(-,l). \text{lambdas } l) \cup \text{lambdasC } c') \\
\text{lambdasV} (L \ l) = \text{lambdas } l \\
\text{lambdasV} - = \{\}
\]

**fun** calls :: \text{lamba} \Rightarrow \text{call set}  \\
**and** callsC :: \text{call} \Rightarrow \text{call set}  \\
**and** callsV :: \text{val} \Rightarrow \text{call set}  \\
**where** calls (\text{Lambda } l \ vs \ c) = callsC c \\
| callsC (\text{App } l \ d \ ds) = \{\text{App } l \ d \ ds\} \cup \text{callsV } d \cup (\text{UNION} (\text{set } ds) \text{callsV}) \\
| callsC (\text{Let } l \ binds \ c') = \{\text{Let } l \ binds \ c'\} \\
\quad \cup (\text{UNION} (\text{set binds}) (\lambda(-,l). \text{calls } l) \\
\quad \cup \text{callsC } c') \\
| callsV (L \ l) = \text{calls } l \\
| callsV - = \{\}

**lemma** finite-lambdas[simp]: finite (\text{lambdas } l) and finite (\text{lambdasC } c) finite (\text{lambdasV } v)  \\
by (induct rule: lambdas-lambdasC-lambdasV.induct, auto)

**lemma** finite-calls[simp]: finite (\text{calls } l) and finite (\text{callsC } c) finite (\text{callsV } v)  \\
by (induct rule: calls-callsC-callsV.induct, auto)

**fun** vars :: \text{lamba} \Rightarrow \text{var set}  \\
**and** varsC :: \text{call} \Rightarrow \text{var set}  \\
**and** varsV :: \text{val} \Rightarrow \text{var set}  \\
**where** vars (\text{Lambda } - vs \ c) = set vs \cup \text{varsC } c \\
| varsC (\text{App } - a \ as) = \text{varsV } a \cup \text{UNION} (\text{set } as) \text{varsV} \\
| varsC (\text{Let } - binds \ c') = \text{UNION} (\text{set binds}) (\lambda(v,l). \{v\} \cup \text{vars } l) \cup \text{varsC } c' \\
| varsV (L \ l) = \text{vars } l \\
| varsV (R - v) = \{v\} \\
| varsV - = \{\}

**lemma** finite-vars[simp]: finite (\text{vars } l) and finite (\text{varsC } c) finite (\text{varsV } v)  \\
by (induct rule: vars-varsC-varsV.induct, auto)

**fun** label :: \text{lamba + call} \Rightarrow \text{label}  \\
**where** label (\text{Inl} (\text{Lambda } l - -)) = l \\
| label (\text{Inr} (\text{App } l - -)) = l \\
| label (\text{Inr} (\text{Let } l - -)) = l

**fun** labels :: \text{lamba} \Rightarrow \text{label set}  \\
**and** labelsC :: \text{call} \Rightarrow \text{label set}  \\
**and** labelsV :: \text{val} \Rightarrow \text{label set}  \\
**where** labels (\text{Lambda } l \ vs \ c) = \{l\} \cup \text{labelsC } c \\
| labelsC (\text{App } l \ a \ as) = \{l\} \cup \text{labelsV } a \cup \text{UNION} (\text{set } as) \text{labelsV} \\
| labelsC (\text{Let } l \ binds \ c') = \{l\} \cup \text{UNION} (\text{set binds}) (\lambda(v,l). \text{labels } l) \cup \text{labelsC } c' \\
| labelsV (L \ l) = \text{labels } l
lemma finite-labels[simp]: finite (labels l) and finite (labelsC c) finite (labelsV v)
by (induct rule: labels-labelsC-labelsV.induct, auto)

fun prims :: lambda ⇒ prim set
and primsC :: call ⇒ prim set
and primsV :: val ⇒ prim set
where prims (Lambda - vs c) = primsC c
| primsC (App - a as) = primsV a ∪ UNION (set as) primsV
| primsC (Let - binds c') = UNION (set binds) (λ(-,l). prims l) ∪ primsC c'
| primsV (L l) = prims l
| primsV (R l v) = {}
| primsV (P prim) = {prim}
| primsV (C l v) = {}

lemma finite-prims[simp]: finite (prims l) and finite (primsC c) finite (primsV v)
by (induct rule: labels-labelsC-labelsV.induct, auto)

fun vals :: lambda ⇒ val set
and valsC :: call ⇒ val set
and valsV :: val ⇒ val set
where vals (Lambda - vs c) = valsC c
| valsC (App - a as) = valsV a ∪ UNION (set as) valsV
| valsC (Let - binds c') = UNION (set binds) (λ(-,l). vals l) ∪ valsC c'
| valsV (L l) = {L l} ∪ vals l
| valsV (R l v) = {R l v}
| valsV (P prim) = {P prim}
| valsV (C l v) = {C l v}

lemma
fixes list2 :: (var × lambda) list and t :: var×lambda
shows lambdas1: Lambda l vs c ∈ lambdas x ⇒ c ∈ calls x
and Lambda l vs c ∈ lambdasC y ⇒ c ∈ callsC y
and Lambda l vs c ∈ lambdasV z ⇒ c ∈ callsV z
and ∀z∈ set list. Lambda l vs c ∈ lambdasV z ⇒ c ∈ callsV z
and ∀z∈ set list2. Lambda l vs c ∈ lambdas (snd x) ⇒ c ∈ calls (snd x)
and Lambda l vs c ∈ lambdas (snd t) ⇒ c ∈ calls (snd t)
apply (induct rule:lambda-call-val.inducts)
apply auto
apply (case-tac c, auto)[1]
apply (rule-tac x=((a, b), ba) in bexI, auto)
done

lemma
shows lambdas2: Lambda l vs c ∈ lambdas x ⇒ l ∈ labels x
and Lambda l vs c ∈ lambdasC y ⇒ l ∈ labelsC y
and Lambda l vs c ∈ lambdasV z ⇒ l ∈ labelsV z

