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

II. The main results 16
  5. The exact call cache is a map 16
     5.1. Preparations ........................................ 16
     5.2. The proof ........................................... 18
  6. The abstract semantics is correct 19
     6.1. Abstraction functions ............................... 19
     6.2. Lemmas about abstraction functions ................. 21
     6.3. Approximation relation .............................. 21
     6.4. Lemmas about the approximation relation ............ 22
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.

```plaintext
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

**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 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 [(I3,"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"'))]))))])])])])])])])])])])])])])])])])])]
```

end

2. Standard semantics

**theory** Eval
  **imports** HOLCF HOLCFUtils CPSScheme
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 \textit{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 \textit{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{verbatim}
type-synonym contour = nat
type-synonym benv = label ➔ contour
type-synonym closure = lambda × benv
\end{verbatim}

The set of semantic values consist of the integers, closures, primitive operations and a special value \textit{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.

\begin{verbatim}
datatype d = DI int | DC closure | DP prim | Stop
type-synonym venv = var × contour ➔ d
\end{verbatim}

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.

\begin{verbatim}
fun evalV :: val ⇒ benv ⇒ venv ⇒ d (A)
  where A (C - i) β ve = DI i
       | A (P prim) β ve = DP prim
\end{verbatim}
\[
A (R \cdot \text{var}) \beta \text{ ve } = \neg\neg \neg
\begin{align*}
\text{case } \beta \text{ (binder var) of} \\
\text{Some } l \Rightarrow (\text{case } \text{ve} \text{ (var,l) of Some } d \Rightarrow d)) \\
\end{align*}
\]

\[
A (L \text{ lam}) \beta \text{ ve } = DC (\text{lam}, \beta)
\]

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 \text{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]:
\[
\begin{align*}
\text{assumes } \forall a b c. \text{cont } (\lambda x. f x a b c) \\
\text{shows } \text{cont } (\lambda x. \text{case-lambda } (f x) l)
\end{align*}
\]
\(\langle \text{proof} \rangle\)

lemma cont2cont-case-d [simp, cont2cont]:
\[
\begin{align*}
\text{assumes } \forall y. \text{cont } (\lambda x. f1 x y) \\
\text{and } \forall y. \text{cont } (\lambda x. f2 x y) \\
\text{and } \forall y. \text{cont } (\lambda x. f3 x y) \\
\text{and } \text{cont } (\lambda x. f4 x) \\
\text{shows } \text{cont } (\lambda x. \text{case-d } (f1 x) (f2 x) (f3 x) (f4 x) d)
\end{align*}
\]
\(\langle \text{proof} \rangle\)

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

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

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 \text{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 $\rightarrow$ 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** $fstate = (d \times d\ list \times venv \times contour)$

**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 $evalF,fstate = (case undiscr fstate of$

$(DC (\text{Lambda} \ lab \ vs \ c, \ \beta), \ as, \ ve, \ b) \Rightarrow$

(if length vs = length as

then let $\beta' = \beta (lab \mapsto \mapsto b)$;

$ve' = \text{map-upds} \ ve (\mapsto \mapsto v.(v,b))$ as

in $C(Discr (c,\beta',ve',b))$

else $\bot)$

| $(DP (\text{Plus} \ c),[DI a1, DI a2, cnt],ve,b) \Rightarrow$

let $b' = Suc \ b$;

$\beta = [c \mapsto b]$

in $F(Discr (cnt,[DI (a1 + a2)],ve,b'))$

| $(DP (\text{prim.If} \ ct \ cf),,[DI v, \ contt, \ contf],ve,b) \Rightarrow$

(if $v \neq 0$

then let $b' = Suc \ b$;

$\beta = [ct \mapsto b]$

in $F(Discr (contt,[]),ve,b'))$

else let $b' = Suc \ b$;

$\beta = [cf \mapsto b]$

in $F(Discr (contf,[]),ve,b'))$

| $(\text{Stop} ,[DI i],\ldots) \Rightarrow \text{Def} \ i$

| $\bot \Rightarrow \bot$)

| $C\cdot cstate = (case undiscr cstate of$

$(App \ lab \ f vs,\beta,ve,b) \Rightarrow$

let $f' = A f \beta ve$;

$as = \text{map} (\lambda v. A v \beta ve) vs$;

$b' = Suc \ b$

in $F(Discr (f',as,ve,b'))$

| $(Let \ lab \ ls \ c',\beta,ve,b) \Rightarrow$

let $b' = Suc \ b$;

$\beta' = \beta (lab \mapsto b')$;

$ve' = ve ++ \text{map-of} (\mapsto \mapsto \mapsto (\lambda(v,l). ((v,b'), A (L l) \beta' ve)) ls)$
To evaluate a full program, it is passed to $F$ 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 ($\mathcal{P}R$)  
**where**  
$\mathcal{P}R$ 1 = (let ve = empty;  
  $\beta$ = empty;  
  $f$ = $A$ (L 1) $\beta$ ve  
  in $F$:$\{\text{Discr} \ (f,\text{[Stop]},ve,0)\}$

**lemma** correct-ex1: $\mathcal{P}R$ ex1 = Def 0  
(proof)

**lemma** correct-ex2: $\mathcal{P}R$ ex2 = Def 2  
(proof)

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 \(b_0 = \text{Abs-contour} \emptyset\)

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

instantiation \(\text{contour} : \text{preorder}\)
begin
definition le-contour-def:
  \(b \leq b' \iff \text{length} (\text{Rep-contour} b) \leq \text{length} (\text{Rep-contour} b')\)

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

instance (proof)
end

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

lemma nb-le-less [iff]: \(nb\ b\ c \leq b' \iff b < b'\)
(proof)

lemma nb-less [iff]: \(b' < nb\ b\ c \iff b' \leq b\)
(proof)

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

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

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

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

datatype \(d = \text{DI} \text{int} \mid \text{DC} \text{closure} \mid \text{DP} \text{prim} \mid \text{Stop}\)

type-synonym \(\text{venv} = \text{var} \times \text{contour} \to d\)

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:
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.

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.

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

(proof)

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

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 \( \text{STOP} \) 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 \text{benv}) \times d) \text{set}
type-synonym ans = ccache
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})

fixrec evalF :: fstate discr \rightarrow ans (F)
and evalC :: cstate discr \rightarrow ans (C)
where \( F \cdot \)fstate = (case undiscr \cdot \)fstate of
  \( \text{DC} \ (\text{Lambda} \ \text{lab} \ \text{vs} \ \text{as} \ \text{ve} \ \text{b}) \Rightarrow \)
  (if length vs = length as
    then let \( \beta' = \beta \ (\text{lab} \mapsto b) \);
    \( \text{ve'} = \text{map-upds} \ \text{ve} \ (\text{map} \ (\lambda v. (v,b)) \ \text{vs}) \) as
    in C \cdot \text{Discr} \ (\text{e}, \beta', \text{ve}', b))
  else \bot)
  \mid (\text{DP} \ (\text{Plus} \ c)\ [\text{DI} \ a1, \text{DI} \ a2, \text{cnt}], \text{ve}, b) \Rightarrow \)
  (if isProc cnt
    then let \( b' = \text{nb} \ b \ c; \)
    \( \beta = [c \mapsto b] \)
    in \( F \cdot (\text{Discr} \ (\text{cnt}, [\text{DI} \ (a1 + a2)], \text{ve}, b')) \)
    \cup \{(\text{e}, \beta, \text{cnt})\}
  else \bot)
  \mid (\text{DP} \ (\text{prim.If} \ \text{ct} \ \text{cf})\ [\text{DI} \ v, \text{contt}, \text{contf}], \text{ve}, b) \Rightarrow \)
  (if isProc contt \land isProc contf
    then
      (if \( v \neq 0 \)
        then let \( b' = \text{nb} \ b \ \text{ct}; \)
        \( \beta = [\text{ct} \mapsto b] \)
        in \( F \cdot (\text{Discr} \ (\text{contt}, []), \text{ve}, b')) \)
        \cup \{(\text{ct}, \beta, \text{contt})\}
      else let \( b' = \text{nb} \ b \ \text{cf}; \)
\[\beta = [cf \mapsto b]\]
\[\text{in } (F : (\text{Discr} \ (\text{contf} , [], ve , b'))) \cup \{((cf , \beta), \text{contf})\}\]

\[
\begin{array}{c}
\text{else } \bot \\
| \text{Stop.[DI i] , \_ , \_ } \Rightarrow \{\}
\end{array}
\]

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 \textit{fstate} into the various cases that were used in the definition of \textit{F}.

