Shivers' Control Flow Analysis

Joachim Breitner

March 19, 2025

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

Part I. The definitions

1. Syntax

theory *CPSScheme*
  imports *Main*
begin

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.

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.

datatype *prim* = *Plus label* | *If label label*
datatype *lambda* = *Lambda label var list call*
  and *call* = *App label val val list*
    | *Let label* (*var* × *lambda*) *list call*
  and *val* = *L lambda* | *R label var* | *C label int* | *P prim*

datatype-compat *lambda call val*

type-synonym *prog* = *lambda*

lemmas *mutual-lambda-call-var-inducts* =
compat-lambda.induct
compat-call.induct
compat-val.induct
compat-val-list.induct
compat-nat-char-list-prod-lambda-prod-list.induct
compat-nat-char-list-prod-lambda-prod.induct

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

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 2* [((*2*,''*rec*''),(*Lambda 3* [(*3*,''*p*''), (*3*,''*i*''), (*3*,''*c-*'')] (*App 4* (*P* (*If 5 6*)) [(*R 7* (*3*,''*i*'')), (*L* (*Lambda 8* [] (*App 9* (*P* (*Plus 10*)) [(*R 11* (*3*,''*p*'')), (*R 12* (*3*,''*i*'')), (*L* (*Lambda 13* [(*13*,''*p-*'')] (*App 14* (*P* (*Plus 15*)) [(*R 16* (*3*,''*i*'')), (*C 17* (− *1*)), (*L* (*Lambda 18* [(*18*,''*i-*'')] (*App 19* (*R 20* (*2*,''*rec*'')) [(*R 21* (*13*,''*p-*'')), (*R*

22 (18,''i-'')), (R 23 (3,''c-''))])))]))))]))), (L (Lambda 24 [] (App 25 (R 26 (3,''c-'')) [(R 27 (3,''p''))])))]))))] (App 28 (R 29 (2,''rec'')) [(C 30 0), (C 31 10), (R 32 (1,''cont''))]))))

end

2. Standard semantics

theory *Eval*
imports *HOLCF HOLCFUtils CPSScheme*
begin

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.

type-synonym *contour* = *nat*
type-synonym *benv* = *label* $\rightharpoonup$ *contour*
type-synonym *closure* = *lambda* $\times$ *benv*

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.

datatype *d* = *DI int*
| *DC closure*
| *DP prim*
| *Stop*

type-synonym *venv* = *var* $\times$ *contour* $\rightharpoonup$ *d*

The function $\mathcal{A}$ 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 β, 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.

fun *evalV* :: *val* ⇒ *benv* ⇒ *venv* ⇒ *d* (‹$\mathcal{A}$›)
  where $\mathcal{A}$ (*C* - *i*) β *ve* = *DI i*
  | $\mathcal{A}$ (*P prim*) β *ve* = *DP prim*
  | $\mathcal{A}$ (*R* - *var*) β *ve* =
      (*case* β (*binder var*) *of*
        Some l ⇒ (*case ve* (*var*,*l*) *of Some d* ⇒ *d*))
  | $\mathcal{A}$ (*L lam*) β *ve* = *DC* (*lam*, β)

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 $\perp$.

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 $\bigwedge$*a b c*. *cont* (λ*x*. *f x a b c*)
  shows *cont* (λ*x*. *case-lambda* (*f x*) *l*)
⟨*proof*⟩

lemma *cont2cont-case-d* [*simp*, *cont2cont*]:
  assumes $\bigwedge$*y*. *cont* (λ*x*. *f1 x y*)
    and $\bigwedge$*y*. *cont* (λ*x*. *f2 x y*)
    and $\bigwedge$*y*. *cont* (λ*x*. *f3 x y*)
    and *cont* (λ*x*. *f4 x*)
  shows *cont* (λ*x*. *case-d* (*f1 x*) (*f2 x*) (*f3 x*) (*f4 x*) *d*)
⟨*proof*⟩

lemma *cont2cont-case-call* [*simp*, *cont2cont*]:
  assumes $\bigwedge$*a b c*. *cont* (λ*x*. *f1 x a b c*)
    and $\bigwedge$*a b c*. *cont* (λ*x*. *f2 x a b c*)
  shows *cont* (λ*x*. *case-call* (*f1 x*) (*f2 x*) *c*)
⟨*proof*⟩

lemma *cont2cont-case-prim* [*simp*, *cont2cont*]:
  assumes $\bigwedge$*y*. *cont* (λ*x*. *f1 x y*)
    and $\bigwedge$*y z*. *cont* (λ*x*. *f2 x y z*)
  shows *cont* (λ*x*. *case-prim* (*f1 x*) (*f2 x*) *p*)
⟨*proof*⟩

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

Note how the contour counter is incremented before each call to $\mathcal{F}$ 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 proofs that this functional is continuous and thus a least fixed point exists. This is then used to define $\mathcal{F}$ and $\mathcal{C}$ and proof the equations given here. To use the *HOLCF* setup, the continuous function arrow $\rightarrow$ 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* × *d list* × *venv* × *contour*)
type-synonym *cstate* = (*call* × *benv* × *venv* × *contour*)

```
fixrec  evalF :: fstate discr → ans (‹𝓕›)
    and evalC :: cstate discr → ans (‹𝓒›)
  where evalF·fstate = (case undiscr fstate of
            (DC (Lambda lab vs c, β), as, ve, b) ⇒
              (if length vs = length as
               then let β' = β (lab ↦ b);
                        ve' = map-upds ve (map (λv.(v,b)) vs) as
                    in 𝓒·(Discr (c,β',ve',b))
               else ⊥)
          | (DP (Plus c),[DI a1, DI a2, cnt],ve,b) ⇒
                    let b' = Suc b;
                        β  = [c ↦ b]
                    in 𝓕·(Discr (cnt,[DI (a1 + a2)],ve,b'))
          | (DP (prim.If ct cf),[DI v, contt, contf],ve,b) ⇒
                (if v ≠ 0
                 then let b' = Suc b;
                          β = [ct ↦ b]
                      in 𝓕·(Discr (contt,[],ve,b'))
                 else let b' = Suc b;
                          β = [cf ↦ b]
                      in 𝓕·(Discr (contf,[],ve,b')))
          | (Stop,[DI i],-,-) ⇒ Def i
          | - ⇒ ⊥
      )
    | 𝓒·cstate = (case undiscr cstate of
            (App lab f vs,β,ve,b) ⇒
```

$$
\begin{array}{l}
\qquad\quad \textit{let } f' = \mathcal{A}\ f\ \beta\ ve;\\
\qquad\qquad as = map\ (\lambda v.\ \mathcal{A}\ v\ \beta\ ve)\ vs;\\
\qquad\qquad b' = Suc\ b\\
\qquad\quad\ \textit{in } \mathcal{F}\cdot(Discr\ (f',as,ve,b'))\\
\quad \mid (Let\ lab\ ls\ c',\beta,ve,b) \Rightarrow\\
\qquad\quad \textit{let } b' = Suc\ b;\\
\qquad\qquad \beta' = \beta\ (lab \mapsto b');\\
\qquad\qquad ve' = ve\ {+}{+}\ map\text{-}of\ (map\ (\lambda(v,l).\ ((v,b'),\ \mathcal{A}\ (L\ l)\ \beta'\ ve))\ ls)\\
\qquad\quad \textit{in } \mathcal{C}\cdot(Discr\ (c',\beta',ve',b'))\\
)
\end{array}
$$

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.

definition *evalCPS* :: *prog* $\Rightarrow$ *ans* (‹$\mathcal{PR}$›)
where $\mathcal{PR}\ l = (\textit{let } ve = Map.empty;$
$\qquad \beta = Map.empty;$
$\qquad f = \mathcal{A}\ (L\ l)\ \beta\ ve$
$\quad \textit{in } \mathcal{F}\cdot(Discr\ (f,[Stop],ve,0)))$

lemma *correct-ex1*: $\mathcal{PR}\ ex1 = Def\ 0$
⟨*proof*⟩

lemma *correct-ex2*: $\mathcal{PR}\ ex2 = Def\ 2$
⟨*proof*⟩

end

3. Exact nonstandard semantics

theory *ExCF*
imports *HOLCF HOLCFUtils CPSScheme Utils*
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 *contour* = (*UNIV*::*label list set*)

typedef *contour* = *contour*
⟨*proof*⟩

definition *initial-contour* (‹b_0›)
where b_0 = *Abs-contour* []

definition *nb*
where *nb b c* = *Abs-contour* (*c* # *Rep-contour b*)

instantiation *contour* :: *preorder*
begin
definition *le-contour-def*: $b \leq b' \longleftrightarrow$ *length* (*Rep-contour b*) $\leq$ *length* (*Rep-contour b'*)
definition *less-contour-def*: $b < b' \longleftrightarrow$ *length* (*Rep-contour b*) $<$ *length* (*Rep-contour b'*)
instance ⟨*proof*⟩
end

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

lemma *nb-le-less*[*iff*]: *nb b c* $\leq b' \longleftrightarrow b < b'$
⟨*proof*⟩

lemma *nb-less*[*iff*]: $b' <$ *nb b c* $\longleftrightarrow b' \leq b$
⟨*proof*⟩

declare *less-imp-le*[**where** *'a* = *contour*, *intro*]

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

type-synonym *benv* = *label* ⇀ *contour*
type-synonym *closure* = *lambda* × *benv*

datatype *d* = *DI int*
| *DC closure*
| *DP prim*
| *Stop*

type-synonym *venv* = *var* × *contour* ⇀ *d*

As we do not use the type system to distinguish procedural from non-procedural values, we define a predicate for that.