52
apply auto
apply (erule-tac x=((a, b), ba) in bexI, auto)
done

lemma
shows lambdas3: \( \lambda a \in \text{set list}. \lambda \Gamma \in \text{lambdas} x \Rightarrow \text{set} \subseteq \text{vars} x \)
and \( \lambda \Gamma \in \text{lambdasC} y \Rightarrow \text{set} \subseteq \text{varsC} y \)
and \( \lambda \Gamma \in \text{lambdasV} z \Rightarrow \text{set} \subseteq \text{varsV} z \)
and \( \forall x \in \text{set list}. \lambda \Gamma \in \text{lambdasV} z \Rightarrow \text{set} \subseteq \text{varsV} z \)
and \( \forall x \in \text{set (list2 :: (var \times \text{lambda}) list)} . \lambda \Gamma \in \text{lambdas (snd x)} \Rightarrow \text{set} \subseteq \text{vars (snd x)} \)
and \( \lambda \Gamma \in \text{lambdas (snd (t:: var\times\text{lambda})) \Rightarrow set \subseteq \text{vars (snd t)} }\)
apply (induct \( x \) and \( y \) and \( z \) and \( \text{list} \) and \( \text{list2} \) and \( t \) rule:lambda-call-val.inducts)
apply auto
apply (erule-tac x=((aa, ba), bb) in ballE)
apply (rule-tac x=((aa, ba), bb) in bexI, auto)
done

lemma
shows app1: \( \text{App} \ l \ d \ ds \in \text{calls} x \Rightarrow \text{d} \in \text{vals} x \)
and \( \text{App} \ l \ d \ ds \in \text{callsC} y \Rightarrow \text{d} \in \text{valsC} y \)
and \( \text{App} \ l \ d \ ds \in \text{callsV} z \Rightarrow \text{d} \in \text{valsV} z \)
and \( \forall x \in \text{set list}. \text{App} \ l \ d \ ds \in \text{callsV} z \Rightarrow \text{d} \in \text{valsV} z \)
and \( \forall x \in \text{set (list2 :: (var \times \text{lambda}) list)} . \text{App} \ l \ d \ ds \in \text{calls (snd x)} \Rightarrow \text{d} \in \text{vals (snd x)} \)
and \( \text{App} \ l \ d \ ds \in \text{calls (snd (t:: var\times\text{lambda})) \Rightarrow set} \subseteq \text{vals (snd t)} \)
apply (induct \( x \) and \( y \) and \( z \) and \( \text{list} \) and \( \text{list2} \) and \( t \) rule:lambda-call-val.inducts)
apply auto
apply (case-tac d, auto)
apply (erule-tac x=((a, b), ba) in ballE)
apply (rule-tac x=((a, b), ba) in bexI, auto)
done

lemma
shows app2: \( \text{App} \ l \ d \ ds \in \text{calls} x \Rightarrow \text{set} \subseteq \text{vals} x \)
and \( \text{App} \ l \ d \ ds \in \text{callsC} y \Rightarrow \text{set} \subseteq \text{valsC} y \)
and \( \text{App} \ l \ d \ ds \in \text{callsV} z \Rightarrow \text{set} \subseteq \text{valsV} z \)
and \( \forall x \in \text{set list}. \text{App} \ l \ d \ ds \in \text{callsV} z \Rightarrow \text{set} \subseteq \text{valsV} z \)
and \( \forall x \in \text{set (list2 :: (var \times \text{lambda}) list)} . \text{App} \ l \ d \ ds \in \text{calls (snd x)} \Rightarrow \text{set} \subseteq \text{vals (snd x)} \)
and \( \text{App} \ l \ d \ ds \in \text{calls (snd (t:: var\times\text{lambda})) \Rightarrow set} \subseteq \text{vals (snd t)} \)
apply (induct \( x \) and \( y \) and \( z \) and \( \text{list} \) and \( \text{list2} \) and \( t \) rule:lambda-call-val.inducts)
apply auto
apply (case-tac x, auto)
apply (erule-tac x=((a, b), ba) in ballE)
apply (rule-tac x=((a, b), ba) in bexI, auto)
done

lemma
  shows let1: Let l binds c’ ∈ calls x ⇒ l ∈ labels x
  and Let l binds c’ ∈ callsC y ⇒ l ∈ labelsC y
  and Let l binds c’ ∈ callsV z ⇒ l ∈ labelsV z
  and ∀ z∈ set list. Let l binds c’ ∈ callsV z → l ∈ labelsV z
  and ∀ x∈ set (list2 :: (var × lambda) list) . Let l binds c’ ∈ calls (snd x) → l ∈ labels (snd x)
  and Let l binds c’ ∈ calls (snd (t:: var×lambda)) ⇒ l ∈ labels (snd t)
apply (induct x and y and z and list and list2 and t rule:lambda-call-val.inducts)
apply auto
apply (erule-tac x=((a, b), ba) in ballE)
apply (rule-tac x=((a, b), ba) in bexI, auto)
done

lemma
  shows let2: Let l binds c’ ∈ calls x ⇒ c’ ∈ calls x
  and Let l binds c’ ∈ callsC y ⇒ c’ ∈ callsC y
  and Let l binds c’ ∈ callsV z ⇒ c’ ∈ callsV z
  and ∀ z∈ set list. Let l binds c’ ∈ callsV z → c’ ∈ callsV z
  and ∀ x∈ set (list2 :: (var × lambda) list) . Let l binds c’ ∈ calls (snd x) → c’ ∈ calls (snd x)
  and Let l binds c’ ∈ calls (snd (t:: var×lambda)) ⇒ c’ ∈ calls (snd t)
apply (induct x and y and z and list and list2 and t rule:lambda-call-val.inducts)
apply auto
apply (case-tac c’, auto)
apply (erule-tac x=((a, b), ba) in ballE)
apply (rule-tac x=((a, b), ba) in bexI, auto)
done