\textbf{lemmas} \textit{evalF-evalC-induct} = \textit{evalF-evalC.induct}[\text{case-names Admissibility Bottom Next}]

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

\textbf{lemmas} \textit{ds-cases-plus} = \textit{list.exhaust}[
\begin{align*}
\text{OF} & \ - \text{d.exhaust}, \text{of} - - \lambda a - . \ a, \\
\text{OF} & \ - \text{list.exhaust}, \text{of} - - \lambda - \ x - . \ x, \\
\text{OF} & \ - - \text{d.exhaust}, \text{of} - - \lambda - - a - . \ a, \\
\text{OF} & \ - - \text{list.exhaust}, \text{of} - - \lambda - - - x - . \ x, \\
\text{OF} & \ - - - \text{list.exhaust}, \text{of} - - - \lambda - - - - - x, \\
\end{align*}
]

\textbf{lemmas} \textit{ds-cases-if} = \textit{list.exhaust}[OF \text{d.exhaust}, \text{of} - - \lambda a - . \ a, \\
\text{OF} & \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - \text{list.exhaust}[OF \ - - - - x, \\
\text{OF} & \ - \text{cl-cases prim.exhaust}, \text{of} - - \lambda - - - - a . \ a \lambda - - - - a, \\
\text{OF} & \ - \text{case-split ds-cases-plus ds-cases-if ds-cases-stop}, \\
\text{of} & \ - \lambda - \ as - - - - vs - . \ length vs = length as \lambda - ds - - - - . \ ds \lambda - ds - - - - . \ ds \lambda - ds - - - - . \ ds, \\
\text{case-names} \ x \ Closure \ xx \ xx \ xx \ Plus \ xxxxxx xx \ xx \ If-True \ If-False \ xx \ xx \ x \ Stop \ xx \ xx \ xx \ x]
\]

11
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 = empty;
$\beta$ = empty;
$\mathcal{f} = A (L l) \beta$ ve
in $\mathcal{F}$(Discr ($\mathcal{f}$,$\text{[Stop]}$,ve,$b_0$)))

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

(proof)

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

(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 $\Rightarrow$ label $\Rightarrow$ 'a ($\tilde{nb}$)
and a-initial-contour :: 'a ($\tilde{b_0}$)
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 \texttt{SetMap} contains functions and lemmas to work with such maps, symbolized by an appended dot (e.g. \{\}, \cup).

type-synonym \('c a-benv = label \rightarrow \ 'c \ (- \ benv [1000])\)
type-synonym \('c a-closure = lambda \times \ 'c benv (- \ closure [1000])\)

datatype \('c proc (- \ proc [1000])\)
= \(\text{PC} \ 'c \ closure\)
| \(\text{PP} \ prim\)
| \(\text{AStop}\)

type-synonym \('c a-d = \ 'c \ proc \ set (- \ d [1000])\)
type-synonym \('c a-venv = \text{var} \times \ 'c \Rightarrow \ 'c \ d (- \ venv [1000])\)

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

fun \(\text{evalV-a} : \text{val} \Rightarrow \ 'c \ benv \Rightarrow \ 'c \ venv \Rightarrow \ 'c \ d \ (\tilde{A})\)
where \(\tilde{A} \ (C - i) \ \beta \ \text{ve} = \{\}\)
| \(\tilde{A} \ (P \ \text{prim}) \ \beta \ \text{ve} = \{PP \ prim\}\)
| \(\tilde{A} \ (R - \text{var}) \ \beta \ \text{ve} = \{\text{case} \ \beta \ (\text{binder} \ \text{var}) \ \text{of}
\quad \text{Some} \ l \Rightarrow \ \text{ve} \ (\text{var}, l)
\quad \text{None} \Rightarrow \{\}\}\)
| \(\tilde{A} \ (L \ \text{lam}) \ \beta \ \text{ve} = \{PC \ (\text{lam}, \ \beta)\}\)

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

type-synonym \('c a-ccache = ([label \times \ 'c \ benv] \times \ 'c \ proc) \ set (- \ ccache [1000])\)
type-synonym \('c a-ans = \ 'c \ ccache (- \ ans [1000])\)
And yet again, cont2cont results need to be shown for our custom data types.

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

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

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 \( \cont (\lambda x. \case-call (f1 x) (f2 x) c) \)
⟨proof⟩

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 \( \cont (\lambda x. \case-prim (f1 x) (f2 x) p) \)
⟨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 \( \text{AStop} \) case, \( {} \) is returned, while for wrong arguments, \( \bot \) is returned. Both actually represent the same value, the empty set, so this is just an aesthetic difference.

fixrec a-evalF :: 'c::contour \fad State\ discr \rightarrow \bint\ \hats{F}
and a-evalC :: 'c::contour \estate\ discr \rightarrow \bint\ \hats{C}
where \( \hats{F}\cdot \fad State = \case undiscr \fad State of \)
  \( \text{(PC (Lambda lab vs c, \beta, as, ve, b) \rightarrow} \)
  \( \text{if length vs = length as} \)
  \( \text{then let } \beta' = \beta (\text{lab } \mapsto b); \)
  \( \text{ve'} = ve \cup (\bigcup \text{map } (\lambda(v,a). \{(v,b) := a\}) (\text{zip vs as}))) \)
  \( \text{in } \hats{C}\cdot (\text{Discr (c,\beta',ve',b)}) \)
  \( \text{else } \bot \)
  \( \big| \text{(PP (Plus c),cnts,ve,b) } \rightarrow \)
  \( \text{let } b' = n\beta b c; \)
  \( \beta = [c \mapsto b]; \)
  \( \text{in } (\bigcup \text{cnt} \in \text{cnts}. \hats{F}\cdot (\text{Discr (cnt,\{\},ve,b'}))) \)
Again, we name the cases of the induction rule and build a nicer case analysis rule for arguments of type \( f\text{-state} \).

\textbf{lemmas} a-evalF-evalC-induct = a-evalF-a-evalC.induct[case-names Admissibility Bottom Next]

\textbf{fun} a-evalF-cases

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

\textbf{lemmas} a-fstate-case-x = a-evalF-cases.cases[

OF case-split, of - \text{\lambda} - vs - - as - -, length vs = length as,
case-names Closure Closure-inv Plus If Stop]

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

\textbf{lemmas} a-ds-cases = list.exhaust[

OF - list.exhaust, of - - \text{\lambda} - x, x,
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.

\[
\text{definition} \quad \text{evalCPS-a} :: \text{prog} \Rightarrow (\text{contour} \ w) \ \widehat{\text{ans}} (\widehat{P}R)
\]

\[
\text{where} \quad \widehat{P}R \ l = (\text{let ve} = \{\}; \beta = \text{empty}; \ f = \hat{A} (L l) \ \beta \ \text{ve}) \quad \text{where} \quad \in \ \hat{F} \cdot (\text{Discr (the-elem f,[\{AStop\}],ve,\hat{b_0})))
\]

\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 = \bigcup \{benv-in-d d \mid 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 = \bigcup \{contours-in-d d \mid 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 \mapsto \mapsto) ds). Q β  
  (proof)

lemma benv-in-eval:  
  assumes ∀ β′∈benv-in-ve ve. Q β′  
  and Q β  
  shows ∀ β ∈ benv-in-d (A v β ve). Q β  
  (proof)