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:

instantiation *contour* :: *discrete-cpo*
begin
definition [*simp*]: $(x::contour) \sqsubseteq y \longleftrightarrow x = y$
instance ⟨*proof*⟩
end
instantiation *d* :: *discrete-cpo* **begin**
definition [*simp*]: $(x::d) \sqsubseteq y \longleftrightarrow x = y$
instance ⟨*proof*⟩
end

instantiation *call* :: *discrete-cpo* **begin**
definition [*simp*]: $(x::call) \sqsubseteq y \longleftrightarrow x = y$
instance ⟨*proof*⟩
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.

fun *evalV* :: *val* ⇒ *benv* ⇒ *venv* ⇒ *d* (‹$\mathcal{A}$›)
  where $\mathcal{A}$ (*C* - *i*) *β* *ve* = *DI i*
  | $\mathcal{A}$ (*P prim*) *β* *ve* = *DP prim*
  | $\mathcal{A}$ (*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*)
  | $\mathcal{A}$ (*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.

lemma *cont2cont-case-lambda* [*simp*, *cont2cont*]:
  assumes $\bigwedge a\ b\ c.\ cont\ (\lambda x.\ f\ x\ a\ b\ c)$
  shows $cont\ (\lambda x.\ case\text{-}lambda\ (f\ x)\ l)$
⟨*proof*⟩

lemma *cont2cont-case-d* [*simp*, *cont2cont*]:
  assumes $\bigwedge y.\ cont\ (\lambda x.\ f1\ x\ y)$
    and $\bigwedge y.\ cont\ (\lambda x.\ f2\ x\ y)$

and $\bigwedge y.\ cont\ (\lambda x.\ f3\ x\ y)$
and $cont\ (\lambda x.\ f4\ x)$
shows $cont\ (\lambda x.\ case\text{-}d\ (f1\ x)\ (f2\ x)\ (f3\ x)\ (f4\ x)\ d)$
$\langle proof \rangle$

lemma *cont2cont-case-call* [*simp*, *cont2cont*]:
assumes $\bigwedge a\ b\ c.\ cont\ (\lambda x.\ f1\ x\ a\ b\ c)$
and $\bigwedge a\ b\ c.\ cont\ (\lambda x.\ f2\ x\ a\ b\ c)$
shows $cont\ (\lambda x.\ case\text{-}call\ (f1\ x)\ (f2\ x)\ c)$
$\langle proof \rangle$

lemma *cont2cont-case-prim* [*simp*, *cont2cont*]:
assumes $\bigwedge y.\ cont\ (\lambda x.\ f1\ x\ y)$
and $\bigwedge y\ z.\ cont\ (\lambda x.\ f2\ x\ y\ z)$
shows $cont\ (\lambda x.\ case\text{-}prim\ (f1\ x)\ (f2\ x)\ p)$
$\langle proof \rangle$

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 $\mathcal{F}$ and $\mathcal{C}$, upon every call, a new element is added to the resulting set. The *STOP* continuation now ignores its argument and retuns 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 $evalF :: fstate\ discr \to ans\ (\langle\mathcal{F}\rangle)$
and $evalC :: cstate\ discr \to ans\ (\langle\mathcal{C}\rangle)$
where $\mathcal{F}\cdot fstate = (case\ undiscr\ fstate\ of$
$(DC\ (Lambda\ lab\ vs\ c,\ \beta),\ as,\ ve,\ b) \Rightarrow$
$(if\ length\ vs = length\ as$
$then\ let\ \beta' = \beta\ (lab \mapsto b);$
$ve' = map\text{-}upds\ ve\ (map\ (\lambda v.(v,b))\ vs)\ as$
$in\ \mathcal{C}\cdot(Discr\ (c,\beta',ve',b))$
$else\ \bot)$
$|\ (DP\ (Plus\ c),[DI\ a1,\ DI\ a2,\ cnt],ve,b) \Rightarrow$
$(if\ isProc\ cnt$
$then\ let\ b' = nb\ b\ c;$
$\beta\ = [c \mapsto b]$
$in\ \mathcal{F}\cdot(Discr\ (cnt,[DI\ (a1\ +\ a2)],ve,b'))$
$\cup\ \{((c,\ \beta),cnt)\}$
$else\ \bot)$
$|\ (DP\ (prim.If\ ct\ cf),[DI\ v,\ contt,\ contf],ve,b) \Rightarrow$

$(\mathit{if}\ \mathit{isProc}\ \mathit{contt} \wedge \mathit{isProc}\ \mathit{contf}$
then
$(\mathit{if}\ v \neq 0$
$\mathit{then}\ \mathit{let}\ b' = \mathit{nb}\ b\ \mathit{ct};$
$\beta = [\mathit{ct} \mapsto b]$
$\mathit{in}\ (\mathcal{F}\cdot(\mathit{Discr}\ (\mathit{contt},[],\mathit{ve},b'))$
$\cup \{((\mathit{ct},\ \beta),\mathit{contt})\})$
$\mathit{else}\ \mathit{let}\ b' = \mathit{nb}\ b\ \mathit{cf};$
$\beta = [\mathit{cf} \mapsto b]$
$\mathit{in}\ (\mathcal{F}\cdot(\mathit{Discr}\ (\mathit{contf},[],\mathit{ve},b')))$
$\cup \{((\mathit{cf},\ \beta),\mathit{contf})\})$
$\mathit{else}\ \bot)$
$|\ (\mathit{Stop},[\mathit{DI}\ i],\text{-},\text{-}) \Rightarrow \{\}$
$|\ \text{-} \Rightarrow \bot$
$)$
$|\ \mathcal{C}\cdot\mathit{cstate} = (\mathit{case}\ \mathit{undiscr}\ \mathit{cstate}\ \mathit{of}$
$(\mathit{App}\ \mathit{lab}\ f\ \mathit{vs},\beta,\mathit{ve},b) \Rightarrow$
$\mathit{let}\ f' = \mathcal{A}\ f\ \beta\ \mathit{ve};$
$\mathit{as} = \mathit{map}\ (\lambda v.\ \mathcal{A}\ v\ \beta\ \mathit{ve})\ \mathit{vs};$
$b' = \mathit{nb}\ b\ \mathit{lab}$
$\mathit{in}\ \mathit{if}\ \mathit{isProc}\ f'$
$\mathit{then}\ \mathcal{F}\cdot(\mathit{Discr}\ (f',\mathit{as},\mathit{ve},b')) \cup \{((\mathit{lab},\ \beta),f')\}$
$\mathit{else}\ \bot$
$|\ (\mathit{Let}\ \mathit{lab}\ \mathit{ls}\ c',\beta,\mathit{ve},b) \Rightarrow$
$\mathit{let}\ b' = \mathit{nb}\ b\ \mathit{lab};$
$\beta' = \beta\ (\mathit{lab} \mapsto b');$
$\mathit{ve}' = \mathit{ve}\ {+}{+}\ \mathit{map\text{-}of}\ (\mathit{map}\ (\lambda(v,l).\ ((v,b'),\ \mathcal{A}\ (L\ l)\ \beta'\ \mathit{ve}))\ \mathit{ls})$
$\mathit{in}\ \mathcal{C}\cdot(\mathit{Discr}\ (c',\beta',\mathit{ve}',b'))$
$)$

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*, *of* - λ *a* - . *a*]
lemmas *ds-cases-plus* = *list.exhaust*[
  OF - *d.exhaust*, *of* - - λ*a* -. *a*,
  OF - *list.exhaust*, *of* - - λ- *x* -. *x*,
  OF - - *d.exhaust*, *of* - - λ- - - *a* -. *a*,
  OF - - *list.exhaust*,*of* - - λ- - - - *x* -. *x*,
  OF - - - *list.exhaust*,*of* - - λ- - - - - - - *x*. *x*
  ]
lemmas *ds-cases-if* = *list.exhaust*[*OF* - *d.exhaust*, *of* - - λ*a* -. *a*,
  OF - *list.exhaust*[*OF* - *list.exhaust*[*OF* - *list.exhaust*, *of* - - λ- *x*. *x*], *of* - - λ- *x*. *x*], *of* - - λ- *x* -. *x*]
lemmas *ds-cases-stop* = *list.exhaust*[*OF* - *d.exhaust*, *of* - - λ*a* -. *a*,
  OF - *list.exhaust*, *of* - - λ- *x* -. *x*]

lemmas *fstate-case* = *prod-cases4*[*OF d.exhaust*, *of* - λx - - - . x,
OF - *cl-cases prim.exhaust*, *of* - - λ - - - - a . a λ - - - - a. a,
OF - *case-split ds-cases-plus ds-cases-if ds-cases-stop*,
of - - λ- *as* - - - - - - *vs* - . *length vs* = *length as* λ - *ds* - - - - . *ds* λ - *ds* - - - - -. *ds* λ - *ds* - -. *ds*,
case-names x Closure x x x x Plus x x x x x x x x x x If-True If-False x x x x x Stop x x x x x]

The exact semantics of a program again uses $\mathcal{F}$ with properly initialized arguments. For the first two examples, we see that the function works as expected.

definition *evalCPS* :: *prog* $\Rightarrow$ *ans* (‹$\mathcal{PR}$›)
where $\mathcal{PR}$ l = (*let* ve = *Map.empty*;
β = *Map.empty*;
f = $\mathcal{A}$ (L l) β ve
in $\mathcal{F}\cdot$(*Discr* (f,[*Stop*],ve,b_0)))

lemma *correct-ex1*: $\mathcal{PR}$ *ex1* = {((*2*,[*1* $\mapsto$ b_0]), *Stop*)}
⟨*proof*⟩

lemma *correct-ex2*: $\mathcal{PR}$ *ex2* = {((*2*, [*1* $\mapsto$ b_0]), *DP* (*Plus 3*)),
((*3*, [*3* $\mapsto$ *nb* b_0 *2*]), *Stop*)}
⟨*proof*⟩

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.

class *contour* = *finite* +
  fixes *nb-a* :: *'a* ⇒ *label* ⇒ *'a* (‹$\widehat{nb}$›)
    and *a-initial-contour* :: *'a* (‹$\widehat{b_0}$›)

instantiation *unit* :: *contour*
begin
definition $\widehat{nb}$ - - = ()
definition $\widehat{b_0}$ = ()
instance ⟨*proof*⟩
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 *Shivers−CFA.SetMap* contains functions and lemmas to work with such maps, symbolized by an appended dot (e.g. {}., ∪.).

type-synonym *'c a-benv* = *label* ⇀ *'c* (‹- $\widehat{benv}$› [*1000*])
type-synonym *'c a-closure* = *lambda* × *'c* $\widehat{benv}$ (‹- $\widehat{closure}$› [*1000*])

datatype *'c proc* (‹- $\widehat{proc}$› [*1000*])
  = *PC 'c* $\widehat{closure}$
  | *PP prim*
  | *AStop*

type-synonym *'c a-d* = *'c* $\widehat{proc}$ *set* (‹- $\widehat{d}$› [*1000*])

type-synonym *'c a-venv* = *var* × *'c* ⇒ *'c* $\widehat{d}$ (‹- $\widehat{venv}$› [*1000*])

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

fun *evalV-a* :: *val* ⇒ *'c* $\widehat{benv}$ ⇒ *'c* $\widehat{venv}$ ⇒ *'c* $\widehat{d}$ (‹$\widehat{\mathcal{A}}$›)
  where $\widehat{\mathcal{A}}$ (*C* - *i*) β *ve* = {}
  | $\widehat{\mathcal{A}}$ (*P prim*) β *ve* = {*PP prim*}
  | $\widehat{\mathcal{A}}$ (*R* - *var*) β *ve* =
      (*case* β (*binder var*) *of*
        Some l ⇒ *ve* (*var*,*l*)
       | *None* ⇒ {})

| $\widehat{\mathcal{A}}$ *(L lam)* β *ve* = {*PC* (*lam*, β)}

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

type-synonym $'c$ *a-ccache* = ((*label* × $'c$ $\widehat{benv}$) × $'c$ $\widehat{proc}$) *set* (‹- $\widehat{ccache}$› [*1000*])
type-synonym $'c$ *a-ans* = $'c$ $\widehat{ccache}$ (‹- $\widehat{ans}$› [*1000*])

type-synonym $'c$ *a-fstate* = ($'c$ $\widehat{proc}$ × $'c$ $\widehat{d}$ *list* × $'c$ $\widehat{venv}$ × $'c$) (‹- $\widehat{fstate}$› [*1000*])
type-synonym $'c$ *a-cstate* = (*call* × $'c$ $\widehat{benv}$ × $'c$ $\widehat{venv}$ × $'c$) (‹- $\widehat{cstate}$› [*1000*])