lemma
  shows let3: Let l binds c’ ∈ calls x ⇒ fst ‘ set binds ⊆ vars x
  and Let l binds c’ ∈ callsC y ⇒ fst ‘ set binds ⊆ varsC y
  and Let l binds c’ ∈ callsV z ⇒ fst ‘ set binds ⊆ varsV z
  and ∀ z∈ set list. Let l binds c’ ∈ callsV z → fst ‘ set binds ⊆ varsV z
  and ∀ x∈ set (list2 :: (var × lambda) list) . Let l binds c’ ∈ calls (snd x) → fst ‘ set binds ⊆ vars (snd x)
  and Let l binds c’ ∈ calls (snd (t:: var×lambda)) ⇒ fst ‘ set binds ⊆ vars (snd t)
apply (induct x and y and z and list and list2 and t rule:lambda-call-val.inducts)
apply auto
apply (erule-tac x=((ab, bc), bd) in ballE)
apply (rule-tac x=((ab, bc), bd) in bexI, auto)
done

lemma
  shows let4: Let l binds c’ ∈ calls x ⇒ snd ‘ set binds ⊆ lambdas x
  and Let l binds c’ ∈ callsC y ⇒ snd ‘ set binds ⊆ lambdasC y
and Let $l$ binds $c' \in \text{calls}V z \implies \text{snd}' \set binds \subseteq \text{lam}d\text{as}V z$

and $\forall z \in \text{set list}. \ Let \ l \ binds \ c' \in \text{calls}V z \implies \text{snd}' \set binds \subseteq \text{lam}d\text{as}V z$

and $\forall x \in \text{set} \ (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) \ . \ Let \ l \ binds \ c' \in \text{calls} (\text{snd} x) \implies \text{snd}' \set binds \subseteq \text{lam}d\text{as} (\text{snd} x)$

and Let $l$ binds $c' \in \text{calls} (\text{snd} (t:: \text{var} \times \text{lambda})) \implies \text{snd}' \set binds \subseteq \text{lam}d\text{as} (\text{snd} t)$

apply (induct $x$ and $y$ and $z$ and list and list2 and $t$ rule:lambda-call-val.inducts)
apply auto
apply (rule-tac $x$=((a, b), ba) in bexI, auto)
apply (case-tac ba, auto)
apply (erule-tac $x$=((aa, bb), bc) in ballE)
apply (rule-tac $x$=((aa, bb), bc) in bexI, auto)
done

lemma
shows $\text{vals1}$: $P \text{ prim} \in \text{vals} p \implies \text{prim} \in \text{prims} p$

and $P \text{ prim} \in \text{vals}C y \implies \text{prim} \in \text{prims}C y$

and $P \text{ prim} \in \text{vals}V z \implies \text{prim} \in \text{prims}V z$

and $\forall z \in \text{set list}. \ P \text{ prim} \in \text{vals}V z \implies \text{prim} \in \text{prims}V z$

and $\forall x \in \text{set} \ (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) \ . \ P \text{ prim} \in \text{vals} (\text{snd} x) \implies \text{prim} \in \text{prims} (\text{snd} x)$

apply (induct rule:lambda-call-val.inducts)
apply auto
apply (erule-tac $x$=((a, b), ba) in ballE)
apply (rule-tac $x$=((a, b), ba) in bexI, auto)
done

lemma
shows $\text{vals2}$: $R \ l \ \text{var} \in \text{vals} p \implies \text{var} \in \text{vars} p$

and $R \ l \ \text{var} \in \text{vals}C y \implies \text{var} \in \text{vars}C y$

and $R \ l \ \text{var} \in \text{vals}V z \implies \text{var} \in \text{vars}V z$

and $\forall z \in \text{set list}. \ R \ l \ \text{var} \in \text{vals}V z \implies \text{var} \in \text{vars}V z$

and $\forall x \in \text{set} \ (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) \ . \ R \ l \ \text{var} \in \text{vals} (\text{snd} x) \implies \text{var} \in \text{vars} (\text{snd} x)$

and $R \ l \ \text{var} \in \text{vals} (\text{snd} (t:: \text{var} \times \text{lambda})) \implies \text{var} \in \text{vars} (\text{snd} t)$

apply (induct rule:lambda-call-val.inducts)
apply auto
apply (erule-tac $x$=((a, b), ba) in ballE)
apply (rule-tac $x$=((a, b), ba) in bexI, auto)
done

lemma
shows $\text{vals3}$: $L \ l \in \text{vals} p \implies l \in \text{lam}d\text{as} p$

and $L \ l \in \text{vals}C y \implies l \in \text{lam}d\text{as}C y$

and $L \ l \in \text{vals}V z \implies l \in \text{lam}d\text{as}V z$

and $\forall z \in \text{set list}. \ L \ l \in \text{vals}V z \implies l \in \text{lam}d\text{as}V z$

and $\forall x \in \text{set} \ (\text{list2} :: (\text{var} \times \text{lambda}) \text{ list}) \ . \ L \ l \in \text{vals} (\text{snd} x) \implies l \in \text{lam}d\text{as} (\text{snd} x)$

and $L \ l \in \text{vals} (\text{snd} (t:: \text{var} \times \text{lambda})) \implies l \in \text{lam}d\text{as} (\text{snd} t)$

apply (induct rule:lambda-call-val.inducts)
apply auto

55
apply (erule-tac x=((a, b), ba) in ballE)
apply (rule-tac x=((a, b), ba) in bexI, auto)
apply (case-tac l, auto)
done

definition nList :: 'a set => nat => 'a list set
where nList A n ≡ \{ l. set l ⊆ A ∧ length l = n\}

