

# The Correctness of Launchbury’s Natural Semantics for Lazy Evaluation

Joachim Breitner  
Programming Paradigms Group  
Karlsruhe Institute for Technology  
[breitner@kit.edu](mailto:breitner@kit.edu)

February 6, 2026

In his seminal paper “Natural Semantics for Lazy Evaluation” [Lau93], John Launchbury proves his semantics correct with respect to a denotational semantics, and outlines an adequacy proof. We have formalized both semantics and machine-checked the correctness proof, clarifying some details. Furthermore, we provide a new and more direct adequacy proof that does not require intermediate operational semantics.

## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Main definitions and theorems . . . . .	6
1.1.1	The big picture . . . . .	6
1.1.2	Expressions . . . . .	6
1.1.3	The natural semantics . . . . .	7
1.1.4	The denotational semantics . . . . .	7
1.1.5	Correctness and Adequacy . . . . .	8
1.2	Differences to our previous work . . . . .	8
1.2.1	The treatment of $\sqcup$ . . . . .	8
1.2.2	The types of environments . . . . .	9
1.2.3	No type <i>assn</i> . . . . .	10
1.3	Related work . . . . .	11
1.4	Theory overview . . . . .	11
1.5	Acknowledgements . . . . .	13
<b>2</b>	<b>Auxiliary theories</b>	<b>14</b>
2.1	Pointwise . . . . .	14

2.2	AList-Utils . . . . .	14
2.2.1	The domain of an associative list . . . . .	14
2.2.2	Other lemmas about associative lists . . . . .	16
2.2.3	Syntax for map comprehensions . . . . .	17
2.3	Mono-Nat-Fun . . . . .	18
2.4	Nominal-Utils . . . . .	19
2.4.1	Lemmas helping with equivariance proofs . . . . .	19
2.4.2	Freshness via equivariance . . . . .	20
2.4.3	Additional simplification rules . . . . .	21
2.4.4	Additional equivariance lemmas . . . . .	21
2.4.5	Freshness lemmas . . . . .	24
2.4.6	Freshness and support for subsets of variables . . . . .	24
2.4.7	The set of free variables of an expression . . . . .	25
2.4.8	Other useful lemmas . . . . .	26
2.5	AList-Utils-Nominal . . . . .	28
2.5.1	Freshness lemmas related to associative lists . . . . .	28
2.5.2	Equivariance lemmas . . . . .	29
2.5.3	Freshness and distinctness . . . . .	29
2.5.4	Pure codomains . . . . .	30
2.6	HOLCF-Utils . . . . .	30
2.6.1	Composition of fun and cfun . . . . .	34
2.6.2	Additional transitivity rules . . . . .	34
2.7	HOLCF-Meet . . . . .	34
2.7.1	Towards meets: Lower bounds . . . . .	35
2.7.2	Greatest lower bounds . . . . .	35
2.8	Nominal-HOLCF . . . . .	39
2.8.1	Type class of continuous permutations and variations thereof . . . . .	39
2.8.2	Instance for <i>cfun</i> . . . . .	41
2.8.3	Instance for <i>fun</i> . . . . .	42
2.8.4	Instance for <i>u</i> . . . . .	42
2.8.5	Instance for <i>lift</i> . . . . .	43
2.8.6	Instance for <i>prod</i> . . . . .	44
2.9	Env . . . . .	44
2.9.1	The domain of a pcpo-valued function . . . . .	44
2.9.2	Updates . . . . .	45
2.9.3	Restriction . . . . .	45
2.9.4	Deleting . . . . .	48
2.9.5	Merging of two functions . . . . .	49
2.9.6	Environments with binary joins . . . . .	50
2.9.7	Singleton environments . . . . .	50
2.10	Env-Nominal . . . . .	51
2.10.1	Equivariance lemmas . . . . .	51
2.10.2	Permutation and restriction . . . . .	52
2.10.3	Pure codomains . . . . .	53

2.11	Env-HOLCF	54
2.11.1	Continuity and pcpo-valued functions	54
2.12	EvalHeap	56
2.12.1	Conversion from heaps to environments	56
2.12.2	Reordering lemmas	58
<b>3</b>	<b>Launchbury's natural semantics</b>	<b>59</b>
3.1	Vars	59
3.2	Terms	59
3.2.1	Expressions	59
3.2.2	Rewriting in terms of heaps	60
3.2.3	Nice induction rules	64
3.2.4	Testing alpha equivalence	65
3.2.5	Free variables	66
3.2.6	Lemmas helping with nominal definitions	66
3.2.7	A smart constructor for lets	68
3.2.8	A predicate for value expressions	68
3.2.9	The notion of thunks	69
3.2.10	Non-recursive Let bindings	70
3.2.11	Renaming a lambda-bound variable	72
3.3	Substitution	72
3.4	Launchbury	78
3.4.1	The natural semantics	78
3.4.2	Example evaluations	79
3.4.3	Better introduction rules	80
3.4.4	Properties of the semantics	81
<b>4</b>	<b>Denotational domain</b>	<b>84</b>
4.1	Value	84
4.1.1	The semantic domain for values and environments	84
4.2	Value-Nominal	86
<b>5</b>	<b>Denotational semantics</b>	<b>87</b>
5.1	Iterative	87
5.2	HasESem	88
5.3	HeapSemantics	89
5.3.1	A locale for heap semantics, abstract in the expression semantics	89
5.3.2	Induction and other lemmas about <i>HSem</i>	89
5.3.3	Substitution	92
5.3.4	Re-calculating the semantics of the heap is idempotent	93
5.3.5	Iterative definition of the heap semantics	93
5.3.6	Fresh variables on the heap are irrelevant	94
5.3.7	Freshness	95
5.3.8	Adding a fresh variable to a heap does not affect its semantics	97

5.3.9	Mutual recursion with fresh variables	97
5.3.10	Parallel induction	99
5.3.11	Congruence rule	99
5.3.12	Equivariance of the heap semantics	100
5.4	AbstractDenotational	100
5.4.1	The denotational semantics for expressions	100
5.5	Abstract-Denotational-Props	102
5.5.1	The semantics ignores fresh variables	102
5.5.2	Nicer equations for ESem, without freshness requirements	103
5.5.3	Denotation of Substitution	104
5.6	Denotational	106
<b>6</b>	<b>Resourced denotational domain</b>	<b>107</b>
6.1	C	107
6.2	C-Meet	109
6.3	C-restr	110
6.3.1	The demand of a C-function	111
6.3.2	Restricting functions with domain C	113
6.3.3	Restricting maps of C-ranged functions	115
6.4	CValue	115
6.5	CValue-Nominal	116
6.6	ResourcedDenotational	117
<b>7</b>	<b>Correctness of the natural semantics</b>	<b>119</b>
7.1	CorrectnessOriginal	119
7.2	CorrectnessResourced	123
<b>8</b>	<b>Equivalence of the denotational semantics</b>	<b>129</b>
8.1	ValueSimilarity	129
8.1.1	A note about section 2.3	129
8.1.2	Working with <i>Value</i> and <i>CValue</i>	130
8.1.3	Restricted similarity is defined recursively	131
8.1.4	Moving up and down the similarity relations	132
8.1.5	Admissibility	133
8.1.6	The real similarity relation	135
8.1.7	The similarity relation lifted pointwise to functions.	139
8.2	Denotational-Related	139
<b>9</b>	<b>Adequacy</b>	<b>142</b>
9.1	ResourcedAdequacy	142
9.2	Adequacy	149

# 1 Introduction

The Natural Semantics for Lazy Evaluation [Lau93] created by John Launchbury in 1992 is often taken as the base for formal treatments of call-by-need evaluation, either to prove properties of lazy evaluation or as a base to describe extensions of the language or the implementation of the language. Therefore, assurance about the correctness and adequacy of the semantics is important in this field of research. Launchbury himself supports his semantics by defining a standard denotational semantics to prove both correctness and adequacy.

Although his proofs are already on the more rigorous side for pen-and-paper proofs, they have not yet been verified by transforming them to machine-checked proofs. The present work fills this gap by formalizing both semantics in the proof assistant Isabelle and proving both correctness and adequacy.

Our correctness formal proof is very close to the original proof. This is possible if the operator  $\sqcup$  is understood as a right-sided update. If we were to understand  $\sqcup$  as the least upper bound, then Theorem 2 in [Lau93], which is the generalization of the correctness statement used for Launchbury’s inductive proof, is wrong. The main correctness result still holds, but needs a different proof; this is discussed in greater detail in [Bre13].

Launchbury outlines an adequacy proof via an intermediate operational semantics and resourced denotational semantics. The alternative operational semantics uses indirection instead of substitution for applications, does not update variable results and does not perform blackholing during evaluation of a variable. The equivalence of these two operational semantics is hard and tricky to prove. We found a direct proof for the adequacy of the original operational semantics and the (slightly modified) resourced denotational semantics. This is, as far as we know, the first complete and rigorous proof of adequacy of Launchbury’s semantics.

In this development we extend Launchburys syntax and semantics with boolean values and an if-then-else construct, in order to base a subsequent work [?] on this. This extension does not affect the validity of the proven theorems, and the extra cases can simply be ignored if one is interested in the plain semantics. The next introductory section does exactly that. Unfortunately, such meta-level arguments are not easily implemented inside a theorem prover.

Our contributions are:

- We define the natural and denotational semantics given by Launchbury in the theorem prover Isabelle.
- We demonstrate how to use both the Nominal package (to handle name binding) [UK12] and the HOLCF [Huf12] package (for the domain-theoretic aspects) in the same development.
- We verify Launchbury’s proof of correctness.

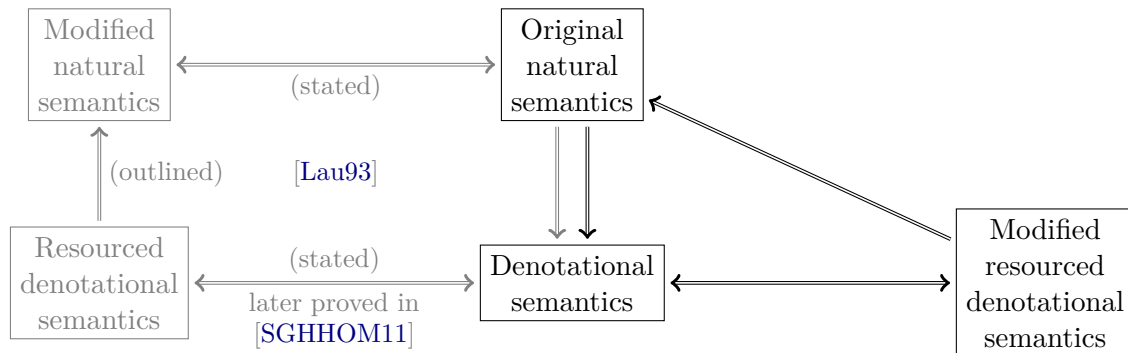
- We provide a new and more direct proof of adequacy.
- In order to do so, we formalize parts of [SGHHOM11], fixing a mistake in the proof.

## 1.1 Main definitions and theorems

For your convenience, the main definitions and theorems of the present work are assembled in this section. The following formulas are mechanically pretty-printed versions of the statements as defined resp. proven in Isabelle. Free variables are all-quantified. Some type conversion functions (like *set*) are omitted. The relations  $\sharp$  and  $\sharp^*$  come from the Nominal package and express freshness of the variables on the left with regard to the expressions on the right.

### 1.1.1 The big picture

The following picture gives an overview of the different semantics. Elements printed in black are formally defined and proved in the present work, while the gray square on the left shows the proofs and propositions in Launchbury's original work [Lau93].



### 1.1.2 Expressions

The type *var* of variables is abstract and provided by the Nominal package. All we know about it is that it is countably infinite. Expressions of type *exp* are given by the following grammar:

$$\begin{array}{ll}
 e ::= \lambda x. e & \text{lambda abstraction} \\
 | e x & \text{application} \\
 | x & \text{variable} \\
 | \textit{let as in } e & \text{recursive let}
 \end{array}$$

In the introduction we pretty-print expressions to resemble the notation in [Lau93] and omit the constructor names *Var*, *App*, *Lam* and *Let*. In the actual theories, these are visible. These expressions are, due to the machinery of the Nominal package, actually alpha-equivalency classes, so  $\lambda x. x = \lambda y. y$  holds provably. This differs from Launchbury's original definition, which expects distinctly-named expressions and performs explicit alpha-renaming in the semantics.

The type *heap* is an abbreviation for  $(var \times exp)$  *list*. These are *not* alpha-equivalency classes, i.e. we manage the bindings in heaps explicitly.

### 1.1.3 The natural semantics

Launchbury's original semantics, extended with some technical overhead related to name binding (following [Ses97]), is defined as follows:

$$\begin{array}{c}
\frac{}{\Gamma : \lambda x. e \Downarrow_L \Gamma : \lambda x. e} \quad \text{LAMBDA} \\
\\
\frac{\Gamma : e \Downarrow_L \quad \Delta : \lambda y. e' \quad \Delta : e'[y := x] \Downarrow_L \quad \Theta : z}{\Gamma : e x \Downarrow_L \quad \Theta : z} \quad \text{APPLICATION} \\
\\
\frac{(x, e) \in \Gamma \quad \Gamma \setminus x : e \Downarrow_x . L \quad \Delta : z}{\Gamma : x \Downarrow_L (x, z) \cdot \Delta : z} \quad \text{VARIABLE} \\
\\
\frac{dom \Delta \#* (\Gamma, L) \quad \Delta @ \Gamma : body \Downarrow_L \quad \Theta : z}{\Gamma : let \Delta in body \Downarrow_L \quad \Theta : z} \quad \text{LET}
\end{array}$$

### 1.1.4 The denotational semantics

The value domain of the denotational semantics is the initial solution to

$$D = [D \rightarrow D]_{\perp}$$

as introduced in [Abr90]. The type *Value*, together with the bottom value  $\perp$ , the injection *Fn* and the projection  $\_ \Downarrow Fn \_ :: Value \rightarrow Value \rightarrow Value$ , is constructed as a pointed chain-complete partial order from this equation by the HOLCF package. The type of semantic environments is  $var \Rightarrow Value$ .

The semantics of an expression  $e :: exp$  in an environment  $\rho :: var \Rightarrow Value$  is written  $\llbracket e \rrbracket_{\rho} :: Value$  and defined by the following equations:

$$\begin{aligned}
\llbracket \lambda x. e \rrbracket_{\rho} &= Fn \cdot (\Lambda v. \llbracket e \rrbracket_{\rho(x := v)}) \\
\llbracket e x \rrbracket_{\rho} &= \llbracket e \rrbracket_{\rho} \Downarrow Fn \rho x \\
\llbracket x \rrbracket_{\rho} &= \rho x \\
\llbracket let \Gamma in body \rrbracket_{\rho} &= \llbracket body \rrbracket_{\llbracket \Gamma \rrbracket_{\rho}}
\end{aligned}$$

The expression  $\llbracket \Gamma \rrbracket_{\varrho}$  maps the evaluation function over a heap, returning an environment:

$$\begin{aligned} (\llbracket \Gamma \rrbracket_{\varrho}) v &= \llbracket e \rrbracket_{\varrho} && \text{if } (v, e) \in \Gamma \\ (\llbracket \Gamma \rrbracket_{\varrho}) v &= \perp && \text{if } v \notin \text{dom } \Gamma \end{aligned}$$

The semantics  $\{\Gamma\}_{\varrho} :: \text{var} \Rightarrow \text{Value}$  of a heap  $\Gamma :: \text{heap}$  in an environment  $\varrho :: \text{var} \Rightarrow \text{Value}$  is defined by the recursive equation

$$\{\Gamma\}_{\varrho} = \varrho \text{ ++}_{\text{dom } \Gamma} \llbracket \Gamma \rrbracket_{\{\Gamma\}_{\varrho}}$$

where

$$\begin{aligned} (f \text{ ++}_A g) a &= f a && \text{if } a \notin A \\ (f \text{ ++}_A g) a &= g a && \text{if } a \in A. \end{aligned}$$

The semantics of the heap in the empty environment  $\perp$  is abbreviated as  $\{\Gamma\}$ .

### 1.1.5 Correctness and Adequacy

The statement of correctness reads: If  $\Gamma : e \Downarrow_L \Delta : v$  and, as a side condition,  $fv(\Gamma, e) \subseteq L \cup \text{dom } \Gamma$  holds, then

$$\llbracket e \rrbracket_{\{\Gamma\}_{\varrho}} = \llbracket v \rrbracket_{\{\Delta\}_{\varrho}}.$$

The statement of adequacy reads:

$$\text{If } \llbracket e \rrbracket_{\{\Gamma\}} \neq \perp \text{ then } \exists \Delta v. \Gamma : e \Downarrow_S \Delta : v.$$

## 1.2 Differences to our previous work

We have previously published [Bre13] of which the present work is a continuation. They differ in scope and focus:

### 1.2.1 The treatment of $\sqcup$

In [Bre13], the question of the precise meaning of  $\sqcup$  is discussed in detail. The original paper is not clear about whether this operator denotes the least upper bound, or the right-sided override operator. A lemma stated in [Lau93] only holds if  $\sqcup$  is the least upper bound, but with that definition, Launchbury's Theorem 2 – the generalized correctness theorem – is false; a counter-example is given in [Bre13].

We came up with an alternative operational semantics that keeps more of the evaluation context in the judgments and allows the correctness theorem to be proved inductively without the problematic generalization. We proved the two operational semantics equivalent and thus obtained the (non-generalized) correctness of Launchbury’s semantics.

We also showed that if one takes  $\sqcup$  to be the update operator, Theorem 2 holds and the proof goes through as it is. Furthermore, we showed that the resulting denotational semantics are identical for expressions, and can differ only for heaps. Therefore, the question of the precise meaning of  $\sqcup$  can be considered of little importance and for the present work we solely work with right sided updates. We also avoid the ambiguous syntax  $\sqcup$  and write  $\_ ++ \_ \_$  instead (the index indicates on what set the function on the right overrides the function on the left). The alternative operational semantics is not included in this work.

### 1.2.2 The types of environments

Another difference is the choice of the type for environments, which map variables to semantics values. A naive choice is  $var \Rightarrow Value$ , but this causes problems when defining the value semantics, for which

$$\llbracket \lambda x. e \rrbracket_{\varrho} = Fn \cdot (\Lambda v. \llbracket e \rrbracket_{\varrho(x := v)})$$

is a defining equation. The argument on the left hand side is the representative of an equivalence class (defined using the Nominal package), so this is only allowed if the right hand side is indeed independent of the actual choice of  $x$ . This is shown most commonly and easily if  $x$  is fresh in all the other arguments ( $x \# \varrho$ ), and indeed the Nominal package allows us to specify this as a side condition to the defining equation, which is what we did in [Bre13].

But this convenience comes as a price: Such side-conditions are only allowed if the argument has finite support (otherwise there might no variable fulfilling  $x \# \varrho$ ). More precisely: The type of the argument must be a member of the *fs* typeclass provided by the Nominal package. The type  $var \Rightarrow Value$  cannot be made a member of this class, as there obviously are elements that have infinite support. The fix here was to introduce a new type constructor, *fmap*, for partial functions with finite domain. This is fine: Only functions with finite domain matter in our formalisation.

The introduction of *fmap* had further consequences. The main type class of the HOLCF package, which we use to define domains and continuous functions on them, is the class *cpo*, of chain-complete partial orders. With the usual ordering on partial functions,  $(var, Value)$  *fmap* cannot be a member of this class. The fix here is to use a different ordering and only let elements be comparable that have the same domain. In our formalisation, the domain is always known (e.g. all variables bound on some heap), so this worked out.

But not without causing yet another issue: With this ordering,  $(var, Value)$  *fmap* is a *cpo*, but lacks a bottom element, i.e. now it is no *pcpo*, and HOLCF’s built-in operator

$\mu x. f x$  for expressing least fixed-points, as they occur in the semantics of heaps, is not available. Furthermore,  $\sqcup$  is not a total function, i.e. defined only on a subset of all possible arguments. The solution was a rather convoluted set of theories that formalize functions that are continuous on a specific set, fixed-points on such sets etc.

In the present work, this problems is solved in a much more elegant way. Using a small trick we defined the semantics functions so that

$$\llbracket \lambda x. e \rrbracket_{\varrho} = Fn \cdot (\Lambda v. \llbracket e \rrbracket_{\varrho(x := v)})$$

holds unconditionally. The actual, technical definition is

$$\llbracket \lambda x. e \rrbracket_{\varrho} = Fn \cdot (\Lambda v. \llbracket e \rrbracket_{\varrho|_{fv(\lambda x. e)}(x := v)})$$

where the right-hand-side can be shown to be invariant of the choice of  $x$ , as  $x \notin fv(\lambda x. e)$ . Once the function is defined, the equality  $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho|_{fv e}}$  can be proved.

With that, the desired equation for  $\llbracket \lambda x. e \rrbracket_{\varrho}$  follows. The same trick is applied to the equation for  $\llbracket \text{let } \Gamma \text{ in body} \rrbracket_{\varrho}$ .

This allows us to use the type  $var \Rightarrow Value$  for the semantic environments and considerably simplifies the formalization compared to [Bre13].

### 1.2.3 No type *assn*

The nominal package provides means to define types that are alpha-equivalence classes, and we use that to define our type *exp*, which contains a constructor *let binds in expr*. The desired type of the parameter for the binding is  $(var \times exp)$  *list*, but the Nominal package does not support such nested recursion, and requires a mutual recursive definition with a custom type (*assn*) with constructors *ANil* and *ACons* that is isomorphic to  $(var \times exp)$  *list*. In [Bre13], this type and conversion functions from and to  $(var \times exp)$  *list* cluttered the whole development. In the present work we improved this by defining the type with a “temporary” constructor *LetA*. Afterwards we define conversions functions and the desired constructor *Let*, and re-state all lemmas produced by the Nominal package (such as type exhaustiveness, distinctiveness of constructors and the induction rules) with that constructor. From that point on, the development is free of the crutch *assn*.

In short, the notable changes in this work over [Bre13] are:

- We consider  $\sqcup$  to be a right-sided update and do discuss neither the problem with  $\sqcup$  denoting the least upper bound, nor possible solutions.
- This, a simpler choice for the type of semantic environments and a better definition of the type for terms, considerably simplifies the work.
- Most importantly, this work contains a complete and formal proof of the adequacy of Launchbury’s semantics.

### 1.3 Related work

Lidia Sánchez-Gil, Mercedes Hidalgo-Herrero and Yolanda Ortega-Mallén have worked on formal aspects of Launchbury’s semantics as well.