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

lemma *cont2cont-case-lambda* [*simp*, *cont2cont*]:
  assumes $\bigwedge$*a b c*. *cont* (λ*x*. *f x a b c*)
  shows *cont* (λ*x*. *case-lambda* (*f x*) *l*)
⟨*proof*⟩

lemma *cont2cont-case-proc* [*simp*, *cont2cont*]:
  assumes $\bigwedge$*y*. *cont* (λ*x*. *f1 x y*)
    and $\bigwedge$*y*. *cont* (λ*x*. *f2 x y*)
    and *cont* (λ*x*. *f3 x*)
  shows *cont* (λ*x*. *case-proc* (*f1 x*) (*f2 x*) (*f3 x*) *d*)
⟨*proof*⟩

lemma *cont2cont-case-call* [*simp*, *cont2cont*]:
  assumes $\bigwedge$*a b c*. *cont* (λ*x*. *f1 x a b c*)
    and $\bigwedge$*a b c*. *cont* (λ*x*. *f2 x a b c*)
  shows *cont* (λ*x*. *case-call* (*f1 x*) (*f2 x*) *c*)
⟨*proof*⟩

lemma *cont2cont-case-prim* [*simp*, *cont2cont*]:
  assumes $\bigwedge$*y*. *cont* (λ*x*. *f1 x y*)
    and $\bigwedge$*y z*. *cont* (λ*x*. *f2 x y z*)
  shows *cont* (λ*x*. *case-prim* (*f1 x*) (*f2 x*) *p*)
⟨*proof*⟩

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, $\bot$ is returned. Both actually represent the same value, the empty set, so this is just a aesthetic difference.