lemma finite-nList[intro]:
  assumes finA: finite A
  shows finite (nList A n)
proof (induct n)
case 0 thus ?case by (simp add: nList-def)
next
case (Suc n) hence finn: finite (nList (A) n) by simp
proof [rule subset-antisym[OF subsetI subsetI]]
  fix l assume l ∈ ?lhs thus l ∈ ?rhs
    by (cases l, auto simp add: nList-def)
next
  fix l assume l ∈ ?rhs thus l ∈ ?lhs
    by (auto simp add: nList-def)
qed
thus finite ?lhs using finA and finn
  by auto
qed

definition NList :: 'a set => nat set => 'a list set
where NList A N ≡ \bigcup n ∈ N. nList A n

lemma finite-Nlist[intro]:
  \[ \text{finite } A \wedge \text{finite } N \] \implies \text{finite } (NList A N)
unfolding NList-def using assms by auto

definition call-list-lengths
where call-list-lengths p = \{0,1,2,3\} \cup (λc. case c of (App - - ds) ⇒ length ds | - ⇒ 0) \'
calls p

lemma finite-call-list-lengths[simp]: finite (call-list-lengths p)
unfolding call-list-lengths-def by auto

end

10. General utility lemmas

theory Utils imports Main
begin


56
This is a potpourri of various lemmas not specific to our project. Some of them could very well be included in the default Isabelle library.

Lemmas about the \textit{single-valued} predicate.

\textbf{lemma \textit{single-valued-empty}:} [simp]\textit{: single-valued \{\}}
\begin{itemize}
  \item \textbf{by (rule single-valuedI)} \textit{auto}
\end{itemize}

\textbf{lemma \textit{single-valued-insert}:}
\begin{itemize}
  \item \textbf{assumes} single-valued rel
  \item \textbf{and} \(\bigwedge x y. [(x,y) \in \text{rel}; x=a] \implies y = b\)
  \item \textbf{shows} single-valued (insert \((a,b)\) rel)
  \item \textbf{using} \textit{assms}
  \item \textbf{by (auto intro:single-valuedI dest:single-valuedD)}
\end{itemize}

Lemmas about ran, the range of a finite map.

\textbf{lemma \textit{ran-upd}:} ran \((m \ (k \mapsto v)) \subseteq \text{ran} \ m \cup \{v\}\)
\begin{itemize}
  \item \textbf{unfolding} \textit{ran-def} \textit{by} \textit{auto}
\end{itemize}

\textbf{lemma \textit{ran-map-of}:} ran \((\map-of \ xs) \subseteq \text{snd \ ' set} \ xs\)
\begin{itemize}
  \item \textbf{by (induct xs)}\textbf{(auto simp del:fun-upd-apply dest: ran-upd[THEN subsetD])}
\end{itemize}

\textbf{lemma \textit{ran-concat}:} ran \((m1 ++ m2) \subseteq \text{ran} \ m1 \cup \text{ran} \ m2\)
\begin{itemize}
  \item \textbf{unfolding} \textit{ran-def}
  \item \textbf{by} \textit{auto}
\end{itemize}

\textbf{lemma \textit{ran-upsds}:}
\begin{itemize}
  \item \textbf{assumes} eq-length: \textit{length} \(\text{ks} = \text{length} \ vs\)
  \item \textbf{shows} ran \((\map-upsds \ m \ \text{ks} \ \text{vs}) \subseteq \text{ran} \ m \cup \text{set} \ \text{vs}\)
  \item \textbf{proof–}
  \item \textbf{have} ran \((\map-upsds \ m \ \text{ks} \ \text{vs}) \subseteq \text{ran} \ ((m ++ \map-of \ (\rev \ (\text{zip} \ \text{ks} \ \text{vs}))))\)
  \item \textbf{unfolding} \textit{map-upsds-def} \textbf{by} \textit{simp}
  \item \textbf{also have} \ldots \subseteq \text{ran} \ m \cup \text{ran} \ ((\map-of \ (\rev \ (\text{zip} \ \text{ks} \ \text{vs}))))\textbf{by (rule ran-concat)}
  \item \textbf{also have} \ldots \subseteq \text{ran} \ m \cup \text{snd \ ' set} \ (\rev \ (\text{zip} \ \text{ks} \ \text{vs}))
  \item \textbf{by (intro Un-mono[of ran m ran m] subset-refl ran-map-of)}
  \item \textbf{also have} \ldots \subseteq \text{ran} \ m \cup \text{set} \ \text{vs}
  \item \textbf{by (auto intro:Un-mono[of ran m ran m] subset-refl simp del: set-map simp add: set-map[THEN subsetI]}\textbf{sym[ map-snd-zip[OF eq-length]]}
  \item \textbf{finally show} \textit{thesis .}
\end{itemize}
\textit{qed}

\textbf{lemma \textit{ran-upd-mem}:} \textit{v} \in \text{ran} \ ((m \ (k \mapsto v)))
\begin{itemize}
  \item \textbf{unfolding} \textit{ran-def} \textbf{by} \textit{auto}
\end{itemize}

Lemmas about \textit{map, zip} and \textit{fst/snd}

\textbf{lemma \textit{map-fst-zip}:} length \(\text{xs} = \text{length} \ \text{ys} \implies \text{map} \ \text{fst} \ (\text{zip} \ \text{xs} \ \text{ys}) = \text{xs}\)
\begin{itemize}
  \item \textbf{apply (induct \text{xs} \ \text{ys} \ \textbf{rule: list-induct2}) by} \textit{auto}
\end{itemize}
lemma map-snd-zip: length xs = length ys ⇒ map snd (zip xs ys) = ys
apply (induct xs ys rule:list-induct2) by auto
end