They identified a step in his adequacy proof relating the standard and the resourced denotational semantics that is not as trivial as it seems at first and worked out a detailed pen-and-paper proof [SGHHOM11], where they first construct a similarity relation  $\_ \triangleleft \_$  between the standard semantic domain (*Value*) and the resourced domain (*CValue*) and show that the denotation semantics yield similar results ( $\varrho \triangleleft^* \sigma \implies \llbracket e \rrbracket_\varrho \triangleleft (\mathcal{N} \llbracket e \rrbracket_\sigma) \cdot C^\infty$ ), which is one step in the adequacy proof. We formalized this (Sections 8.1 and 8.2), identifying and fixing a mistake in the paper (Lemma 2.3(3) does not hold; the problem can be fixed by applying an extra round of take-induction in the proof of Proposition 9).

Currently, they are working on completing the adequacy proof as outlined by Launchbury, i.e. by going via the alternative natural semantics given in [Lau93], which differs from the semantics above in that the application rule works with an indirection on the heap instead of a substitution and that the variable rule has no blackholing and no update. In [SGHHOM14], they relate the original semantics with one where indirections have been introduced. The next step, modifying the variable rule, is under development. Once that is done they can close the loop and have completed Launchbury’s work.

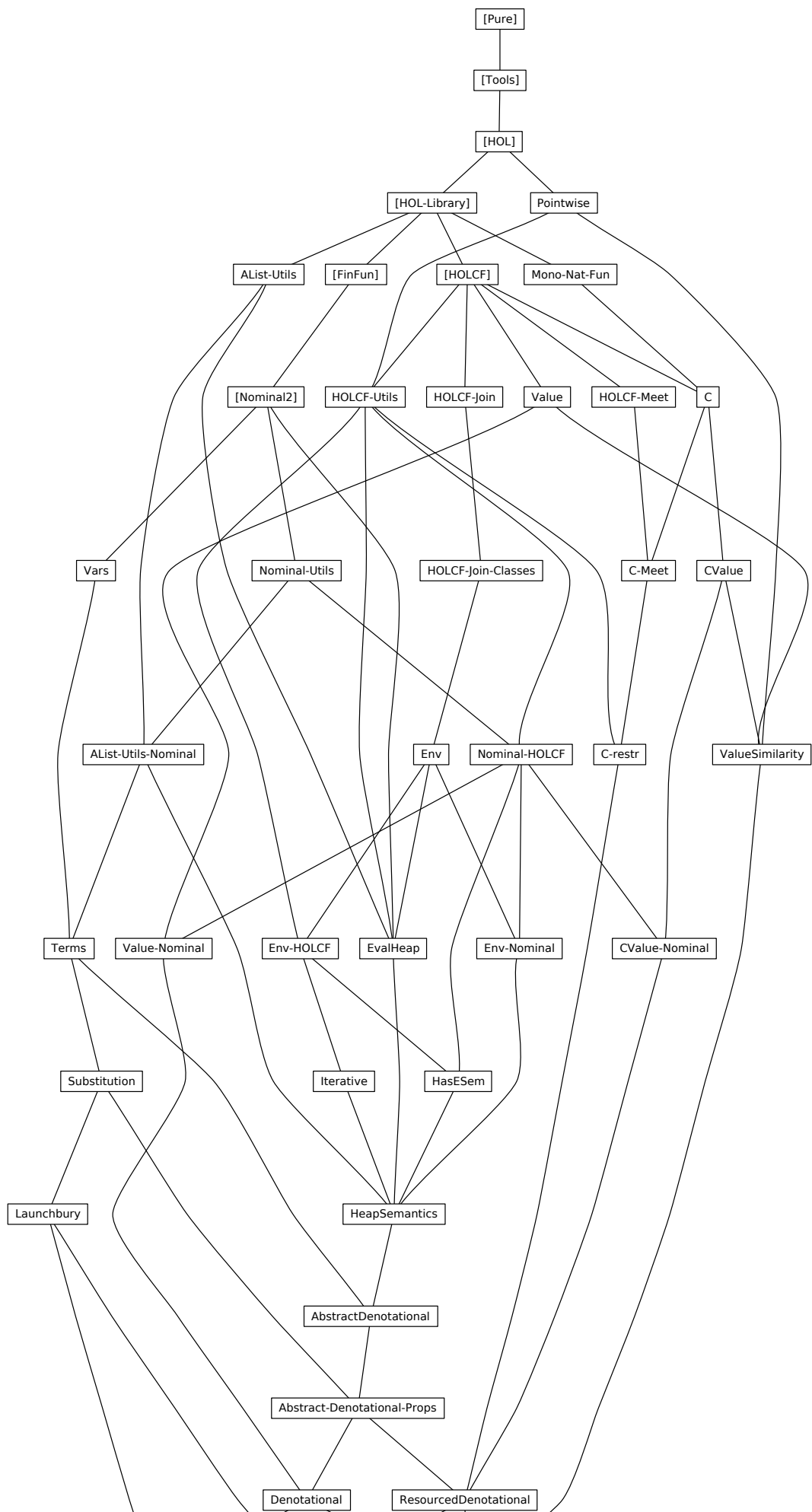
This work proves the adequacy as stated by Launchbury as well, but in contrast to his proof outline no alternative operational semantics is introduced. The problems of indirection vs. substitution and of blackholing is solved on the denotational side instead, which turned out to be much easier than proving the various operational semantics to be equivalent.

### 1.4 Theory overview

The following chapters contain the complete Isabelle theories, with one section per theory. Their interdependencies are visualized in Figure 1.

Chapter 2 contains auxiliary theories, not necessarily tied to Launchbury’s semantics. The base theories are kept independent of Nominal and HOLCF where possible, the lemmas combining them are in theories of their own, creatively named by appending *-Nominal* resp. *-HOLCF*. You will find these theories:

- A definition for lifting a relation point-wise (*Pointwise*).
- A collection of definition related to associative lists (*AList-Utils*, *AList-Utils-Nominal*).
- A characterization of monotonous functions  $\mathbb{N} \rightarrow \mathbb{N}$  (*Mono-Nat-Fun*).
- General utility functions extending Nominal (*Nominal-Utils*).



- General utility functions extending HOLCF (*HOLCF-Utills*).
- Binary meets in the context of HOLCF (*HOLCF-Meet*).
- A theory combining notions from HOLCF and Nominal, e.g. continuity of permutation (*Nominal-HOLCF*).
- A theory for working with pcpo-valued functions as semantic environments (*Env*, *Env-Nominal*, *Env-HOLCF*).
- A function *evalHeap* that converts between associative lists and functions. (*Eval-Heap*)

Chapter 3 defines the syntax and Launchbury’s natural semantics.

Chapter 4 sets the stage for the denotational semantics by defining a locale *semantic-domain* for denotational domains, and an instantiation for the standard domain.

Chapter 5 defines the denotational semantics. It also introduces the locale *has-ESem* which abstracts over the value semantics when defining the semantics of heaps.

Chapter 6 defines the resourced denotational semantics.

Chapter 7 proves the correctness of Launchbury’s semantics with regard to both denotational semantics. We need the correctness with regard to the resourced semantics in the adequacy proof.

Chapter 8 proves the two denotational semantics related, which is used in

Chapter 9, where finally the adequacy is proved.

## 1.5 Acknowledgements

I’d like to thank Lidia Sánchez-Gil, Mercedes Hidalgo-Herrero and Yolanda Ortega-Mallén for inviting me to Madrid to discuss our respective approaches.

This work was supported by the Deutsche Telekom Stiftung.

## 2 Auxiliary theories

### 2.1 Pointwise

theory *Pointwise* imports *Main* begin

Lifting a relation to a function.

**definition** *pointwise* **where** *pointwise*  $P\ m\ m' = (\forall\ x.\ P\ (m\ x)\ (m'\ x))$

**lemma** *pointwiseI*[*intro*]:  $(\bigwedge\ x.\ P\ (m\ x)\ (m'\ x)) \implies \textit{pointwise}\ P\ m\ m'$  **unfolding** *pointwise-def*  
**by** *blast*

end

### 2.2 AList-Utills

theory *AList-Utills*

imports *Main* *HOL-Library.AList*

begin

**declare** *implies-True-equals* [*simp*] *False-implies-equals*[*simp*]

We want to have *delete* and *update* back in the namespace.

**abbreviation** *delete* **where** *delete*  $\equiv$  *AList.delete*

**abbreviation** *update* **where** *update*  $\equiv$  *AList.update*

**abbreviation** *restrictA* **where** *restrictA*  $\equiv$  *AList.restrict*

**abbreviation** *clearjunk* **where** *clearjunk*  $\equiv$  *AList.clearjunk*

**lemmas** *restrict-eq* = *AList.restrict-eq*

**and** *delete-eq* = *AList.delete-eq*

**lemma** *restrictA-append*:  $\textit{restrictA}\ S\ (a\ @\ b) = \textit{restrictA}\ S\ a\ @\ \textit{restrictA}\ S\ b$

**unfolding** *restrict-eq* **by** (*rule filter-append*)

**lemma** *length-restrictA-le*:  $\textit{length}\ (\textit{restrictA}\ S\ a) \leq \textit{length}\ a$

**by** (*metis length-filter-le restrict-eq*)

#### 2.2.1 The domain of an associative list

**definition** *domA*

**where** *domA*  $h = \textit{fst}\ \textit{set}\ h$

**lemma** *domA-append*[*simp*]:  $\textit{domA}\ (a\ @\ b) = \textit{domA}\ a\ \cup\ \textit{domA}\ b$

**and** [*simp*]:  $\textit{domA}\ ((v,e)\ \#\ h) = \textit{insert}\ v\ (\textit{domA}\ h)$

**and** [*simp*]:  $\textit{domA}\ (p\ \#\ h) = \textit{insert}\ (\textit{fst}\ p)\ (\textit{domA}\ h)$

**and** [*simp*]:  $\textit{domA}\ [] = \{\}$

**by** (*auto simp add: domA-def*)

**lemma** *domA-from-set*:

$(x, e) \in \text{set } h \implies x \in \text{domA } h$

**by** (*induct h, auto*)

**lemma** *finite-domA[simp]*:

$\text{finite } (\text{domA } \Gamma)$

**by** (*auto simp add: domA-def*)

**lemma** *domA-delete[simp]*:

$\text{domA } (\text{delete } x \Gamma) = \text{domA } \Gamma - \{x\}$

**by** (*induct \Gamma auto*)

**lemma** *domA-restrictA[simp]*:

$\text{domA } (\text{restrictA } S \Gamma) = \text{domA } \Gamma \cap S$

**by** (*induct \Gamma auto*)

**lemma** *delete-not-domA[simp]*:

$x \notin \text{domA } \Gamma \implies \text{delete } x \Gamma = \Gamma$

**by** (*induct \Gamma auto*)

**lemma** *deleted-not-domA*:  $x \notin \text{domA } (\text{delete } x \Gamma)$

**by** (*induct \Gamma auto*)

**lemma** *dom-map-of-conv-domA*:

$\text{dom } (\text{map-of } \Gamma) = \text{domA } \Gamma$

**by** (*induct \Gamma (auto simp add: dom-if)*)

**lemma** *domA-map-of-Some-the*:

$x \in \text{domA } \Gamma \implies \text{map-of } \Gamma x = \text{Some } (\text{the } (\text{map-of } \Gamma x))$

**by** (*induct \Gamma (auto simp add: dom-if)*)

**lemma** *domA-clearjunk[simp]*:  $\text{domA } (\text{clearjunk } \Gamma) = \text{domA } \Gamma$

**unfolding** *domA-def* **using** *dom-clearjunk*.

**lemma** *the-map-option-domA[simp]*:  $x \in \text{domA } \Gamma \implies \text{the } (\text{map-option } f (\text{map-of } \Gamma x)) = f$   
 $(\text{the } (\text{map-of } \Gamma x))$

**by** (*induction \Gamma auto*)

**lemma** *map-of-domAD*:  $\text{map-of } \Gamma x = \text{Some } e \implies x \in \text{domA } \Gamma$

**using** *dom-map-of-conv-domA* **by** *fastforce*

**lemma** *restrictA-noop*:  $\text{domA } \Gamma \subseteq S \implies \text{restrictA } S \Gamma = \Gamma$

**unfolding** *restrict-eq* **by** (*induction \Gamma auto*)

**lemma** *restrictA-cong*:

$(\bigwedge x. x \in \text{domA } m1 \implies x \in V \longleftrightarrow x \in V') \implies m1 = m2 \implies \text{restrictA } V m1 = \text{restrictA } V' m2$

**unfolding** *restrict-eq* **by** (*induction m1 arbitrary: m2 auto*)

## 2.2.2 Other lemmas about associative lists

**lemma** *delete-set-none*:  $(\text{map-of } l)(x := \text{None}) = \text{map-of } (\text{delete } x \ l)$

**proof** *(induct l)*  
**case** *Nil* **thus** *?case* **by** *simp*  
**case** *(Cons l ls)*  
**from** *this[symmetric]*  
**show** *?case*  
**by** *(cases fst l = x) auto*  
**qed**

**lemma** *list-size-delete[simp]*:  $\text{size-list size } (\text{delete } x \ l) < \text{Suc } (\text{size-list size } l)$   
**by** *(induct l) auto*

**lemma** *delete-append[simp]*:  $\text{delete } x \ (l1 \ @ \ l2) = \text{delete } x \ l1 \ @ \ \text{delete } x \ l2$   
**unfolding** *AList.delete-eq* **by** *simp*

**lemma** *map-of-delete-insert*:  
**assumes**  $\text{map-of } \Gamma \ x = \text{Some } e$   
**shows**  $\text{map-of } ((x,e) \# \text{delete } x \ \Gamma) = \text{map-of } \Gamma$   
**using** *assms* **by** *(induct \Gamma) (auto split:prod.split)*

**lemma** *map-of-delete-iff[simp]*:  $\text{map-of } (\text{delete } x \ \Gamma) \ xa = \text{Some } e \iff (\text{map-of } \Gamma \ xa = \text{Some } e) \wedge xa \neq x$   
**by** *(metis delete-conv fun-upd-same map-of-delete option.distinct(1))*

**lemma** *map-add-domA[simp]*:  
 $x \in \text{domA } \Gamma \implies (\text{map-of } \Delta \ ++ \ \text{map-of } \Gamma) \ x = \text{map-of } \Gamma \ x$   
 $x \notin \text{domA } \Gamma \implies (\text{map-of } \Delta \ ++ \ \text{map-of } \Gamma) \ x = \text{map-of } \Delta \ x$   
**apply** *(metis dom-map-of-conv-domA map-add-dom-app-simps(1))*  
**apply** *(metis dom-map-of-conv-domA map-add-dom-app-simps(3))*  
**done**

**lemma** *set-delete-subset*:  $\text{set } (\text{delete } k \ al) \subseteq \text{set } al$   
**by** *(auto simp add: delete-eq)*

**lemma** *dom-delete-subset*:  $\text{snd } ' \ \text{set } (\text{delete } k \ al) \subseteq \text{snd } ' \ \text{set } al$   
**by** *(auto simp add: delete-eq)*

**lemma** *map-ran-cong[fundef-cong]*:  
 $\llbracket \bigwedge x . x \in \text{set } m1 \implies f1 \ (fst \ x) \ (snd \ x) = f2 \ (fst \ x) \ (snd \ x) ; m1 = m2 \rrbracket$   
 $\implies \text{map-ran } f1 \ m1 = \text{map-ran } f2 \ m2$   
**by** *(induction m1 arbitrary: m2) auto*

**lemma** *domA-map-ran[simp]*:  $\text{domA } (\text{map-ran } f \ m) = \text{domA } m$   
**unfolding** *domA-def* **by** *(rule dom-map-ran)*

**lemma** *map-ran-delete*:  
 $\text{map-ran } f \ (\text{delete } x \ \Gamma) = \text{delete } x \ (\text{map-ran } f \ \Gamma)$

by (induction  $\Gamma$ ) auto

**lemma** *map-ran-restrictA*:

$map\text{-}ran\ f\ (restrictA\ V\ \Gamma) = restrictA\ V\ (map\text{-}ran\ f\ \Gamma)$

by (induction  $\Gamma$ ) auto

**lemma** *map-ran-append*:

$map\text{-}ran\ f\ (\Gamma @ \Delta) = map\text{-}ran\ f\ \Gamma @ map\text{-}ran\ f\ \Delta$

by (induction  $\Gamma$ ) auto

### 2.2.3 Syntax for map comprehensions

**definition** *mapCollect* :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'c)  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  'c set

where  $mapCollect\ f\ m = \{f\ k\ v \mid k\ v . m\ k = Some\ v\}$

**syntax**

-MapCollect :: 'c  $\Rightarrow$  pttm  $\Rightarrow$  pttm  $\Rightarrow$  'a  $\rightarrow$  'b  $\Rightarrow$  'c set  $\langle\langle(1\{-\ /-\ /-\ /-\ /-\ /-\})\rangle\rangle$

**syntax-consts**

-MapCollect == mapCollect

**translations**

$\{e \mid k \mapsto v \in m\} == CONST\ mapCollect\ (\lambda k\ v . e)\ m$

**lemma** *mapCollect-empty[simp]*:  $\{f\ k\ v \mid k \mapsto v \in Map.empty\} = \{\}$

unfolding *mapCollect-def* by *simp*

**lemma** *mapCollect-const[simp]*:

$m \neq Map.empty \implies \{e \mid k \mapsto v \in m\} = \{e\}$

unfolding *mapCollect-def* by *auto*

**lemma** *mapCollect-cong[fundef-cong]*:

$(\bigwedge k\ v . m1\ k = Some\ v \implies f1\ k\ v = f2\ k\ v) \implies m1 = m2 \implies mapCollect\ f1\ m1 = mapCollect\ f2\ m2$

unfolding *mapCollect-def* by *force*

**lemma** *mapCollectE[elim!]*:

assumes  $x \in \{f\ k\ v \mid k \mapsto v \in m\}$

obtains  $k\ v$  where  $m\ k = Some\ v$  and  $x = f\ k\ v$

using *assms* by (*auto simp add: mapCollect-def*)

**lemma** *mapCollectI[intro]*:

assumes  $m\ k = Some\ v$

shows  $f\ k\ v \in \{f\ k\ v \mid k \mapsto v \in m\}$

using *assms* by (*auto simp add: mapCollect-def*)

**lemma** *ball-mapCollect[simp]*:

$(\forall x \in \{f\ k\ v \mid k \mapsto v \in m\} . P\ x) \iff (\forall k\ v . m\ k = Some\ v \longrightarrow P\ (f\ k\ v))$

by (*auto simp add: mapCollect-def*)

**lemma** *image-mapCollect[simp]*:  
 $g \{ f k v \mid k \mapsto v \in m \} = \{ g (f k v) \mid k \mapsto v \in m \}$   
**by** (*auto simp add: mapCollect-def*)

**lemma** *mapCollect-map-upd[simp]*:  
 $mapCollect f (m(k \mapsto v)) = insert (f k v) (mapCollect f (m(k := None)))$   
**unfolding** *mapCollect-def* **by** *auto*

**definition** *mapCollectFilter* ::  $('a \Rightarrow 'b \Rightarrow (bool \times 'c)) \Rightarrow ('a \rightarrow 'b) \Rightarrow 'c \text{ set}$   
**where**  $mapCollectFilter f m = \{snd (f k v) \mid k v . m k = Some v \wedge fst (f k v)\}$

**syntax**

*-MapCollectFilter* ::  $'c \Rightarrow p\text{trn} \Rightarrow p\text{trn} \Rightarrow ('a \rightarrow 'b) \Rightarrow bool \Rightarrow 'c \text{ set}$  ( $\langle (1\{-\}/\{-/\mapsto/\{-/\in/\{-/\cdot/\cdot\}\rangle\rangle$ )

**syntax-consts**

*-MapCollectFilter* == *mapCollectFilter*

**translations**

$\{e \mid k \mapsto v \in m . P\} == CONST mapCollectFilter (\lambda k v . (P,e)) m$

**lemma** *mapCollectFilter-const-False[simp]*:  
 $\{e \mid k \mapsto v \in m . False\} = \{\}$   
**unfolding** *mapCollect-def mapCollectFilter-def* **by** *simp*

**lemma** *mapCollectFilter-const-True[simp]*:  
 $\{e \mid k \mapsto v \in m . True\} = \{e \mid k \mapsto v \in m\}$   
**unfolding** *mapCollect-def mapCollectFilter-def* **by** *simp*

**end**

## 2.3 Mono-Nat-Fun

**theory** *Mono-Nat-Fun*

**imports** *HOL-Library.Infinite-Set*

**begin**

The following lemma proves that a monotonous function from and to the natural numbers is either eventually constant or unbounded.