fixrec *a-evalF* :: $'c$::*contour* $\widehat{fstate}$ *discr* → $'c$ $\widehat{ans}$ (‹$\widehat{\mathcal{F}}$›)
    and *a-evalC* :: $'c$::*contour* $\widehat{cstate}$ *discr* → $'c$ $\widehat{ans}$ (‹$\widehat{\mathcal{C}}$›)
  where $\widehat{\mathcal{F}}$·*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 Ĉ·(Discr (c,β′,ve′,b))
      else ⊥)
  | (PP (Plus c),[-,-,cnts],ve,b) ⇒
          let b′ = n̂b b c;
              β  = [c ↦ b]
          in (⋃ cnt∈cnts. F̂·(Discr (cnt,[{}],ve,b′)))
             ∪
             {((c, β), cont) | cont . cont ∈ cnts}
  | (PP (prim.If ct cf),[-, cntts, cntfs],ve,b) ⇒
        ((   let b′ = n̂b b ct;
                  β = [ct ↦ b]
              in (⋃ cnt∈cntts . F̂·(Discr (cnt,[],ve,b′)))
                 ∪{((ct, β), cnt) | cnt . cnt ∈ cntts}
         )∪(
             let b′ = n̂b b cf;
                  β = [cf ↦ b]
              in (⋃ cnt∈cntfs . F̂·(Discr (cnt,[],ve,b′)))
                 ∪{((cf, β), cnt) | cnt . cnt ∈ cntfs}
         ))
  | (AStop,[-],-,-) ⇒ {}
  | - ⇒ ⊥
)
| Ĉ·cstate = (case undiscr cstate of
      (App lab f vs,β,ve,b) ⇒
         let fs = Â f β ve;
             as = map (λv. Â v β ve) vs;
             b′ = n̂b b lab
         in (⋃f′ ∈ fs. F̂·(Discr (f′,as,ve,b′)))
            ∪{((lab, β),f′) | f′ . f′∈ fs}
  | (Let lab ls c′,β,ve,b) ⇒
         let b′ = n̂b b lab;
             β′ = β (lab ↦ b′);
             ve′ = ve ∪. (⋃. (map (λ(v,l). {(v,b′) := (Â (L l) β′ ve)}.) ls))
         in Ĉ·(Discr (c′,β′,ve′,b′))
)
```

Again, we name the cases of the induction rule and build a nicer case analysis rule for arguments of type $\widehat{fstate}$.

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, β)) 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* - λ- *vs* - - *as* - - . *length vs* = *length as*,
case-names Closure Closure-inv Plus If Stop]

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

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

definition *evalCPS-a* :: *prog* ⇒ (*'c::contour*) $\widehat{ans}$ (‹$\widehat{\mathcal{PR}}$›)
where $\widehat{\mathcal{PR}}$ *l* = (*let ve* = {}.;
β = *Map.empty*;
f = $\widehat{\mathcal{A}}$ (*L l*) β *ve*
in $\widehat{\mathcal{F}}$·(*Discr* (*the-elem f*,[{*AStop*}],*ve*,$\widehat{b_0}$)))

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* :: $d \Rightarrow benv\ set$
  where *benv-in-d* $(DC\ (l,\beta)) = \{\beta\}$
    | *benv-in-d* - $= \{\}$

definition *benv-in-ve* :: $venv \Rightarrow benv\ set$
  where *benv-in-ve* $ve = \bigcup\{benv\text{-}in\text{-}d\ d \mid d\ .\ d \in ran\ ve\}$

fun *contours-in-d* :: $d \Rightarrow contour\ set$
  where *contours-in-d* $(DC\ (l,\beta)) = ran\ \beta$
    | *contours-in-d* - $= \{\}$

definition *contours-in-ve* :: $venv \Rightarrow contour\ set$
  where *contours-in-ve* $ve = \bigcup\{contours\text{-}in\text{-}d\ d \mid d\ .\ d \in ran\ ve\}$

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 $\forall\,\beta{\in}benv\text{-}in\text{-}ve\ ve.\ Q\ \beta$
    and $\forall\,d'{\in}set\ ds.\ \forall\,\beta{\in}benv\text{-}in\text{-}d\ d'.\ Q\ \beta$
  shows $\forall\,\beta{\in}benv\text{-}in\text{-}ve\ (ve(map\ (\lambda v.\ (v,\ b''))\ vs\ [\mapsto]\ ds)).\ Q\ \beta$
$\langle proof\rangle$

lemma *benv-in-eval*:
  assumes $\forall\,\beta'{\in}benv\text{-}in\text{-}ve\ ve.\ Q\ \beta'$
    and $Q\ \beta$
  shows $\forall\,\beta{\in}benv\text{-}in\text{-}d\ (\mathcal{A}\ v\ \beta\ ve).\ Q\ \beta$
$\langle proof\rangle$

lemma *contours-in-ve-empty*[*simp*]: $contours\text{-}in\text{-}ve\ Map.empty = \{\}$
  $\langle proof\rangle$

lemma *contours-in-ve-upds*:
  assumes *eq-length*: $length\ vs = length\ ds$
    and $\forall\,b'{\in}contours\text{-}in\text{-}ve\ ve.\ Q\ b'$
    and $\forall\,d'{\in}set\ ds.\ \forall\,b'{\in}contours\text{-}in\text{-}d\ d'.\ Q\ b'$
  shows $\forall\,b'{\in}contours\text{-}in\text{-}ve\ (ve(map\ (\lambda v.\ (v,\ b''))\ vs\ [\mapsto]\ ds)).\ Q\ b'$
$\langle proof\rangle$

lemma *contours-in-ve-upds-binds*:
assumes $\forall\, b' {\in} contours\text{-}in\text{-}ve\ ve.\ Q\ b'$
and $\forall\, b' {\in} ran\ \beta'.\ Q\ b'$
shows $\forall\, b' {\in} contours\text{-}in\text{-}ve\ (ve\ {+}{+}\ map\text{-}of\ (map\ (\lambda(v,l).\ ((v,b''),\ \mathcal{A}\ (L\ l)\ \beta'\ ve))\ ls)).\ Q\ b'$
$\langle proof \rangle$

lemma *contours-in-eval*:
assumes $\forall\, b' {\in} contours\text{-}in\text{-}ve\ ve.\ Q\ b'$
and $\forall\, b' {\in}\ ran\ \beta.\ Q\ b'$
shows $\forall\, b' {\in} contours\text{-}in\text{-}d\ (\mathcal{A}\ f\ \beta\ ve).\ Q\ b'$
$\langle proof \rangle$

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 *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 admissibiliy, *HOLCF* provides a number of introduction lemmas that, together with some additions in *Shivers−CFA.HOLCFUtils* and the continuity lemmas, mechanically proove admissibiliy. 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 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 hypothesises.

lemma *cc-single-valued'*:
$[\![\ \forall\, b' \in contours\text{-}in\text{-}ve\ ve.\ b' < b$
$;\ \forall\, b' \in contours\text{-}in\text{-}d\ d.\ b' < b$
$;\ \forall\, d' \in set\ ds.\ \forall\, b' \in contours\text{-}in\text{-}d\ d'.\ b' < b$
$]\!]$
$\Longrightarrow$
$(\quad single\text{-}valued\ (\mathcal{F}{\cdot}(Discr\ (d,ds,ve,b)))$
$\wedge\ (\forall\ \ ((lab,\beta),t) \in \mathcal{F}{\cdot}(Discr\ (d,ds,ve,\ b)).\ \exists\ \ b'.\ b' \in ran\ \beta\ \wedge\ b \leq b')$
$)$
and $[\![\ b \in ran\ \beta'$

; $\forall\, b' {\in} ran\ \beta'.\ b' \leq b$
; $\forall\, b' \in contours\text{-}in\text{-}ve\ ve.\ b' \leq b$
$\rrbracket$
$\Longrightarrow$
($single\text{-}valued\ (\mathcal{C}{\cdot}(Discr\ (c,\beta',ve,b)))$
$\wedge\ (\forall\ ((lab,\beta),t) \in \mathcal{C}{\cdot}(Discr\ (c,\beta',ve,b)).\ \exists\ b'.\ b' \in ran\ \beta \wedge b \leq b')$
)
⟨*proof*⟩

lemma *single-valued* ($\mathcal{PR}$ *prog*)
⟨*proof*⟩
end

6. The abstract semantics is correct

theory *AbsCFCorrect*
imports *AbsCF ExCF*
begin

default-sort *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 *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 *contour* type class by the abstraction functions and two equations involving the *nb* and b_0 symbols.

class *contour-a* = *contour* +
fixes *abs-cnt* :: *contour* ⇒ *'a*
assumes *abs-cnt-nb*[*simp*]: $abs\text{-}cnt\ (nb\ b\ lab) = \widehat{nb}\ (abs\text{-}cnt\ b)\ lab$
and *abs-cnt-initial*[*simp*]: $abs\text{-}cnt(b_0) = \widehat{b_0}$

instantiation *unit* :: *contour-a*
begin
definition *abs-cnt* - = ()
instance ⟨*proof*⟩

end

It would be unwieldly to always write out *abs-<type> x*. We would rather like to write $|x|$ if the type of 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 *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 |-| for the family of abstraction functions.

consts *abs* :: $'a \Rightarrow 'b$ (‹|-|›)

adhoc-overloading
  $abs \rightleftharpoons abs\text{-}cnt$

definition *abs-benv* :: $benv \Rightarrow 'c{::}contour\text{-}a\ \widehat{benv}$
  where $abs\text{-}benv\ \beta = map\text{-}option\ abs\text{-}cnt \circ \beta$

adhoc-overloading
  $abs \rightleftharpoons abs\text{-}benv$

primrec *abs-closure* :: $closure \Rightarrow 'c{::}contour\text{-}a\ \widehat{closure}$
  where $abs\text{-}closure\ (l,\beta) = (l,|\beta|\ )$

adhoc-overloading
  $abs \rightleftharpoons abs\text{-}closure$

primrec *abs-d* :: $d \Rightarrow 'c{::}contour\text{-}a\ \widehat{d}$
  where $abs\text{-}d\ (DI\ i) = \{\}$
    $|\ abs\text{-}d\ (DP\ p) = \{PP\ p\}$
    $|\ abs\text{-}d\ (DC\ cl) = \{PC\ |cl|\}$
    $|\ abs\text{-}d\ (Stop) = \{AStop\}$

adhoc-overloading
  $abs \rightleftharpoons abs\text{-}d$

definition *abs-venv* :: $venv \Rightarrow 'c{::}contour\text{-}a\ \widehat{venv}$
  where $abs\text{-}venv\ ve = (\lambda(v,b\text{-}a).\ \bigcup\{(case\ ve\ (v,b)\ of\ Some\ d \Rightarrow |d|\ |\ None \Rightarrow \{\})\ |\ b.\ |b| = b\text{-}a\ \})$

adhoc-overloading
  $abs \rightleftharpoons abs\text{-}venv$

definition *abs-ccache* :: *ccache* ⇒ $'c$::*contour-a* $\widehat{ccache}$
  where *abs-ccache cc* = $(\bigcup((c,\beta),d) \in cc\ .\ \{((c,abs\text{-}benv\ \beta),\ p) \mid p\ .\ p{\in}abs\text{-}d\ d\})$

adhoc-overloading
  abs ⇌ *abs-ccache*

fun *abs-fstate* :: *fstate* ⇒ $'c$::*contour-a* $\widehat{fstate}$
  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* $\widehat{cstate}$
  where *abs-cstate* $(c,\beta,ve,b) = (c,\ |\beta|,\ |ve|,\ |b|\ )$

adhoc-overloading
  abs ⇌ *abs-cstate*

6.2. Lemmas about abstraction functions

Some results about the abstractions functions.

lemma *abs-benv-empty*[*simp*]: $|Map.empty| = Map.empty$
⟨*proof*⟩

lemma *abs-benv-upd*[*simp*]: $|\beta(c{\mapsto}b)| = |\beta|\ (c \mapsto |b|\ )$
  ⟨*proof*⟩

lemma *the-elem-is-Proc*:
  assumes *isProc cnt*
  shows *the-elem* $|cnt| \in |cnt|$
⟨*proof*⟩

lemma [*simp*]: $|\{\}| = \{\}$ ⟨*proof*⟩

lemma *abs-cache-singleton* [*simp*]: $|\{((c,\beta),d)\}| = \{((c,\ |\beta|\ ),\ p)\ |p.\ p \in |d|\}$
  ⟨*proof*⟩

lemma *abs-venv-empty*[*simp*]: $|Map.empty| = \{\}$.
  ⟨*proof*⟩

6.3. Approximation relation

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

consts *approx* :: $'a \Rightarrow 'a \Rightarrow bool$ (‹- $\lessapprox$ -›)

definition *venv-approx* :: $'c\ \widehat{venv} \Rightarrow 'c\ \widehat{venv} \Rightarrow bool$
where *venv-approx* = *smap-less*

adhoc-overloading
approx $\rightleftharpoons$ *venv-approx*

definition *ccache-approx* :: $'c\ \widehat{ccache} \Rightarrow 'c\ \widehat{ccache} \Rightarrow bool$
where *ccache-approx* = *less-eq*

adhoc-overloading
approx $\rightleftharpoons$ *ccache-approx*

definition *d-approx* :: $'c\ \widehat{d} \Rightarrow 'c\ \widehat{d} \Rightarrow bool$
where *d-approx* = *less-eq*

adhoc-overloading
approx $\rightleftharpoons$ *d-approx*

definition *ds-approx* :: $'c\ \widehat{d}\ list \Rightarrow 'c\ \widehat{d}\ list \Rightarrow bool$
where *ds-approx* = *list-all2 d-approx*

adhoc-overloading
approx $\rightleftharpoons$ *ds-approx*

inductive *fstate-approx* :: $'c\ \widehat{fstate} \Rightarrow 'c\ \widehat{fstate} \Rightarrow bool$
where $[\![\ ve \lessapprox ve'\ ;\ ds \lessapprox ds'\ ]\!]$
$\Longrightarrow$ *fstate-approx* $(proc,ds,ve,b)\ (proc,ds',ve',b)$

adhoc-overloading
approx $\rightleftharpoons$ *fstate-approx*

inductive *cstate-approx* :: $'c\ \widehat{cstate} \Rightarrow 'c\ \widehat{cstate} \Rightarrow bool$
where $[\![\ ve \lessapprox ve'\ ]\!] \Longrightarrow$ *cstate-approx* $(c,\beta,ve,b)\ (c,\beta,ve',b)$

adhoc-overloading
approx $\rightleftharpoons$ *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* :: $'c\ \widehat{venv}$
shows $[\![\ ve1 \lessapprox ve2;\ ve2 \lessapprox ve3\ ]\!] \Longrightarrow (ve1 \lessapprox ve3)$
$\langle proof \rangle$

lemma *abs-venv-union*: $|ve1\ ++\ ve2| \lessapprox |ve1| \cup.\ |ve2|$
⟨*proof*⟩

lemma *abs-venv-map-of-rev*: $|map\text{-}of\ (rev\ l)| \lessapprox \bigcup.\ (map\ (\lambda(v,k).\ |[v \mapsto k]|\ )\ l)$
⟨*proof*⟩

lemma *abs-venv-map-of*: $|map\text{-}of\ l| \lessapprox \bigcup.\ (map\ (\lambda(v,k).\ |[v \mapsto k]|\ )\ l)$
⟨*proof*⟩

lemma *abs-venv-singleton*: $|[(v,b) \mapsto d]| = \{(v,|b|\ ) := |d|\}$.
⟨*proof*⟩

lemma *ccache-approx-empty*[*simp*]:
fixes $x :: {}'c\ \widehat{ccache}$
shows $\{\} \lessapprox x$
⟨*proof*⟩

lemmas *ccache-approx-trans*[*trans*] = *subset-trans*[**where** $'a = ((label \times {}'c\ \widehat{benv}) \times {}'c\ \widehat{proc})$, *folded ccache-approx-def*]
lemmas *Un-mono-approx* = *Un-mono*[**where** $'a = ((label \times {}'c\ \widehat{benv}) \times {}'c\ \widehat{proc})$, *folded ccache-approx-def*]
lemmas *Un-upper1-approx* = *Un-upper1*[**where** $'a = ((label \times {}'c\ \widehat{benv}) \times {}'c\ \widehat{proc})$, *folded ccache-approx-def*]
lemmas *Un-upper2-approx* = *Un-upper2*[**where** $'a = ((label \times {}'c\ \widehat{benv}) \times {}'c\ \widehat{proc})$, *folded ccache-approx-def*]

lemma *abs-ccache-union*: $|c1 \cup c2| \lessapprox |c1| \cup |c2|$
⟨*proof*⟩

lemma *d-approx-empty*[*simp*]: $\{\} \lessapprox (d{::}'c\ \widehat{d})$
⟨*proof*⟩

lemma *ds-approx-empty*[*simp*]: $[] \lessapprox []$
⟨*proof*⟩

6.5. Lemma 7

Shivers' lemma 7 says that $\widehat{\mathcal{A}}$ safely approximates $\mathcal{A}$.

lemma *lemma7*:
assumes $|ve{::}venv| \lessapprox ve\text{-}a$
shows $|\mathcal{A}\ f\ \beta\ ve| \lessapprox \widehat{\mathcal{A}}\ f\ |\beta|\ ve\text{-}a$
⟨*proof*⟩

6.6. Lemmas 8 and 9

The main goal of this section is to show that $\widehat{\mathcal{F}}$ safely approximates $\mathcal{F}$ and that $\widehat{\mathcal{C}}$ safely approximates $\mathcal{C}$. 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* $(\lambda x.\ abs\text{-}ccache(f\ x))$
⟨*proof*⟩

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 $fstate\text{-}a :: {}'c{::}contour\text{-}a\ \widehat{fstate}$ **and** $cstate\text{-}a :: {}'c{::}contour\text{-}a\ \widehat{cstate}$
 shows $|fstate| \lessapprox fstate\text{-}a \Longrightarrow |\mathcal{F}\cdot(Discr\ fstate)| \lessapprox \widehat{\mathcal{F}}\cdot(Discr\ fstate\text{-}a)$
   and $|cstate| \lessapprox cstate\text{-}a \Longrightarrow |\mathcal{C}\cdot(Discr\ cstate)| \lessapprox \widehat{\mathcal{C}}\cdot(Discr\ cstate\text{-}a)$
⟨*proof*⟩

And finally, we lift this result to $\widehat{\mathcal{PR}}$ and $\mathcal{PR}$.

lemma *lemma6*: $|\mathcal{PR}\ l| \lessapprox \widehat{\mathcal{PR}}\ l$
  ⟨*proof*⟩
end

7. Generic Computability

theory *Computability*
imports *HOLCF HOLCFUtils*
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 $\widehat{\mathcal{PR}}$.

7.1. Non-branching case

After the following lemma (which could go into *HOL.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: 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*:
  $insert\ (n{::}nat)\ \{n{<}..\} = \{n..\}$
⟨*proof*⟩

lemma *theorem10*:
  fixes $g :: 'a{::}cpo \to 'b{::}type\ set$ **and** $r :: 'a \to 'a$
  shows $fix{\cdot}(\Lambda\ f\ x.\ g{\cdot}x \cup f{\cdot}(r{\cdot}x)) = (\Lambda\ x.\ (\bigcup i.\ g{\cdot}(r^i{\cdot}x)))$
⟨*proof*⟩

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

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

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

$$\underline{g}Y = \bigcup_{a \in A} g\ a \quad \text{and} \underline{R}Y = \bigcup_{a \in R} R\ a$$

the equation becomes

$$\underline{f}Y = \underline{g}Y \cup \underline{f}\ (\underline{R}Y)$$

(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* $:: ('a{::}cpo \to 'b{::}type\ set) \Rightarrow 'a\ set \to 'b\ set\ (\underline{\ })$
  where $\underline{f} = (\Lambda\ S.\ (\bigcup y{\in}S\ .\ f{\cdot}y))$

lemma *powerset-lift-singleton*[*simp*]:
  $\underline{f}{\cdot}\{x\} = f{\cdot}x$
⟨*proof*⟩

lemma *powerset-lift-union*[*simp*]:
  $\underline{f}{\cdot}(A \cup B) = \underline{f}{\cdot}A \cup \underline{f}{\cdot}B$
⟨*proof*⟩

lemma *UNION-commute*:$(\bigcup x{\in}A.\ \bigcup y{\in}B\ .\ P\ x\ y) = (\bigcup y{\in}B.\ \bigcup x{\in}A\ .\ P\ x\ y)$
  ⟨*proof*⟩

lemma *powerset-lift-UNION*:
  $(\bigcup x{\in}S.\ \underline{g}{\cdot}(A\ x)) = \underline{g}{\cdot}(\bigcup x{\in}S.\ A\ x)$

⟨*proof*⟩

lemma *powerset-lift-iterate-UNION*:
$(\bigcup x{\in}S.\ (\underline{g})^{i}{\cdot}(A\ x)) = (\underline{g})^{i}{\cdot}(\bigcup x{\in}S.\ A\ x)$
⟨*proof*⟩

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 :: $'a \to 'b$ *set* **and** g :: $'a \to 'b$ *set* **and** R :: $'a \to 'a$ *set*
assumes $\bigwedge x.\ f{\cdot}x = g{\cdot}x \cup (\bigcup y{\in}R{\cdot}x.\ f{\cdot}y)$
shows $\underline{f}{\cdot}S = \underline{g}{\cdot}S \cup \underline{f}{\cdot}(\underline{R}{\cdot}S)$
⟨*proof*⟩

Theorem 10 as it will be used in Theorem 12.

lemmas *theorem10ps* = *theorem10*[*of* $\underline{g}$ $\underline{r}$] **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* $\to$ $'b$::*type set* **and** R :: $'a \to 'a$ *set*
assumes *F-fix*: $F = \mathit{fix}{\cdot}(\Lambda\ F\ x.\ \underline{g}{\cdot}x \cup F{\cdot}(\underline{R}{\cdot}x))$
shows $\mathit{fix}{\cdot}(\Lambda\ f\ x.\ g{\cdot}x \cup (\bigcup y{\in}R{\cdot}x.\ f{\cdot}y)) = (\Lambda\ x.\ F{\cdot}\{x\})$
⟨*proof*⟩

lemma *theorem12*:
fixes g :: $'a$::*discrete-cpo* $\to$ $'b$::*type set* **and** R :: $'a \to 'a$ *set*
shows $\mathit{fix}{\cdot}(\Lambda\ f\ x.\ g{\cdot}x \cup (\bigcup y{\in}R{\cdot}x.\ f{\cdot}y)){\cdot}x = \underline{g}{\cdot}(\bigcup i.((\underline{R})^{i}{\cdot}\{x\}))$
⟨*proof*⟩

end

8. The abstract semantics is computable

theory *AbsCFComp*
imports *AbsCF Computability FixTransform CPSUtils MapSets*
begin

default-sort *type*

The point of the abstract semantics is that it is computable. To show this, we exploit the special structure of $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$: 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 $\widehat{fstate}$ and $\widehat{cstate}$, i.e. we treat the two mutually recursive functions now as one.

abs-g gives the local result for the given argument.