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

lemma contours-in-ve-upds:  
  assumes eq-length: length vs = length ds  
  and ∀ b′∈contours-in-ve ve. Q b′  
  and ∀ d′∈set ds. ∀ b′∈contours-in-d d′. Q b′  
  shows ∀ b′∈contours-in-ve (ve(map (λv. (v, b′′)) vs \mapsto \mapsto) ds). Q b′  
  (proof)

lemma contours-in-ve-upds-binds:  
  assumes ∀ b′∈contours-in-ve ve. Q b′  
  and ∀ b′∈ran β. Q b′  
  shows ∀ b′∈contours-in-ve (ve ++ map-of (map (λ(v,l). ((v,b′′), A (L l β ve)) ls)). Q b′  
  (proof)

lemma contours-in-eval:
assumes \( \forall b' \in \text{contours-in-ve} \). \( Q b' \)
and \( \forall b' \in \text{ran} \beta. \ Q b' \)
shows \( \forall b' \in \text{contours-in-d} (A f \beta \vee). \ Q b' \)
(proof)

5.2. The proof

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

We would like to show an auxiliary result about the contour counter passed to \( F \) and \( C \) (such that it is an unused counter when passed to \( F \) 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, \textit{HOLCF} provides a number of introduction lemmas that, together with some additions in \textit{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 \( F \) and \( C \) and use the induction hypotheses.

\textbf{lemma cc-single-valued':}
\[
\begin{align*}
\text{assumes} & \quad \forall b' \in \text{contours-in-ve} \ . \ b' < b \\
& \quad \forall b' \in \text{contours-in-d} \ d. \ b' < b \\
& \quad \forall d' \in \text{set} \ ds. \ \forall b' \in \text{contours-in-d} \ d'. \ b' < b \\
\text{shows} & \quad \forall b' \in \text{contours-in-d} (A f \beta \vee). \ Q b' \\
\text{(proof)} & \quad \rightarrow
\end{align*}
\]
\[
\begin{align*}
\left( \begin{array}{l}
\text{single-valued} \ (F \cdot (\text{Discr} \ (d,ds,ve,b))) \\
\land (\forall \ ((\text{lab},\beta),t) \in F \cdot (\text{Discr} \ (d,ds,ve,b)). \ \exists \ b'. \ b' \in \text{ran} \beta \land b \leq b')
\end{array} \right)
\end{align*}
\]
\textbf{and}
\[
\begin{align*}
\text{assumes} & \quad b \in \text{ran} \beta' \\
& \quad \forall b' \in \text{ran} \beta'. \ b' \leq b \\
& \quad \forall b' \in \text{contours-in-ve} \ . \ b' \leq b \\
\text{shows} & \quad \forall (\text{lab},\beta),t \in \text{C} \cdot \ (\text{Discr} \ (e,\beta',ve,b)). \ \exists \ b'. \ b' \in \text{ran} \beta \land b \leq b'
\end{align*}
\]
\textbf{and}
\[
\begin{align*}
\left( \begin{array}{l}
\text{single-valued} \ (C \cdot (\text{Discr} \ (e,\beta',ve,b))) \\
\land (\forall \ ((\text{lab},\beta),t) \in C \cdot (\text{Discr} \ (e,\beta',ve,b)). \ \exists b'. \ b' \in \text{ran} \beta \land b \leq b')
\end{array} \right)
\end{align*}
\]
\textbf{(proof)}
lemma single-valued (PR prog) (proof)
end

6. The abstract semantics is correct

theory AbsCFCorrect
  imports AbsCF ExCF ~/src/Tools/Adhoc-Overloading
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 b0 symbols.

class contour-a = contour +
  fixes abs-cnt :: contour ⇒ 'a
  assumes abs-cnt-nb[simp]: abs-cnt (nb b lab) = ˆnb (abs-cnt b) lab
  and abs-cnt-initial[simp]: abs-cnt(b0) = b0

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 $\text{abs-<type>}$ 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.

\begin{verbatim}
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 = (\{((e,abs-benv β), p) | p . p∈abs-d d\})

adhoc-overloading
abs abs-ccache
\end{verbatim}
fun abs-fstate :: state => '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|)

adhoc-overloading
  abs abs-cstate

6.2. Lemmas about abstraction functions

Some results about the abstractions functions.

lemma abs-benv-empty[simp]: |empty| = empty
  ⟨proof⟩

lemma abs-benv-upd[simp]: |β(c->b)| = |β| (c -> |b|)
  ⟨proof⟩

lemma the-elem-is-Proc:
  assumes isProc cnt
  shows the-elem |cnt| ∈ |cnt|
  ⟨proof⟩

lemma [simp]: |{c}| = {} ⟨proof⟩

lemma abs-cache-singleton [simp]: |{(c,β,d)}| = |{(c, |β|, p) |p, p ∈ |d|}|
  ⟨proof⟩

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

6.3. Approximation relation

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

consts approx :: 'a => 'a => bool (- ≼ -)

definition venv-approx :: 'c venvv => 'c venvv => bool
  where venv-approx = smap-less

adhoc-overloading
  approx venv-approx

definition ccache-approx :: 'c ccache => 'c ccache => bool
where \texttt{ccache-approx} = \texttt{less-eq}

ad hoc-overloading
approx \texttt{ccache-approx}

\textbf{definition} \texttt{d-approx} :: 'c \hat{d} \Rightarrow 'c \hat{d} \Rightarrow \texttt{bool}
where \texttt{d-approx} = \texttt{less-eq}

ad hoc-overloading
approx \texttt{d-approx}

\textbf{definition} \texttt{ds-approx} :: 'c \hat{d} \texttt{list} \Rightarrow 'c \hat{d} \texttt{list} \Rightarrow \texttt{bool}
where \texttt{ds-approx} = \texttt{list-all2 d-approx}

ad hoc-overloading
approx \texttt{ds-approx}

\textbf{inductive} \texttt{fstate-approx} :: 'c \hat{fstate} \Rightarrow 'c \hat{fstate} \Rightarrow \texttt{bool}
where \[
\begin{array}{l}
\left[ \left[ \texttt{ve} \preceq \texttt{ve}'; \texttt{ds} \preceq \texttt{ds}' \right] \right] \\
\Rightarrow \texttt{fstate-approx} \left( \texttt{proc}, \texttt{ds}, \texttt{ve}, \texttt{b} \right) \left( \texttt{proc}, \texttt{ds}', \texttt{ve}', \texttt{b} \right)
\end{array}
\]
ad hoc-overloading
approx \texttt{fstate-approx}

\textbf{inductive} \texttt{cstate-approx} :: 'c \hat{cstate} \Rightarrow 'c \hat{cstate} \Rightarrow \texttt{bool}
where \[
\begin{array}{l}
\left[ \texttt{ve} \preceq \texttt{ve}' \right] \\
\Rightarrow \texttt{cstate-approx} \left( \texttt{c}, \texttt{\beta}, \texttt{ve}, \texttt{b} \right) \left( \texttt{c}, \texttt{\beta}, \texttt{ve}', \texttt{b} \right)
\end{array}
\]
ad hoc-overloading
approx \texttt{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.