11. Set-valued maps

theory SetMap
  imports Main
begin

For the abstract semantics, we need methods to work with set-valued maps, i.e. functions from a key type to sets of values. For this type, some well known operations are introduced and properties shown, either borrowing the nomenclature from finite maps (sdom, sran,...) or of sets (\{\}, \cup,...).

definition sdom :: ('a => 'b set) => 'a set where
  sdom m = {a. m a = {}}
definition sran :: ('a => 'b set) => 'b set where
  sran m = {b. EX a. b \in m a}

lemma sranI: b \in m a ⇒ b \in sran m by (auto simp: sran-def)
lemma sdom-not-mem[elim]: a /\notin sdom m ⇒ m a = {} by (auto simp: sdom-def)
definition smap-empty :: (\{\}.
  where \{\}. k = {}
definition smap-union :: ('a::type => 'b::type set) => ('a => 'b set) => ('a => 'b set) (- \cup -.)
  where smap1 \cup. smap2 k = smap1 k \cup. smap2 k
primrec smap-Union :: ('a::type => 'b::type set) list => 'a => 'b set (\cup.-)
  where [simp]:\cup. [] = {}
  | \cup. (m\#ms) = m . \cup. \cup. ms

definition smap-singleton :: 'a::type => 'b::type set => 'a => 'b set (\{ . := -.}
  where {k := vs}. = {}. (k := vs)
definition smap-less :: ('a => 'b set) => ('a => 'b set) => bool (-/ \subseteq. -. [50, 51] 50)
  where smap-less m1 m2 = (\forall k. m1 k \subseteq m2 k)

lemma sdom-empty[simp]: sdom \{\}. = {}
unfolding \textit{sdom-def \ smap-empty-def} by auto

\textbf{lemma} \textit{sdom-singleton[simp]}: \textit{sdom \{k := vs\} \subseteq \{k\}} \\
by (auto simp add: sdom-def smap-singleton-def smap-empty-def)

\textbf{lemma} \textit{sran-singleton[simp]}: \textit{sran \{k := vs\} = vs} \\
by (auto simp add: sran-def smap-singleton-def smap-empty-def)

\textbf{lemma} \textit{sran-empty[simp]}: \textit{sran \{\} = \{\}} \\
unfolding sran-def smap-empty-def by auto

\textbf{lemma} \textit{sdom-union[simp]}: \textit{sdom (m \cup n) = sdom m \cup sdom n} \\
by (auto simp add: smap-union-def sdom-def)

\textbf{lemma} \textit{sran-union[simp]}: \textit{sran (m \cup n) = sran m \cup sran n} \\
by (auto simp add: smap-union-def sran-def)

\textbf{lemma} \textit{smap-empty[simp]}: \textit{\{} \subseteq \{\} \\
unfolding smap-less-def by auto

\textbf{lemma} \textit{smap-less-refl}: \textit{m \subseteq m} \\
unfolding smap-less-def by simp

\textbf{lemma} \textit{smap-less-trans[trans]}: \{ m1 \subseteq m2; m2 \subseteq m3 \} \implies m1 \subseteq m3 \\
unfolding smap-less-def by auto

\textbf{lemma} \textit{smap-union-mono}: \{ ve1 \subseteq ve1'; ve2 \subseteq ve2' \} \implies ve1 \cup ve2 \subseteq ve1' \cup ve2' \\
by (auto simp add: smap-less-def smap-union-def)

\textbf{lemma} \textit{smap-Union-union}: \textit{m1 \cup \bigcup ms = \bigcup \{m1 \# ms\}} \\
by (rule ext, auto simp add: smap-union-def smap-Union-def)

\textbf{lemma} \textit{smap-singleton-mono}: \textit{v \subseteq v' \implies \{k := v\} \subseteq \{k := v'\}}. \\
by (auto simp add: smap-singleton-def)

\textbf{lemma} \textit{smap-union-comm}: \textit{m1 \cup m2 = m2 \cup m1} \\
by (rule ext, auto simp add: smap-union-def)

\textbf{lemma} \textit{smap-union-empty1[simp]}: \textit{\{} \cup m = m} \\
by (rule ext, auto simp add: smap-union-def)

\textbf{lemma} \textit{smap-union-empty2[simp]}: \textit{m \cup \{\} = m} \\
by auto
by (rule ext, auto simp add: smap-union-def smap-empty-def)

lemma smap-union-assoc [simp]: \( (m1 \cup m2) \cup m3 = m1 \cup (m2 \cup m3) \)
by (rule ext, auto simp add: smap-union-def)

lemma smap-Union-append [simp]: \( \bigcup (m1 @ m2) = (\bigcup m1) \cup (\bigcup m2) \)
by (induct m1) auto

lemma smap-Union-rev [simp]: \( \bigcup (\text{rev} l) = \bigcup (\bigcup l) \)
by (subst rev-map [THEN sym], subst smap-Union-rev, rule refl)

end

12. Sets of maps

theory MapSets
imports SetMap Utils
begin

In the section about the finiteness of the argument space, we need the fact that the set of maps from a finite domain to a finite range is finite, and the same for the set-valued maps defined in SetMap. Both these sets are defined (maps-over, smaps-over) and the finiteness is shown.