**lemma** *nat-mono-characterization*:

**fixes**  $f :: nat \Rightarrow nat$

**assumes** *mono f*

**obtains**  $n$  **where**  $\bigwedge m . n \leq m \implies f n = f m \mid \bigwedge m . \exists n . m \leq f n$

**proof** (*cases finite (range f)*)

**case** *True*

**from** *Max-in[OF True]*

**obtain**  $n$  **where**  $Max: f n = Max (range f)$  **by** *auto*

```

show thesis
proof(rule that(1))
  fix m
  assume  $n \leq m$ 
  hence  $f n \leq f m$  using ‹mono f› by (metis monoD)
  also
  have  $f m \leq f n$  unfolding Max by (rule Max-ge[OF True rangeI])
  finally
  show  $f n = f m$ .
qed
next
case False
thus thesis by (fastforce intro: that(2) simp add: infinite-nat-iff-unbounded-le)
qed
end

```

## 2.4 Nominal-Utills

```

theory Nominal-Utills
imports Nominal2.Nominal2 HOL-Library.AList
begin

```

### 2.4.1 Lemmas helping with equivariance proofs

```

lemma perm-rel-lemma:
  assumes  $\bigwedge \pi x y. r (\pi \cdot x) (\pi \cdot y) \implies r x y$ 
  shows  $r (\pi \cdot x) (\pi \cdot y) \longleftrightarrow r x y$  (is ?l  $\longleftrightarrow$  ?r)
by (metis (full-types) assms permute-minus-cancel(2))

```

```

lemma perm-rel-lemma2:
  assumes  $\bigwedge \pi x y. r x y \implies r (\pi \cdot x) (\pi \cdot y)$ 
  shows  $r x y \longleftrightarrow r (\pi \cdot x) (\pi \cdot y)$  (is ?l  $\longleftrightarrow$  ?r)
by (metis (full-types) assms permute-minus-cancel(2))

```

```

lemma fun-eqvtI:
  assumes f-eqvt[eqvt]:  $(\bigwedge p x. p \cdot (f x) = f (p \cdot x))$ 
  shows  $p \cdot f = f$  by perm-simp rule

```

```

lemma eqvt-at-apply:
  assumes eqvt-at  $f x$ 
  shows  $(p \cdot f) x = f x$ 
by (metis (opaque-lifting, no-types) assms eqvt-at-def permute-fun-def permute-minus-cancel(1))

```

```

lemma eqvt-at-apply':
  assumes eqvt-at  $f x$ 
  shows  $p \cdot f x = f (p \cdot x)$ 
by (metis (opaque-lifting, no-types) assms eqvt-at-def)

```

**lemma** *eqvt-at-apply''*:  
**assumes** *eqvt-at f x*  
**shows**  $(p \cdot f) (p \cdot x) = f (p \cdot x)$   
**by** (*metis (opaque-lifting, no-types) assms eqvt-at-def permute-fun-def permute-minus-cancel(1)*)

**lemma** *size-list- $eqvt$* [*eqvt*]:  $p \cdot \text{size-list } f \ x = \text{size-list } (p \cdot f) (p \cdot x)$   
**proof** (*induction x*)  
**case** (*Cons x xs*)  
**have**  $f \ x = p \cdot (f \ x)$  **by** (*simp add: permute-pure*)  
**also have**  $\dots = (p \cdot f) (p \cdot x)$  **by** *simp*  
**with** *Cons*  
**show** ?*case* **by** (*auto simp add: permute-pure*)  
**qed** *simp*

## 2.4.2 Freshness via equivariance

**lemma** *eqvt-fresh-cong1*:  $(\bigwedge p \ x. p \cdot (f \ x) = f (p \cdot x)) \implies a \ \# \ x \implies a \ \# \ f \ x$   
**apply** (*rule fresh-fun- $eqvt$ -app[of f]*)  
**apply** (*rule eqvtI*)  
**apply** (*rule eq-reflection*)  
**apply** (*rule ext*)  
**apply** (*metis permute-fun-def permute-minus-cancel(1)*)  
**apply** *assumption*  
**done**

**lemma** *eqvt-fresh-cong2*:  
**assumes** *eqvt*:  $(\bigwedge p \ x \ y. p \cdot (f \ x \ y) = f (p \cdot x) (p \cdot y))$   
**and** *fresh1*:  $a \ \# \ x$  **and** *fresh2*:  $a \ \# \ y$   
**shows**  $a \ \# \ f \ x \ y$   
**proof** –  
**have** *eqvt*  $(\lambda (x,y). f \ x \ y)$   
**using** *eqvt*  
**apply** –  
**apply** (*auto simp add: eqvt-def*)  
**apply** (*rule ext*)  
**apply** *auto*  
**by** (*metis permute-minus-cancel(1)*)  
**moreover**  
**have**  $a \ \# \ (x, y)$  **using** *fresh1 fresh2* **by** *auto*  
**ultimately**  
**have**  $a \ \# \ (\lambda (x,y). f \ x \ y) (x, y)$  **by** (*rule fresh-fun- $eqvt$ -app*)  
**thus** ?*thesis* **by** *simp*  
**qed**

**lemma** *eqvt-fresh-star-cong1*:  
**assumes** *eqvt*:  $(\bigwedge p \ x. p \cdot (f \ x) = f (p \cdot x))$   
**and** *fresh1*:  $a \ \#^* \ x$   
**shows**  $a \ \#^* \ f \ x$

by (metis fresh-star-def eqvt-fresh-cong1 assms)

**lemma** eqvt-fresh-star-cong2:

**assumes** eqvt:  $(\bigwedge p x y. p \cdot (f x y) = f (p \cdot x) (p \cdot y))$

**and** fresh1:  $a \#^* x$  **and** fresh2:  $a \#^* y$

**shows**  $a \#^* f x y$

by (metis fresh-star-def eqvt-fresh-cong2 assms)

**lemma** eqvt-fresh-cong3:

**assumes** eqvt:  $(\bigwedge p x y z. p \cdot (f x y z) = f (p \cdot x) (p \cdot y) (p \cdot z))$

**and** fresh1:  $a \# x$  **and** fresh2:  $a \# y$  **and** fresh3:  $a \# z$

**shows**  $a \# f x y z$

**proof**–

**have** eqvt  $(\lambda (x,y,z). f x y z)$

**using** eqvt

**apply** –

**apply** (auto simp add: eqvt-def)

**apply** (rule ext)

**apply** auto

**by** (metis permute-minus-cancel(1))

**moreover**

**have**  $a \# (x, y, z)$  **using** fresh1 fresh2 fresh3 **by** auto

**ultimately**

**have**  $a \# (\lambda (x,y,z). f x y z) (x, y, z)$  **by** (rule fresh-fun-eqvt-app)

**thus** ?thesis **by** simp

**qed**

**lemma** eqvt-fresh-star-cong3:

**assumes** eqvt:  $(\bigwedge p x y z. p \cdot (f x y z) = f (p \cdot x) (p \cdot y) (p \cdot z))$

**and** fresh1:  $a \#^* x$  **and** fresh2:  $a \#^* y$  **and** fresh3:  $a \#^* z$

**shows**  $a \#^* f x y z$

by (metis fresh-star-def eqvt-fresh-cong3 assms)

### 2.4.3 Additional simplification rules

**lemma** not-self-fresh[simp]:  $\text{atom } x \# x \longleftrightarrow \text{False}$

by (metis fresh-at-base(2))

**lemma** fresh-star-singleton:  $\{ x \} \#^* e \longleftrightarrow x \# e$

by (simp add: fresh-star-def)

### 2.4.4 Additional equivariance lemmas

**lemma** eqvt-cases:

**fixes**  $f x \pi$

**assumes** eqvt:  $\bigwedge x. \pi \cdot f x = f (\pi \cdot x)$

**obtains**  $f x f (\pi \cdot x) \mid \neg f x \quad \neg f (\pi \cdot x)$

**using** assms[symmetric]

**by** (cases  $f x$ ) auto

**lemma** *range-eqvt*:  $\pi \cdot \text{range } Y = \text{range } (\pi \cdot Y)$   
**unfolding** *image-eqvt UNIV-eqvt ..*

**lemma** *case-option-eqvt*[*eqvt*]:  
 $\pi \cdot \text{case-option } d f x = \text{case-option } (\pi \cdot d) (\pi \cdot f) (\pi \cdot x)$   
**by**(*cases x*)(*simp-all*)

**lemma** *supp-option-eqvt*:  
 $\text{supp } (\text{case-option } d f x) \subseteq \text{supp } d \cup \text{supp } f \cup \text{supp } x$   
**apply** (*cases x*)  
**apply** (*auto simp add: supp-Some*)  
**apply** (*metis (mono-tags) Un-iff subsetCE supp-fun-app*)  
**done**

**lemma** *funpow-eqvt*[*simp,eqvt*]:  
 $\pi \cdot ((f :: 'a \Rightarrow 'a::pt) \overset{\sim}{\sim} n) = (\pi \cdot f) \overset{\sim}{\sim} (\pi \cdot n)$   
**apply** (*induct n*)  
**apply** *simp*  
**apply** (*rule ext*)  
**apply** *simp*  
**apply** *perm-simp*  
**apply** *simp*  
**done**

**lemma** *delete-eqvt*[*eqvt*]:  
 $\pi \cdot \text{AList.delete } x \Gamma = \text{AList.delete } (\pi \cdot x) (\pi \cdot \Gamma)$   
**by** (*induct*  $\Gamma$ , *auto*)

**lemma** *restrict-eqvt*[*eqvt*]:  
 $\pi \cdot \text{AList.restrict } S \Gamma = \text{AList.restrict } (\pi \cdot S) (\pi \cdot \Gamma)$   
**unfolding** *AList.restrict-eq* **by** *perm-simp rule*

**lemma** *supp-restrict*:  
 $\text{supp } (\text{AList.restrict } S \Gamma) \subseteq \text{supp } \Gamma$   
**by** (*induction*  $\Gamma$ ) (*auto simp add: supp-Pair supp-Cons*)

**lemma** *clearjunk-eqvt*[*eqvt*]:  
 $\pi \cdot \text{AList.clearjunk } \Gamma = \text{AList.clearjunk } (\pi \cdot \Gamma)$   
**by** (*induction*  $\Gamma$  *rule: clearjunk.induct*) *auto*

**lemma** *map-ran-eqvt*[*eqvt*]:  
 $\pi \cdot \text{map-ran } f \Gamma = \text{map-ran } (\pi \cdot f) (\pi \cdot \Gamma)$   
**by** (*induct*  $\Gamma$ , *auto*)

**lemma** *dom-perm*:  
 $\text{dom } (\pi \cdot f) = \pi \cdot (\text{dom } f)$   
**unfolding** *dom-def* **by** (*perm-simp*) (*simp*)

**lemmas** *dom-perm-rev*[*simp,eqvt*] = *dom-perm*[*symmetric*]

**lemma** *ran-perm*[*simp*]:

$$\pi \cdot (\text{ran } f) = \text{ran } (\pi \cdot f)$$

**unfolding** *ran-def* **by** (*perm-simp*) (*simp*)

**lemma** *map-add-eqvt*[*eqvt*]:

$$\pi \cdot (m1 ++ m2) = (\pi \cdot m1) ++ (\pi \cdot m2)$$

**unfolding** *map-add-def*

**by** (*perm-simp*, *rule*)

**lemma** *map-of-eqvt*[*eqvt*]:

$$\pi \cdot \text{map-of } l = \text{map-of } (\pi \cdot l)$$

**apply** (*induct* *l*)

**apply** (*simp* *add: permute-fun-def*)

**apply** *simp*

**apply** *perm-simp*

**apply** *auto*

**done**

**lemma** *concat-eqvt*[*eqvt*]:  $\pi \cdot \text{concat } l = \text{concat } (\pi \cdot l)$

**by** (*induction* *l*)(*auto* *simp* *add: append-eqvt*)

**lemma** *tranclp-eqvt*[*eqvt*]:  $\pi \cdot \text{tranclp } P v_1 v_2 = \text{tranclp } (\pi \cdot P) (\pi \cdot v_1) (\pi \cdot v_2)$

**unfolding** *tranclp-def* **by** *perm-simp* *rule*

**lemma** *rtranclp-eqvt*[*eqvt*]:  $\pi \cdot \text{rtranclp } P v_1 v_2 = \text{rtranclp } (\pi \cdot P) (\pi \cdot v_1) (\pi \cdot v_2)$

**unfolding** *rtranclp-def* **by** *perm-simp* *rule*

**lemma** *Set-filter-eqvt*[*eqvt*]:  $\pi \cdot \text{Set.filter } P S = \text{Set.filter } (\pi \cdot P) (\pi \cdot S)$

**by** *simp*

**lemma** *Sigma-eqvt'*[*eqvt*]:  $\pi \cdot \text{Sigma} = \text{Sigma}$

**apply** (*rule* *ext*)

**apply** (*rule* *ext*)

**apply** (*subst* *permute-fun-def*)

**apply** (*subst* *permute-fun-def*)

**unfolding** *Sigma-def*

**apply** *perm-simp*

**apply** (*simp* *add: permute-self*)

**done**

**lemma** *override-on-eqvt*[*eqvt*]:

$$\pi \cdot (\text{override-on } m1 m2 S) = \text{override-on } (\pi \cdot m1) (\pi \cdot m2) (\pi \cdot S)$$

**by** (*auto* *simp* *add: override-on-def*)

**lemma** *card-eqvt*[*eqvt*]:

$$\pi \cdot (\text{card } S) = \text{card } (\pi \cdot S)$$

**by** (*cases* *finite* *S*, *induct* *rule: finite-induct*) (*auto* *simp* *add: card-insert-if mem-permute-iff permute-pure*)

**lemma** *Projl-permute*:  
 assumes  $a: \exists y. f = \text{Inl } y$   
 shows  $(p \cdot (\text{Sum-Type.proj1 } f)) = \text{Sum-Type.proj1 } (p \cdot f)$   
 using  $a$  by *auto*

**lemma** *Projr-permute*:  
 assumes  $a: \exists y. f = \text{Inr } y$   
 shows  $(p \cdot (\text{Sum-Type.proj2 } f)) = \text{Sum-Type.proj2 } (p \cdot f)$   
 using  $a$  by *auto*

### 2.4.5 Freshness lemmas

**lemma** *fresh-list-elem*:  
 assumes  $a \# \Gamma$   
 and  $e \in \text{set } \Gamma$   
 shows  $a \# e$   
 using *assms*  
 by(*induct*  $\Gamma$ )(*auto simp add: fresh-Cons*)

**lemma** *set-not-fresh*:  
 $x \in \text{set } L \implies \neg(\text{atom } x \# L)$   
 by (*metis fresh-list-elem not-self-fresh*)

**lemma** *pure-fresh-star[simp]*:  $a \#* (x :: 'a :: \text{pure})$   
 by (*simp add: fresh-star-def pure-fresh*)

**lemma** *supp-set-mem*:  $x \in \text{set } L \implies \text{supp } x \subseteq \text{supp } L$   
 by (*induct*  $L$ ) (*auto simp add: supp-Cons*)

**lemma** *set-supp-mono*:  $\text{set } L \subseteq \text{set } L2 \implies \text{supp } L \subseteq \text{supp } L2$   
 by (*induct*  $L$ )(*auto simp add: supp-Cons supp-Nil dest:supp-set-mem*)

**lemma** *fresh-star-at-base*:  
 fixes  $x :: 'a :: \text{at-base}$   
 shows  $S \#* x \iff \text{atom } x \notin S$   
 by (*metis fresh-at-base(2) fresh-star-def*)

### 2.4.6 Freshness and support for subsets of variables

**lemma** *supp-mono*:  $\text{finite } (B :: 'a :: \text{fs set}) \implies A \subseteq B \implies \text{supp } A \subseteq \text{supp } B$   
 by (*metis infinite-super subset-Un-eq supp-of-finite-union*)

**lemma** *fresh-subset*:  
 $\text{finite } B \implies x \# (B :: 'a :: \text{at-base set}) \implies A \subseteq B \implies x \# A$   
 by (*auto dest:supp-mono simp add: fresh-def*)

**lemma** *fresh-star-subset*:

$finite\ B \implies x \#^* (B :: 'a::at-base\ set) \implies A \subseteq B \implies x \#^* A$   
**by** (*metis fresh-star-def fresh-subset*)

**lemma** *fresh-star-set-subset*:

$x \#^* (B :: 'a::at-base\ list) \implies set\ A \subseteq set\ B \implies x \#^* A$   
**by** (*metis fresh-star-set fresh-star-subset[OF finite-set]*)

## 2.4.7 The set of free variables of an expression

**definition** *fv* ::  $'a::pt \Rightarrow 'b::at-base\ set$   
**where**  $fv\ e = \{v.\ atom\ v \in\ supp\ e\}$

**lemma** *fv-eqvt[simp,eqvt]*:  $\pi \cdot (fv\ e) = fv\ (\pi \cdot e)$   
**unfolding** *fv-def* **by** *simp*

**lemma** *fv-Nil[simp]*:  $fv\ [] = \{\}$   
**by** (*auto simp add: fv-def supp-Nil*)

**lemma** *fv-Cons[simp]*:  $fv\ (x \#\ xs) = fv\ x \cup fv\ xs$   
**by** (*auto simp add: fv-def supp-Cons*)

**lemma** *fv-Pair[simp]*:  $fv\ (x, y) = fv\ x \cup fv\ y$   
**by** (*auto simp add: fv-def supp-Pair*)

**lemma** *fv-append[simp]*:  $fv\ (x @\ y) = fv\ x \cup fv\ y$   
**by** (*auto simp add: fv-def supp-append*)

**lemma** *fv-at-base[simp]*:  $fv\ a = \{a::'a::at-base\}$   
**by** (*auto simp add: fv-def supp-at-base*)

**lemma** *fv-pure[simp]*:  $fv\ (a::'a::pure) = \{\}$   
**by** (*auto simp add: fv-def pure-supp*)

**lemma** *fv-set-at-base[simp]*:  $fv\ (l :: ('a :: at-base)\ list) = set\ l$   
**by** (*induction l*) *auto*

**lemma** *flip-not-fv*:  $a \notin fv\ x \implies b \notin fv\ x \implies (a \leftrightarrow b) \cdot x = x$   
**by** (*metis flip-def fresh-def fv-def mem-Collect-eq swap-fresh-fresh*)

**lemma** *fv-not-fresh*:  $atom\ x \# e \longleftrightarrow x \notin fv\ e$   
**unfolding** *fv-def fresh-def* **by** *blast*

**lemma** *fresh-fv*:  $finite\ (fv\ e :: 'a\ set) \implies atom\ (x :: ('a::at-base)) \# (fv\ e :: 'a\ set) \longleftrightarrow atom\ x \# e$   
**unfolding** *fv-def fresh-def*  
**by** (*auto simp add: supp-finite-set-at-base*)

**lemma** *finite-fv[simp]*:  $finite\ (fv\ (e::'a::fs) :: ('b::at-base)\ set)$

**proof** –

**have**  $finite\ (supp\ e)$  **by** (*metis finite-supp*)  
**hence**  $finite\ (atom\ -\ 'supp\ e :: 'b\ set)$   
**apply** (*rule finite-vimageI*)  
**apply** (*rule inj-onI*)  
**apply** (*simp*)

**done**  
**moreover**  
**have**  $(atom - 'supp e :: 'b set) = fv e$  **unfolding** *fv-def* **by** *auto*  
**ultimately**  
**show** *?thesis* **by** *simp*  
**qed**

**definition** *fv-list* ::  $'a::fs \Rightarrow 'b::at-base list$   
**where** *fv-list*  $e = (SOME l. set l = fv e)$

**lemma** *set-fv-list[simp]*:  $set (fv-list e) = (fv e :: ('b::at-base) set)$

**proof**–  
**have** *finite*  $(fv e :: 'b set)$  **by** *(rule finite-fv)*  
**from** *finite-list[OF finite-fv]*  
**obtain**  $l$  **where**  $set l = (fv e :: 'b set)$ .  
**thus** *?thesis*  
**unfolding** *fv-list-def* **by** *(rule someI)*  
**qed**

**lemma** *fresh-fv-list[simp]*:

$a \# (fv-list e :: 'b::at-base list) \longleftrightarrow a \# (fv e :: 'b::at-base set)$

**proof**–

**have**  $a \# (fv-list e :: 'b::at-base list) \longleftrightarrow a \# set (fv-list e :: 'b::at-base list)$   
**by** *(rule fresh-set[symmetric])*  
**also have**  $\dots \longleftrightarrow a \# (fv e :: 'b::at-base set)$  **by** *simp*  
**finally show** *?thesis*.

**qed**

### 2.4.8 Other useful lemmas

**lemma** *pure-permute-id*:  $permute p = (\lambda x. (x::'a::pure))$   
**by** *rule (simp add: permute-pure)*

**lemma** *supp-set-elim-finite*:

**assumes** *finite*  $S$   
**and**  $(m::'a::fs) \in S$   
**and**  $y \in supp m$   
**shows**  $y \in supp S$   
**using** *assms supp-of-finite-sets*  
**by** *auto*

**lemmas** *fresh-star-Cons* = *fresh-star-list(2)*

**lemma** *mem-permute-set*:

**shows**  $x \in p \cdot S \longleftrightarrow (- p \cdot x) \in S$   
**by** *(metis mem-permute-iff permute-minus-cancel(2))*

**lemma** *flip-set-both-not-in*:

**assumes**  $x \notin S$  **and**  $x' \notin S$

**shows**  $((x' \leftrightarrow x) \cdot S) = S$   
**unfolding** *permute-set-def*  
**by** (*auto*) (*metis assms flip-at-base-simps(3)*)+

**lemma** *inj-atom*: *inj atom* **by** (*metis atom-eq-iff injI*)

**lemmas** *image-Int*[*OF inj-atom, simp*]

**lemma** *eqvt-uncurry*:  $eqvt f \implies eqvt (case-prod f)$   
**unfolding** *eqvt-def*  
**by** *perm-simp simp*

**lemma** *supp-fun-app-eqvt2*:  
**assumes** *a*: *eqvt f*  
**shows**  $supp (f x y) \subseteq supp x \cup supp y$

**proof**–

**from** *supp-fun-app-eqvt*[*OF eqvt-uncurry [OF a]*]  
**have**  $supp (case-prod f (x,y)) \subseteq supp (x,y)$ .  
**thus** *?thesis* **by** (*simp add: supp-Pair*)

**qed**

**lemma** *supp-fun-app-eqvt3*:  
**assumes** *a*: *eqvt f*  
**shows**  $supp (f x y z) \subseteq supp x \cup supp y \cup supp z$

**proof**–

**from** *supp-fun-app-eqvt2*[*OF eqvt-uncurry [OF a]*]  
**have**  $supp (case-prod f (x,y) z) \subseteq supp (x,y) \cup supp z$ .  
**thus** *?thesis* **by** (*simp add: supp-Pair*)

**qed**

**lemma** *permute-0*[*simp*]:  $permute 0 = (\lambda x. x)$   
**by** *auto*

**lemma** *permute-comp*[*simp*]:  $permute x \circ permute y = permute (x + y)$  **by** *auto*

**lemma** *map-permute*:  $map (permute p) = permute p$   
**apply** *rule*  
**apply** (*induct-tac x*)  
**apply** *auto*  
**done**

**lemma** *fresh-star-restrictA*[*intro*]:  $a \sharp^* \Gamma \implies a \sharp^* AList.restrict V \Gamma$   
**by** (*induction*  $\Gamma$ ) (*auto simp add: fresh-star-Cons*)

**lemma** *Abs-lst-Nil-eq*[*simp*]:  $[[[]]lst. (x::'a::fs) = [xs]lst. x' \longleftrightarrow (([],x) = (xs, x'))$   
**apply** *rule*

```

apply (frule Abs-lst-fcb2[where  $f = \lambda x y . (x, y)$  and  $as = []$  and  $bs = xs$  and  $c = ()$ ])
apply (auto simp add: fresh-star-def)
done

```

```

lemma Abs-lst-Nil-eq2[simp]:  $[xs]lst. (x::'a::fs) = [[]]lst. x' \longleftrightarrow ((xs, x) = ([], x'))$ 
by (subst eq-commute) auto

```

end

## 2.5 AList-Utills-Nominal

```

theory AList-Utills-Nominal
imports AList-Utills Nominal-Utills
begin

```

### 2.5.1 Freshness lemmas related to associative lists

```

lemma domA-not-fresh:
 $x \in \text{dom} A \Gamma \implies \neg(\text{atom } x \# \Gamma)$ 
by (induct  $\Gamma$ , auto simp add: fresh-Cons fresh-Pair)