\textbf{lemma} \texttt{venv-approx-trans[trans]}:
\textit{\textbf{fixes} \texttt{ve1 ve2 ve3 :: 'c \hat{venv}}
\textit{\textbf{shows} \left[ \left[ \texttt{ve1} \preceq \texttt{ve2}; \texttt{ve2} \preceq \texttt{ve3} \right] \right] \Rightarrow \left( \texttt{ve1} \preceq \texttt{ve3} \right)}
\textit{\textbf{proof}}

\textbf{lemma} \texttt{abs-venv-union}:
\textit{\texttt{|ve1 ++ ve2|} \preceq \texttt{|ve1|} \cup \texttt{|ve2|}}
\textit{\textbf{proof}}

\textbf{lemma} \texttt{abs-venv-map-of-rev}:
\textit{\texttt{|map-of (rev l)|} \preceq \texttt{|map (\lambda (v,k). ||v \mapsto k||) l|}}
\textit{\textbf{proof}}

\textbf{lemma} \texttt{abs-venv-map-of}:
\textit{\texttt{|map-of l|} \preceq \texttt{|map (\lambda (v,k). ||v \mapsto k||) l|}}
\textit{\textbf{proof}}

22
lemma abs-venv-singleton: \([(v,b) \mapsto d]\) = \{(v,|b|) := |d|\}.
(\langle proof \rangle)

lemma ccache-approx-empty[simp]:
  fixes x :: 'c ccache
  shows \(\{\} \subseteq x\)
  (\langle proof \rangle)

lemmas ccache-approx-trans[trans] = subset-trans[where 'a = ((label × 'c benv) × 'c proc),
fold ccache-approx-def]
lemmas Un-mono-approx = Un-mono[where 'a = ((label × 'c benv) × 'c proc), folded ccache-approx-def]
lemmas Un-upper1-approx = Un-upper1[where 'a = ((label × 'c benv) × 'c proc), folded ccache-approx-def]
lemmas Un-upper2-approx = Un-upper2[where 'a = ((label × 'c benv) × 'c proc), folded ccache-approx-def]

lemma abs-ccache-union: \(c1 \cup c2\) \(\subseteq\) \(|c1| \cup |c2|\)
(\langle proof \rangle)

lemma d-approx-empty[simp]: \(\{\} \subseteq (d::'c d)\)
(\langle proof \rangle)

lemma ds-approx-empty[simp]: \([],\[] \subseteq \[]\)
(\langle proof \rangle)

6.5. Lemma 7

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

lemma lemma7:
  assumes \(|ve::venv| \subseteq ve-a\)
  shows \(|A f \beta ve| \subseteq \hat{A} f |\beta| ve-a\)
  (\langle proof \rangle)

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 (λx. abs-ccache(f x))
  (\langle proof \rangle)
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 fstate and estate-a :: 'c::contour-a estate
shows |fstate| \(\subseteq\) fstate-a \(\Rightarrow\) \(\hat{F}((\text{Discr fstate}))\) \(\subseteq\) \(\hat{F}((\text{Discr fstate-a}))\)
and |estate| \(\subseteq\) estate-a \(\Rightarrow\) \(\hat{C}((\text{Discr estate}))\) \(\subseteq\) \(\hat{C}((\text{Discr estate-a}))\)

(proof)

And finally, we lift this result to \(\hat{PR}\) and \(PR\).

**lemma** lemma6: \(|PR| \subseteq \hat{PR}\)
(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 \(\hat{PR}\).

**7.1. Non-branching case**

After the following lemma (which could go into Set-Interval), we show Shivers' Theorem 10. This says that the least fixed point of the equation 

\[ f(x) = g(x) \cup f(r^i(x)) \]

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 \((n\cdot\text{nat})\) \{\(n<..\}\) = \{\(n..\}\)
(proof)
lemma theorem10:
  fixes g :: 'a::cpo ⇒ 'b::type set and r :: 'a ⇒ '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.

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

the equation becomes

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

(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 ⇒ 'b::type set) ⇒ 'a set ⇒ 'b set
  where \fix\cdot(\Lambda S. (\bigcup y \in S. f\cdot y))

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

lemma powerset-lift-union[simp]:
  \fix\cdot(\bigcup A \cup B) = \fix\cdot A \cup \fix\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. g\cdot(A x)) = g\cdot(\bigcup x \in S. A x)
⟨proof⟩

lemma powerset-lift-iterate-UNION:
  (\bigcup x \in S. (g)^i\cdot(A x)) = (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 $f :: 'a \to 'b\text{ set}$ and $g :: 'a \to 'b\text{ set}$ and $R :: 'a \to 'a\text{ set}$
- assumes $\bigwedge x. f\cdot x = g\cdot x \cup (\bigcup y \in R\cdot x. f\cdot y)$
- shows $f\cdot S = g\cdot S \cup f(R\cdot S)$

**proof**

Theorem 10 as it will be used in Theorem 12.

**lemmas theorem10ps = theorem10[of g r] for g r**

Now we can show Lemma 12: If $F$ is the least solution to the recursive power-set equation, then $x \mapsto F\cdot x$ 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 $g :: 'a::\text{discrete-cpo} \to 'b::\text{type set}$ and $R :: 'a \to 'a\text{ set}$
- assumes $F\cdot fix\cdot F = fix\cdot(\Lambda x. g\cdot x \cup F\cdot(R\cdot x))$
- shows $fix\cdot(\Lambda f\cdot x. g\cdot x \cup (\bigcup y \in R\cdot x. f\cdot y)) = (\Lambda x. F\cdot\{x\})$

**proof**

**lemma theorem12:**

- fixes $g :: 'a::\text{discrete-cpo} \to 'b::\text{type set}$ and $R :: 'a \to 'a\text{ set}$
- shows $fix\cdot(\Lambda f\cdot x. g\cdot x \cup (\bigcup y \in R\cdot x. f\cdot y))\cdot x = g\cdot(\bigcup i.((R)^i\cdot\{x\}))$

**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 $\widehat{F}$ and $\widehat{C}$: 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 $\widehat{f\cdot state}$ and $c\cdot state$, i.e. we treat the two mutually recursive functions now as one.

$abs\cdot g$ gives the local result for the given argument.

**fixrec abs-g :: ('c::contour f\cdot state + 'c c\cdot state) discr \to 'c ans**
where \( \text{abs-}\text{-g}\cdot x = (\text{case undiscr } x \text{ of} \)

\[
\begin{align*}
& (\text{Inl} \ (\text{PC} \ (\text{Lambda lab vs } c, \beta), \text{as}, \text{ve}, b)) \Rightarrow \\
& \quad \text{let } b' = \overrightarrow{n}b \ b \ c; \\
& \quad \beta = [c \mapsto b] \\
& \quad \text{in } \{(c, \beta), \text{cont} \mid \text{cont} . \text{cont} \in \text{cnts}\}
\end{align*}
\]

\[
\begin{align*}
& (\text{Inl} \ (\text{PP} \ (\text{Plus } c),\ldots,\text{cnts},\text{ve},b)) \Rightarrow \\
& \quad \text{let } b' = \overrightarrow{n}b \ b \ c; \\
& \quad \beta = [c \mapsto b] \\
& \quad \text{in } \{(c, \beta), \text{cont} \mid \text{cnt} . \text{cont} \in \text{cnts}\}
\end{align*}
\]

\[
\begin{align*}
& (\text{Inl} \ (\text{PP} \ (\text{prim.If ct cf}),\ldots,\text{cnts},\text{ve},b)) \Rightarrow \\
& \quad (\text{let } b' = \overrightarrow{n}b \ b \ ct; \\
& \quad \beta = [ct \mapsto b] \\
& \quad \text{in } \{(ct, \beta), \text{cnt} \mid \text{cnt} . \text{cnt} \in \text{cnts}\}
\end{align*}
\]

\[
\begin{align*}
& (\text{Inl} \ (A\text{Stop},\ldots,\ldots)) \Rightarrow \{\} \\
& (\text{Inl} \ :') \Rightarrow \perp \\
& (\text{Inr} \ (\text{App lab f vs},\beta,\text{ve},b)) \Rightarrow \\
& \quad \text{let } fs = \overrightarrow{A} \ f \ \beta \ \text{ve}; \\
& \quad \text{as} = \text{map}(\lambda v. \overrightarrow{A} \ v \ \beta \ \text{ve}) \ \text{vs}; \\
& \quad b' = \overrightarrow{n}b \ b \ \text{lab} \\
& \quad \text{in } \{((lab, \beta), f') \mid f' . f' \in \text{fs}\}
\end{align*}
\]

\[
\begin{align*}
& (\text{Inr} \ (\text{Let lab ls } c',\beta,\text{ve},b)) \Rightarrow \{\}
\end{align*}
\]