definition maps-over :: `'a::type set \Rightarrow `'b::type set \Rightarrow ('a \rightarrow 'b) set
  where maps-over A B = \{ m. dom m \subseteq A \land ran m \subseteq B \}\n
lemma maps-over-empty [simp]:
  empty \in maps-over A B

unfolding maps-over-def by simp

lemma maps-over-upd:
  assumes m \in maps-over A B
  and v \in A and k \in B
  shows m(v \mapsto k) \in maps-over A B
  using assms unfolding maps-over-def
  by (auto dest: subsetD[OF ran-upd])

lemma maps-over-finite [intro]:
  assumes finite A and finite B shows finite (maps-over A B)
proof
  have inj-map-graph: inj (\lambda f. \{(x, y). Some y = f x\})
  proof (induct rule: inj-onI)
    case (1 x y)
    from 1.hyps(3) have hyp: \( \forall a b. (Some b = x a) \leftrightarrow (Some b = y a) \)
  end

end
by (simp add: set-eq-iff)
show \textit{\textbf{fcase}}
proof (rule ext)
fix \( z \) show \( x \ z = y \ z \)
  using hyp[of \( - z \)]
  by (cases \( x \ z \), cases \( y \ z \), auto)
qed
qed

have (\( \lambda x. \{ x, y \} \) \ittel{\textit{\textbf{maps-over}} \ A \ B \subseteq \textit{\textbf{Pow}}(A \times B)} (is ?graph \subseteq -)
unfolding \ittel{\textit{\textbf{maps-over-def}}}
by (auto dest!:subsetD[of \ A subsetD[of \ B] intro:ranI])
moreover
have finite (\ittel{\textit{\textbf{Pow}}(A \times B)}) using \ittel{\textit{assms}} by auto
ultimately
have finite ?graph by (rule finite-subset)
thus ?thesis
by (rule finite-imageD[OF - subset-inj-on[OF inj-map-graph subset-UNIV]])
qed

definition \ittel{\textit{\textbf{smaps-over}} :: 'a::type set \ittel{\textit{\textbf{⇒}}} 'b::type set \ittel{\textit{\textbf{⇒}}} ('a \ittel{\textit{\textbf{⇒}}} 'b set) set}
where \ittel{\textit{\textbf{smaps-over \ A \ B} = \{ m. sdom m \subseteq A \wedge sran m \subseteq B \}}}

lemma \ittel{\textit{\textbf{smaps-over-empty}}[simp]}:
{} \ittel{\textit{\textbf{∈}}} \ittel{\textit{\textbf{smaps-over \ A \ B}}}
unfolding \ittel{\textit{\textbf{smaps-over-def}}} by simp

lemma \ittel{\textit{\textbf{smaps-over-singleton}}:}
  assumes \( k \in A \) and \ittel{\textit{\textbf{vs \subseteq B}}}
  shows \( \{ k := \textit{\textbf{vs}} \} \ittel{\textit{\textbf{∈}}} \ittel{\textit{\textbf{smaps-over \ A \ B}}} \)
  using \ittel{\textit{\textbf{assms}}} unfolding \ittel{\textit{\textbf{smaps-over-def}}}
  by(auto dest: subsetD[of sdom-singleton])

lemma \ittel{\textit{\textbf{smaps-over-un}}}:
  assumes \( m1 \in \ittel{\textit{\textbf{smaps-over \ A \ B}}} \) and \ittel{\textit{\textbf{m2 \in smaps-over \ A \ B}}} 
  shows \( m1 \cup . m2 \in \ittel{\textit{\textbf{smaps-over \ A \ B}}} \)
  using \ittel{\textit{\textbf{assms}}} unfolding \ittel{\textit{\textbf{smaps-over-def}}}
  by (auto simp add:smap-union-def)

lemma \ittel{\textit{\textbf{smaps-over-Union}}:}
  assumes \( \textit{\textbf{set \ ms \subseteq smaps-over \ A \ B}} \)
  shows \ittel{\textit{\textbf{\bigcup .ms \in smaps-over \ A \ B}}} 
  using \ittel{\textit{\textbf{assms}}}
  by (induct \ittel{\textit{\textbf{ms}}})(auto intro: smaps-over-un)

lemma \ittel{\textit{\textbf{smaps-over-im}}}:
\ittel{[ f \in m a ; m \in smaps-over \ A \ B ]} \ittel{\implies f \in B}
unfolding \ittel{\textit{\textbf{smaps-over-def}}} by (auto simp add:sran-def)
lemma smaps-over-finite[intro]:
assumes finite A and finite B shows finite (smaps-over A B)
proof
  have inj-smap-graph: inj (λf. {(x, y). y = f x ∧ y ≠ {}}) (is inj ?gr)
  proof (induct rule: inj-onI)
    case (1 x y)
    from 1.hyps(3) have hyp: a b. (b = x a ∧ b ≠ {}) = (b = y a ∧ b ≠ {})
      by (subst (asm) (3) set-eq-iff, simp)
    show ?case
      proof (rule ext)
        fix z
        show x z = y z
          using hyp[of - z]
          by (cases x z ≠ {}, cases y z ≠ {}, auto)
      qed
  qed

have ?gr ⊆ smaps-over A B ⊆ Pow( A × Pow B ) (is ?graph ⊆ -)
unfolding smaps-over-def
moreover
have finite (Pow( A × Pow B )) using assms by auto
ultimately
have finite ?graph by (rule finite-subset)
thus ?thesis
  by (rule finite-imageD[OF - subset-inj-on[OF inj-smap-graph subset-UNIV]])
qed

end

13. HOLCF Utility lemmas

theory HOLCFUtils
imports HOLCF
begin

We use HOLCF to define the denotational semantics. By default, HOLCF does not turn the regular set type into a partial order, so this is done here. Some of the lemmas here are contributed by Brian Huffman.