```

```

lemma fresh-delete:
assumes  $a \# \Gamma$ 
shows  $a \# \text{delete } v \Gamma$ 
using assms
by (induct  $\Gamma$ )(auto simp add: fresh-Cons)

```

```

lemma fresh-star-delete:
assumes  $S \#* \Gamma$ 
shows  $S \#* \text{delete } v \Gamma$ 
using assms fresh-delete unfolding fresh-star-def by fastforce

```

```

lemma fv-delete-subset:
 $\text{fv}(\text{delete } v \Gamma) \subseteq \text{fv } \Gamma$ 
using fresh-delete unfolding fresh-def fv-def by auto

```

```

lemma fresh-heap-expr:
assumes  $a \# \Gamma$ 
and  $(x, e) \in \text{set } \Gamma$ 
shows  $a \# e$ 
using assms
by (metis fresh-list-elem fresh-Pair)

```

```

lemma fresh-heap-expr':
assumes  $a \# \Gamma$ 
and  $e \in \text{snd } \Gamma \text{ set } \Gamma$ 

```

shows  $a \# e$   
 using *assms*  
 by (*induct*  $\Gamma$ , *auto simp add: fresh-Cons fresh-Pair*)

**lemma** *fresh-star-heap-expr'*:  
 assumes  $S \#* \Gamma$   
 and  $e \in \text{snd } \text{' set } \Gamma$   
 shows  $S \#* e$   
 using *assms*  
 by (*metis fresh-star-def fresh-heap-expr'*)

**lemma** *fresh-map-of*:  
 assumes  $x \in \text{dom}A \ \Gamma$   
 assumes  $a \# \Gamma$   
 shows  $a \# \text{the } (\text{map-of } \Gamma \ x)$   
 using *assms*  
 by (*induct*  $\Gamma$ )(*auto simp add: fresh-Cons fresh-Pair*)

**lemma** *fresh-star-map-of*:  
 assumes  $x \in \text{dom}A \ \Gamma$   
 assumes  $a \#* \Gamma$   
 shows  $a \#* \text{the } (\text{map-of } \Gamma \ x)$   
 using *assms* by (*simp add: fresh-star-def fresh-map-of*)

**lemma** *domA-fv-subset*:  $\text{dom}A \ \Gamma \subseteq \text{fv } \Gamma$   
 by (*induction*  $\Gamma$ ) *auto*

**lemma** *map-of-fv-subset*:  $x \in \text{dom}A \ \Gamma \implies \text{fv } (\text{the } (\text{map-of } \Gamma \ x)) \subseteq \text{fv } \Gamma$   
 by (*induction*  $\Gamma$ ) *auto*

**lemma** *map-of-Some-fv-subset*:  $\text{map-of } \Gamma \ x = \text{Some } e \implies \text{fv } e \subseteq \text{fv } \Gamma$   
 by (*metis domA-from-set map-of-fv-subset map-of-SomeD option.sel*)

## 2.5.2 Equivariance lemmas

**lemma** *domA[eqvt]*:  
 $\pi \cdot \text{dom}A \ \Gamma = \text{dom}A \ (\pi \cdot \Gamma)$   
 by (*simp add: domA-def*)

**lemma** *mapCollect[eqvt]*:  
 $\pi \cdot \text{mapCollect } f \ m = \text{mapCollect } (\pi \cdot f) \ (\pi \cdot m)$   
**unfolding** *mapCollect-def*  
 by *perm-simp rule*

## 2.5.3 Freshness and distinctness

**lemma** *fresh-distinct*:  
 assumes *atom*  $S \#* \Gamma$   
 shows  $S \cap \text{dom}A \ \Gamma = \{\}$   
**proof** –

```

{ fix x
  assume x ∈ S
  moreover
  assume x ∈ domA Γ
  hence atom x ∈ supp Γ
    by (induct Γ)(auto simp add: supp-Cons domA-def supp-Pair supp-at-base)
  ultimately
  have False
    using assms
    by (simp add: fresh-star-def fresh-def)
}
thus S ∩ domA Γ = {} by auto
qed

```

```

lemma fresh-distinct-list:
  assumes atom ' S #* l
  shows S ∩ set l = {}
  using assms
  by (metis disjoint-iff-not-equal fresh-list-elem fresh-star-def image-eqI not-self-fresh)

```

```

lemma fresh-distinct-fv:
  assumes atom ' S #* l
  shows S ∩ fv l = {}
  using assms
  by (metis disjoint-iff-not-equal fresh-star-def fv-not-fresh image-eqI)

```

#### 2.5.4 Pure codomains

```

lemma domA-fv-pure:
  fixes Γ :: ('a::at-base × 'b::pure) list
  shows fv Γ = domA Γ
  apply (induct Γ)
  apply simp
  apply (case-tac a)
  apply (simp)
  done

```

```

lemma domA-fresh-pure:
  fixes Γ :: ('a::at-base × 'b::pure) list
  shows x ∈ domA Γ ↔ ¬(atom x # Γ)
  unfolding domA-fv-pure[symmetric]
  by (auto simp add: fv-def fresh-def)

```

end

## 2.6 HOLCF-Utills

```

theory HOLCF-Utills
  imports HOLCF Pointwise

```

```

begin

default-sort type

lemmas cont-fun[simp]
lemmas cont2cont-fun[simp]

lemma cont-compose2:
  assumes  $\bigwedge y. cont (\lambda x. c x y)$ 
  assumes  $\bigwedge x. cont (\lambda y. c x y)$ 
  assumes cont f
  assumes cont g
  shows cont  $(\lambda x. c (f x) (g x))$ 
  by (intro cont-apply[OF assms(4) assms(2)]
      cont2cont-fun[OF cont-compose[OF - assms(3)]]
      cont2cont-lambda[OF assms(1)])

lemma pointwise-adm:
  fixes P :: 'a::pcpo  $\Rightarrow$  'b::pcpo  $\Rightarrow$  bool
  assumes adm  $(\lambda x. P (fst x) (snd x))$ 
  shows adm  $(\lambda m. pointwise P (fst m) (snd m))$ 
proof (rule admI, goal-cases)
  case prems: (1 Y)
  show ?case
    apply (rule pointwiseI)
    apply (rule admD[OF adm-subst[where t =  $\lambda p. (fst p x, snd p x)$  for x, OF - assms,
simplified]  $\langle chain Y \rangle$ ])
    using prems(2) unfolding pointwise-def apply auto
  done
qed

lemma cfun-beta-Pair:
  assumes cont  $(\lambda p. f (fst p) (snd p))$ 
  shows  $csplit(\Lambda a b . f a b).(x, y) = f x y$ 
  apply simp
  apply (subst beta-cfun)
  apply (rule cont2cont-LAM')
  apply (rule assms)
  apply (rule beta-cfun)
  apply (rule cont2cont-fun)
  using assms
  unfolding prod-cont-iff
  apply auto
  done

lemma fun-upd-mono:
   $\rho 1 \sqsubseteq \rho 2 \implies v 1 \sqsubseteq v 2 \implies \rho 1(x := v 1) \sqsubseteq \rho 2(x := v 2)$ 
  apply (rule fun-belowI)

```

```

apply (case-tac xa = x)
apply simp
apply (auto elim:fun-belowD)
done

```

```

lemma fun-upd-cont[simp,cont2cont]:
  assumes cont f and cont h
  shows cont ( $\lambda x. (f x)(v := h x) :: 'a \Rightarrow 'b::pcpo$ )
  by (rule cont2cont-lambda)(auto simp add: assms)

```

```

lemma fun-upd-belowI:
  assumes  $\bigwedge z. z \neq x \implies \varrho z \sqsubseteq \varrho' z$ 
  assumes  $y \sqsubseteq \varrho' x$ 
  shows  $\varrho(x := y) \sqsubseteq \varrho'$ 
  apply (rule fun-belowI)
  using assms
  apply (case-tac xa = x)
  apply auto
  done

```

```

lemma cont-if-else-above:
  assumes cont f
  assumes cont g
  assumes  $\bigwedge x. f x \sqsubseteq g x$ 
  assumes  $\bigwedge x y. x \sqsubseteq y \implies P y \implies P x$ 
  assumes adm P
  shows cont ( $\lambda x. \text{if } P x \text{ then } f x \text{ else } g x$ ) (is cont ?I)

```

```

proof(intro contI2 monofunI)
  fix x y :: 'a
  assume  $x \sqsubseteq y$ 
  with assms(4)[OF this]
  show ?I x  $\sqsubseteq$  ?I y
    apply (auto)
    apply (rule cont2monofunE[OF assms(1)], assumption)
    apply (rule below-trans[OF cont2monofunE[OF assms(1)] assms(3)], assumption)
    apply (rule cont2monofunE[OF assms(2)], assumption)
  done

```

```

next
  fix Y :: nat  $\Rightarrow$  'a
  assume chain Y
  assume chain ( $\lambda i. ?I (Y i)$ )

```

```

  have ch-f:  $f (\bigsqcup i. Y i) \sqsubseteq (\bigsqcup i. f (Y i))$  by (metis  $\langle \text{chain } Y \rangle$  assms(1) below-refl cont2contlubE)

```

```

  show ?I ( $\bigsqcup i. Y i$ )  $\sqsubseteq$  ( $\bigsqcup i. ?I (Y i)$ )
  proof(cases  $\forall i. P (Y i)$ )
    case True hence P ( $\bigsqcup i. Y i$ ) by (metis  $\langle \text{chain } Y \rangle$  adm-def assms(5))

```

```

with True ch-f show ?thesis by auto
next
case False
then obtain j where  $\neg P (Y j)$  by auto
hence *:  $\forall i \geq j. \neg P (Y i) \rightarrow \neg P (\bigsqcup i. Y i)$ 
  apply (auto)
  apply (metis assms(4) chain-mono[OF ‹chain Y›])
  apply (metis assms(4) is-ub-theLub[OF ‹chain Y›])
  done

have ?I ( $\bigsqcup i. Y i$ ) =  $g (\bigsqcup i. Y i)$  using * by simp
also have ... =  $g (\bigsqcup i. Y (i + j))$  by (metis lub-range-shift[OF ‹chain Y›])
also have ... = ( $\bigsqcup i. (g (Y (i + j)))$ ) by (rule cont2contlubE[OF assms(2) chain-shift[OF ‹chain Y›]])
  also have ... = ( $\bigsqcup i. (?I (Y (i + j)))$ ) using * by auto
  also have ... = ( $\bigsqcup i. (?I (Y i))$ ) by (metis lub-range-shift[OF ‹chain ( $\lambda i. ?I (Y i)$ )›])
  finally show ?thesis by simp
qed
qed

fun up2option :: 'a::cpo $\perp$   $\Rightarrow$  'a option
  where up2option Ibottom = None
  |   up2option (Iup a) = Some a

lemma up2option-simps[simp]:
  up2option  $\perp$  = None
  up2option (up·x) = Some x
  unfolding up-def by (simp-all add: cont-Iup inst-up-pcpo)

fun option2up :: 'a option  $\Rightarrow$  'a::cpo $\perp$ 
  where option2up None =  $\perp$ 
  |   option2up (Some a) = up·a

lemma option2up-up2option[simp]:
  option2up (up2option x) = x
  by (cases x) auto
lemma up2option-option2up[simp]:
  up2option (option2up x) = x
  by (cases x) auto

lemma adm-subst2: cont f  $\Longrightarrow$  cont g  $\Longrightarrow$  adm ( $\lambda x. f (fst x) = g (snd x)$ )
  apply (rule admI)
  apply (simp add:
    cont2contlubE[where f = f] cont2contlubE[where f = g]
    cont2contlubE[where f = snd] cont2contlubE[where f = fst]
  )
  done

```

### 2.6.1 Composition of fun and cfun

**lemma** *cont2cont-comp* [*simp*, *cont2cont*]:  
**assumes** *cont f*  
**assumes**  $\bigwedge x. \text{cont } (f\ x)$   
**assumes** *cont g*  
**shows** *cont*  $(\lambda x. (f\ x) \circ (g\ x))$   
**unfolding** *comp-def*  
**by** (*rule cont2cont-lambda*)  
*(intro cont2cont <cont g> <cont f> cont-compose2[OF cont2cont-fun[OF assms(1)] assms(2)] cont2cont-fun)*

**definition** *cfun-comp* ::  $('a::pcpo \rightarrow 'b::pcpo) \rightarrow ('c::type \Rightarrow 'a) \rightarrow ('c::type \Rightarrow 'b)$   
**where** *cfun-comp* =  $(\Lambda f\ \varrho. (\lambda x. f\cdot x) \circ \varrho)$

**lemma** [*simp*]: *cfun-comp.f*· $\varrho(x := v) = (cfun-comp.f\cdot\varrho)(x := f\cdot v)$   
**unfolding** *cfun-comp-def* **by** *auto*

**lemma** *cfun-comp-app*[*simp*]:  $(cfun-comp.f\cdot\varrho)\ x = f\cdot(\varrho\ x)$   
**unfolding** *cfun-comp-def* **by** *auto*

**lemma** *fix-eq-fix*:  
 $f\cdot(\text{fix}\cdot g) \sqsubseteq \text{fix}\cdot g \implies g\cdot(\text{fix}\cdot f) \sqsubseteq \text{fix}\cdot f \implies \text{fix}\cdot f = \text{fix}\cdot g$   
**by** (*metis fix-least-below below-antisym*)

### 2.6.2 Additional transitivity rules

These collect side-conditions of the form *cont f*, so the usual way to discharge them is to write *by this (intro cont2cont)+* at the end.

**lemma** *below-trans-cong*[*trans*]:  
 $a \sqsubseteq f\ x \implies x \sqsubseteq y \implies \text{cont } f \implies a \sqsubseteq f\ y$   
**by** (*metis below-trans cont2monofunE*)

**lemma** *not-bot-below-trans*[*trans*]:  
 $a \neq \perp \implies a \sqsubseteq b \implies b \neq \perp$   
**by** (*metis below-bottom-iff*)

**lemma** *not-bot-below-trans-cong*[*trans*]:  
 $f\ a \neq \perp \implies a \sqsubseteq b \implies \text{cont } f \implies f\ b \neq \perp$   
**by** (*metis below-bottom-iff cont2monofunE*)

**end**

## 2.7 HOLCF-Meet

**theory** *HOLCF-Meet*  
**imports** *HOLCF*  
**begin**

This theory defines the  $\sqcap$  operator on HOLCF domains, and introduces a type class for domains where all finite meets exist.

### 2.7.1 Towards meets: Lower bounds

**context** *po*

**begin**

**definition** *is-lb* :: 'a set  $\Rightarrow$  'a  $\Rightarrow$  bool (infix  $\langle \rangle$  55) **where**  
 $S \rangle | x \longleftrightarrow (\forall y \in S. x \sqsubseteq y)$

**lemma** *is-lbI*:  $(!!x. x \in S \implies l \sqsubseteq x) \implies S \rangle | l$   
**by** (*simp add: is-lb-def*)

**lemma** *is-lbD*:  $[|S \rangle | l; x \in S|] \implies l \sqsubseteq x$   
**by** (*simp add: is-lb-def*)

**lemma** *is-lb-empty* [*simp*]:  $\{\} \rangle | l$   
**unfolding** *is-lb-def* **by** *fast*

**lemma** *is-lb-insert* [*simp*]:  $(insert\ x\ A) \rangle | y = (y \sqsubseteq x \wedge A \rangle | y)$   
**unfolding** *is-lb-def* **by** *fast*

**lemma** *is-lb-downward*:  $[|S \rangle | l; y \sqsubseteq l|] \implies S \rangle | y$   
**unfolding** *is-lb-def* **by** (*fast intro: below-trans*)

### 2.7.2 Greatest lower bounds

**definition** *is-glb* :: 'a set  $\Rightarrow$  'a  $\Rightarrow$  bool (infix  $\langle \rangle \rangle$  55) **where**  
 $S \rangle \rangle | x \longleftrightarrow S \rangle | x \wedge (\forall u. S \rangle | u \longrightarrow u \sqsubseteq x)$

**definition** *glb* :: 'a set  $\Rightarrow$  'a ( $\langle \sqcap \rangle \rightarrow$  [60]60) **where**  
 $glb\ S = (THE\ x. S \rangle \rangle | x)$

Access to the definition as inference rule

**lemma** *is-glbD1*:  $S \rangle \rangle | x \implies S \rangle | x$   
**unfolding** *is-glb-def* **by** *fast*

**lemma** *is-glbD2*:  $[|S \rangle \rangle | x; S \rangle | u|] \implies u \sqsubseteq x$   
**unfolding** *is-glb-def* **by** *fast*

**lemma** (in *po*) *is-glbI*:  $[|S \rangle | x; !!u. S \rangle | u \implies u \sqsubseteq x|] \implies S \rangle \rangle | x$   
**unfolding** *is-glb-def* **by** *fast*

**lemma** *is-glb-above-iff*:  $S \rangle \rangle | x \implies u \sqsubseteq x \longleftrightarrow S \rangle | u$   
**unfolding** *is-glb-def is-lb-def* **by** (*metis below-trans*)

glbs are unique

**lemma** *is-glb-unique*:  $[|S \rangle \rangle | x; S \rangle \rangle | y|] \implies x = y$

**unfolding** *is-glb-def is-lb-def* **by** (*blast intro: below-antisym*)

technical lemmas about *glb* and ( $\gg|$ )

**lemma** *is-glb-glb*:  $M \gg| x \implies M \gg| \text{glb } M$   
**unfolding** *glb-def* **by** (*rule theI [OF - is-glb-unique]*)

**lemma** *glb-eqI*:  $M \gg| l \implies \text{glb } M = l$   
**by** (*rule is-glb-unique [OF is-glb-glb]*)

**lemma** *is-glb-singleton*:  $\{x\} \gg| x$   
**by** (*simp add: is-glb-def*)

**lemma** *glb-singleton [simp]*:  $\text{glb } \{x\} = x$   
**by** (*rule is-glb-singleton [THEN glb-eqI]*)

**lemma** *is-glb-bin*:  $x \sqsubseteq y \implies \{x, y\} \gg| x$   
**by** (*simp add: is-glb-def*)

**lemma** *glb-bin*:  $x \sqsubseteq y \implies \text{glb } \{x, y\} = x$   
**by** (*rule is-glb-bin [THEN glb-eqI]*)

**lemma** *is-glb-maximal*:  $[|S \gg| x; x \in S|] \implies S \gg| x$   
**by** (*erule is-glbI, erule (1) is-lbD*)

**lemma** *glb-maximal*:  $[|S \gg| x; x \in S|] \implies \text{glb } S = x$   
**by** (*rule is-glb-maximal [THEN glb-eqI]*)

**lemma** *glb-above*:  $S \gg| z \implies x \sqsubseteq \text{glb } S \iff S \gg| x$   
**by** (*metis glb-eqI is-glb-above-iff*)  
**end**

**lemma** (*in cpo*) *Meet-insert*:  $S \gg| l \implies \{x, l\} \gg| l2 \implies \text{insert } x S \gg| l2$   
**apply** (*rule is-glbI*)  
**apply** (*metis is-glb-above-iff is-glb-def is-lb-insert*)  
**by** (*metis is-glb-above-iff is-glb-def is-glb-singleton is-lb-insert*)

Binary, hence finite meets.