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

\textbf{fixrec} \(\text{abs-R} :: (\text{'c::contour } fstate + \text{'c cstate}) \text{ discr } \rightarrow (\text{'c::contour } fstate + \text{'c cstate}) \text{ discr set}\)

where \(\text{abs-R}\cdot x = (\text{case undiscr } x \text{ of} \)

\[
\begin{align*}
& (\text{Inl} \ (\text{PC} \ (\text{Lambda lab vs } c, \beta), \text{as}, \text{ve}, b)) \Rightarrow \\
& \quad (\text{if length } vs = \text{length } as \text{ then let } \beta' = \beta \ (\text{lab } \mapsto b); \\
& \quad \text{ve'} = \text{ve } \cup \{(\text{Discr } (\text{Inr } (c, \beta', \text{ve}', b))\}
\end{align*}
\]

\[
\begin{align*}
& \text{else } \perp
\end{align*}
\]

\[
\begin{align*}
& (\text{Inl} \ (\text{PP} \ (\text{Plus } c),\ldots,\text{cnts},\text{ve},b)) \Rightarrow \\
& \quad \text{let } b' = \overrightarrow{n}b \ b \ c; \\
& \quad \beta = [c \mapsto b] \\
& \quad \text{in } \bigcup_{\text{cnt} \in \text{cnts}} \{\text{Discr } (\text{Inl } (\text{cnt},[\ldots],b'))\}
\end{align*}
\]

\[
\begin{align*}
& (\text{Inl} \ (\text{PP} \ (\text{prim.If ct cf}),\ldots,\text{cnts},\text{ve},b)) \Rightarrow \\
& \quad (\text{let } b' = \overrightarrow{n}b \ b \ ct; \\
& \quad \beta = [ct \mapsto b] \\
& \quad \text{in } \bigcup_{\text{cnt} \in \text{cnts}} \{\text{Discr } (\text{Inl } (\text{cnt},[\ldots],b'))\}
\end{align*}
\]

\[
\begin{align*}
& (\text{Inl} \ (\text{PP} \ (\text{prim.If ct cf}),\ldots,\text{cnts},\text{ve},b)) \Rightarrow \\
& \quad (\text{let } b' = \overrightarrow{n}b \ b \ cf; \\
& \quad \beta = [cf \mapsto b] \\
& \quad \text{in } \bigcup_{\text{cnt} \in \text{cnts}} \{\text{Discr } (\text{Inl } (\text{cnt},[\ldots],b'))\}
\end{align*}
\]

abs-R gives the set of arguments passed to the recursive calls.
\documentclass{article}
\usepackage{amsmath,amssymb}

\begin{document}

\begin{verbatim}
| \text{(Inl (\text{AStop},[\cdot],\cdot,\cdot))} \Rightarrow \{\}
| \text{(Inl \cdot)} \Rightarrow \perp
| \text{(Inr \text{App lab f vs,\text{\beta},ve,b})} \Rightarrow
  \text{let fs = \text{\overline{A}} f \text{\beta} ve;}
  \text{as = map (\lambda v. \text{\overline{A}} v \text{\beta} ve) vs;}
  \text{b' = \text{\overline{n}} b lab}
  \text{in (\bigcup f' \in fs. (Discr (Inl (f',as,ve,b')))})
| \text{(Inr (Let lab ls c',\text{\beta},ve,b))} \Rightarrow
  \text{let b' = \text{\overline{n}} b lab;}
  \text{\beta' = \beta (\text{lab} \rightarrow b');}
  \text{ve' = ve \cup. (\bigcup. (map (\lambda (v,l). \{(v,b') := (\text{\overline{A}} (L l) \text{\beta'} ve\})) ls))}
  \text{in \{Discr (Inr (c',\beta',ve',b'))\}}

The initial argument vector, as created by \text{\overline{PR}}.

\textbf{definition} initial-r :: \text{prog} \Rightarrow \text{\text{'c::contour \overline{fstate} + 'c cstate} discr}
\text{where} initial-r \text{ prog = Discr (Inl (the-elem (\text{\overline{A}} (L prog) empty \{\}, \{AStop\}, \{\}, \hat{b}_0)))}

\section{8.1. Towards finiteness}

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

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

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

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

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

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

\textbf{lemma} finite-arg-space: finite (arg-poss p)
\text{(proof)}

\end{verbatim}

\end{document}
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 \( \hat{A} \) and an admissibility lemma.

**Lemma evalV-possible:**
- **Assumes** \( f: f \in \hat{A} \) \( d \beta \) \( ve \)
- **And** \( d: d \in \text{vals } p \) \( ve: ve \in \text{smaps-over } (\text{vars } p \times \text{UNIV}) \) \( (\text{proc-poss } p) \)
- **And** \( \beta: \beta \in \text{maps-over } (\text{labels } p) \) \( \text{UNIV} \)
- **Shows** \( f \in \text{proc-poss } p \)

**Lemma adm-subset:** cont \((\lambda x. f x) \implies \text{adm } (\lambda x. f x \subseteq S)\)

**Lemma arg-space-complete:**
- \( \text{state} \in \text{arg-poss } p \implies \text{abs-R } \text{state} \subseteq \text{arg-poss } p \)

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

**Lemma arg-space-complete-ps:** states \( \subseteq \text{arg-poss } p \implies (\text{abs-R }) \text{states} \subseteq \text{arg-poss } p \)

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** \( \text{start} : x \in S \)
- **And** \( \text{step} : \forall y. y \subseteq S \implies (f \cdot y) \subseteq S \)
- **Shows** \((\bigcup i. \text{iterate } i \cdot f \cdot \{x\}) \subseteq S \)

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

**8.2. A decomposition**

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

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

**Lemma a-evalF-decomp:**
\[ \hat{F} = \text{fst} \left( \text{sum-to-tup} \left( \text{fix} \left( \lambda f \cdot x. \left( \bigcup y \in \text{abs-R} \cdot f \cdot y \right) \cup \text{abs-g} \cdot x \right) \right) \right) \]

(The proof)

8.3. The iterative equation

Because of the special form of \( \hat{F} \) (and thus \( \widehat{PR} \)) 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.

\text{lemma a-evalF-iterative:}
\[ \hat{F} \cdot (\text{Discr } x) = \text{abs-g} \cdot \left( \bigcup i. \text{iterate } i \cdot (\text{abs-R}) \cdot \{\text{Discr } (\text{Inl } x)\} \right) \]

(The proof)

\text{lemma a-evalCPS-iterative:}
\[ \widehat{PR} \text{ prog} = \text{abs-g} \cdot \left( \bigcup i. \text{iterate } i \cdot (\text{abs-R}) \cdot \{\text{initial-r prog}\} \right) \]

(The proof)

end

Part III.
The auxiliary theories

9. Syntax tree helpers

theory CPSUtils
imports CPSScheme
begin

This theory defines the sets \textit{lambdas} \( p \), \textit{calls} \( p \), \textit{vars} \( p \), \textit{labels} \( p \) and \textit{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.