```
fixrec abs-g :: ('c::contour \widehat{fstate} + 'c \widehat{cstate}) discr → 'c \widehat{ans}
  where abs-g·x = (case undiscr x of
            (Inl (PC (Lambda lab vs c, β), as, ve, b)) ⇒ {}
          | (Inl (PP (Plus c),[-,-,cnts],ve,b)) ⇒
                 let b' = \widehat{nb} b c;
                     β  = [c ↦ b]
                 in {((c, β), cont) | cont . cont ∈ cnts}
          | (Inl (PP (prim.If ct cf),[-, cntts, cntfs],ve,b)) ⇒
               ((   let b' = \widehat{nb} b ct;
                          β = [ct ↦ b]
                      in {((ct, β), cnt) | cnt . cnt ∈ cntts}
                )∪(
                    let b' = \widehat{nb} b cf;
                          β = [cf ↦ b]
                      in {((cf, β), cnt) | cnt . cnt ∈ cntfs}
                ))
          | (Inl (AStop,[-],-,-)) ⇒ {}
          | (Inl -) ⇒ ⊥
          | (Inr (App lab f vs,β,ve,b)) ⇒
              let fs = \widehat{A} f β ve;
                  as = map (λv. \widehat{A} v β ve) vs;
                  b' = \widehat{nb} b lab
               in {((lab, β),f') | f' . f'∈ fs}
          | (Inr (Let lab ls c',β,ve,b)) ⇒ {}
      )
```

abs-R gives the set of arguments passed to the recursive calls.