**class** *Finite-Meet-cpo* = *cpo* +  
**assumes** *binary-meet-exists*:  $\exists l. l \sqsubseteq x \wedge l \sqsubseteq y \wedge (\forall z. z \sqsubseteq x \longrightarrow z \sqsubseteq y \longrightarrow z \sqsubseteq l)$   
**begin**

**lemma** *binary-meet-exists'*:  $\exists l. \{x, y\} \gg| l$   
**using** *binary-meet-exists[of x y]*  
**unfolding** *is-glb-def is-lb-def*  
**by** *auto*

**lemma** *finite-meet-exists*:  
**assumes**  $S \neq \{\}$

```

    and finite S
    shows  $\exists x. S \gg | x$ 
    using  $\langle S \neq \{\} \rangle$ 
    apply (induct rule: finite-induct[OF  $\langle finite S \rangle$ ])
    apply (erule notE, rule refl)[1]
    apply (case-tac  $F = \{\}$ )
    apply (metis is-glb-singleton)
    apply (metis Meet-insert binary-meet-exists')
    done
end

```

**definition** *meet* ::  $'a::cpo \Rightarrow 'a \Rightarrow 'a$  (**infix**  $\langle \sqcap \rangle$  80) **where**  
 $x \sqcap y = (if \exists z. \{x, y\} \gg | z \text{ then } glb \{x, y\} \text{ else } x)$

**lemma** *meet-def'*:  $(x::'a::Finite-Meet-cpo) \sqcap y = glb \{x, y\}$   
**unfolding** *meet-def* **by** (*metis binary-meet-exists'*)

**lemma** *meet-comm*:  $(x::'a::Finite-Meet-cpo) \sqcap y = y \sqcap x$  **unfolding** *meet-def'* **by** (*metis insert-commute*)

**lemma** *meet-bot1*[*simp*]:  
**fixes**  $y :: 'a :: \{Finite-Meet-cpo, pcpo\}$   
**shows**  $(\perp \sqcap y) = \perp$  **unfolding** *meet-def'* **by** (*metis minimal po-class.glb-bin*)

**lemma** *meet-bot2*[*simp*]:  
**fixes**  $x :: 'a :: \{Finite-Meet-cpo, pcpo\}$   
**shows**  $(x \sqcap \perp) = \perp$  **by** (*metis meet-bot1 meet-comm*)

**lemma** *meet-below1*[*intro*]:  
**fixes**  $x y :: 'a :: Finite-Meet-cpo$   
**assumes**  $x \sqsubseteq z$   
**shows**  $(x \sqcap y) \sqsubseteq z$  **unfolding** *meet-def'* **by** (*metis assms binary-meet-exists' below-trans glb-eqI is-glbD1 is-lb-insert*)

**lemma** *meet-below2*[*intro*]:  
**fixes**  $x y :: 'a :: Finite-Meet-cpo$   
**assumes**  $y \sqsubseteq z$   
**shows**  $(x \sqcap y) \sqsubseteq z$  **unfolding** *meet-def'* **by** (*metis assms binary-meet-exists' below-trans glb-eqI is-glbD1 is-lb-insert*)

**lemma** *meet-above-iff*:  
**fixes**  $x y z :: 'a :: Finite-Meet-cpo$   
**shows**  $z \sqsubseteq x \sqcap y \longleftrightarrow z \sqsubseteq x \wedge z \sqsubseteq y$

**proof** –  
**obtain**  $g$  **where**  $\{x, y\} \gg | g$  **by** (*metis binary-meet-exists'*)  
**thus** *?thesis*  
**unfolding** *meet-def'* **by** (*simp add: glb-above*)  
**qed**

**lemma** *below-meet*[*simp*]:  
**fixes**  $x y :: 'a :: Finite-Meet-cpo$

**assumes**  $x \sqsubseteq z$   
**shows**  $(x \sqcap z) = x$  **by** (*metis assms glb-bin meet-def'*)

**lemma** *below-meet2[simp]*:  
**fixes**  $x y :: 'a :: \text{Finite-Meet-cpo}$   
**assumes**  $z \sqsubseteq x$   
**shows**  $(x \sqcap z) = z$  **by** (*metis assms below-meet meet-comm*)

**lemma** *meet-aboveI*:  
**fixes**  $x y z :: 'a :: \text{Finite-Meet-cpo}$   
**shows**  $z \sqsubseteq x \implies z \sqsubseteq y \implies z \sqsubseteq x \sqcap y$  **by** (*simp add: meet-above-iff*)

**lemma** *is-meetI*:  
**fixes**  $x y z :: 'a :: \text{Finite-Meet-cpo}$   
**assumes**  $z \sqsubseteq x$   
**assumes**  $z \sqsubseteq y$   
**assumes**  $\bigwedge a. [a \sqsubseteq x ; a \sqsubseteq y] \implies a \sqsubseteq z$   
**shows**  $x \sqcap y = z$   
**by** (*metis assms below-antisym meet-above-iff below-refl*)

**lemma** *meet-assoc[simp]*:  $((x :: 'a :: \text{Finite-Meet-cpo}) \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$   
**apply** (*rule is-meetI*)  
**apply** (*metis below-refl meet-above-iff*)  
**apply** (*metis below-refl meet-below2*)  
**apply** (*metis meet-above-iff*)  
**done**

**lemma** *meet-self[simp]*:  $r \sqcap r = (r :: 'a :: \text{Finite-Meet-cpo})$   
**by** (*metis below-refl is-meetI*)

**lemma** *[simp]*:  $(r :: 'a :: \text{Finite-Meet-cpo}) \sqcap (r \sqcap x) = r \sqcap x$   
**by** (*metis below-refl is-meetI meet-below1*)

**lemma** *meet-monofunI*:  
**fixes**  $y :: 'a :: \text{Finite-Meet-cpo}$   
**shows** *monofun*  $(\lambda x. (x \sqcap y))$   
**by** (*rule monofunI*)(*auto simp add: meet-above-iff*)

**lemma** *chain-meet1*:  
**fixes**  $y :: 'a :: \text{Finite-Meet-cpo}$   
**assumes** *chain*  $Y$   
**shows** *chain*  $(\lambda i. Y i \sqcap y)$   
**by** (*rule chainI*) (*auto simp add: meet-above-iff intro: chainI chainE[OF assms]*)

**class** *cont-binary-meet* = *Finite-Meet-cpo* +  
**assumes** *meet-cont'*:  $\text{chain } Y \implies (\bigsqcap i. Y i) \sqcap y = (\bigsqcap i. Y i \sqcap y)$

**lemma** *meet-cont1*:  
**fixes**  $y :: 'a :: \text{cont-binary-meet}$

**shows** *cont* ( $\lambda x. (x \sqcap y)$ )  
**by** (*rule contI2*[*OF meet-monofun1*]) (*simp add: meet-cont'*)

**lemma** *meet-cont2*:  
**fixes**  $x :: 'a :: \text{cont-binary-meet}$   
**shows** *cont* ( $\lambda y. (x \sqcap y)$ ) **by** (*subst meet-comm, rule meet-cont1*)

**lemma** *meet-cont*[*cont2cont, simp*]: *cont*  $f \implies \text{cont } g \implies \text{cont } (\lambda x. (f x \sqcap (g x :: 'a :: \text{cont-binary-meet})))$   
**apply** (*rule cont2cont-case-prod*[**where**  $g = \lambda x. (f x, g x)$  **and**  $f = \lambda p x y. x \sqcap y$ , *simplified*])  
**apply** (*rule meet-cont1*)  
**apply** (*rule meet-cont2*)  
**apply** (*metis cont2cont-Pair*)  
**done**

**end**

## 2.8 Nominal-HOLCF

**theory** *Nominal-HOLCF*  
**imports**  
*Nominal-Utils HOLCF-Utils*  
**begin**

### 2.8.1 Type class of continuous permutations and variations thereof

**class** *cont-pt* =  
*cpo* +  
*pt* +  
**assumes** *perm-cont*:  $\bigwedge p. \text{cont } ((\text{permute } p) :: 'a :: \{\text{cpo}, \text{pt}\} \Rightarrow 'a)$

**class** *discr-pt* =  
*discrete-cpo* +  
*pt*

**class** *pcpo-pt* =  
*cont-pt* +  
*pcpo*

**instance** *pcpo-pt*  $\subseteq$  *cont-pt*  
**by** *standard* (*auto intro: perm-cont*)

**instance** *discr-pt*  $\subseteq$  *cont-pt*  
**by** *standard auto*

**lemma** (**in** *cont-pt*) *perm-cont-simp*[*simp*]:  $\pi \cdot x \sqsubseteq \pi \cdot y \longleftrightarrow x \sqsubseteq y$   
**apply** *rule*  
**apply** (*drule cont2monofunE*[*OF perm-cont, of - - - $\pi$* ], *simp*)[1]  
**apply** (*erule cont2monofunE*[*OF perm-cont, of - -  $\pi$* ])  
**done**

**lemma** (in *cont-pt*) *perm-below-to-right*:  $\pi \cdot x \sqsubseteq y \longleftrightarrow x \sqsubseteq -\pi \cdot y$   
**by** (*metis perm-cont-simp pt-class.permute-minus-cancel(2)*)

**lemma** *perm-is-ub-simp*[*simp*]:  $\pi \cdot S <| \pi \cdot (x::'a::\text{cont-pt}) \longleftrightarrow S <| x$   
**by** (*auto simp add: is-ub-def permute-set-def*)

**lemma** *perm-is-ub-eqvt*[*simp,eqvt*]:  $S <| (x::'a::\text{cont-pt}) \implies \pi \cdot S <| \pi \cdot x$   
**by** *simp*

**lemma** *perm-is-lub-simp*[*simp*]:  $\pi \cdot S <<| \pi \cdot (x::'a::\text{cont-pt}) \longleftrightarrow S <<| x$   
**apply** (*rule perm-rel-lemma*)  
**by** (*metis is-lubI is-lubD1 is-lubD2 perm-cont-simp perm-is-ub-simp*)

**lemma** *perm-is-lub-eqvt*[*simp,eqvt*]:  $S <<| (x::'a::\text{cont-pt}) \implies \pi \cdot S <<| \pi \cdot x$   
**by** *simp*

**lemmas** *perm-cont2cont*[*simp,cont2cont*] = *cont-compose*[*OF perm-cont*]

**lemma** *perm-still-cont*:  $\text{cont}(\pi \cdot f) = \text{cont}(f :: ('a :: \text{cont-pt}) \Rightarrow ('b :: \text{cont-pt}))$   
**proof**  
**have** *imp*:  $\bigwedge (f :: 'a \Rightarrow 'b) \pi. \text{cont } f \implies \text{cont}(\pi \cdot f)$   
**unfolding** *permute-fun-def*  
**by** (*metis cont-compose perm-cont*)  
**show**  $\text{cont } f \implies \text{cont}(\pi \cdot f)$  **using** *imp*[*of f*  $\pi$ ].  
**show**  $\text{cont}(\pi \cdot f) \implies \text{cont}(f)$  **using** *imp*[*of*  $\pi \cdot f$   $-\pi$ ] **by** *simp*  
**qed**

**lemma** *perm-bottom*[*simp,eqvt*]:  $\pi \cdot \perp = (\perp::'a::\{\text{cont-pt},\text{pcpo}\})$   
**proof**–  
**have**  $\perp \sqsubseteq -\pi \cdot (\perp::'a::\{\text{cont-pt},\text{pcpo}\})$  **by** *simp*  
**hence**  $\pi \cdot \perp \sqsubseteq \pi \cdot (-\pi \cdot (\perp::'a::\{\text{cont-pt},\text{pcpo}\}))$  **by** (*rule cont2monofunE*[*OF perm-cont*])  
**hence**  $\pi \cdot \perp \sqsubseteq (\perp::'a::\{\text{cont-pt},\text{pcpo}\})$  **by** *simp*  
**thus**  $\pi \cdot \perp = (\perp::'a::\{\text{cont-pt},\text{pcpo}\})$  **by** *simp*  
**qed**

**lemma** *bot-supp*[*simp*]:  $\text{supp}(\perp :: 'a :: \text{pcpo-pt}) = \{\}$   
**by** (*rule supp-fun-eqvt*) (*simp add: eqvt-def*)

**lemma** *bot-fresh*[*simp*]:  $a \# (\perp :: 'a :: \text{pcpo-pt})$   
**by** (*simp add: fresh-def*)

**lemma** *bot-fresh-star*[*simp*]:  $a \#* (\perp :: 'a :: \text{pcpo-pt})$   
**by** (*simp add: fresh-star-def*)

**lemma** *below-eqvt* [*eqvt*]:  
 $\pi \cdot (x \sqsubseteq y) = (\pi \cdot x \sqsubseteq \pi \cdot (y::'a::\text{cont-pt}))$  **by** (*auto simp add: permute-pure*)

**lemma** *lub-eqvt*[*simp*]:

$(\exists z. S \ll z \mid (z :: 'a :: \{cont-pt\})) \implies \pi \cdot lub S = lub (\pi \cdot S)$   
**by** (*metis lub-eqI perm-is-lub-simp*)

**lemma** *chain-eqvt*[*eqvt*]:  
**fixes**  $F :: nat \Rightarrow 'a :: cont-pt$   
**shows**  $chain F \implies chain (\pi \cdot F)$   
**apply** (*rule chainI*)  
**apply** (*drule-tac i = i in chainE*)  
**apply** (*subst (asm) perm-cont-simp*[*symmetric*, **where**  $\pi = \pi$ ])  
**by** (*metis permute-fun-app-eq permute-pure*)

## 2.8.2 Instance for *cfun*

**instantiation** *cfun* :: (*cont-pt*, *cont-pt*) *pt*  
**begin**  
**definition**  $p \cdot (f :: 'a \rightarrow 'b) = (\Lambda x. p \cdot (f \cdot (- p \cdot x)))$   
  
**instance**  
**apply** *standard*  
**apply** (*simp add: permute-cfun-def eta-cfun*)  
**apply** (*simp add: permute-cfun-def cfun-eqI minus-add*)  
**done**  
**end**

**lemma** *permute-cfun-eq*:  $permute p = (\lambda f. (Abs-cfun (permute p)) \circ f \circ (Abs-cfun (permute (-p))))$   
**by** (*rule, rule cfun-eqI, auto simp add: permute-cfun-def*)

**lemma** *Cfun-app-eqvt*[*eqvt*]:  
 $\pi \cdot (f \cdot x) = (\pi \cdot f) \cdot (\pi \cdot x)$   
**unfolding** *permute-cfun-def*  
**by** *auto*

**lemma** *permute-Lam*:  $cont f \implies p \cdot (\Lambda x. f x) = (\Lambda x. (p \cdot f) x)$   
**apply** (*rule cfun-eqI*)  
**unfolding** *permute-cfun-def*  
**by** (*metis Abs-cfun-inverse2 eqvt-lambda unpermute-def*)

**lemma** *Abs-cfun-eqvt*:  $cont f \implies (p \cdot Abs-cfun) f = Abs-cfun f$   
**apply** (*subst permute-fun-def*)  
**by** (*metis permute-Lam perm-still-cont permute-minus-cancel(1)*)

**lemma** *cfun-eqvtI*:  $(\Lambda x. p \cdot (f \cdot x) = f' \cdot (p \cdot x)) \implies p \cdot f = f'$   
**by** (*metis Cfun-app-eqvt cfun-eqI permute-minus-cancel(1)*)

**lemma** *ID-eqvt*[*eqvt*]:  $\pi \cdot ID = ID$   
**unfolding** *ID-def*  
**apply** *perm-simp*  
**apply** (*simp add: Abs-cfun-eqvt*)

done

**instance** *cfun* :: (*cont-pt*, *cont-pt*) *cont-pt*  
by *standard* (*subst permute-cfun-eq*, *auto*)

**instance** *cfun* :: ({*pure,cont-pt*}, {*pure,cont-pt*}) *pure*  
by *standard* (*auto simp add: permute-cfun-def permute-pure Cfun.cfun.Rep-cfun-inverse*)

**instance** *cfun* :: (*cont-pt*, *pcpo-pt*) *pcpo-pt*  
by *standard*

### 2.8.3 Instance for *fun*

**lemma** *permute-fun-eq*:  $\text{permute } p = (\lambda f. (\text{permute } p) \circ f \circ (\text{permute } (-p)))$   
by (*rule*, *rule*, *metis comp-apply eqvt-lambda unpermute-def*)

**instance** *fun* :: (*pt*, *cont-pt*) *cont-pt*  
apply *standard*  
apply (*rule cont2cont-lambda*)  
apply (*subst permute-fun-def*)  
apply (*rule perm-cont2cont*)  
apply (*rule cont-fun*)  
done

**lemma** *fix-eqvt[eqvt]*:  
 $\pi \cdot \text{fix} = (\text{fix} :: ('a \rightarrow 'a) \rightarrow 'a :: \{\text{cont-pt}, \text{pcpo}\})$   
apply (*rule cfun-eqI*)  
apply (*subst permute-cfun-def*)  
apply *simp*  
apply (*rule parallel-fix-ind[OF adm-subst2]*)  
apply (*auto simp add: permute-self*)  
done

### 2.8.4 Instance for *u*

**instantiation** *u* :: (*cont-pt*) *pt*  
**begin**  
  **definition**  $p \cdot (x :: 'a \ u) = \text{fup} \cdot (\Lambda x. \text{up} \cdot (p \cdot x)) \cdot x$   
  **instance**  
  apply *standard*  
  apply (*case-tac x*) apply (*auto simp add: permute-u-def*)  
  apply (*case-tac x*) apply (*auto simp add: permute-u-def*)  
  **done**  
**end**

**instance** *u* :: (*cont-pt*) *cont-pt*  
**proof**  
  **fix** *p*  
  
  **have**  $\text{permute } p = (\lambda x. \text{fup} \cdot (\Lambda x. \text{up} \cdot (p \cdot x)) \cdot (x :: 'a \ u))$

```

    by (rule ext, rule permute-u-def)
  moreover have cont ( $\lambda x. fup(\Lambda x. up.(p \cdot x)).(x:: 'a u)$ ) by simp
  ultimately show cont (permute p :: 'a u  $\Rightarrow$  'a u) by simp
qed

```

```

instance u :: (cont-pt) pcpo-pt ..

```

```

class pure-cont-pt = pure + cont-pt

```

```

instance u :: (pure-cont-pt) pure
  apply standard
  apply (case-tac x)
  apply (auto simp add: permute-u-def permute-pure)
done

```

```

lemma up-eqvt[eqvt]:  $\pi \cdot up = up$ 
  apply (rule cfun-eqI)
  apply (subst permute-cfun-def, simp)
  apply (simp add: permute-u-def)
done

```

```

lemma fup-eqvt[eqvt]:  $\pi \cdot fup = fup$ 
  apply (rule cfun-eqI)
  apply (rule cfun-eqI)
  apply (subst permute-cfun-def, simp)
  apply (subst permute-cfun-def, simp)
  apply (case-tac xa)
  apply simp
  apply (simp add: permute-self)
done

```

### 2.8.5 Instance for lift

```

instantiation lift :: (pt) pt
begin
  definition p  $\cdot$  (x :: 'a lift) = case-lift  $\perp$  ( $\lambda x. Def (p \cdot x)$ ) x
  instance
  apply standard
  apply (case-tac x) apply (auto simp add: permute-lift-def)
  apply (case-tac x) apply (auto simp add: permute-lift-def)
  done
end

```

```

instance lift :: (pt) cont-pt

```

```

proof
  fix p
  have permute p = ( $\lambda x. case-lift \perp (\lambda x. Def (p \cdot x)) (x::'a lift)$ )

```

```

    by (rule ext, rule permute-lift-def)
  moreover have cont (λ x. case-lift ⊥ (λ x. Def (p · x)) (x::'a lift)) by simp
  ultimately show cont (permute p :: 'a lift ⇒ 'a lift) by simp
qed

```

```

instance lift :: (pt) pcpo-pt ..

```

```

instance lift :: (pure) pure
  apply standard
  apply (case-tac x)
  apply (auto simp add: permute-lift-def permute-pure)
done

```

```

lemma Def-eqvt[eqvt]: π · (Def x) = Def (π · x)
  by (simp add: permute-lift-def)

```

```

lemma case-lift-eqvt[eqvt]: π · case-lift d f x = case-lift (π · d) (π · f) (π · x)
  by (cases x) (auto simp add: permute-self)

```

### 2.8.6 Instance for prod

```

instance prod :: (cont-pt, cont-pt) cont-pt
proof
  fix p

```

```

    have permute p = (λ (x :: ('a, 'b) prod). (p · fst x, p · snd x)) by auto
    moreover have cont ... by (intro cont2cont)
    ultimately show cont (permute p :: ('a,'b) prod ⇒ ('a,'b) prod) by simp
qed

```

```

end

```

## 2.9 Env

```

theory Env
  imports Main HOLCF-Join-Classes
begin

```

```

default-sort type

```

Our type for environments is a function with a pcpo as the co-domain; this theory collects related definitions.

### 2.9.1 The domain of a pcpo-valued function

```

definition edom :: ('key ⇒ 'value::pcpo) ⇒ 'key set
  where edom m = {x. m x ≠ ⊥}

```

**lemma** *bot-edom*[simp]:  $\text{edom } \perp = \{\}$  **by** (*simp add: edom-def*)

**lemma** *bot-edom2*[simp]:  $\text{edom } (\lambda \cdot \perp) = \{\}$  **by** (*simp add: edom-def*)

**lemma** *edomIff*:  $(a \in \text{edom } m) = (m \ a \neq \perp)$  **by** (*simp add: edom-def*)

**lemma** *edom-iff2*:  $(m \ a = \perp) \longleftrightarrow (a \notin \text{edom } m)$  **by** (*simp add: edom-def*)

**lemma** *edom-empty-iff-bot*:  $\text{edom } m = \{\} \longleftrightarrow m = \perp$   
**by** (*metis below-bottom-iff bot-edom edomIff empty-iff fun-belowI*)

**lemma** *lookup-not-edom*:  $x \notin \text{edom } m \implies m \ x = \perp$  **by** (*auto iff:edomIff*)

**lemma** *lookup-edom*[simp]:  $m \ x \neq \perp \implies x \in \text{edom } m$  **by** (*auto iff:edomIff*)

**lemma** *edom-mono*:  $x \sqsubseteq y \implies \text{edom } x \subseteq \text{edom } y$   
**unfolding** *edom-def*  
**by** *auto* (*metis below-bottom-iff fun-belowD*)

**lemma** *edom-subset-adm*[simp]:  
*adm*  $(\lambda ae'. \text{edom } ae' \subseteq S)$   
**apply** (*rule admI*)  
**apply** *rule*  
**apply** (*subst (asm) edom-def*) **back**  
**apply** *simp*  
**apply** (*subst (asm) lub-fun*) **apply** *assumption*  
**apply** (*subst (asm) lub-eq-bottom-iff*)  
**apply** (*erule ch2ch-fun*)  
**unfolding** *not-all*  
**apply** (*erule exE*)  
**apply** (*rule subsetD*)  
**apply** (*rule allE*) **apply** *assumption* **apply** *assumption*  
**unfolding** *edom-def*  
**apply** *simp*  
**done**

## 2.9.2 Updates

**lemma** *edom-fun-upd-subset*:  $\text{edom } (h \ (x := v)) \subseteq \text{insert } x \ (\text{edom } h)$   
**by** (*auto simp add: edom-def*)

**declare** *fun-upd-same*[simp] *fun-upd-other*[simp]

## 2.9.3 Restriction

**definition** *env-restr* ::  $'a \text{ set} \Rightarrow ('a \Rightarrow 'b::\text{pcpo}) \Rightarrow ('a \Rightarrow 'b)$   
**where** *env-restr*  $S \ m = (\lambda x. \text{if } x \in S \text{ then } m \ x \text{ else } \perp)$

**abbreviation** *env-restr-rev* (**infixl**  $\langle f | \cdot \rangle$  110)

**where**  $env\text{-}restr\text{-}rev\ m\ S \equiv env\text{-}restr\ S\ m$

**notation** (*latex output*)  $env\text{-}restr\text{-}rev\ (\langle\cdot|\cdot\rangle)$

**lemma**  $env\text{-}restr\text{-}empty\text{-}iff[simp]$ :  $m\ f|\ 'S = \perp \iff edom\ m \cap S = \{\}$

**apply** (*auto simp add: edom-def env-restr-def lambda-strict[symmetric] split-if-splits*)

**apply** *metis*

**apply** (*fastforce simp add: edom-def env-restr-def lambda-strict[symmetric] split-if-splits*)

**done**

**lemmas**  $env\text{-}restr\text{-}empty = iffD2[OF\ env\text{-}restr\text{-}empty\text{-}iff,\ simp]$

**lemma**  $lookup\text{-}env\text{-}restr[simp]$ :  $x \in S \implies (m\ f|\ 'S)\ x = m\ x$

**by** (*fastforce simp add: env-restr-def*)

**lemma**  $lookup\text{-}env\text{-}restr\text{-}not\text{-}there[simp]$ :  $x \notin S \implies (env\text{-}restr\ S\ m)\ x = \perp$

**by** (*fastforce simp add: env-restr-def*)

**lemma**  $lookup\text{-}env\text{-}restr\text{-}eq$ :  $(m\ f|\ 'S)\ x = (if\ x \in S\ then\ m\ x\ else\ \perp)$

**by** *simp*

**lemma**  $env\text{-}restr\text{-}eqI$ :  $(\bigwedge x. x \in S \implies m_1\ x = m_2\ x) \implies m_1\ f|\ 'S = m_2\ f|\ 'S$

**by** (*auto simp add: lookup-env-restr-eq*)

**lemma**  $env\text{-}restr\text{-}eqD$ :  $m_1\ f|\ 'S = m_2\ f|\ 'S \implies x \in S \implies m_1\ x = m_2\ x$

**by** (*auto dest!: fun-cong[where x = x]*)

**lemma**  $env\text{-}restr\text{-}belowI$ :  $(\bigwedge x. x \in S \implies m_1\ x \sqsubseteq m_2\ x) \implies m_1\ f|\ 'S \sqsubseteq m_2\ f|\ 'S$