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

\text{lemma finite-lambdas[simp]: finite (lambdas l) and finite (lambdasC c) finite (lambdasV v)} \\
\langle \text{proof} \rangle \\

\text{lemma finite-calls[simp]: finite (calls l) and finite (callsC c) finite (callsV v)} \\
\langle \text{proof} \rangle \\

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

\text{lemma finite-vars[simp]: finite (vars l) and finite (varsC c) finite (varsV v)} \\
\langle \text{proof} \rangle \\

fun \text{label} :: \text{lambda + call} \Rightarrow \text{label} \\
\text{where} \ \text{label} (\text{Inl} (\text{Lambda} \ l \ - \ -)) = l \\
\text{label} (\text{Inr} (\text{App} \ l \ - \ -)) = l \\
\text{fun labels :: \text{lambda} \Rightarrow \text{label set} \\
\text{and} \ \text{labelsC} :: \text{call} \Rightarrow \text{label set} \\
\text{and} \ \text{labelsV} :: \text{val} \Rightarrow \text{label set} \\
\text{where} \ \text{labels} (\text{Lambda} \ l \ vs \ c) = \{l\} \cup \text{labelsC} \ c \\
\text{labelsC} (\text{App} \ l \ a \ as) = \{l\} \cup \text{labelsV} \ a \cup \text{(UNION (set as) labelsV)} \\
\text{labelsC} (\text{Let} \ l \ binds \ c') = \{l\} \cup \text{(UNION (set binds) (\lambda(v,l). \text{labels l} \cup \text{labelsC} c')} \\
\text{labelsV} \ (\text{L} \ l) = \text{labels l} \\
\text{labelsV} (\text{R} \ l \ -) = \{l\} \\
\text{labelsV} - = \{} \\

31
**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 ∪ 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)

(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 ∪ 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 ∀ 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 ∀ x ∈ set (list2 :: (var × lambda) list) . Lambda l vs c ∈ lambdas (snd x) ⇒ l ∈ labels (snd x)
and Lambda l vs c ∈ lambdas (snd (t:: var × lambda)) ⇒ l ∈ labels (snd t)
(proof)
lemma
shows \( \text{lambdas}3 \): \( \text{Lambda } l \text{ vs } c \in \text{lambdas } x \implies \text{set } vs \subseteq \text{vars } x \)
and \( \text{Lambda } l \text{ vs } c \in \text{lambdas}C \ y \implies \text{set } vs \subseteq \text{vars}C \ y \)
and \( \text{Lambda } l \text{ vs } c \in \text{lambdas}V z \implies \text{set } vs \subseteq \text{vars}V z \)
and \( \forall z \in \text{set list}. \text{Lambda } l \text{ vs } c \in \text{lambdas}V z \implies \text{set } vs \subseteq \text{vars}V z \)
and \( \forall x \in \text{set } (\text{list}2 :: (\text{var } \times \text{ lambda}) \text{ list}). \text{Lambda } l \text{ vs } c \in \text{lambdas} (\text{snd } x) \implies \text{set } vs \subseteq \text{vars } (\text{snd } t) \)
(proof)

lemma
shows \( \text{app}1 \): \( \text{App } l \text{ d } ds \in \text{calls } x \implies d \in \text{vals } x \)
and \( \text{App } l \text{ d } ds \in \text{calls}C \ y \implies d \in \text{vals}C \ y \)
and \( \text{App } l \text{ d } ds \in \text{calls}V z \implies d \in \text{vals}V z \)
and \( \forall z \in \text{set list}. \text{App } l \text{ d } ds \in \text{calls}V z \implies d \in \text{vals}V z \)
and \( \forall x \in \text{set } (\text{list}2 :: (\text{var } \times \text{ lambda}) \text{ list}). \text{App } l \text{ d } ds \in \text{calls } (\text{snd } x) \implies d \in \text{vals } (\text{snd } x) \)
and \( \text{App } l \text{ d } ds \in \text{calls } (\text{snd } (t:: \text{var } \times \text{ lambda}))) \implies d \in \text{vals } (\text{snd } t) \)
(proof)

lemma
shows \( \text{app}2 \): \( \text{App } l \text{ d } ds \in \text{calls } x \implies \text{set } ds \subseteq \text{vals } x \)
and \( \text{App } l \text{ d } ds \in \text{calls}C \ y \implies \text{set } ds \subseteq \text{vals}C \ y \)
and \( \text{App } l \text{ d } ds \in \text{calls}V z \implies \text{set } ds \subseteq \text{vals}V z \)
and \( \forall z \in \text{set list}. \text{App } l \text{ d } ds \in \text{calls}V z \implies \text{set } ds \subseteq \text{vals}V z \)
and \( \forall x \in \text{set } (\text{list}2 :: (\text{var } \times \text{ lambda}) \text{ list}). \text{App } l \text{ d } ds \in \text{calls } (\text{snd } x) \implies \text{set } ds \subseteq \text{vals } (\text{snd } x) \)
and \( \text{App } l \text{ d } ds \in \text{calls } (\text{snd } (t:: \text{var } \times \text{ lambda}))) \implies \text{set } ds \subseteq \text{vals } (\text{snd } t) \)
(proof)

lemma
shows \( \text{let}1 \): Let \( l \) binds \( c' \in \text{calls } x \implies l \in \text{labels } x \)
and Let \( l \) binds \( c' \in \text{calls}C \ y \implies l \in \text{labels}C \ y \)
and Let \( l \) binds \( c' \in \text{calls}V z \implies l \in \text{labels}V z \)
and \( \forall z \in \text{set list}. \text{Let } l \text{ binds } c' \in \text{calls}V z \implies l \in \text{labels}V z \)
and \( \forall x \in \text{set } (\text{list}2 :: (\text{var } \times \text{ lambda}) \text{ list}). \text{Let } l \text{ binds } c' \in \text{calls } (\text{snd } x) \implies l \in \text{labels } (\text{snd } x) \)
and Let \( l \) binds \( c' \in \text{calls } (\text{snd } (t:: \text{var } \times \text{ lambda}))) \implies l \in \text{labels } (\text{snd } t) \)
(proof)

lemma
shows \( \text{let}2 \): Let \( l \) binds \( c' \in \text{calls } x \implies c' \in \text{calls } x \)
and Let \( l \) binds \( c' \in \text{calls}C \ y \implies c' \in \text{calls}C \ y \)
and Let \( l \) binds \( c' \in \text{calls}V z \implies c' \in \text{calls}V z \)
and \( \forall z \in \text{set list}. \text{Let } l \text{ binds } c' \in \text{calls}V z \implies c' \in \text{calls}V z \)
and \( \forall x \in \text{set } (\text{list}2 :: (\text{var } \times \text{ lambda}) \text{ list}). \text{Let } l \text{ binds } c' \in \text{calls } (\text{snd } x) \implies c' \in \text{calls } (\text{snd } x) \)
and Let \( l \) binds \( c' \in \text{calls } (\text{snd } (t:: \text{var } \times \text{ lambda}))) \implies c' \in \text{calls } (\text{snd } t) \)
(proof)
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)
(proof)

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' ∈ callsV z ⇒ snd ' set binds ⊆ lambdasV z
and ∀ z ∈ set list. Let l binds c' ∈ callsV z ⇒ snd ' set binds ⊆ lambdasV z
and ∀ x ∈ set (list2 :: (var × lambda) list) . Let l binds c' ∈ calls (snd x) ⇒ snd ' set binds ⊆ lambdas (snd x)
and Let l binds c' ∈ calls (snd (t:: var×lambda)) ⇒ snd ' set binds ⊆ lambdas (snd t)
(proof)

lemma

shows vals1: P prim ∈ vals p ⇒ prim ∈ prims p
and P prim ∈ valsC y ⇒ prim ∈ primsC y
and P prim ∈ valsV z ⇒ prim ∈ primsV z
and ∀ z ∈ set list. P prim ∈ valsV z ⇒ prim ∈ primsV z
and ∀ x ∈ set (list2 :: (var × lambda) list) . P prim ∈ vals (snd x) ⇒ prim ∈ prims (snd x)
and P prim ∈ vals (snd (t:: var×lambda)) ⇒ prim ∈ prims (snd t)
(proof)

lemma

shows vals2: R l var ∈ vals p ⇒ var ∈ vars p
and R l var ∈ valsC y ⇒ var ∈ varsC y
and R l var ∈ valsV z ⇒ var ∈ varsV z
and ∀ z ∈ set list. R l var ∈ valsV z ⇒ var ∈ varsV z
and ∀ x ∈ set (list2 :: (var × lambda) list) . R l var ∈ vals (snd x) ⇒ var ∈ vars (snd x)
and R l var ∈ vals (snd (t:: var×lambda)) ⇒ var ∈ vars (snd t)
(proof)