We start by making the type bool a pointed chain-complete partial order.

instantiation bool :: po
begin
definition x ⊑ y ←→ (x → y)
instance by (default, unfold below-bool-def, fast+)
end
instance bool :: chfin
apply default
apply (drule finite-range-imp-finch)
apply (rule finite)
apply (simp add: finite-chain-def)
done

instance bool :: pcpo
proof
  have \forall y. False \sqsubseteq y by (simp add: below-bool-def)
  thus \exists x::bool. \forall y. x \sqsubseteq y ..
qed

lemma is-lub-bool: S <<| (True \in S)
  unfolding is-lub-def is-ub-def below-bool-def by auto

lemma lub-bool: lub S = (True \in S)
  using is-lub-bool by (rule lub-eqI)

lemma bottom-eq-False[simp]: \bot = False
by (rule below-antisym [OF minimal], simp add: below-bool-def)

To convert between the squared syntax used by HOLCF and the regular, round syntax
for sets, we state some of the equivalencies.

instantiation set :: (type) po
begin
definition
  A \sqsubseteq B \iff A \subseteq B
instance by (default, unfold below-set-def, fast+)
end

lemma sqsubset-is-subset: A \sqsubseteq B \iff A \subseteq B
  by (fact below-set-def)

lemma is-lub-set: S <<| \bigcup S
  unfolding is-lub-def is-ub-def below-set-def by fast

lemma lub-is-union: lub S = \bigcup S
  using is-lub-set by (rule lub-eqI)

instance set :: (type) cpo
  by (default, fast intro: is-lub-set)

lemma emptyset-is-bot[simp]: \{\} \sqsubseteq S
  by (simp add: sqsubset-is-subset)

instance set :: (type) pcpo
  by (default, fast intro: emptyset-is-bot)
lemma bot-bool-is-emptyset [simp]: \bot = \{\}

using emptyset-is-bot by (rule bottomI [symmetric])

To actually use these instance in fixrec definitions or fixed-point inductions, we need continuity requrements for various boolean and set operations.

lemma cont2cont-disj [simp, cont2cont]:
  assumes f: cont (λx. f x) and g: cont (λx. g x)
  shows cont (λx. f x ∨ g x)
  apply (rule cont-apply [OF f])
  apply (rule chfindom-monofun2cont)
  apply (rule monofunI, simp add: below-bool-def)
  apply (rule cont-compose [OF - g])
  apply (rule chfindom-monofun2cont)
  apply (rule monofunI, simp add: below-bool-def)
  done

lemma cont2cont-imp [simp, cont2cont]:
  assumes f: cont (λx. ¬ f x) and g: cont (λx. g x)
  shows cont (λx. f x −→ g x)
  unfolding imp-conv-disj by (rule cont2cont-disj [OF f g])

lemma cont2cont-Collect [simp, cont2cont]:
  assumes ⋀y. cont (λx. f x y)
  shows cont (λx. \{y. f x y\})
  apply (rule contI)
  apply (subst cont2contlubE [OF assms], assumption)
  apply (auto simp add: is-lub-def is-ub-def below-set-def lub-bool)
  done

lemma cont2cont-mem [simp, cont2cont]:
  assumes cont (λx. f x)
  shows cont (λx. y ∈ f x)
  apply (rule cont-compose [OF - assms])
  apply (rule contI)
  apply (auto simp add: is-lub-def is-ub-def below-set-def lub-is-union)
  done

lemma cont2cont-union [simp, cont2cont]:
  cont (λx. f x) \imp cont (λx. g x)
  \imp cont (λx. f x ∪ g x)
  unfolding Un-def by simp

lemma cont2cont-insert [simp, cont2cont]:
  assumes cont (λx. f x)
  shows cont (λx. insert y (f x))
  unfolding insert-def using assms
  by (intro cont2cont)
monofunE

As with the continuity lemmas, we need admissibility lemmas.
assumes cont (λx. f x)
shows adm (λx. y /∈ f x)
using assms
apply (erule-tac t = f in adm-subst)
proof (rule admI)
fix Y :: nat ⇒ 'b set
assume chain: chain Y
assume ∀ i. y /∈ Y i hence (∪ i. y ∈ Y i) = False 
  by auto
thus y /∈ (∪ i. Y i)
using chain unfolding lub-bool lub-is-union by auto
qed

lemma adm-id[simp]: adm (λx . x)
by (rule adm-chfin)

lemma adm-Not[simp]: adm Not
by (rule adm-chfin)

lemma adm-prod-split:
  assumes adm (λp. f (fst p) (snd p))
  shows adm (λ(x,y). f x y)
  using assms unfolding split-def .

lemma adm-ball':
  assumes (∀ y. adm (λx. y ∈ A x −→ P x y))
  shows adm (λx. ∀ y ∈ A x . P x y)
by (subst Ball-def, rule adm-all[OF assms])

lemma adm-not-conj:
  [[adm (λx. ¬ P x); adm (λx. ¬ Q x)] ⇒ adm (λx. ¬ (P x ∧ Q x))]
by simp

lemma adm-single-valued:
  assumes cont (λx. f x)
  shows adm (λx. single-valued (f x))
  using assms unfolding single-valued-def
by (intro adm-lemmas adm-not-mem cont2cont adm-subst[of f])

To match Shivers’ syntax we introduce the power-syntax for iterated function application.

abbreviation niceiterate ((·^-) [1000] 1000)
  where niceiterate f i ≡ iterate i f

end
14. Fixed point transformations

theory FixTransform
imports HOLCF
begin

default-sort type

In his treatment of the computability, Shivers gives proofs only for a generic example and leaves it to the reader to apply this to the mutually recursive functions used for the semantics. As we carry this out, we need to transform a fixed point for two functions (implemented in HOLCF as a fixed point over a tuple) to a simple fixed point equation. The approach here works as long as both functions in the tuple have the same return type, using the equation

\[ X^A \cdot X^B = X^{A+B}. \]

Generally, a fixed point can be transformed using any retractable continuous function:

lemma fix-transform:
  assumes \( \forall x. \ g \cdot (f \cdot x) = x \)
  shows \( \text{fix} \cdot F = g \cdot (\text{fix} \cdot (f \circ F \circ g)) \)
using assms apply --
apply (rule parallel-fix-ind)
apply (rule adm-eq)
apply auto
apply (erule retraction-strict[of g f, rule-format])
done

The functions we use here convert a tuple of functions to a function taking a direct sum as parameters and back. We only care about discrete arguments here.