**by** (*auto intro: fun-belowI simp add: lookup-env-restr-eq*)

**lemma**  $env\text{-}restr\text{-}belowD$ :  $m_1\ f|\ 'S \sqsubseteq m_2\ f|\ 'S \implies x \in S \implies m_1\ x \sqsubseteq m_2\ x$

**by** (*auto dest!: fun-belowD[where x = x]*)

**lemma**  $env\text{-}restr\text{-}env\text{-}restr[simp]$ :

$x\ f|\ 'd2\ f|\ 'd1 = x\ f|\ '(d1 \cap d2)$

**by** (*fastforce simp add: env-restr-def*)

**lemma**  $env\text{-}restr\text{-}env\text{-}restr\text{-}subset$ :

$d1 \subseteq d2 \implies x\ f|\ 'd2\ f|\ 'd1 = x\ f|\ 'd1$

**by** (*metis Int-absorb2 env-restr-env-restr*)

**lemma**  $env\text{-}restr\text{-}useless$ :  $edom\ m \subseteq S \implies m\ f|\ 'S = m$

**by** (*rule ext*) (*auto simp add: lookup-env-restr-eq dest!: subsetD*)

**lemma**  $env\text{-}restr\text{-}UNIV[simp]$ :  $m\ f|\ 'UNIV = m$

**by** (*rule env-restr-useless*) *simp*

**lemma**  $env\text{-}restr\text{-}fun\text{-}upd[simp]$ :  $x \in S \implies m1(x := v)\ f|\ 'S = (m1\ f|\ 'S)(x := v)$

**apply** (*rule ext*)

**apply** (*case-tac xa = x*)

**apply** (*auto simp add: lookup-env-restr-eq*)  
**done**

**lemma** *env-restr-fun-upd-other[simp]*:  $x \notin S \implies m1(x := v) f|' S = m1 f|' S$   
**apply** (*rule ext*)  
**apply** (*case-tac xa = x*)  
**apply** (*auto simp add: lookup-env-restr-eq*)  
**done**

**lemma** *env-restr-eq-subset*:  
**assumes**  $S \subseteq S'$   
**and**  $m1 f|' S' = m2 f|' S'$   
**shows**  $m1 f|' S = m2 f|' S$   
**using** *assms*  
**by** (*metis env-restr-env-restr le-iff-inf*)

**lemma** *env-restr-below-subset*:  
**assumes**  $S \subseteq S'$   
**and**  $m1 f|' S' \sqsubseteq m2 f|' S'$   
**shows**  $m1 f|' S \sqsubseteq m2 f|' S$   
**using** *assms*  
**by** (*auto intro!: env-restr-belowI dest!: env-restr-belowD*)

**lemma** *edom-env[simp]*:  
 $edom (m f|' S) = edom m \cap S$   
**unfolding** *edom-def env-restr-def* **by** *auto*

**lemma** *env-restr-below-self*:  $f f|' S \sqsubseteq f$   
**by** (*rule fun-belowI*) (*auto simp add: env-restr-def*)

**lemma** *env-restr-below-trans*:  
 $m1 f|' S1 \sqsubseteq m2 f|' S1 \implies m2 f|' S2 \sqsubseteq m3 f|' S2 \implies m1 f|' (S1 \cap S2) \sqsubseteq m3 f|' (S1 \cap S2)$   
**by** (*auto intro!: env-restr-belowI dest!: env-restr-belowD elim: below-trans*)

**lemma** *env-restr-cont*: *cont (env-restr S)*  
**apply** (*rule cont2cont-lambda*)  
**unfolding** *env-restr-def*  
**apply** (*intro cont2cont cont-fun*)  
**done**

**lemma** *env-restr-mono*:  $m1 \sqsubseteq m2 \implies m1 f|' S \sqsubseteq m2 f|' S$   
**by** (*metis env-restr-belowI fun-belowD*)

**lemma** *env-restr-mono2*:  $S2 \subseteq S1 \implies m f|' S2 \sqsubseteq m f|' S1$   
**by** (*metis env-restr-below-self env-restr-env-restr-subset*)

**lemmas** *cont-compose[OF env-restr-cont, cont2cont, simp]*

**lemma** *env-restr-cong*:  $(\bigwedge x. \text{edom } m \subseteq S \cap S' \cup -S \cap -S') \implies m f|' S = m f|' S'$   
**by** (*rule ext*)(*auto simp add: lookup-env-restr-eq edom-def*)

#### 2.9.4 Deleting

**definition** *env-delete* ::  $'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b::\text{pcpo})$   
**where** *env-delete*  $x m = m(x := \perp)$

**lemma** *lookup-env-delete[simp]*:  
 $x' \neq x \implies \text{env-delete } x m x' = m x'$   
**by** (*simp add: env-delete-def*)

**lemma** *lookup-env-delete-None[simp]*:  
 $\text{env-delete } x m x = \perp$   
**by** (*simp add: env-delete-def*)

**lemma** *edom-env-delete[simp]*:  
 $\text{edom } (\text{env-delete } x m) = \text{edom } m - \{x\}$   
**by** (*auto simp add: env-delete-def edom-def*)

**lemma** *edom-env-delete-subset*:  
 $\text{edom } (\text{env-delete } x m) \subseteq \text{edom } m$  **by** *auto*

**lemma** *env-delete-fun-upd[simp]*:  
 $\text{env-delete } x (m(x := v)) = \text{env-delete } x m$   
**by** (*auto simp add: env-delete-def*)

**lemma** *env-delete-fun-upd2[simp]*:  
 $(\text{env-delete } x m)(x := v) = m(x := v)$   
**by** (*auto simp add: env-delete-def*)

**lemma** *env-delete-fun-upd3[simp]*:  
 $x \neq y \implies \text{env-delete } x (m(y := v)) = (\text{env-delete } x m)(y := v)$   
**by** (*auto simp add: env-delete-def*)

**lemma** *env-delete-noop[simp]*:  
 $x \notin \text{edom } m \implies \text{env-delete } x m = m$   
**by** (*auto simp add: env-delete-def edom-def*)

**lemma** *fun-upd-env-delete[simp]*:  $x \in \text{edom } \Gamma \implies (\text{env-delete } x \Gamma)(x := \Gamma x) = \Gamma$   
**by** (*auto*)

**lemma** *env-restr-env-delete-other[simp]*:  $x \notin S \implies \text{env-delete } x m f|' S = m f|' S$   
**apply** (*rule ext*)  
**apply** (*auto simp add: lookup-env-restr-eq*)  
**by** (*metis lookup-env-delete*)

**lemma** *env-delete-restr*:  $\text{env-delete } x m = m f|' (-\{x\})$   
**by** (*auto simp add: lookup-env-restr-eq*)

**lemma** *below-env-deleteI*:  $f x = \perp \implies f \sqsubseteq g \implies f \sqsubseteq \text{env-delete } x \ g$   
**by** (*metis env-delete-def env-delete-restr env-restr-mono fun-upd-triv*)

**lemma** *env-delete-below-cong*[*intro*]:  
**assumes**  $x \neq v \implies e1 \ x \sqsubseteq e2 \ x$   
**shows**  $\text{env-delete } v \ e1 \ x \sqsubseteq \text{env-delete } v \ e2 \ x$   
**using** *assms unfolding env-delete-def by auto*

**lemma** *env-delete-env-restr-swap*:  
 $\text{env-delete } x \ (\text{env-restr } S \ e) = \text{env-restr } S \ (\text{env-delete } x \ e)$   
**by** (*metis (erased, opaque-lifting) env-delete-def env-restr-fun-upd env-restr-fun-upd-other fun-upd-triv lookup-env-restr-eq*)

**lemma** *env-delete-mono*:  
 $m \sqsubseteq m' \implies \text{env-delete } x \ m \sqsubseteq \text{env-delete } x \ m'$   
**unfolding** *env-delete-restr*  
**by** (*rule env-restr-mono*)

**lemma** *env-delete-below-arg*:  
 $\text{env-delete } x \ m \sqsubseteq m$   
**unfolding** *env-delete-restr*  
**by** (*rule env-restr-below-self*)

## 2.9.5 Merging of two functions

We'd like to have some nice syntax for *override-on*.

**abbreviation** *override-on-syn*  $(\langle \cdot \ \cdot \ \cdot \ \rangle [100, 0, 100] \ 100)$  **where**  $f1 \ ++_S \ f2 \equiv \text{override-on } f1 \ f2 \ S$

**lemma** *override-on-bot*[*simp*]:  
 $\perp \ ++_S \ m = m \ f|' \ S$   
 $m \ ++_S \ \perp = m \ f|' \ (-S)$   
**by** (*auto simp add: override-on-def env-restr-def*)

**lemma** *edom-override-on*[*simp*]:  $\text{edom } (m1 \ ++_S \ m2) = (\text{edom } m1 \ - \ S) \cup (\text{edom } m2 \ \cap \ S)$   
**by** (*auto simp add: override-on-def edom-def*)

**lemma** *lookup-override-on-eq*:  $(m1 \ ++_S \ m2) \ x = (\text{if } x \in S \ \text{then } m2 \ x \ \text{else } m1 \ x)$   
**by** (*cases x \notin S simp-all*)

**lemma** *override-on-upd-swap*:  
 $x \notin S \implies \varrho(x := z) \ ++_S \ \varrho' = (\varrho \ ++_S \ \varrho')(x := z)$   
**by** (*auto simp add: override-on-def edom-def*)

**lemma** *override-on-upd*:  
 $x \in S \implies \varrho \ ++_S \ (\varrho'(x := z)) = (\varrho \ ++_S \ - \ \{x\} \ \varrho')(x := z)$   
**by** (*auto simp add: override-on-def edom-def*)

**lemma** *env-restr-add*:  $(m1 \text{ ++}_{S2} m2) f|' S = m1 f|' S \text{ ++}_{S2} m2 f|' S$   
**by** (*auto simp add: override-on-def edom-def env-restr-def*)

**lemma** *env-delete-add*:  $\text{env-delete } x (m1 \text{ ++}_S m2) = \text{env-delete } x m1 \text{ ++}_S - \{x\} \text{ env-delete } x m2$   
**by** (*auto simp add: override-on-def edom-def env-restr-def env-delete-def*)

### 2.9.6 Environments with binary joins

**lemma** *edom-join[simp]*:  $\text{edom } (f \sqcup (g :: ('a::\text{type} \Rightarrow 'b::\{\text{Finite-Join-cpo,pcpo}\}))) = \text{edom } f \cup \text{edom } g$   
**unfolding** *edom-def* **by** *auto*

**lemma** *env-delete-join[simp]*:  $\text{env-delete } x (f \sqcup (g :: ('a::\text{type} \Rightarrow 'b::\{\text{Finite-Join-cpo,pcpo}\}))) = \text{env-delete } x f \sqcup \text{env-delete } x g$   
**by** (*metis env-delete-def fun-upd-meet-simp*)

**lemma** *env-restr-join*:  
**fixes**  $m1 m2 :: 'a::\text{type} \Rightarrow 'b::\{\text{Finite-Join-cpo,pcpo}\}$   
**shows**  $(m1 \sqcup m2) f|' S = (m1 f|' S) \sqcup (m2 f|' S)$   
**by** (*auto simp add: env-restr-def*)

**lemma** *env-restr-join2*:  
**fixes**  $m :: 'a::\text{type} \Rightarrow 'b::\{\text{Finite-Join-cpo,pcpo}\}$   
**shows**  $m f|' S \sqcup m f|' S' = m f|' (S \cup S')$   
**by** (*auto simp add: env-restr-def*)

**lemma** *join-env-restr-UNIV*:  
**fixes**  $m :: 'a::\text{type} \Rightarrow 'b::\{\text{Finite-Join-cpo,pcpo}\}$   
**shows**  $S1 \cup S2 = \text{UNIV} \Longrightarrow (m f|' S1) \sqcup (m f|' S2) = m$   
**by** (*fastforce simp add: env-restr-def*)

**lemma** *env-restr-split*:  
**fixes**  $m :: 'a::\text{type} \Rightarrow 'b::\{\text{Finite-Join-cpo,pcpo}\}$   
**shows**  $m = m f|' S \sqcup m f|' (- S)$   
**by** (*simp add: env-restr-join2 Compl-partition*)

**lemma** *env-restr-below-split*:  
 $m f|' S \sqsubseteq m' \Longrightarrow m f|' (- S) \sqsubseteq m' \Longrightarrow m \sqsubseteq m'$   
**by** (*metis ComplI fun-below-iff lookup-env-restr*)

### 2.9.7 Singleton environments

**definition** *esing* ::  $'a \Rightarrow 'b::\{\text{pcpo}\} \rightarrow ('a \Rightarrow 'b)$   
**where**  $\text{esing } x = (\Lambda a. (\lambda y. (\text{if } x = y \text{ then } a \text{ else } \perp)))$

**lemma** *esing-bot[simp]*:  $\text{esing } x \cdot \perp = \perp$   
**by** (*rule ext*)(*simp add: esing-def*)

**lemma** *esing-simps*[*simp*]:  
 (*esing*  $x \cdot n$ )  $x = n$   
 $x' \neq x \implies (\text{esing } x \cdot n) x' = \perp$   
**by** (*simp-all add: esing-def*)

**lemma** *esing-eq-up-iff*[*simp*]: (*esing*  $x \cdot (\text{up} \cdot a)$ )  $y = \text{up} \cdot a' \iff (x = y \wedge a = a')$   
**by** (*auto simp add: fun-below-iff esing-def*)

**lemma** *esing-below-iff*[*simp*]: *esing*  $x \cdot a \sqsubseteq ae \iff a \sqsubseteq ae x$   
**by** (*auto simp add: fun-below-iff esing-def*)

**lemma** *edom-esing-subset*: *edom* (*esing*  $x \cdot n$ )  $\subseteq \{x\}$   
**unfolding** *edom-def esing-def* **by** *auto*

**lemma** *edom-esing-up*[*simp*]: *edom* (*esing*  $x \cdot (\text{up} \cdot n)$ ) =  $\{x\}$   
**unfolding** *edom-def esing-def* **by** *auto*

**lemma** *env-delete-esing*[*simp*]: *env-delete*  $x$  (*esing*  $x \cdot n$ ) =  $\perp$   
**unfolding** *env-delete-def esing-def*  
**by** *auto*

**lemma** *env-restr-esing*[*simp*]:  
 $x \in S \implies \text{esing } x \cdot v \upharpoonright S = \text{esing } x \cdot v$   
**by** (*auto intro: env-restr-useless dest: subsetD[OF edom-esing-subset]*)

**lemma** *env-restr-esing2*[*simp*]:  
 $x \notin S \implies \text{esing } x \cdot v \upharpoonright S = \perp$   
**by** (*auto dest: subsetD[OF edom-esing-subset]*)

**lemma** *esing-eq-iff*[*simp*]:  
 $\text{esing } x \cdot v = \text{esing } x \cdot v' \iff v = v'$   
**by** (*metis esing-simps(1)*)

**end**

## 2.10 Env-Nominal

**theory** *Env-Nominal*  
**imports** *Env Nominal-Utills Nominal-HOLCF*  
**begin**

### 2.10.1 Equivariance lemmas

**lemma** *edom-perm*:  
**fixes**  $f :: 'a::pt \Rightarrow 'b::\{pcpo-pt\}$   
**shows** *edom* ( $\pi \cdot f$ ) =  $\pi \cdot (\text{edom } f)$   
**by** (*simp add: edom-def*)

**lemmas** *edom-perm-rev*[*simp,eqvt*] = *edom-perm*[*symmetric*]

**lemma** *mem-edom-perm*[*simp*]:

**fixes**  $\varrho :: 'a::at-base \Rightarrow 'b::\{pcpo-pt\}$

**shows**  $xa \in \text{edom } (p \cdot \varrho) \iff - p \cdot xa \in \text{edom } \varrho$

**by** (*metis* (*mono-tags*) *edom-perm-rev mem-Collect-eq permute-set-eq*)

**lemma** *env-restr-eqvt*[*eqvt*]:

**fixes**  $m :: 'a::pt \Rightarrow 'b::\{cont-pt,pcpo\}$

**shows**  $\pi \cdot m f|' d = (\pi \cdot m) f|' (\pi \cdot d)$

**by** (*auto simp add: env-restr-def*)

**lemma** *env-delete-eqvt*[*eqvt*]:

**fixes**  $m :: 'a::pt \Rightarrow 'b::\{cont-pt,pcpo\}$

**shows**  $\pi \cdot \text{env-delete } x m = \text{env-delete } (\pi \cdot x) (\pi \cdot m)$

**by** (*auto simp add: env-delete-def*)

**lemma** *esing-eqvt*[*eqvt*]:  $\pi \cdot (\text{esing } x) = \text{esing } (\pi \cdot x)$

**unfolding** *esing-def*

**apply** *perm-simp*

**apply** (*simp add: Abs-cfun-eqvt*)

**done**

## 2.10.2 Permutation and restriction

**lemma** *env-restr-perm*:

**fixes**  $\varrho :: 'a::at-base \Rightarrow 'b::\{pcpo-pt,pure\}$

**assumes**  $\text{supp } p \#* S$  **and** [*simp*]: *finite S*

**shows**  $(p \cdot \varrho) f|' S = \varrho f|' S$

**using** *assms*

**apply**  $-$

**apply** (*rule ext*)

**apply** (*case-tac*  $x \in S$ )

**apply** (*simp*)

**apply** (*subst permute-fun-def*)

**apply** (*simp add: permute-pure*)

**apply** (*subst perm-supp-eq*)

**apply** (*auto simp add:perm-supp-eq supp-minus-perm fresh-star-def fresh-def supp-set-elem-finite*)

**done**

**lemma** *env-restr-perm'*:

**fixes**  $\varrho :: 'a::at-base \Rightarrow 'b::\{pcpo-pt,pure\}$

**assumes**  $\text{supp } p \#* S$  **and** [*simp*]: *finite S*

**shows**  $p \cdot (\varrho f|' S) = \varrho f|' S$

**by** (*simp add: perm-supp-eq*[*OF assms(1)*] *env-restr-perm*[*OF assms*])

**lemma** *env-restr-flip*:

**fixes**  $\varrho :: 'a::at-base \Rightarrow 'b::\{pcpo-pt,pure\}$

**assumes**  $x \notin S$  **and**  $x' \notin S$

**shows**  $((x' \leftrightarrow x) \cdot \varrho) f|' S = \varrho f|' S$   
**using** *assms*  
**apply** –  
**apply** *rule*  
**apply** (*auto simp add: permute-flip-at env-restr-def split-if-splits*)  
**by** (*metis eqvt-lambda flip-at-base-simps(3) minus-flip permute-pure unpermute-def*)

**lemma** *env-restr-flip'*:  
**fixes**  $\varrho :: 'a::at-base \Rightarrow 'b::\{pcpo-pt,pure\}$   
**assumes**  $x \notin S$  **and**  $x' \notin S$   
**shows**  $(x' \leftrightarrow x) \cdot (\varrho f|' S) = \varrho f|' S$   
**by** (*simp add: flip-set-both-not-in[OF assms] env-restr-flip[OF assms]*)

### 2.10.3 Pure codomains

**lemma** *edom-fv-pure*:  
**fixes**  $f :: ('a::at-base \Rightarrow 'b::\{pcpo,pure\})$   
**assumes** *finite* (*edom f*)  
**shows**  $fv f \subseteq edom f$   
**using** *assms*  
**proof** (*induction edom f arbitrary: f*)  
**case** *empty*  
**hence**  $f = \perp$  **unfolding** *edom-def* **by** *auto*  
**thus** *?case* **by** (*auto simp add: fv-def fresh-def supp-def*)  
**next**  
**case** (*insert x S*)  
**have**  $f = (env-delete x f)(x := f x)$  **by** *auto*  
**hence**  $fv f \subseteq fv (env-delete x f) \cup fv x \cup fv (f x)$   
**using** *eqvt-fresh-cong3*[**where**  $f = fun-upd$  **and**  $x = env-delete x f$  **and**  $y = x$  **and**  $z = f x$ ,  
*OF fun-upd-eqvt*]  
**apply** (*auto simp add: fv-def fresh-def*)  
**by** (*metis fresh-def pure-fresh*)  
**also**  
  
**from**  $\langle insert x S = edom f \rangle$  **and**  $\langle x \notin S \rangle$   
**have**  $S = edom (env-delete x f)$  **by** *auto*  
**hence**  $fv (env-delete x f) \subseteq edom (env-delete x f)$  **by** (*rule insert*)  
**also**  
**have**  $fv (f x) = \{\}$  **by** (*rule fv-pure*)  
**also**  
**from**  $\langle insert x S = edom f \rangle$  **have**  $x \in edom f$  **by** *auto*  
**hence**  $edom (env-delete x f) \cup fv x \cup \{\} \subseteq edom f$  **by** *auto*  
**finally**  
**show** *?case* **by** *this (intro Un-mono subset-refl)*  
**qed**

**end**

## 2.11 Env-HOLCF

```
theory Env-HOLCF
  imports Env HOLCF-Utils
begin
```

### 2.11.1 Continuity and pcpo-valued functions

```
lemma override-on-belowI:
  assumes  $\bigwedge a. a \in S \implies y a \sqsubseteq z a$ 
  and  $\bigwedge a. a \notin S \implies x a \sqsubseteq z a$ 
  shows  $x ++_S y \sqsubseteq z$ 
  using assms
  apply -
  apply (rule fun-belowI)
  apply (case-tac xa  $\in S$ )
  apply auto
  done
```

```
lemma override-on-cont1: cont ( $\lambda x. x ++_S m$ )
  by (rule cont2cont-lambda) (auto simp add: override-on-def)
```

```
lemma override-on-cont2: cont ( $\lambda x. m ++_S x$ )
  by (rule cont2cont-lambda) (auto simp add: override-on-def)
```

```
lemma override-on-cont2cont[simp, cont2cont]:
  assumes cont f
  assumes cont g
  shows cont ( $\lambda x. f x ++_S g x$ )
  by (rule cont-apply[OF assms(1) override-on-cont1 cont-compose[OF override-on-cont2 assms(2)]])
```

```
lemma override-on-mono:
  assumes  $x1 \sqsubseteq (x2 :: 'a::type \Rightarrow 'b::cpo)$ 
  assumes  $y1 \sqsubseteq y2$ 
  shows  $x1 ++_S y1 \sqsubseteq x2 ++_S y2$ 
  by (rule below-trans[OF cont2monofunE[OF override-on-cont1 assms(1)] cont2monofunE[OF override-on-cont2 assms(2)]])
```

```
lemma fun-upd-below-env-deleteI:
  assumes env-delete x  $\rho \sqsubseteq$  env-delete x  $\rho'$ 
  assumes  $y \sqsubseteq \rho' x$ 
  shows  $\rho(x := y) \sqsubseteq \rho'$ 
  using assms
  apply (auto intro!: fun-upd-belowI simp add: env-delete-def)
  by (metis fun-belowD fun-upd-other)
```

```
lemma fun-upd-belowI2:
  assumes  $\bigwedge z. z \neq x \implies \rho z \sqsubseteq \rho' z$ 
  assumes  $\rho x \sqsubseteq y$ 
  shows  $\rho \sqsubseteq \rho'(x := y)$ 
```