lemma

shows vals3: L l ∈ vals p ⇒ l ∈ lambdas p
and L l ∈ valsC y ⇒ l ∈ lambdasC y
and L l ∈ valsV z ⇒ l ∈ lambdasV z
and ∀ z ∈ set list. L l ∈ valsV z ⇒ l ∈ lambdasV z
and ∀ x ∈ 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 \( n\text{List} :: 'a \text{ set} \Rightarrow \text{nat} \Rightarrow 'a \text{ list set} \)
where \( n\text{List} \ A \ n \equiv \{ l. \ \text{set} \ l \leq A \land \text{length} \ l = n \} \)

lemma \( \text{finite-nList}[\text{intro}]: \)
\hspace{1em} assumes \( \text{finA}: \text{finite} \ A \)
\hspace{1em} shows \( \text{finite} \ (n\text{List} \ A \ n) \)
\hspace{1em} (\text{proof})

definition \( \text{NList} :: 'a \text{ set} \Rightarrow \text{nat set} \Rightarrow 'a \text{ list set} \)
where \( \text{NList} \ A \ N \equiv \bigcup \ n \in N. \ n\text{List} \ A \ n \)

lemma \( \text{finite-Nlist}[\text{intro}]: \)
\hspace{1em} \[ \text{finite} \ A; \ \text{finite} \ N \] \Rightarrow \( \text{finite} \ (\text{NList} \ A \ N) \)
\hspace{1em} (\text{proof})

definition \( \text{call-list-lengths} \)
\hspace{1em} where \( \text{call-list-lengths} \ p = \{0,1,2,3\} \cup (\lambda \ c. \ \text{case} \ c \ \text{of} \ \text{App} \ - \ - \ ds \Rightarrow \text{length} \ ds \mid - \Rightarrow 0) \)
\hspace{1em} \( \text{calls} \ p \)

lemma \( \text{finite-call-list-lengths}[\text{simp}]: \text{finite} \ (\text{call-list-lengths} \ p) \)
\hspace{1em} (\text{proof})

end

10. General utility lemmas

theory \( \text{Utils} \) imports \( \text{Main} \)
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 \( \text{single-valued} \) predicate.

lemma \( \text{single-valued-empty}[\text{simp}]:\text{single-valued} \ \{} \)
\hspace{1em} (\text{proof})

lemma \( \text{single-valued-insert} : \)
\hspace{1em} assumes \( \text{single-valued} \ \text{rel} \)
\hspace{1em} and \( \land \ x \ y. \ [(x,y) \in \text{rel}; \ x=a] \Longrightarrow y = b \)
\hspace{1em} shows \( \text{single-valued} \ (\text{insert} \ (a,b) \ \text{rel}) \)
\hspace{1em} (\text{proof})

Lemmas about \( \text{ran} \), the range of a finite map.

lemma \( \text{ran-upd}: \text{ran} \ (m \ (k \mapsto v)) \subseteq \text{ran} \ m \cup \{v\} \)
\hspace{1em} (\text{proof})

35
lemma ran-map-of: ran (map-of xs) ⊆ snd i set xs
(proof)

lemma ran-concat: ran (m1 ++ m2) ⊆ ran m1 ∪ ran m2
(proof)

lemma ran-upds:
assumes eq-length: length ks = length vs
shows ran (map-upds m ks vs) ⊆ ran m ∪ set vs
(proof)

lemma ran-upd-mem[simp]: v ∈ ran (m (k ↦→ v))
(proof)

Lemmas about map, zip and fst/snd

lemma map-fst-zip: length xs = length ys ⇒ map fst (zip xs ys) = xs
(proof)

lemma map-snd-zip: length xs = length ys ⇒ 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 ∼= {}}

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

lemma sranI: b ∈ m a ⇒ b ∈ sran m
(proof)

lemma sdom-not-mem[elim]: a /∈ sdom m ⇒ m a = {}
(proof)
definition **smap-empty** ({\{} \}.)
where \{\}. \cdot k = \{\}.

definition **smap-union** :: ('a::type => 'b::type set) => ('a => 'b set) => ('a => 'b set) (- `∪` -)
where smap1 `∪` smap2 k = smap1 k `∪` smap2 k

primrec **smap-Union** :: ('a::type => 'b::type set) list => ('a => 'b set) (`∪`-)
where \[simp\]: `∪` [] = \{\}. `∪` [m\#ms] = m `∪` `∪`. 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-less m1 m2 = (\forall k. m1 k `⊆` m2 k)

lemma **sdom-empty**[simp]: sdom \{\}. = \{
⟨proof⟩
lemma **sdom-singleton**[simp]: sdom \{ k := vs \}. `⊆` \{k\}
⟨proof⟩
lemma **sran-singleton**[simp]: sran \{ k := vs \}. = vs
⟨proof⟩
lemma **sran-empty**[simp]: sran \{\}. = \{
⟨proof⟩
lemma **sdom-union**[simp]: sdom (m `∪` n) = sdom m `∪` sdom n
⟨proof⟩
lemma **sran-union**[simp]: sran (m `∪` n) = sran m `∪` sran n
⟨proof⟩
lemma **smap-empty**[simp]: \{\}. `⊆` \{\}.
⟨proof⟩
lemma **smap-less-refl**: m `⊆` m
⟨proof⟩
lemma **smap-less-trans**[trans]: \[ m1 `⊆` m2; m2 `⊆` m3 \] \implies m1 `⊆` m3
⟨proof⟩
lemma **smap-union-mono**: \[ ve1 `⊆` ve1\'; ve2 `⊆` ve2\' \] \implies ve1 `∪` ve2 `⊆` ve1\' `∪` ve2\'
⟨proof⟩
lemma **smap-Union-union**: mI `∪` \. ms = `∪` (mI\#ms)
⟨proof⟩
lemma smap-Union-mono:
  assumes list-all2 smap-less ms1 ms2
  shows \( \bigcup \) ms1 \( \subseteq \) \( \bigcup \) ms2
⟨proof⟩

lemma smap-singleton-mono: \( v \subseteq v' \implies \{ k := v \} \subseteq \{ k := v' \} \).
⟨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]: \( \bigcup (m1 \circ m2) = (\bigcup m1) \cup (\bigcup m2) \)
⟨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 SetMap. Both these sets are defined \((\text{maps-over, smaps-over})\) and the finiteness is shown.

definition maps-over :: \( 'a::\text{type set} \Rightarrow 'b::\text{type set} \Rightarrow ('a \rightarrow 'b) \text{ set} \)
  where \( \text{maps-over A B = \{ m. dom m \subseteq A \land ran m \subseteq B \}} \)

lemma maps-over-empty[simp]:
  empty \( \in \text{maps-over A B} \)

end
lemma maps-over-upd:
  assumes \( m \in \text{maps-over} \ A \ B \) and \( v \in A \) and \( k \in B \)
  shows \( m(v \mapsto k) \in \text{maps-over} \ A \ B \)

(\text{proof})

lemma maps-over-finite[intro]:
  assumes finite \( A \) and finite \( B \) shows finite (maps-over \( A \ B \))

(\text{proof})

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

lemma smaps-over-empty[simp]:
  \{\}\. \in \text{smaps-over} \ A \ B

(\text{proof})

lemma smaps-over-singleton:
  assumes \( k \in A \) and \( vs \subseteq B \)
  shows \( \{k := vs\}. \in \text{smaps-over} \ A \ B \)

(\text{proof})

lemma smaps-over-un:
  assumes \( m1 \in \text{smaps-over} \ A \ B \) and \( m2 \in \text{smaps-over} \ A \ B \)
  shows \( m1 \cup. m2 \in \text{smaps-over} \ A \ B \)

(\text{proof})

lemma smaps-over-Union:
  assumes set ms \( \subseteq \text{smaps-over} \ A \ B \)
  shows \( \bigcup. ms \in \text{smaps-over} \ A \ B \)

(\text{proof})

lemma smaps-over-im:
  \[ f \in m a ; m \in \text{smaps-over} \ A \ B \] \Longrightarrow f \in B