```
fixrec abs-R :: ('c::contour \widehat{fstate} + 'c \widehat{cstate}) discr → ('c::contour \widehat{fstate} + 'c \widehat{cstate}) discr
set
  where abs-R·x = (case undiscr x of
            (Inl (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 {Discr (Inr (c,β',ve',b))}
             else ⊥)
          | (Inl (PP (Plus c),[-,-,cnts],ve,b)) ⇒
                 let b' = \widehat{nb} b c;
                     β  = [c ↦ b]
                 in (⋃ cnt∈cnts. {Discr (Inl (cnt,[{}],ve,b'))})
```

| $(Inl\ (PP\ (prim.If\ ct\ cf),[\text{-},\ cntts,\ cntfs],ve,b)) \Rightarrow$
(($let\ b' = \widehat{nb}\ b\ ct;$
$\beta = [ct \mapsto b]$
$in\ (\bigcup cnt{\in}cntts\ .\ \{Discr\ (Inl\ (cnt,[],ve,b'))\})$
$)\cup($
$let\ b' = \widehat{nb}\ b\ cf;$
$\beta = [cf \mapsto b]$
$in\ (\bigcup cnt{\in}cntfs\ .\ \{Discr\ (Inl\ (cnt,[],ve,b'))\})$
))
| $(Inl\ (AStop,[\text{-}],\text{-},\text{-})) \Rightarrow \{\}$
| $(Inl\ \text{-}) \Rightarrow \bot$
| $(Inr\ (App\ lab\ f\ vs,\beta,ve,b)) \Rightarrow$
$let\ fs = \widehat{\mathcal{A}}\ f\ \beta\ ve;$
$as = map\ (\lambda v.\ \widehat{\mathcal{A}}\ v\ \beta\ ve)\ vs;$
$b' = \widehat{nb}\ b\ lab$
$in\ (\bigcup f' \in fs.\ \{Discr\ (Inl\ (f',as,ve,b'))\})$
| $(Inr\ (Let\ lab\ ls\ c',\beta,ve,b)) \Rightarrow$
$let\ b' = \widehat{nb}\ b\ lab;$
$\beta' = \beta\ (lab \mapsto b');$
$ve' = ve \cup.\ (\bigcup.\ (map\ (\lambda(v,l).\ \{(v,b') := (\widehat{\mathcal{A}}\ (L\ l)\ \beta'\ ve)\}.)\ ls))$
$in\ \{Discr\ (Inr\ (c',\beta',ve',b'))\}$
)

The initial argument vector, as created by $\widehat{\mathcal{PR}}$.

definition $initial\text{-}r :: prog \Rightarrow ('c{::}contour\ \widehat{fstate} + 'c\ \widehat{cstate})\ discr$
where $initial\text{-}r\ prog = Discr\ (Inl$
$(the\text{-}elem\ (\widehat{\mathcal{A}}\ (L\ prog)\ Map.empty\ \{\}.),\ [\{AStop\}],\ \{\}.,\ \widehat{b_0}))$

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 $\widehat{\mathcal{F}}$, or possible arguments to $\widehat{\mathcal{C}}$ and of possible arguments.

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

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

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

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

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

lemma *finite-arg-space*: *finite* (*arg-poss p*)
  ⟨*proof*⟩

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

lemma *evalV-possible*:
  assumes *f*: *f* ∈ $\widehat{\mathcal{A}}$ *d β ve*
  and *d*: *d* ∈ *vals p*
  and *ve*: *ve* ∈ *smaps-over* (*vars p* × *UNIV*) (*proc-poss p*)
  and *β*: *β* ∈ *maps-over* (*labels p*) *UNIV*
shows *f* ∈ *proc-poss p*
⟨*proof*⟩

lemma *adm-subset*: *cont* (*λx. f x*) ⟹ *adm* (*λx. f x* ⊆ *S*)
⟨*proof*⟩

lemma *arg-space-complete*:
  state ∈ *arg-poss p* ⟹ *abs-R·state* ⊆ *arg-poss p*
⟨*proof*⟩

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

lemma *arg-space-complete-ps*: *states* ⊆ *arg-poss p* ⟹ (*a̲bs-R*)·*states* ⊆ *arg-poss p*
⟨*proof*⟩

We are not so much interested in the finiteness of the set of possible arguments but rather of the 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* ∈ *S*
  and *step*: ⋀*y. y*⊆*S* ⟹ (*f·y*) ⊆ *S*
  shows (⋃*i. iterate i·f·*{*x*}) ⊆ *S*
⟨*proof*⟩

lemma *args-finite*: *finite* (⋃*i. iterate i·*(*a̲bs-R*)·{*initial-r p*}) (**is** *finite ?S*)
⟨*proof*⟩

8.2. A decomposition

The functions *abs-g* and *abs-R* are derived from $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$. This connection has yet to expressed explicitly.

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

lemma *a-evalF-decomp*:
  $\widehat{\mathcal{F}} = \mathit{fst}\ (\mathit{sum\text{-}to\text{-}tup}\cdot(\mathit{fix}\cdot(\Lambda\ f\ x.\ (\bigcup y{\in}\mathit{abs\text{-}R}\cdot x.\ f\cdot y) \cup \mathit{abs\text{-}g}\cdot x)))$
⟨*proof*⟩

8.3. The iterative equation

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

lemma *a-evalF-iterative*:
  $\widehat{\mathcal{F}}\cdot(\mathit{Discr}\ x) = \underline{a}\mathit{bs\text{-}g}\cdot(\bigcup i.\ \mathit{iterate}\ i\cdot(\underline{a}\mathit{bs\text{-}R})\cdot\{\mathit{Discr}\ (\mathit{Inl}\ x)\})$
⟨*proof*⟩

lemma *a-evalCPS-interative*:
$\widehat{\mathcal{PR}}\ \mathit{prog} = \underline{a}\mathit{bs\text{-}g}\cdot(\bigcup i.\ \mathit{iterate}\ i\cdot(\underline{a}\mathit{bs\text{-}R})\cdot\{\mathit{initial\text{-}r}\ \mathit{prog}\})$
⟨*proof*⟩

end

Part III. The auxiliary theories

9. Syntax tree helpers

theory *CPSUtils*
imports *CPSScheme*
begin

This theory defines the sets *lambdas p*, *calls p*, *calls p*, *vars p*, *labels p* and *prims p* as the subexpressions of the program *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.

fun *lambdas* :: *lambda* $\Rightarrow$ *lambda set*
and *lambdasC* :: *call* $\Rightarrow$ *lambda set*
and *lambdasV* :: *val* $\Rightarrow$ *lambda set*
where *lambdas* $(Lambda\ l\ vs\ c) = (\{Lambda\ l\ vs\ c\} \cup lambdasC\ c)$
  | $lambdasC\ (App\ l\ d\ ds) = lambdasV\ d \cup \bigcup\ (lambdasV\ `\ set\ ds)$
  | $lambdasC\ (Let\ l\ binds\ c') = (\bigcup (\text{-},\ y){\in}set\ binds.\ lambdas\ y) \cup lambdasC\ c'$
  | $lambdasV\ (L\ l) = lambdas\ l$
  | $lambdasV$ - $= \{\}$

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

lemma *finite-lambdas*[*simp*]: *finite* (*lambdas l*) **and** *finite* (*lambdasC c*) *finite* (*lambdasV v*)
⟨*proof*⟩

lemma *finite-calls*[*simp*]: *finite* (*calls l*) **and** *finite* (*callsC c*) *finite* (*callsV v*)
⟨*proof*⟩

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

lemma *finite-vars*[*simp*]: *finite* (*vars l*) **and** *finite* (*varsC c*) *finite* (*varsV v*)
⟨*proof*⟩

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

fun *labels* :: *lambda* $\Rightarrow$ *label set*
and *labelsC* :: *call* $\Rightarrow$ *label set*
and *labelsV* :: *val* $\Rightarrow$ *label set*
where *labels* $(Lambda\ l\ vs\ c) = \{l\} \cup labelsC\ c$

| *labelsC* (*App l a as*) = {*l*} ∪ *labelsV a* ∪ ⋃(*labelsV* ‘ (*set as*))
| *labelsC* (*Let l binds c'*) = {*l*} ∪ (⋃(*v*, *y*)∈*set binds*. *labels y*) ∪ *labelsC c'*
| *labelsV* (*L l*) = *labels l*
| *labelsV* (*R l* -) = {*l*}
| *labelsV* - = {}

lemma *finite-labels*[*simp*]: *finite* (*labels l*) **and** *finite* (*labelsC c*) *finite* (*labelsV v*)
⟨*proof*⟩

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* ∪ ⋃(*primsV* ‘ (*set as*))
| *primsC* (*Let* - *binds c'*) = (⋃(-, *y*)∈*set binds*. *prims y*) ∪ *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*)
⟨*proof*⟩

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* ∪ ⋃(*valsV* ‘ (*set as*))
| *valsC* (*Let* - *binds c'*) = (⋃(-, *y*)∈*set binds*. *vals y*) ∪ *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 ∀ *x*∈ *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*)
⟨*proof*⟩

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*
and ∀ *z*∈ *set list*. *Lambda l vs c* ∈ *lambdasV z* ⟶ *l* ∈ *labelsV z*

and $\forall x \in$ *set* (*list2* :: (*var* $\times$ *lambda*) *list*) . *Lambda l vs c* $\in$ *lambdas* (*snd x*) $\longrightarrow$ *l* $\in$ *labels* (*snd x*)
and *Lambda l vs c* $\in$ *lambdas* (*snd* (*t*:: *var*$\times$*lambda*)) $\Longrightarrow$ *l* $\in$ *labels* (*snd t*)
⟨*proof*⟩

lemma
shows *lambdas3*: *Lambda l vs c* $\in$ *lambdas x* $\Longrightarrow$ *set vs* $\subseteq$ *vars x*
and *Lambda l vs c* $\in$ *lambdasC y* $\Longrightarrow$ *set vs* $\subseteq$ *varsC y*
and *Lambda l vs c* $\in$ *lambdasV z* $\Longrightarrow$ *set vs* $\subseteq$ *varsV z*
and $\forall z \in$ *set list*. *Lambda l vs c* $\in$ *lambdasV z* $\longrightarrow$ *set vs* $\subseteq$ *varsV z*
and $\forall x \in$ *set* (*list2* :: (*var* $\times$ *lambda*) *list*) . *Lambda l vs c* $\in$ *lambdas* (*snd x*) $\longrightarrow$ *set vs* $\subseteq$ *vars* (*snd x*)
and *Lambda l vs c* $\in$ *lambdas* (*snd* (*t*:: *var*$\times$*lambda*)) $\Longrightarrow$ *set vs* $\subseteq$ *vars* (*snd t*)
⟨*proof*⟩

lemma
shows *app1*: *App l d ds* $\in$ *calls x* $\Longrightarrow$ *d* $\in$ *vals x*
and *App l d ds* $\in$ *callsC y* $\Longrightarrow$ *d* $\in$ *valsC y*
and *App l d ds* $\in$ *callsV z* $\Longrightarrow$ *d* $\in$ *valsV z*
and $\forall z \in$ *set list*. *App l d ds* $\in$ *callsV z* $\longrightarrow$ *d* $\in$ *valsV z*
and $\forall x \in$ *set* (*list2* :: (*var* $\times$ *lambda*) *list*) . *App l d ds* $\in$ *calls* (*snd x*) $\longrightarrow$ *d* $\in$ *vals* (*snd x*)
and *App l d ds* $\in$ *calls* (*snd* (*t*:: *var*$\times$*lambda*)) $\Longrightarrow$ *d* $\in$ *vals* (*snd t*)
⟨*proof*⟩

lemma
shows *app2*: *App l d ds* $\in$ *calls x* $\Longrightarrow$ *set ds* $\subseteq$ *vals x*
and *App l d ds* $\in$ *callsC y* $\Longrightarrow$ *set ds* $\subseteq$ *valsC y*
and *App l d ds* $\in$ *callsV z* $\Longrightarrow$ *set ds* $\subseteq$ *valsV z*
and $\forall z \in$ *set list*. *App l d ds* $\in$ *callsV z* $\longrightarrow$ *set ds* $\subseteq$ *valsV z*