**apply** (*rule fun-belowI*)  
**using** *assms* **by** *auto*

**lemma** *env-restr-belowI*:

**assumes**  $\bigwedge x. x \in S \implies (m1\ f|' S)\ x \sqsubseteq (m2\ f|' S)\ x$   
**shows**  $m1\ f|' S \sqsubseteq m2\ f|' S$   
**apply** (*rule fun-belowI*)  
**by** (*metis assms below-bottom-iff lookup-env-restr-not-there*)

**lemma** *env-restr-belowI2*:

**assumes**  $\bigwedge x. x \in S \implies m1\ x \sqsubseteq m2\ x$   
**shows**  $m1\ f|' S \sqsubseteq m2$   
**by** (*rule fun-belowI*)  
(*simp add: assms env-restr-def*)

**lemma** *env-restr-below-itself*:

**shows**  $m\ f|' S \sqsubseteq m$   
**apply** (*rule fun-belowI*)  
**apply** (*case-tac x \in S*)  
**apply** *auto*  
**done**

**lemma** *env-restr-cont*: *cont (env-restr S)*

**apply** (*rule cont2cont-lambda*)  
**apply** (*case-tac y \in S*)  
**apply** *auto*  
**done**

**lemma** *env-restr-belowD*:

**assumes**  $m1\ f|' S \sqsubseteq m2\ f|' S$   
**assumes**  $x \in S$   
**shows**  $m1\ x \sqsubseteq m2\ x$   
**using** *fun-belowD[OF assms(1), where x = x] assms(2)* **by** *simp*

**lemma** *env-restr-eqD*:

**assumes**  $m1\ f|' S = m2\ f|' S$   
**assumes**  $x \in S$   
**shows**  $m1\ x = m2\ x$   
**by** (*metis assms(1) assms(2) lookup-env-restr*)

**lemma** *env-restr-below-subset*:

**assumes**  $S \subseteq S'$   
**and**  $m1\ f|' S' \sqsubseteq m2\ f|' S'$   
**shows**  $m1\ f|' S \sqsubseteq m2\ f|' S$   
**using** *assms*  
**by** (*auto intro!: env-restr-belowI dest: env-restr-belowD*)

```

lemma override-on-below-restrI:
  assumes  $x f|' (-S) \sqsubseteq z f|' (-S)$ 
  and  $y f|' S \sqsubseteq z f|' S$ 
  shows  $x ++_S y \sqsubseteq z$ 
using assms
by (auto intro: override-on-belowI dest:env-restr-belowD)

lemma fmap-below-add-restrI:
  assumes  $x f|' (-S) \sqsubseteq y f|' (-S)$ 
  and  $x f|' S \sqsubseteq z f|' S$ 
  shows  $x \sqsubseteq y ++_S z$ 
using assms
by (auto intro!: fun-belowI dest:env-restr-belowD simp add: lookup-override-on-eq)

lemmas env-restr-cont2cont[simp,cont2cont] = cont-compose[OF env-restr-cont]

lemma env-delete-cont: cont (env-delete  $x$ )
  apply (rule cont2cont-lambda)
  apply (case-tac  $y = x$ )
  apply auto
  done
lemmas env-delete-cont2cont[simp,cont2cont] = cont-compose[OF env-delete-cont]

end

```

## 2.12 EvalHeap

```

theory EvalHeap
  imports AList-Utils Env Nominal2.Nominal2 HOLCF-Utils
begin

```

### 2.12.1 Conversion from heaps to environments

```

fun
  evalHeap :: ('var × 'exp) list ⇒ ('exp ⇒ 'value::{pure,pcpo}) ⇒ 'var ⇒ 'value
where
  evalHeap [] = ⊥
| evalHeap (( $x, e$ )# $h$ ) eval = (evalHeap  $h$  eval) ( $x := \text{eval } e$ )

lemma cont2cont-evalHeap[simp, cont2cont]:
  ( $\bigwedge e . e \in \text{snd } ' \text{set } h \implies \text{cont } (\lambda \varrho . \text{eval } \varrho e) \implies \text{cont } (\lambda \varrho . \text{evalHeap } h (\text{eval } \varrho))$ )
  by(induct  $h$ , auto)

lemma evalHeap-eqvt[eqvt]:
   $\pi \cdot \text{evalHeap } h \text{ eval} = \text{evalHeap } (\pi \cdot h) (\pi \cdot \text{eval})$ 
  by (induct  $h$ ) (auto simp add:fun-upd-eqvt simp del: fun-upd-apply)

```

**lemma** *edom-evalHeap-subset:edom* (*evalHeap h eval*)  $\subseteq$  *domA h*  
**by** (*induct h eval rule:evalHeap.induct*) (*auto dest:subsetD[OF edom-fun-upd-subset] simp del: fun-upd-apply*)

**lemma** *evalHeap-cong[fundef-cong]*:  
 $\llbracket \text{heap1} = \text{heap2} ; (\bigwedge e. e \in \text{snd } \text{'set heap2} \implies \text{eval1 } e = \text{eval2 } e) \rrbracket$   
 $\implies \text{evalHeap heap1 eval1} = \text{evalHeap heap2 eval2}$   
**by** (*induct heap2 eval2 arbitrary:heap1 rule:evalHeap.induct, auto*)

**lemma** *lookupEvalHeap*:  
**assumes**  $v \in \text{domA } h$   
**shows** (*evalHeap h f*)  $v = f$  (*the (map-of h v)*)  
**using** *assms*  
**by** (*induct h f rule: evalHeap.induct*) *auto*

**lemma** *lookupEvalHeap'*:  
**assumes** *map-of*  $\Gamma v = \text{Some } e$   
**shows** (*evalHeap*  $\Gamma f$ )  $v = f e$   
**using** *assms*  
**by** (*induct*  $\Gamma f$  *rule: evalHeap.induct*) *auto*

**lemma** *lookupEvalHeap-other[simp]*:  
**assumes**  $v \notin \text{domA } \Gamma$   
**shows** (*evalHeap*  $\Gamma f$ )  $v = \perp$   
**using** *assms*  
**by** (*induct*  $\Gamma f$  *rule: evalHeap.induct*) *auto*

**lemma** *env-restr-evalHeap-noop*:  
 $\text{domA } h \subseteq S \implies \text{env-restr } S$  (*evalHeap h eval*) = *evalHeap h eval*  
**apply** (*rule ext*)  
**apply** (*case-tac x \in S*)  
**apply** (*auto simp add: lookupEvalHeap intro: lookupEvalHeap-other*)  
**done**

**lemma** *env-restr-evalHeap-same[simp]*:  
 $\text{env-restr } (\text{domA } h)$  (*evalHeap h eval*) = *evalHeap h eval*  
**by** (*simp add: env-restr-evalHeap-noop*)

**lemma** *evalHeap-cong'*:  
 $\llbracket (\bigwedge x. x \in \text{domA } \text{heap} \implies \text{eval1 } (\text{the } (\text{map-of } \text{heap } x)) = \text{eval2 } (\text{the } (\text{map-of } \text{heap } x))) \rrbracket$   
 $\implies \text{evalHeap heap eval1} = \text{evalHeap heap eval2}$   
**apply** (*rule ext*)  
**apply** (*case-tac x \in domA heap*)  
**apply** (*auto simp add: lookupEvalHeap*)  
**done**

**lemma** *lookupEvalHeapNotAppend[simp]*:  
**assumes**  $x \notin \text{domA } \Gamma$   
**shows** (*evalHeap* ( $\Gamma @ h$ ) *f*)  $x = \text{evalHeap } h f x$

**using** *assms* **by** (*induct*  $\Gamma$ , *auto*)

**lemma** *evalHeap-delete[simp]*: *evalHeap* (*delete*  $x$   $\Gamma$ ) *eval* = *env-delete*  $x$  (*evalHeap*  $\Gamma$  *eval*)  
**by** (*induct*  $\Gamma$ ) *auto*

**lemma** *evalHeap-mono*:

$x \notin \text{dom}A \Gamma \implies$   
*evalHeap*  $\Gamma$  *eval*  $\sqsubseteq$  *evalHeap* ( $(x, e) \# \Gamma$ ) *eval*  
**apply** *simp*  
**apply** (*rule fun-belowI*)  
**apply** (*case-tac*  $xa \in \text{dom}A \Gamma$ )  
**apply** (*case-tac*  $xa = x$ )  
**apply** *auto*  
**done**

### 2.12.2 Reordering lemmas

**lemma** *evalHeap-reorder*:

**assumes** *map-of*  $\Gamma = \text{map-of } \Delta$   
**shows** *evalHeap*  $\Gamma$   $h = \text{evalHeap } \Delta$   $h$

**proof** (*rule ext*)

**from** *assms*

**have**  $*$ :  $\text{dom}A \Gamma = \text{dom}A \Delta$  **by** (*metis dom-map-of-conv-domA*)

**fix**  $x$

**show** *evalHeap*  $\Gamma$   $h$   $x = \text{evalHeap } \Delta$   $h$   $x$

**using** *assms(1)*  $*$

**apply** (*cases*  $x \in \text{dom}A \Gamma$ )

**apply** (*auto simp add: lookupEvalHeap*)

**done**

**qed**

**lemma** *evalHeap-reorder-head*:

**assumes**  $x \neq y$

**shows** *evalHeap* ( $(x, e1) \# (y, e2) \# \Gamma$ ) *eval* = *evalHeap* ( $(y, e2) \# (x, e1) \# \Gamma$ ) *eval*

**by** (*rule evalHeap-reorder*) (*simp add: fun-upd-twist[OF assms]*)

**lemma** *evalHeap-reorder-head-append*:

**assumes**  $x \notin \text{dom}A \Gamma$

**shows** *evalHeap* ( $(x, e) \# \Gamma @ \Delta$ ) *eval* = *evalHeap* ( $\Gamma @ ((x, e) \# \Delta)$ ) *eval*

**by** (*rule evalHeap-reorder*) (*simp, metis assms dom-map-of-conv-domA map-add-upd-left*)

**lemma** *evalHeap-subst-exp*:

**assumes** *eval*  $e = \text{eval } e'$

**shows** *evalHeap* ( $(x, e) \# \Gamma$ ) *eval* = *evalHeap* ( $(x, e') \# \Gamma$ ) *eval*

**by** (*simp add: assms*)

**end**

## 3 Launchbury’s natural semantics

### 3.1 Vars

```
theory Vars
imports Nominal2.Nominal2
begin
```

The type of variables is abstract and provided by the `Nominal` package. All we know is that it is countable.

```
atom-decl var

end
```

### 3.2 Terms

```
theory Terms
imports Nominal-Utils Vars AList-Utils-Nominal
begin
```

#### 3.2.1 Expressions

This is the main data type of the development; our minimal lambda calculus with recursive let-bindings. It is created using the `nominal_datatype` command, which creates alpha-equivalence classes.

The package does not support nested recursion, so the bindings of the let cannot simply be of type  $(var, exp)$  list. Instead, the definition of lists have to be inlined here, as the custom type `assn`. Later we create conversion functions between these two types, define a properly typed `let` and redo the various lemmas in terms of that, so that afterwards, the type `assn` is no longer referenced.

```
nominal-datatype exp =
  Var var
| App exp var
| LetA as::assn body::exp binds bn as in body as
| Lam x::var body::exp binds x in body (⟨Lam [-]. -⟩ [100, 100] 100)
| Bool bool
| IfThenElse exp exp exp (⟨((-) / ? (-) / : (-)⟩ [0, 0, 10] 10)
and assn =
  ANil | ACons var exp assn
binder
  bn :: assn ⇒ atom list
where bn ANil = [] | bn (ACons x t as) = (atom x) # (bn as)

notation (latex output) Terms.Var (⟨-⟩)
notation (latex output) Terms.App (⟨- -⟩)
```

**notation** (*latex output*) *Terms.Lam* ( $\langle \lambda \cdot \rightarrow [100, 100] 100$ )

**type-synonym** *heap* = (*var* × *exp*) *list*

**lemma** *exp-assn-size-eqvt*[*eqvt*]:  $p \cdot (\text{size} :: \text{exp} \Rightarrow \text{nat}) = \text{size}$   
by (*metis exp-assn.size-eqvt(1) fun-eqvtI permute-pure*)

### 3.2.2 Rewriting in terms of heaps

We now work towards using *heap* instead of *assn*. All this could be skipped if Nominal supported nested recursion.

Conversion from *assn* to *heap*.

**nominal-function** *asToHeap* :: *assn*  $\Rightarrow$  *heap*  
where *ANilToHeap*: *asToHeap* *ANil* = []  
| *AConsToHeap*: *asToHeap* (*ACons* *v e as*) = (*v, e*) # *asToHeap as*  
**unfolding** *eqvt-def asToHeap-graph-aux-def*  
**apply** *rule*  
**apply** *simp*  
**apply** *rule*  
**apply**(*case-tac x rule: exp-assn.exhaust(2)*)  
**apply** *auto*  
**done**  
**nominal-termination**(*eqvt*) by *lexicographic-order*

**lemma** *asToHeap-eqvt*: *eqvt asToHeap*  
**unfolding** *eqvt-def*  
by (*auto simp add: permute-fun-def asToHeap.eqvt*)

The other direction.

**fun** *heapToAssn* :: *heap*  $\Rightarrow$  *assn*  
where *heapToAssn* [] = *ANil*  
| *heapToAssn* ((*v,e*)# $\Gamma$ ) = *ACons v e (heapToAssn  $\Gamma$ )*

**declare** *heapToAssn.simps*[*simp del*]

**lemma** *heapToAssn-eqvt*[*simp,eqvt*]:  $p \cdot \text{heapToAssn } \Gamma = \text{heapToAssn } (p \cdot \Gamma)$   
by (*induct  $\Gamma$  rule: heapToAssn.induct*)  
(*auto simp add: heapToAssn.simps*)

**lemma** *bn-heapToAssn*:  $\text{bn } (\text{heapToAssn } \Gamma) = \text{map } (\lambda x. \text{atom } (\text{fst } x)) \Gamma$   
by (*induct rule: heapToAssn.induct*)  
(*auto simp add: heapToAssn.simps exp-assn.bn-defs*)

**lemma** *set-bn-to-atom-domA*:  
 $\text{set } (\text{bn } \text{as}) = \text{atom } \text{'domA } (\text{asToHeap } \text{as})$   
by (*induct as rule: asToHeap.induct*)  
(*auto simp add: exp-assn.bn-defs*)

They are inverse to each other.

**lemma** *heapToAssn-asToHeap[simp]*:  
 $heapToAssn (asToHeap as) = as$   
**by** (*induct rule: exp-assn.inducts(2)[of  $\lambda \cdot$ . True]*)  
*(auto simp add: heapToAssn.simps)*

**lemma** *asToHeap-heapToAssn[simp]*:  
 $asToHeap (heapToAssn as) = as$   
**by** (*induct rule: heapToAssn.induct*)  
*(auto simp add: heapToAssn.simps)*

**lemma** *heapToAssn-inject[simp]*:  
 $heapToAssn x = heapToAssn y \longleftrightarrow x = y$   
**by** (*metis asToHeap-heapToAssn*)

They are transparent to various notions from the Nominal package.

**lemma** *supp-heapToAssn*:  $supp (heapToAssn \Gamma) = supp \Gamma$   
**by** (*induct rule: heapToAssn.induct*)  
*(simp-all add: heapToAssn.simps exp-assn.supp supp-Nil supp-Cons supp-Pair sup-assoc)*

**lemma** *supp-asToHeap*:  $supp (asToHeap as) = supp as$   
**by** (*induct as rule: asToHeap.induct*)  
*(simp-all add: exp-assn.supp supp-Nil supp-Cons supp-Pair sup-assoc)*

**lemma** *fv-asToHeap*:  $fv (asToHeap \Gamma) = fv \Gamma$   
**unfolding** *fv-def* **by** (*auto simp add: supp-asToHeap*)

**lemma** *fv-heapToAssn*:  $fv (heapToAssn \Gamma) = fv \Gamma$   
**unfolding** *fv-def* **by** (*auto simp add: supp-heapToAssn*)

**lemma** [*simp*]:  $size (heapToAssn \Gamma) = size-list (\lambda (v,e) . size e) \Gamma$   
**by** (*induct rule: heapToAssn.induct*)  
*(simp-all add: heapToAssn.simps)*

**lemma** *Lam-eq-same-var[simp]*:  $Lam [y]. e = Lam [y]. e' \longleftrightarrow e = e'$   
**by** *auto (metis fresh-PairD(2) obtain-fresh)*

Now we define the Let constructor in the form that we actually want.

**hide-const** *HOL.Let*

**definition** *Let* ::  $heap \Rightarrow exp \Rightarrow exp$   
**where** *Let*  $\Gamma e = LetA (heapToAssn \Gamma) e$

**notation** (*latex output*) *Let* ( $\langle let - in \rangle$ )

**abbreviation**

*LetBe* ::  $var \Rightarrow exp \Rightarrow exp \Rightarrow exp$  ( $\langle let - be - in \rangle [100,100,100] 100$ )

**where**

$let\ x\ be\ t1\ in\ t2 \equiv Let [(x,t1)] t2$

We rewrite all (relevant) lemmas about *LetA* in terms of *Let*.

**lemma** *size-Let[simp]*:  $size (Let \Gamma e) = size-list (\lambda p. size (snd p)) \Gamma + size e + Suc 0$   
**unfolding** *Let-def* **by** (*auto simp add: split-beta'*)

**lemma** *Let-distinct[simp]*:

*Var*  $v \neq Let \Gamma e$   
*Let*  $\Gamma e \neq Var v$   
*App*  $e v \neq Let \Gamma e'$   
*Lam*  $[v]. e' \neq Let \Gamma e$   
*Let*  $\Gamma e \neq Lam [v]. e'$   
*Let*  $\Gamma e' \neq App e v$   
*Bool*  $b \neq Let \Gamma e$   
*Let*  $\Gamma e \neq Bool b$   
*(scrut ? e1 : e2)*  $\neq Let \Gamma e$   
*Let*  $\Gamma e \neq (scrut ? e1 : e2)$   
**unfolding** *Let-def* **by** *simp-all*

**lemma** *Let-perm-simps[simp,eqvt]*:

$p \cdot Let \Gamma e = Let (p \cdot \Gamma) (p \cdot e)$   
**unfolding** *Let-def* **by** *simp*

**lemma** *Let-supp*:

$supp (Let \Gamma e) = (supp e \cup supp \Gamma) - atom ' (domA \Gamma)$   
**unfolding** *Let-def* **by** (*simp add: exp-assn.supp supp-heapToAssn bn-heapToAssn domA-def image-image*)

**lemma** *Let-fresh[simp]*:

$a \# Let \Gamma e = (a \# e \wedge a \# \Gamma \vee a \in atom ' (domA \Gamma))$   
**unfolding** *fresh-def* **by** (*auto simp add: Let-supp*)

**lemma** *Abs-eq-cong*:

**assumes**  $\bigwedge p. (p \cdot x = x') \longleftrightarrow (p \cdot y = y')$   
**assumes**  $supp y = supp x$   
**assumes**  $supp y' = supp x'$   
**shows**  $([a]lst. x = [a']lst. x') \longleftrightarrow ([a]lst. y = [a']lst. y')$   
**by** (*simp add: Abs-eq-iff alpha-lst assms*)

**lemma** *Let-eq-iff[simp]*:

$(Let \Gamma e = Let \Gamma' e') = ([map (\lambda x. atom (fst x)) \Gamma ]lst. (e, \Gamma) = [map (\lambda x. atom (fst x)) \Gamma']lst. (e', \Gamma'))$

**unfolding** *Let-def*  
**apply** (*simp add: bn-heapToAssn*)  
**apply** (*rule Abs-eq-cong*)  
**apply** (*simp-all add: supp-Pair supp-heapToAssn*)  
**done**

**lemma** *exp-strong-exhaust*:

**fixes**  $c :: 'a :: fs$   
**assumes**  $\bigwedge var. y = Var var \implies P$

**assumes**  $\bigwedge \text{exp var. } y = \text{App exp var} \implies P$   
**assumes**  $\bigwedge \Gamma \text{ exp. atom } \text{' domA } \Gamma \#* c \implies y = \text{Let } \Gamma \text{ exp} \implies P$   
**assumes**  $\bigwedge \text{var exp. } \{\text{atom var}\} \#* c \implies y = \text{Lam [var]. exp} \implies P$   
**assumes**  $\bigwedge b. (y = \text{Bool } b) \implies P$   
**assumes**  $\bigwedge \text{scrut } e1 \text{ } e2. y = (\text{scrut } ? e1 : e2) \implies P$   
**shows**  $P$   
**apply** (*rule exp-assn.strong-exhaust(1)[where y = y and c = c]*)  
**apply** (*metis assms(1)*)  
**apply** (*metis assms(2)*)  
**apply** (*metis assms(3) set-bn-to-atom-domA Let-def heapToAssn-asToHeap*)  
**apply** (*metis assms(4)*)  
**apply** (*metis assms(5)*)  
**apply** (*metis assms(6)*)  
**done**

And finally the induction rules with *Let*.