(\text{proof})

lemma smaps-over-finite[intro]:
  assumes finite \( A \) and finite \( B \) shows finite (smaps-over \( A \ B \))

(\text{proof})

end

13. HOLCF Utility lemmas

theory HOLCFUtils
  imports HOLCF
  begin

end
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.

```plaintext
instantiation bool :: po
begin
  definition x ⊑ y ←→ (x → y)
instance ⟨proof⟩
end

instance bool :: chfin ⟨proof⟩
instance bool :: pcpo ⟨proof⟩

lemma is-lub-bool: S <<| (True ∈ S) ⟨proof⟩
lemma lub-bool: lub S = (True ∈ S) ⟨proof⟩
lemma bottom-eq-False[simp]: ⊥ = False ⟨proof⟩
```

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

```plaintext
instantiation set :: (type) po
begin
  definition A ⊑ B ←→ A ⊆ B
instance ⟨proof⟩
end

lemma sqsubset-is-subset: A ⊑ B ←→ A ⊆ B ⟨proof⟩
lemma is-lub-set: S <<| ∪ S ⟨proof⟩
lemma lub-is-union: lub S = ∪ S ⟨proof⟩
instance set :: (type) cpo ⟨proof⟩
```

40
lemma emptyset-is-bot[simp]: \{\} \subseteq 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 requirements 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 \lor 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 \rightarrow g x)
(proof)

lemma cont2cont-Collect [simp, cont2cont]:
  assumes \forall 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) \Rightarrow cont (\lambda x. g x)
  \Rightarrow 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 \forall y. cont (\lambda x. g x y)
  shows cont (\lambda x. \bigcup y \in f. g x y)
(proof)

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

**lemma** cont2cont-case-list [simp, cont2cont]:
  **assumes** ⋀y. cont (λx. f1 x)
  and ⋀y z. cont (λx. f2 x y z)
  **shows** cont (λx. case-list (f1 x) (f2 x) l)
(\ proof

As with the continuity lemmas, we need admissibility lemmas.

**lemma** adm-not-mem:
  **assumes** cont (λx. f x)
  **shows** adm (λx. 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 (λp. f (fst p) (snd p))
  **shows** adm (λ(x,y). f x y)
(\ proof

**lemma** adm-ball':
  **assumes** ⋀y. adm (λx. y ∈ A x → P x y)
  **shows** adm (λx. ∀y ∈ A x . P x y)
(\ proof

**lemma** adm-not-conj:
  [adm (λx. ¬ P x); adm (λx. ¬ Q x)] \implies adm (λx. ¬ (P x ∧ Q x))
(\ proof

**lemma** adm-single-valued:
  **assumes** cont (λx. f x)
  **shows** adm (λx. 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 ≡ iterate i f
end

14. Fixed point transformations

theory FixTransform
imports HOLCF
begin

default-sort type

In his treatment of the computably, 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 ∏x. g (f x) = x
  shows fix F = g (fix (f oo F 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 discr → 'c) × ('b discr → 'c) → ('a + 'b) discr → 'c::cpo
  where tup-to-sum = (Λ p s. (λ(f,g).
      case undiscr s of Inl x ⇒ f (Discr x)
      | Inr x ⇒ g (Discr x)) p)

definition sum-to-tup :: ('a + 'b) discr → 'c → ('a discr → 'c) × ('b discr → 'c::cpo)
  where sum-to-tup = (Λ f. (Λ x. f (Discr (Inl (undiscr x)))),
                  Λ x. f (Discr (Inr (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 (λx. case-sum (f x) (g x) s)
⟨proof⟩
**Lemma** \( \text{cont2cont-circ}[\text{simp}, \text{cont2cont}]: \)
\[
\text{cont} \ (\lambda f. \ f \circ g)
\]
\(\langle \text{proof} \rangle\)

**Lemma** \( \text{cont2cont-split-pair}[\text{cont2cont}, \text{simp}]: \)
\[
\begin{align*}
\text{assumes } &f_1: \text{cont } f \\
\text{and } &f_2: \forall x. \text{cont } (f \ x) \\
\text{and } &g_1: \text{cont } g \\
\text{and } &g_2: \forall x. \text{cont } (g \ x)
\end{align*}
\]
\[
\text{shows } \text{cont} \ (\lambda (a, b). \ (f \ a \ b, g \ a \ b))
\]
\(\langle \text{proof} \rangle\)

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

**Lemma** \( \text{sum-to-tup-app}: \)
\[
\text{sum-to-tup} \ f = (\Lambda x. \ f \cdot (\text{Discr } (\text{Inl } (\text{undiscr } x))), \ \Lambda x. \ f \cdot (\text{Discr } (\text{Inr } (\text{undiscr } x))))
\]
\(\langle \text{proof} \rangle\)

**Lemma** \( \text{tup-to-sum-app}: \)
\[
\begin{align*}
\text{tup-to-sum} \ p = (\Lambda s. \ (\lambda (f, g). \\
\text{case } \text{undiscr } s \text{ of } \text{Inl } x \Rightarrow f \cdot (\text{Discr } x) \\
| \text{Inr } x \Rightarrow g \cdot (\text{Discr } x)) \ p)
\end{align*}
\]
\(\langle \text{proof} \rangle\)

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

**Lemma** \( \text{discr-app}[\text{simp}]: \)
\[
(\Lambda s. \ f \ s) \cdot (\text{Discr } x) = f \cdot (\text{Discr } x)
\]
\(\langle \text{proof} \rangle\)

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

**Lemma** \( \text{tup-to-sum-to-tup}[\text{simp}]: \)
\[
\text{shows } \text{sum-to-tup} \cdot (\text{tup-to-sum} \cdot F) = F
\]
\(\langle \text{proof} \rangle\)

**Lemma** \( \text{fix-transform-pair-sum}: \)
\[
\begin{align*}
\text{shows } &\text{fix } F = \text{sum-to-tup} \cdot (\text{fix} \cdot (\text{tup-to-sum oo F oo sum-to-tup}))
\end{align*}
\]
\(\langle \text{proof} \rangle\)

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

**Lemma** \( \text{tup-sum-oo}[\text{simp}]: \)
\[
\begin{align*}
\text{assumes } &f_1: \text{cont } F
\end{align*}
\]
\[ \text{and } f_2 : \bigwedge x. \text{cont} (F x) \]
\[ \text{and } g_1 : \text{cont } G \]
\[ \text{and } g_2 : \bigwedge x. \text{cont} (G x) \]
\text{shows} \quad \text{tup-to-sum oo } (\Lambda p . (\lambda(a, b). (F a, b, G a, b)) p) \text{ oo sum-to-tup} \]
\[ = (\Lambda f s . (\text{case } \text{undiscr } s \text{ of} \]
\[ \quad \text{Inl } x \Rightarrow \]
\[ \quad F (\Lambda s. f \cdot (\text{Discr } (\text{Inl } (\text{undiscr } s)))) \]
\[ \quad (\Lambda s. f \cdot (\text{Discr } (\text{Inr } (\text{undiscr } s)))) \]
\[ \quad (\text{Discr } x) \]
\[ | \quad \text{Inr } x \Rightarrow \]
\[ \quad G (\Lambda s. f \cdot (\text{Discr } (\text{Inl } (\text{undiscr } s)))) \]
\[ \quad (\Lambda s. f \cdot (\text{Discr } (\text{Inr } (\text{undiscr } s)))) \]
\[ \quad (\text{Discr } x) \]
\[ \rangle \]
\text{⟨proof⟩} \]
\text{lemma } \text{fst-sum-to-tup[simp]:} \]
\[ \text{fst } (\text{sum-to-tup } x) = (\Lambda xa. x \cdot (\text{Discr } (\text{Inl } (\text{undiscr } xa)))) \]
\text{⟨proof⟩} \]
\text{end} 

References

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

haskell.org/package/shivers-cfg-0.1, November 2010.