and $\forall x \in$ *set* (*list2* :: (*var* $\times$ *lambda*) *list*) . *App l d ds* $\in$ *calls* (*snd x*) $\longrightarrow$ *set ds* $\subseteq$ *vals* (*snd x*)
and *App l d ds* $\in$ *calls* (*snd* (*t*:: *var*$\times$*lambda*)) $\Longrightarrow$ *set ds* $\subseteq$ *vals* (*snd t*)
⟨*proof*⟩

lemma
shows *let1*: *Let l binds* c' $\in$ *calls x* $\Longrightarrow$ *l* $\in$ *labels x*
and *Let l binds* c' $\in$ *callsC y* $\Longrightarrow$ *l* $\in$ *labelsC y*
and *Let l binds* c' $\in$ *callsV z* $\Longrightarrow$ *l* $\in$ *labelsV z*
and $\forall z \in$ *set list*. *Let l binds* c' $\in$ *callsV z* $\longrightarrow$ *l* $\in$ *labelsV z*
and $\forall x \in$ *set* (*list2* :: (*var* $\times$ *lambda*) *list*) . *Let l binds* c' $\in$ *calls* (*snd x*) $\longrightarrow$ *l* $\in$ *labels* (*snd x*)
and *Let l binds* c' $\in$ *calls* (*snd* (*t*:: *var*$\times$*lambda*)) $\Longrightarrow$ *l* $\in$ *labels* (*snd t*)
⟨*proof*⟩

lemma
shows *let2*: *Let l binds* c' $\in$ *calls x* $\Longrightarrow$ c' $\in$ *calls x*
and *Let l binds* c' $\in$ *callsC y* $\Longrightarrow$ c' $\in$ *callsC y*
and *Let l binds* c' $\in$ *callsV z* $\Longrightarrow$ c' $\in$ *callsV z*
and $\forall z \in$ *set list*. *Let l binds* c' $\in$ *callsV z* $\longrightarrow$ c' $\in$ *callsV z*

and $\forall x \in$ *set* (*list2* :: (*var* × *lambda*) *list*) . *Let l binds* $c' \in$ *calls* (*snd x*) $\longrightarrow$ $c' \in$ *calls* (*snd x*)
and *Let l binds* $c' \in$ *calls* (*snd* (*t*:: *var*×*lambda*)) $\Longrightarrow$ $c' \in$ *calls* (*snd t*)
⟨*proof*⟩

lemma
shows *let3*: *Let l binds* $c' \in$ *calls x* $\Longrightarrow$ *fst* ‘ *set binds* ⊆ *vars x*
and *Let l binds* $c' \in$ *callsC y* $\Longrightarrow$ *fst* ‘ *set binds* ⊆ *varsC y*
and *Let l binds* $c' \in$ *callsV z* $\Longrightarrow$ *fst* ‘ *set binds* ⊆ *varsV z*
and $\forall z \in$ *set list*. *Let l binds* $c' \in$ *callsV z* $\longrightarrow$ *fst* ‘ *set binds* ⊆ *varsV z*
and $\forall x \in$ *set* (*list2* :: (*var* × *lambda*) *list*) . *Let l binds* $c' \in$ *calls* (*snd x*) $\longrightarrow$ *fst* ‘ *set binds* ⊆ *vars* (*snd x*)
and *Let l binds* $c' \in$ *calls* (*snd* (*t*:: *var*×*lambda*)) $\Longrightarrow$ *fst* ‘ *set binds* ⊆ *vars* (*snd t*)
⟨*proof*⟩

lemma
shows *let4*: *Let l binds* $c' \in$ *calls x* $\Longrightarrow$ *snd* ‘ *set binds* ⊆ *lambdas x*
and *Let l binds* $c' \in$ *callsC y* $\Longrightarrow$ *snd* ‘ *set binds* ⊆ *lambdasC y*
and *Let l binds* $c' \in$ *callsV z* $\Longrightarrow$ *snd* ‘ *set binds* ⊆ *lambdasV z*
and $\forall z \in$ *set list*. *Let l binds* $c' \in$ *callsV z* $\longrightarrow$ *snd* ‘ *set binds* ⊆ *lambdasV z*
and $\forall x \in$ *set* (*list2* :: (*var* × *lambda*) *list*) . *Let l binds* $c' \in$ *calls* (*snd x*) $\longrightarrow$ *snd* ‘ *set binds* ⊆ *lambdas* (*snd x*)
and *Let l binds* $c' \in$ *calls* (*snd* (*t*:: *var*×*lambda*)) $\Longrightarrow$ *snd* ‘ *set binds* ⊆ *lambdas* (*snd t*)
⟨*proof*⟩

lemma
shows *vals1*: *P prim* ∈ *vals p* $\Longrightarrow$ *prim* ∈ *prims p*
and *P prim* ∈ *valsC y* $\Longrightarrow$ *prim* ∈ *primsC y*
and *P prim* ∈ *valsV z* $\Longrightarrow$ *prim* ∈ *primsV z*
and $\forall z \in$ *set list*. *P prim* ∈ *valsV z* $\longrightarrow$ *prim* ∈ *primsV z*
and $\forall x \in$ *set* (*list2* :: (*var* × *lambda*) *list*) . *P prim* ∈ *vals* (*snd x*) $\longrightarrow$ *prim* ∈ *prims* (*snd x*)
and *P prim* ∈ *vals* (*snd* (*t*:: *var*×*lambda*)) $\Longrightarrow$ *prim* ∈ *prims* (*snd t*)
⟨*proof*⟩

lemma
shows *vals2*: *R l var* ∈ *vals p* $\Longrightarrow$ *var* ∈ *vars p*
and *R l var* ∈ *valsC y* $\Longrightarrow$ *var* ∈ *varsC y*
and *R l var* ∈ *valsV z* $\Longrightarrow$ *var* ∈ *varsV z*
and $\forall z \in$ *set list*. *R l var* ∈ *valsV z* $\longrightarrow$ *var* ∈ *varsV z*
and $\forall x \in$ *set* (*list2* :: (*var* × *lambda*) *list*) . *R l var* ∈ *vals* (*snd x*) $\longrightarrow$ *var* ∈ *vars* (*snd x*)
and *R l var* ∈ *vals* (*snd* (*t*:: *var*×*lambda*)) $\Longrightarrow$ *var* ∈ *vars* (*snd t*)
⟨*proof*⟩

lemma
shows *vals3*: *L l* ∈ *vals p* $\Longrightarrow$ *l* ∈ *lambdas p*
and *L l* ∈ *valsC y* $\Longrightarrow$ *l* ∈ *lambdasC y*
and *L l* ∈ *valsV z* $\Longrightarrow$ *l* ∈ *lambdasV z*
and $\forall z \in$ *set list*. *L l* ∈ *valsV z* $\longrightarrow$ *l* ∈ *lambdasV z*

and $\forall x\in$ *set* (*list2* :: (*var* × *lambda*) *list*) . *L l* ∈ *vals* (*snd x*) ⟶ *l* ∈ *lambdas* (*snd x*)
and *L l* ∈ *vals* (*snd* (*t*:: *var*×*lambda*)) ⟹ *l* ∈ *lambdas* (*snd t*)
⟨*proof*⟩

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*⟩

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

lemma *finite-Nlist*[*intro*]:
⟦ *finite A*; *finite N* ⟧ ⟹ *finite* (*NList A N*)
⟨*proof*⟩

definition *call-list-lengths*
where *call-list-lengths p* = {*0*,*1*,*2*,*3*} ∪ (λ*c. case c of* (*App - - ds*) ⇒ *length ds* | - ⇒ *0*) ‘ *calls p*

lemma *finite-call-list-lengths*[*simp*]: *finite* (*call-list-lengths p*)
⟨*proof*⟩

end

10. General utility lemmas

theory *Utils* **imports** *Main*
begin

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 *single-valued* predicate.

lemma *single-valued-empty*[*simp*]:*single-valued* {}
⟨*proof*⟩

lemma *single-valued-insert*:
assumes *single-valued rel*
and ⋀ *x y* . ⟦(*x*,*y*) ∈ *rel*; *x*=*a*⟧ ⟹ *y* = *b*
shows *single-valued* (*insert* (*a*,*b*) *rel*)
⟨*proof*⟩

Lemmas about *ran*, the range of a finite map.

lemma *ran-upd*: $ran\ (m\ (k \mapsto v)) \subseteq ran\ m \cup \{v\}$
⟨*proof*⟩

lemma *ran-map-of*: $ran\ (map\text{-}of\ xs) \subseteq snd\ `\ set\ xs$
⟨*proof*⟩

lemma *ran-concat*: $ran\ (m1\ ++\ m2) \subseteq ran\ m1 \cup ran\ m2$
⟨*proof*⟩

lemma *ran-upds*:
  assumes *eq-length*: $length\ ks = length\ vs$
  shows $ran\ (map\text{-}upds\ m\ ks\ vs) \subseteq ran\ m \cup set\ vs$
⟨*proof*⟩

lemma *ran-upd-mem*[*simp*]: $v \in ran\ (m\ (k \mapsto v))$
⟨*proof*⟩

Lemmas about *map*, *zip* and *fst*/*snd*

lemma *map-fst-zip*: $length\ xs = length\ ys \Longrightarrow map\ fst\ (zip\ xs\ ys) = xs$
⟨*proof*⟩

lemma *map-snd-zip*: $length\ xs = length\ ys \Longrightarrow map\ snd\ (zip\ xs\ ys) = ys$
⟨*proof*⟩

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 ({}., ∪.,...).

definition
  $sdom :: ('a => 'b\ set) => 'a\ set$ **where**
  $sdom\ m = \{a.\ m\ a\ \tilde{}= \{\}\}$

definition
  $sran :: ('a => 'b\ set) => 'b\ set$ **where**
  $sran\ m = \{b.\ \exists\, a.\ b \in m\ a\}$

lemma *sranI*: $b \in m\ a \Longrightarrow b \in sran\ m$

⟨*proof*⟩

lemma *sdom-not-mem*[*elim*]: $a \notin sdom\ m \Longrightarrow m\ a = \{\}$
⟨*proof*⟩

definition *smap-empty* (‹{}.›)
where $\{\}.\ k = \{\}$

definition *smap-union* :: ($'a$::*type* ⇒ $'b$::*type set*) ⇒ ($'a$ ⇒ $'b$ *set*) ⇒ ($'a$ ⇒ $'b$ *set*) (‹- ∪. -›)
where $smap1 \cup.\ smap2\ k = smap1\ k \cup smap2\ k$

primrec *smap-Union* :: ($'a$::*type* ⇒ $'b$::*type set*) *list* ⇒ $'a$ ⇒ $'b$ *set* (‹⋃.-›)
where [*simp*]:$\bigcup.\ [] = \{\}.$
| $\bigcup.\ (m\#ms) = m \cup. \bigcup.\ ms$

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* (‹-/ ⊆. -› [*50*, *51*] *50*)
where $smap\text{-}less\ m1\ m2 = (\forall k.\ m1\ k \subseteq m2\ k)$

lemma *sdom-empty*[*simp*]: $sdom\ \{\}. = \{\}$
⟨*proof*⟩

lemma *sdom-singleton*[*simp*]: $sdom\ \{k := vs\}. \subseteq \{k\}$
⟨*proof*⟩

lemma *sran-singleton*[*simp*]: $sran\ \{k := vs\}. = vs$
⟨*proof*⟩

lemma *sran-empty*[*simp*]: $sran\ \{\}. = \{\}$
⟨*proof*⟩

lemma *sdom-union*[*simp*]: $sdom\ (m \cup.\ n) = sdom\ m \cup sdom\ n$
⟨*proof*⟩

lemma *sran-union*[*simp*]: $sran\ (m \cup.\ n) = sran\ m \cup sran\ n$
⟨*proof*⟩

lemma *smap-empty*[*simp*]: $\{\}. \subseteq. \{\}.$
⟨*proof*⟩

lemma *smap-less-refl*: $m \subseteq. m$
⟨*proof*⟩

lemma *smap-less-trans*[*trans*]: $[\![m1 \subseteq. m2; m2 \subseteq. m3]\!] \Longrightarrow m1 \subseteq. m3$
⟨*proof*⟩

lemma *smap-union-mono*: $[\![ve1 \subseteq. ve1'; ve2 \subseteq. ve2']\!] \Longrightarrow ve1 \cup. ve2 \subseteq. ve1' \cup. ve2'$

⟨*proof*⟩

lemma *smap-Union-union*: $m1 \cup. \bigcup.ms = \bigcup.(m1\#ms)$
⟨*proof*⟩

lemma *smap-Union-mono*:
assumes *list-all2 smap-less ms1 ms2*
shows $\bigcup. \ ms1 \subseteq. \bigcup. \ ms2$
⟨*proof*⟩

lemma *smap-singleton-mono*: $v \subseteq v' \Longrightarrow \{k := v\}. \subseteq. \{k := v'\}.$
⟨*proof*⟩

lemma *smap-union-comm*: $m1 \cup. \ m2 = m2 \cup. \ m1$
⟨*proof*⟩

lemma *smap-union-empty1*[*simp*]: $\{\}. \cup. \ m = m$
⟨*proof*⟩

lemma *smap-union-empty2*[*simp*]: $m \cup. \{\}. = m$
⟨*proof*⟩

lemma *smap-union-assoc* [*simp*]: $(m1 \cup. \ m2) \cup. \ m3 = m1 \cup. \ (m2 \cup. \ m3)$
⟨*proof*⟩

lemma *smap-Union-append*[*simp*]: $\bigcup. \ (m1@m2) = (\bigcup. \ m1) \cup. \ (\bigcup. \ m2)$
⟨*proof*⟩

lemma *smap-Union-rev*[*simp*]: $\bigcup. \ (rev \ l) = \bigcup. \ l$
⟨*proof*⟩