definition tup-to-sum :: \((a \ \text{discr} \rightarrow c) \times (b \ \text{discr} \rightarrow c) \rightarrow (a + b) \ \text{discr} \rightarrow c::cpo\)
  where tup-to-sum = (\(\lambda p \ s. \ (\lambda f,g).\)
      \case \text{undiscr }s \ of \ \text{Inl }x \Rightarrow f(\text{Discr }x)
        | \ \text{Inr }x \Rightarrow g(\text{Discr }x)) \ p)

definition sum-to-tup :: \((a + b) \ \text{discr} \rightarrow c \rightarrow (a \ \text{discr} \rightarrow c) \times (b \ \text{discr} \rightarrow c::cpo)\)
  where sum-to-tup = (\(\lambda f. \ (\lambda x. f(\text{Discr } (\text{Inl } \text{undiscr }x))),\)
      \(\lambda x. f(\text{Discr } (\text{Inr } \text{undiscr }x))))\)

As so often when working with HOLCF, some continuity lemmas are required.

lemma cont2cont-case-sum[simp,cont2cont]:
  assumes \(\text{cont } f \ \text{and } \text{cont } g\)
  shows \(\text{cont } (\lambda x. \ \text{case-sum } (f \ x) (g \ x) \ s)\)
using assms
by (cases s, auto intro:cont2cont-fun)
lemma cont2cont-circ[simp,cont2cont]:
  cont (λf. f ◦ g)
apply (rule cont2cont-lambda)
apply (subst comp-def)
apply (rule cont2cont-fun[of λx. x, OF cont-id])
done

lemma cont2cont-split-pair[cont2cont,simp]:
assumes f1: cont f
  and f2: λ x. cont (f x)
  and g1: cont g
  and g2: λ x. cont (g x)
shows cont (λ(a, b). (f a b, g a b))
apply (intro cont2cont)
apply (rule cont-apply[OF cont-snd - cont-const])
apply (rule cont-apply[OF cont-snd f2])
apply (rule cont-apply[OF cont-fst cont2cont-fun[OF f1] cont-const])
apply (rule cont-apply[OF cont-snd - cont-const])
apply (rule cont-apply[OF cont-snd g2])
apply (rule cont-apply[OF cont-fst cont2cont-fun[OF g1] cont-const])
done

Using these continuity lemmas, we can show that our function are actually continuous
and thus allow us to apply them to a value.

lemma sum-to-tup-app:
  sum-to-tup f = (λ x. f · (Discr (Inl (undiscr x))), λ x. f · (Discr (Inr (undiscr x))))
unfolding sum-to-tup-def by simp

lemma tup-to-sum-app:
  tup-to-sum p = (λ s. (λ(f,g).
          case undiscr s of Inl x ⇒ f · (Discr x)
                                | Inr x ⇒ g · (Discr x)) p)
unfolding tup-to-sum-def by simp

Generally, lambda abstractions with discrete domain are continous and can be resolved
immediately.

lemma discr-app[simp]:
  (λ s. f s) · (Discr x) = f · (Discr x)
by simp

Our transformation functions are inverse to each other, so we can use them to transform
a fixed point.

lemma tup-to-sum-to-tup[simp]:
  shows sum-to-tup · (tup-to-sum F) = F
unfolding \textit{sum-to-tup-app} and \textit{tup-to-sum-app} \\
by (cases \( F \), auto intro:cfun-eqI)

\textbf{lemma} \textit{fix-transform-pair-sum:} \\
\textbf{shows} \( \text{fix } F = \text{sum-to-tup}(\text{fix}(\text{tup-to-sum oo } F \text{ oo sum-to-tup})) \)
\\
by (rule fix-transform[\( \text{OF } \text{tup-to-sum-to-tup} \])

After such a transformation, we want to get rid of these helper functions again. This is 
done by the next two simplification lemmas.

\textbf{lemma} \textit{tup-sum-oo[simp]}
\\
\textbf{assumes} \( f1 \): \text{cont } F \\
\textbf{and} \( f2 \): \( \forall x.\ \text{cont } (F x) \)
\\
\textbf{and} \( g1 \): \text{cont } G \\
\textbf{and} \( g2 \): \( \forall x.\ \text{cont } (G x) \)
\\
\textbf{shows} \( \text{tup-to-sum oo } (\Lambda p.\ (\lambda(a, b).\ (F a b, G a b)) \text{ p}) \text{ oo sum-to-tup} = (\Lambda f s.\ (\text{case } \text{undiscr } s \text{ of } \text{Inl } x \Rightarrow F (\Lambda s.\ f \text{ (Discr (Inl (undiscr s))))})\)
\\
| \text{Inr } x \Rightarrow G (\Lambda s.\ f \text{ (Discr (Inl (undiscr s))))}
\\
(\Lambda s.\ f \text{ (Discr (Inr (undiscr s))))} \\
(\text{Discr } x))
\\
by (rule cfun-eqI, rule cfun-eqI, \\
\text{simp add: sum-to-tup-app tup-to-sum-app} \\
\text{cont2cont-split-pair[\( \text{OF } f1 f2 g1 g2 \])} \\
\text{cont2cont-lambda} \\
\text{cont-apply[\( \text{OF - } f2 \text{ cont2cont-fun[\( \text{OF cont-compose[\( \text{OF f1}]})}]} \\
\text{cont-apply[\( \text{OF - } g2 \text{ cont2cont-fun[\( \text{OF cont-compose[\( \text{OF g1}]})}]})]

\textbf{lemma} \textit{fst-sum-to-tup[simp]} \\
\text{fst } (\text{sum-to-tup } x) = (\Lambda xa.\ x \text{ (Discr (Inl (undiscr xa))))}
\\
by (simp add: sum-to-tup-app)
\end{it}

\begin{itemize}
\item \textbf{References}
\end{itemize}