lemma *smap-Union-map-rev*[*simp*]: $\bigcup. \ (map \ f \ (rev \ l)) = \bigcup. \ (map \ f \ l)$
⟨*proof*⟩

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 *Shivers−CFA.SetMap*. Both these sets are defined (*maps-over*, *smaps-over*) and the finiteness is shown.

definition *maps-over* :: $'a::type \ set \Rightarrow \ 'b::type \ set \Rightarrow ('a \rightharpoonup \ 'b) \ set$

where *maps-over* A B = $\{m.\ dom\ m \subseteq A \wedge ran\ m \subseteq B\}$

lemma *maps-over-empty*[*simp*]:
Map.empty $\in$ *maps-over* A B
⟨*proof*⟩

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
⟨*proof*⟩

lemma *maps-over-finite*[*intro*]:
assumes *finite* A **and** *finite* B **shows** *finite* (*maps-over* A B)
⟨*proof*⟩

definition *smaps-over* :: *'a::type set* $\Rightarrow$ *'b::type set* $\Rightarrow$ (*'a* $\Rightarrow$ *'b set*) *set*
where *smaps-over* A B = $\{m.\ sdom\ m \subseteq A \wedge sran\ m \subseteq B\}$

lemma *smaps-over-empty*[*simp*]:
$\{\}.\ \in$ *smaps-over* A B
⟨*proof*⟩

lemma *smaps-over-singleton*:
assumes $k \in A$ **and** $vs \subseteq B$
shows $\{k := vs\}.\ \in$ *smaps-over* A B
⟨*proof*⟩

lemma *smaps-over-un*:
assumes $m1 \in$ *smaps-over* A B **and** $m2 \in$ *smaps-over* A B
shows $m1 \cup.\ m2 \in$ *smaps-over* A B
⟨*proof*⟩

lemma *smaps-over-Union*:
assumes *set ms* $\subseteq$ *smaps-over* A B
shows $\bigcup.ms \in$ *smaps-over* A B
⟨*proof*⟩

lemma *smaps-over-im*:
$[\![\ f \in m\ a\ ;\ m \in$ *smaps-over* A $B\ ]\!] \Longrightarrow f \in B$
⟨*proof*⟩

lemma *smaps-over-finite*[*intro*]:
assumes *finite* A **and** *finite* B **shows** *finite* (*smaps-over* A B)
⟨*proof*⟩

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
  $x \sqsubseteq y \longleftrightarrow (x \longrightarrow y)$
instance ⟨*proof*⟩
end

instance *bool* :: *chfin*
⟨*proof*⟩

instance *bool* :: *pcpo*
⟨*proof*⟩

lemma *is-lub-bool*: $S <<| (True \in S)$
  ⟨*proof*⟩

lemma *lub-bool*: $lub\ S = (True \in S)$
  ⟨*proof*⟩

lemma *bottom-eq-False*[*simp*]: $\bot = False$
⟨*proof*⟩

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
  $A \sqsubseteq B \longleftrightarrow A \subseteq B$
instance ⟨*proof*⟩
end

lemma *sqsubset-is-subset*: $A \sqsubseteq B \longleftrightarrow A \subseteq B$
  ⟨*proof*⟩

lemma *is-lub-set*: $S <<| \bigcup S$
  ⟨*proof*⟩

lemma *lub-is-union*: *lub* $S = \bigcup S$
⟨*proof*⟩

instance *set* :: (*type*) *cpo*
⟨*proof*⟩

lemma *emptyset-is-bot*[*simp*]: $\{\} \sqsubseteq S$
⟨*proof*⟩

instance *set* :: (*type*) *pcpo*
⟨*proof*⟩

lemma *bot-bool-is-emptyset*[*simp*]: $\bot = \{\}$
⟨*proof*⟩

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* $(\lambda x.\ f\ x)$ **and** *g*: *cont* $(\lambda x.\ g\ x)$
shows *cont* $(\lambda x.\ f\ x \vee g\ x)$
⟨*proof*⟩

lemma *cont2cont-imp*[*simp*, *cont2cont*]:
assumes *f*: *cont* $(\lambda x.\ \neg\ f\ x)$ **and** *g*: *cont* $(\lambda x.\ g\ x)$
shows *cont* $(\lambda x.\ f\ x \longrightarrow g\ x)$
⟨*proof*⟩

lemma *cont2cont-Collect* [*simp*, *cont2cont*]:
assumes $\bigwedge y.$ *cont* $(\lambda x.\ f\ x\ y)$
shows *cont* $(\lambda x.\ \{y.\ f\ x\ y\})$
⟨*proof*⟩

lemma *cont2cont-mem* [*simp*, *cont2cont*]:
assumes *cont* $(\lambda x.\ f\ x)$
shows *cont* $(\lambda x.\ y \in f\ x)$
⟨*proof*⟩

lemma *cont2cont-union* [*simp*, *cont2cont*]:
cont $(\lambda x.\ f\ x) \Longrightarrow$ *cont* $(\lambda x.\ g\ x)$
$\Longrightarrow$ *cont* $(\lambda x.\ f\ x \cup g\ x)$
⟨*proof*⟩

lemma *cont2cont-insert* [*simp*, *cont2cont*]:
assumes *cont* $(\lambda x.\ f\ x)$
shows *cont* $(\lambda x.$ *insert* $y\ (f\ x))$
⟨*proof*⟩

lemmas *adm-subset* = *adm-below*[**where** *?'b* = *'a::type set*, *unfolded sqsubset-is-subset*]

lemma *cont2cont-UNION*[*cont2cont*,*simp*]:
  assumes *cont f*
    and $\bigwedge$ *y. cont* ($\lambda x.$ *g x y*)
  shows *cont* ($\lambda x.$ $\bigcup y \in$ *f x. g x y*)
⟨*proof*⟩

lemma *cont2cont-Let-simple*[*simp*,*cont2cont*]:
  assumes *cont* ($\lambda x.$ *g x t*)
  shows *cont* ($\lambda x.$ *let y* = *t in g x y*)
⟨*proof*⟩

lemma *cont2cont-case-list* [*simp*, *cont2cont*]:
  assumes $\bigwedge$*y. cont* ($\lambda x.$ *f1 x*)
    and $\bigwedge$*y z. cont* ($\lambda x.$ *f2 x y z*)
  shows *cont* ($\lambda x.$ *case-list* (*f1 x*) (*f2 x*) *l*)
⟨*proof*⟩

As with the continuity lemmas, we need admissibility lemmas.

lemma *adm-not-mem*:
  assumes *cont* ($\lambda x.$ *f x*)
  shows *adm* ($\lambda x.$ *y* $\notin$ *f x*)
⟨*proof*⟩

lemma *adm-id*[*simp*]: *adm* (λx . *x*)
⟨*proof*⟩

lemma *adm-Not*[*simp*]: *adm Not*
⟨*proof*⟩

lemma *adm-prod-split*:
  assumes *adm* ($\lambda p.$ *f* (*fst p*) (*snd p*))
  shows *adm* (λ(*x*,*y*). *f x y*)
⟨*proof*⟩

lemma *adm-ball'*:
  assumes $\bigwedge$ *y. adm* ($\lambda x.$ *y* $\in$ *A x* $\longrightarrow$ *P x y*)
  shows *adm* ($\lambda x.$ $\forall$ *y* $\in$ *A x* . *P x y*)
⟨*proof*⟩

lemma *adm-not-conj*:
  $[\![$*adm* ($\lambda x.$ $\neg$ *P x*); *adm* ($\lambda x.$ $\neg$ *Q x*)$]\!]$ $\Longrightarrow$ *adm* ($\lambda x.$ $\neg$ (*P x* $\wedge$ *Q x*))
⟨*proof*⟩

lemma *adm-single-valued*:
 assumes *cont* ($\lambda x.$ *f x*)

shows *adm* ($\lambda x.$ *single-valued* ($f\ x$))
⟨*proof*⟩

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

abbreviation *niceiterate* (‹(-^-)› [*1000*] *1000*)
where *niceiterate f i* $\equiv$ *iterate* $i\cdot f$

end

14. Fixed point transformations

theory *FixTransform*
imports *HOLCF*
begin

default-sort *type*

In his treatment of the computabily, 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 $\bigwedge x.\ g\cdot(f\cdot x)=x$
shows $\mathit{fix}\cdot F = g\cdot(\mathit{fix}\cdot(f\ \mathit{oo}\ F\ \mathit{oo}\ g))$
⟨*proof*⟩

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* :: $('a\ \mathit{discr} \to 'c) \times ('b\ \mathit{discr} \to 'c) \to ('a + 'b)\ \mathit{discr} \to 'c{::}\mathit{cpo}$
where *tup-to-sum* = (Λ *p s.* ($\lambda(f,g)$.
case undiscr s of Inl x $\Rightarrow f\cdot(\mathit{Discr}\ x)$
| *Inr x* $\Rightarrow g\cdot(\mathit{Discr}\ x)$) *p*)

definition *sum-to-tup* :: $(('a + 'b)\ \mathit{discr} \to 'c) \to ('a\ \mathit{discr} \to 'c) \times ('b\ \mathit{discr} \to 'c{::}\mathit{cpo})$
where *sum-to-tup* = (Λ *f.* (Λ *x.* $f\cdot(\mathit{Discr}\ (\mathit{Inl}\ (\mathit{undiscr}\ x)))$,
Λ *x.* $f\cdot(\mathit{Discr}\ (\mathit{Inr}\ (\mathit{undiscr}\ x)))$))

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

lemma *cont2cont-case-sum*[*simp*,*cont2cont*]:
  assumes *cont f* **and** *cont g*
  shows $cont\ (\lambda x.\ case\text{-}sum\ (f\ x)\ (g\ x)\ s)$
⟨*proof*⟩

lemma *cont2cont-circ*[*simp*,*cont2cont*]:
 $cont\ (\lambda f.\ f \circ g)$
⟨*proof*⟩

lemma *cont2cont-split-pair*[*cont2cont*,*simp*]:
 assumes *f1*: *cont f*
  and *f2*: $\bigwedge x.\ cont\ (f\ x)$
  and *g1*: *cont g*
  and *g2*: $\bigwedge x.\ cont\ (g\ x)$
 shows $cont\ (\lambda(a,\ b).\ (f\ a\ b,\ g\ a\ b))$
⟨*proof*⟩

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*:
 $sum\text{-}to\text{-}tup \cdot f = (\Lambda\ x.\ f \cdot (Discr\ (Inl\ (undiscr\ x))),\ \Lambda\ x.\ f \cdot (Discr\ (Inr\ (undiscr\ x))))$
⟨*proof*⟩

lemma *tup-to-sum-app*:
 $tup\text{-}to\text{-}sum \cdot p = (\Lambda\ s.\ (\lambda(f,g).$
  $case\ undiscr\ s\ of\ Inl\ x \Rightarrow f \cdot (Discr\ x)$
  $\mid Inr\ x \Rightarrow g \cdot (Discr\ x))\ p)$
⟨*proof*⟩

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

lemma *discr-app*[*simp*]:
 $(\Lambda\ s.\ f\ s) \cdot (Discr\ x) = f\ (Discr\ x)$
⟨*proof*⟩

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 $sum\text{-}to\text{-}tup \cdot (tup\text{-}to\text{-}sum \cdot F) = F$
⟨*proof*⟩

lemma *fix-transform-pair-sum*:
  shows $fix \cdot F = sum\text{-}to\text{-}tup \cdot (fix \cdot (tup\text{-}to\text{-}sum\ oo\ F\ oo\ sum\text{-}to\text{-}tup))$
⟨*proof*⟩

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

lemma *tup-sum-oo*[*simp*]:
 assumes *f1*: *cont F*
  and *f2*: $\bigwedge$ *x*. *cont* (*F x*)
  and *g1*: *cont G*
  and *g2*: $\bigwedge$ *x*. *cont* (*G x*)
shows *tup-to-sum oo* (Λ *p*. (λ(*a*, *b*). (*F a b*, *G a b*)) *p*) *oo sum-to-tup*
 = (Λ *f s*. (*case undiscr s of*
   Inl x $\Rightarrow$
    F (Λ *s*. *f*·(*Discr* (*Inl* (*undiscr s*))))
     (Λ *s*. *f*·(*Discr* (*Inr* (*undiscr s*))))·
    (*Discr x*)
   | *Inr x* $\Rightarrow$
    G (Λ *s*. *f*·(*Discr* (*Inl* (*undiscr s*))))
     (Λ *s*. *f*·(*Discr* (*Inr* (*undiscr s*))))·
    (*Discr x*)))
⟨*proof*⟩

lemma *fst-sum-to-tup*[*simp*]:
 fst (*sum-to-tup*·*x*) = (Λ *xa*. *x*·(*Discr* (*Inl* (*undiscr xa*))))
⟨*proof*⟩

end

References

[1] J. Breitner. Control flow in functional languages. Student research project, Karlsruher Institut für Technologie (KIT), November 2010.

[2] J. Breitner. Implementation of Shivers' Control-Flow Analysis. http://hackage.haskell.org/package/shivers-cfg-0.1, November 2010.

[3] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, 1991.