**lemma** *exp-heap-induct[case-names Var App Let Lam Bool IfThenElse Nil Cons]:*

**assumes**  $\bigwedge b \text{ var. } P1 \text{ (Var var)}$   
**assumes**  $\bigwedge \text{exp var. } P1 \text{ exp} \implies P1 \text{ (App exp var)}$   
**assumes**  $\bigwedge \Gamma \text{ exp. } P2 \text{ } \Gamma \implies P1 \text{ exp} \implies P1 \text{ (Let } \Gamma \text{ exp)}$   
**assumes**  $\bigwedge \text{var exp. } P1 \text{ exp} \implies P1 \text{ (Lam [var]. exp)}$   
**assumes**  $\bigwedge b. P1 \text{ (Bool } b)$   
**assumes**  $\bigwedge \text{scrut } e1 \text{ } e2. P1 \text{ scrut} \implies P1 \text{ } e1 \implies P1 \text{ } e2 \implies P1 \text{ (scrut } ? e1 : e2)$   
**assumes**  $P2 \text{ []}$   
**assumes**  $\bigwedge \text{var exp } \Gamma. P1 \text{ exp} \implies P2 \text{ } \Gamma \implies P2 \text{ ((var, exp)\#}\Gamma)$   
**shows**  $P1 \text{ } e \text{ and } P2 \text{ } \Gamma$

**proof**–

**have**  $P1 \text{ } e \text{ and } P2 \text{ (asToHeap (heapToAssn } \Gamma))$   
**apply** (*induct rule: exp-assn.inducts[of P1 λ assn. P2 (asToHeap assn)]*)  
**apply** (*metis assms(1)*)  
**apply** (*metis assms(2)*)  
**apply** (*metis assms(3) Let-def heapToAssn-asToHeap*)  
**apply** (*metis assms(4)*)  
**apply** (*metis assms(5)*)  
**apply** (*metis assms(6)*)  
**apply** (*metis assms(7) asToHeap.simps(1)*)  
**apply** (*metis assms(8) asToHeap.simps(2)*)  
**done**

**thus**  $P1 \text{ } e \text{ and } P2 \text{ } \Gamma$  **unfolding** *asToHeap-heapToAssn*.

**qed**

**lemma** *exp-heap-strong-induct[case-names Var App Let Lam Bool IfThenElse Nil Cons]:*

**assumes**  $\bigwedge \text{var } c. P1 \text{ } c \text{ (Var var)}$   
**assumes**  $\bigwedge \text{exp var } c. (\bigwedge c. P1 \text{ } c \text{ exp}) \implies P1 \text{ } c \text{ (App exp var)}$   
**assumes**  $\bigwedge \Gamma \text{ exp } c. \text{atom } \text{' domA } \Gamma \#* c \implies (\bigwedge c. P2 \text{ } c \text{ } \Gamma) \implies (\bigwedge c. P1 \text{ } c \text{ exp}) \implies P1 \text{ } c \text{ (Let } \Gamma \text{ exp)}$   
**assumes**  $\bigwedge \text{var exp } c. \{\text{atom var}\} \#* c \implies (\bigwedge c. P1 \text{ } c \text{ exp}) \implies P1 \text{ } c \text{ (Lam [var]. exp)}$   
**assumes**  $\bigwedge b \text{ } c. P1 \text{ } c \text{ (Bool } b)$   
**assumes**  $\bigwedge \text{scrut } e1 \text{ } e2 \text{ } c. (\bigwedge c. P1 \text{ } c \text{ scrut}) \implies (\bigwedge c. P1 \text{ } c \text{ } e1) \implies (\bigwedge c. P1 \text{ } c \text{ } e2) \implies P1$

$c$  (*scrut* ?  $e1 : e2$ )  
**assumes**  $\bigwedge c. P2\ c \ []$   
**assumes**  $\bigwedge var\ exp\ \Gamma\ c. (\bigwedge c. P1\ c\ exp) \implies (\bigwedge c. P2\ c\ \Gamma) \implies P2\ c\ ((var,exp)\#\Gamma)$   
**fixes**  $c :: 'a :: fs$   
**shows**  $P1\ c\ e$  and  $P2\ c\ \Gamma$

**proof**–

**have**  $P1\ c\ e$  and  $P2\ c$  (*asToHeap* (*heapToAssn*  $\Gamma$ ))  
**apply** (*induct rule: exp-assn.strong-induct*[of  $P1\ \lambda\ c\ assn. P2\ c$  (*asToHeap*  $assn$ )])  
**apply** (*metis assms*(1))  
**apply** (*metis assms*(2))  
**apply** (*metis assms*(3) *set-bn-to-atom-domA Let-def heapToAssn-asToHeap* )  
**apply** (*metis assms*(4))  
**apply** (*metis assms*(5))  
**apply** (*metis assms*(6))  
**apply** (*metis assms*(7) *asToHeap.simps*(1))  
**apply** (*metis assms*(8) *asToHeap.simps*(2))  
**done**

**thus**  $P1\ c\ e$  and  $P2\ c\ \Gamma$  **unfolding** *asToHeap-heapToAssn*.

**qed**

### 3.2.3 Nice induction rules

These rules can be used instead of the original induction rules, which require a separate goal for *assn*.

**lemma** *exp-induct*[*case-names Var App Let Lam Bool IfThenElse*]:

**assumes**  $\bigwedge var. P\ (Var\ var)$   
**assumes**  $\bigwedge exp\ var. P\ exp \implies P\ (App\ exp\ var)$   
**assumes**  $\bigwedge \Gamma\ exp. (\bigwedge x. x \in domA\ \Gamma \implies P\ (the\ (map-of\ \Gamma\ x))) \implies P\ exp \implies P\ (Let\ \Gamma\ exp)$   
**assumes**  $\bigwedge var\ exp. P\ exp \implies P\ (Lam\ [var].\ exp)$   
**assumes**  $\bigwedge b. P\ (Bool\ b)$   
**assumes**  $\bigwedge scrut\ e1\ e2. P\ scrut \implies P\ e1 \implies P\ e2 \implies P\ (scrut\ ?\ e1 : e2)$   
**shows**  $P\ exp$   
**apply** (*rule exp-heap-induct*[of  $P\ \lambda\ \Gamma. (\forall x \in domA\ \Gamma. P\ (the\ (map-of\ \Gamma\ x)))$ ])  
**apply** (*metis assms*(1))  
**apply** (*metis assms*(2))  
**apply** (*metis assms*(3))  
**apply** (*metis assms*(4))  
**apply** (*metis assms*(5))  
**apply** (*metis assms*(6))  
**apply** *auto*  
**done**

**lemma** *exp-strong-induct-set*[*case-names Var App Let Lam Bool IfThenElse*]:

**assumes**  $\bigwedge var\ c. P\ c\ (Var\ var)$   
**assumes**  $\bigwedge exp\ var\ c. (\bigwedge c. P\ c\ exp) \implies P\ c\ (App\ exp\ var)$   
**assumes**  $\bigwedge \Gamma\ exp\ c.$   
*atom*  $'\ domA\ \Gamma\ \#\ast\ c \implies (\bigwedge c\ x\ e. (x,e) \in set\ \Gamma \implies P\ c\ e) \implies (\bigwedge c. P\ c\ exp) \implies P\ c\ (Let$

$\Gamma \text{ exp}$   
**assumes**  $\bigwedge \text{var exp } c. \{atom \text{ var}\} \#* c \implies (\bigwedge c. P c \text{ exp}) \implies P c (\text{Lam } [var]. \text{exp})$   
**assumes**  $\bigwedge b c. P c (\text{Bool } b)$   
**assumes**  $\bigwedge \text{scrut } e1 \text{ } e2 \text{ } c. (\bigwedge c. P c \text{ scrut}) \implies (\bigwedge c. P c \text{ } e1) \implies (\bigwedge c. P c \text{ } e2) \implies P c (\text{scrut } ? e1 : e2)$   
**shows**  $P (c::'a::fs) \text{ exp}$   
**apply**  $(rule \text{ exp-heap-strong-induct}(1)[of P \lambda c \Gamma. (\forall (x,e) \in set \Gamma. P c e)])$   
**apply**  $(metis \text{ assms}(1))$   
**apply**  $(metis \text{ assms}(2))$   
**apply**  $(metis \text{ assms}(3) \text{ split-conv})$   
**apply**  $(metis \text{ assms}(4))$   
**apply**  $(metis \text{ assms}(5))$   
**apply**  $(metis \text{ assms}(6))$   
**apply**  $auto$   
**done**

**lemma**  $\text{exp-strong-induct}[case-names \text{ Var App Let Lam Bool IfThenElse}]$ :  
**assumes**  $\bigwedge \text{var } c. P c (\text{Var } var)$   
**assumes**  $\bigwedge \text{exp var } c. (\bigwedge c. P c \text{ exp}) \implies P c (\text{App } \text{exp } var)$   
**assumes**  $\bigwedge \Gamma \text{ exp } c.$   
 $atom \text{ ' domA } \Gamma \#* c \implies (\bigwedge c \text{ } x. x \in domA \Gamma \implies P c (\text{the } (map-of \Gamma x))) \implies (\bigwedge c. P c \text{ exp}) \implies P c (\text{Let } \Gamma \text{ exp})$   
**assumes**  $\bigwedge \text{var exp } c. \{atom \text{ var}\} \#* c \implies (\bigwedge c. P c \text{ exp}) \implies P c (\text{Lam } [var]. \text{exp})$   
**assumes**  $\bigwedge b c. P c (\text{Bool } b)$   
**assumes**  $\bigwedge \text{scrut } e1 \text{ } e2 \text{ } c. (\bigwedge c. P c \text{ scrut}) \implies (\bigwedge c. P c \text{ } e1) \implies (\bigwedge c. P c \text{ } e2) \implies P c (\text{scrut } ? e1 : e2)$   
**shows**  $P (c::'a::fs) \text{ exp}$   
**apply**  $(rule \text{ exp-heap-strong-induct}(1)[of P \lambda c \Gamma. (\forall x \in domA \Gamma. P c (\text{the } (map-of \Gamma x)))])$   
**apply**  $(metis \text{ assms}(1))$   
**apply**  $(metis \text{ assms}(2))$   
**apply**  $(metis \text{ assms}(3))$   
**apply**  $(metis \text{ assms}(4))$   
**apply**  $(metis \text{ assms}(5))$   
**apply**  $(metis \text{ assms}(6))$   
**apply**  $auto$   
**done**

### 3.2.4 Testing alpha equivalence

**lemma**  $\text{alpha-test}$ :  
**shows**  $\text{Lam } [x]. (\text{Var } x) = \text{Lam } [y]. (\text{Var } y)$   
**by**  $(simp \text{ add: Abs1-eq-iff fresh-at-base pure-fresh})$

**lemma**  $\text{alpha-test2}$ :  
**shows**  $\text{let } x \text{ be } (\text{Var } x) \text{ in } (\text{Var } x) = \text{let } y \text{ be } (\text{Var } y) \text{ in } (\text{Var } y)$   
**by**  $(simp \text{ add: fresh-Cons fresh-Nil Abs1-eq-iff fresh-Pair add: fresh-at-base pure-fresh})$

**lemma**  $\text{alpha-test3}$ :

shows

$Let [(x, Var y), (y, Var x)] (Var x)$   
 $=$   
 $Let [(y, Var x), (x, Var y)] (Var y)$  (is  $Let ?la ?lb = -$ )  
 by (simp add: bn-heapToAssn Abs1-eq-iff fresh-Pair fresh-at-base  
 Abs-swap2[of atom x (?lb, [(x, Var y), (y, Var x)]) [atom x, atom y] atom y])

### 3.2.5 Free variables

**lemma** *fv-supp-exp*:  $supp e = atom \text{ ` } (fv (e::exp) :: var set)$  **and** *fv-supp-as*:  $supp as = atom \text{ ` } (fv (as::assn) :: var set)$

**by** (induction e **and** as rule:exp-assn.inducts)  
 (auto simp add: fv-def exp-assn.supp supp-at-base pure-supp)

**lemma** *fv-supp-heap*:  $supp (\Gamma::heap) = atom \text{ ` } (fv \Gamma :: var set)$

**by** (metis fv-def fv-supp-as supp-heapToAssn)

**lemma** *fv-Lam[simp]*:  $fv (Lam [x]. e) = fv e - \{x\}$

**unfolding** fv-def **by** (auto simp add: exp-assn.supp)

**lemma** *fv-Var[simp]*:  $fv (Var x) = \{x\}$

**unfolding** fv-def **by** (auto simp add: exp-assn.supp supp-at-base pure-supp)

**lemma** *fv-App[simp]*:  $fv (App e x) = insert x (fv e)$

**unfolding** fv-def **by** (auto simp add: exp-assn.supp supp-at-base)

**lemma** *fv-Let[simp]*:  $fv (Let \Gamma e) = (fv \Gamma \cup fv e) - domA \Gamma$

**unfolding** fv-def **by** (auto simp add: Let-supp exp-assn.supp supp-at-base set-bn-to-atom-domA)

**lemma** *fv-Bool[simp]*:  $fv (Bool b) = \{\}$

**unfolding** fv-def **by** (auto simp add: exp-assn.supp pure-supp)

**lemma** *fv-IfThenElse[simp]*:  $fv (scrut ? e1 : e2) = fv scrut \cup fv e1 \cup fv e2$

**unfolding** fv-def **by** (auto simp add: exp-assn.supp)

**lemma** *fv-delete-heap*:

**assumes** map-of  $\Gamma x = Some e$

**shows**  $fv (delete x \Gamma, e) \cup \{x\} \subseteq (fv (\Gamma, Var x) :: var set)$

**proof** –

**have**  $fv (delete x \Gamma) \subseteq fv \Gamma$  **by** (metis fv-delete-subset)

**moreover**

**have**  $(x, e) \in set \Gamma$  **by** (metis assms map-of-SomeD)

**hence**  $fv e \subseteq fv \Gamma$  **by** (metis assms domA-from-set map-of-fv-subset option.sel)

**moreover**

**have**  $x \in fv (Var x)$  **by** simp

**ultimately show** ?thesis **by** auto

qed

### 3.2.6 Lemmas helping with nominal definitions

**lemma** *eqvt-lam-case*:

**assumes**  $Lam [x]. e = Lam [x']. e'$

**assumes**  $\bigwedge \pi . supp (-\pi) \#* (fv (Lam [x]. e) :: var set) \implies$

$supp \pi \#* (Lam [x]. e) \implies$

$F (\pi \cdot e) (\pi \cdot x) (Lam [x]. e) = F e x (Lam [x]. e)$   
**shows**  $F e x (Lam [x]. e) = F e' x' (Lam [x']. e')$   
**proof**–  
**from** *assms(1)*  
**have**  $[[atom\ x]]lst. (e, x) = [[atom\ x']]lst. (e', x')$  **by** *auto*  
**then obtain**  $p$   
**where**  $(supp\ (e, x) - \{atom\ x\}) \#* p$   
**and**  $[simp]: p \cdot x = x'$   
**and**  $[simp]: p \cdot e = e'$   
**unfolding** *Abs-eq-iff(3) alpha-lst.simps* **by** *auto*  
  
**from**  $\langle - \#* p \rangle$   
**have**  $*$ :  $supp\ (-p) \#* (fv\ (Lam\ [x].\ e) :: var\ set)$   
**by** (*auto simp add: fresh-star-def fresh-def supp-finite-set-at-base supp-Pair fv-supply-exp*  
*fv-supply-heap supp-minus-perm*)  
  
**from**  $\langle - \#* p \rangle$   
**have**  $**$ :  $supp\ p \#* Lam\ [x].\ e$   
**by** (*auto simp add: fresh-star-def fresh-def supp-Pair fv-supply-exp*)  
  
**have**  $F\ e\ x\ (Lam\ [x].\ e) = F\ (p \cdot e)\ (p \cdot x)\ (Lam\ [x].\ e)$  **by** (*rule assms(2)[OF \* \*\*, symmetric]*)  
**also have**  $\dots = F\ e'\ x'\ (Lam\ [x'].\ e')$  **by** (*simp add: assms(1)*)  
**finally show** *?thesis*.  
**qed**

**lemma** *eqvt-let-case*:

**assumes**  $Let\ as\ body = Let\ as'\ body'$   
**assumes**  $\bigwedge \pi .$   
 $supp\ (-\pi) \#* (fv\ (Let\ as\ body) :: var\ set) \implies$   
 $supp\ \pi \#* Let\ as\ body \implies$   
 $F\ (\pi \cdot as)\ (\pi \cdot body)\ (Let\ as\ body) = F\ as\ body\ (Let\ as\ body)$   
**shows**  $F\ as\ body\ (Let\ as\ body) = F\ as'\ body'\ (Let\ as'\ body')$   
**proof**–  
**from** *assms(1)*  
**have**  $[map\ (\lambda\ p.\ atom\ (fst\ p))\ as]lst. (body, as) = [map\ (\lambda\ p.\ atom\ (fst\ p))\ as']lst. (body', as')$   
**by** *auto*  
**then obtain**  $p$   
**where**  $(supp\ (body, as) - atom\ 'domA\ as) \#* p$   
**and**  $[simp]: p \cdot body = body'$   
**and**  $[simp]: p \cdot as = as'$   
**unfolding** *Abs-eq-iff(3) alpha-lst.simps* **by** (*auto simp add: domA-def image-image*)  
  
**from**  $\langle - \#* p \rangle$   
**have**  $*$ :  $supp\ (-p) \#* (fv\ (Terms.Let\ as\ body) :: var\ set)$   
**by** (*auto simp add: fresh-star-def fresh-def supp-finite-set-at-base supp-Pair fv-supply-exp*)

*fv-supp-heap supp-minus-perm*)

**from**  $\langle - \#* p \rangle$   
**have**  $**$ : *supp*  $p \#* \text{Terms.Let } as \text{ body}$   
**by** (*auto simp add: fresh-star-def fresh-def supp-Pair fv-supp-exp fv-supp-heap* )  
  
**have**  $F \text{ as body (Let as body) = } F (p \cdot as) (p \cdot body) \text{ (Let as body)}$  **by** (*rule assms(2)[OF \*\* , symmetric]*)  
**also have**  $\dots = F \text{ as' body' (Let as' body')}$  **by** (*simp add: assms(1)*)  
**finally show** *?thesis*.  
**qed**

### 3.2.7 A smart constructor for lets

Certain program transformations might change the bound variables, possibly making it an empty list. This smart constructor avoids the empty let in the resulting expression. Semantically, it should not make a difference.

**definition** *SmartLet* :: *heap* => *exp* => *exp*  
**where** *SmartLet*  $\Gamma e = (\text{if } \Gamma = [] \text{ then } e \text{ else Let } \Gamma e)$

**lemma** *SmartLet-eqvt*[*eqvt*]:  $\pi \cdot (\text{SmartLet } \Gamma e) = \text{SmartLet } (\pi \cdot \Gamma) (\pi \cdot e)$   
**unfolding** *SmartLet-def* **by** *perm-simp rule*

**lemma** *SmartLet-supp*:  
 $\text{supp } (\text{SmartLet } \Gamma e) = (\text{supp } e \cup \text{supp } \Gamma) - \text{atom } '( \text{domA } \Gamma)$   
**unfolding** *SmartLet-def* **using** *Let-supp* **by** (*auto simp add: supp-Nil*)

**lemma** *fv-SmartLet*[*simp*]:  $\text{fv } (\text{SmartLet } \Gamma e) = (\text{fv } \Gamma \cup \text{fv } e) - \text{domA } \Gamma$   
**unfolding** *SmartLet-def* **by** *auto*

### 3.2.8 A predicate for value expressions

**nominal-function** *isLam* :: *exp* => *bool* **where**

*isLam* (*Var*  $x$ ) = *False* |  
*isLam* (*Lam* [ $x$ ].  $e$ ) = *True* |  
*isLam* (*App*  $e x$ ) = *False* |  
*isLam* (*Let*  $as e$ ) = *False* |  
*isLam* (*Bool*  $b$ ) = *False* |  
*isLam* (*scrut* ?  $e1 : e2$ ) = *False*  
**unfolding** *isLam-graph-aux-def eqvt-def*  
**apply** *simp*  
**apply** *simp*  
**apply** (*metis exp-strong-exhaust*)  
**apply** *auto*  
**done**

**nominal-termination** (*eqvt*) **by** *lexicographic-order*

**lemma** *isLam-Lam*: *isLam* (*Lam* [ $x$ ].  $e$ ) **by** *simp*

**lemma** *isLam-obtain-fresh*:  
**assumes** *isLam* *z*  
**obtains** *y e'*  
**where**  $z = (\text{Lam } [y]. e')$  **and** *atom*  $y \# (c::'a::fs)$   
**using** *assms* **by** (*nominal-induct* *z* *avoiding*: *c* *rule*:*exp-strong-induct*) *auto*

**nominal-function** *isVal* :: *exp*  $\Rightarrow$  *bool* **where**

*isVal* (*Var* *x*) = *False* |  
*isVal* (*Lam* [*x*]. *e*) = *True* |  
*isVal* (*App* *e* *x*) = *False* |  
*isVal* (*Let* *as* *e*) = *False* |  
*isVal* (*Bool* *b*) = *True* |  
*isVal* (*scrut* ? *e1* : *e2*) = *False*

**unfolding** *isVal-graph-aux-def* *eqvt-def*

**apply** *simp*

**apply** *simp*

**apply** (*metis* *exp-strong-exhaust*)

**apply** *auto*

**done**

**nominal-termination** (*eqvt*) **by** *lexicographic-order*

**lemma** *isVal-Lam*: *isVal* (*Lam* [*x*]. *e*) **by** *simp*

**lemma** *isVal-Bool*: *isVal* (*Bool* *b*) **by** *simp*

### 3.2.9 The notion of thunks

**definition** *thunks* :: *heap*  $\Rightarrow$  *var set* **where**

*thunks*  $\Gamma = \{x . \text{case } \text{map-of } \Gamma \text{ of } \text{Some } e \Rightarrow \neg \text{isVal } e \mid \text{None} \Rightarrow \text{False}\}$

**lemma** *thunks-Nil*[*simp*]: *thunks* [] = {} **by** (*auto* *simp* *add*: *thunks-def*)

**lemma** *thunks-domA*: *thunks*  $\Gamma \subseteq \text{domA } \Gamma$

**by** (*induction*  $\Gamma$ ) (*auto* *simp* *add*: *thunks-def*)

**lemma** *thunks-Cons*: *thunks* ((*x,e*)# $\Gamma$ ) = (*if isVal* *e* *then* *thunks*  $\Gamma - \{x\}$  *else* *insert* *x* (*thunks*  $\Gamma$ ))

**by** (*auto* *simp* *add*: *thunks-def*)

**lemma** *thunks-append*[*simp*]: *thunks* ( $\Delta @ \Gamma$ ) = *thunks*  $\Delta \cup (\text{thunks } \Gamma - \text{domA } \Delta)$

**by** (*induction*  $\Delta$ ) (*auto* *simp* *add*: *thunks-def*)

**lemma** *thunks-delete*[*simp*]: *thunks* (*delete* *x*  $\Gamma$ ) = *thunks*  $\Gamma - \{x\}$

**by** (*induction*  $\Gamma$ ) (*auto* *simp* *add*: *thunks-def*)

**lemma** *thunksI*[*intro*]: *map-of*  $\Gamma$  *x* = *Some* *e*  $\Longrightarrow \neg \text{isVal } e \Longrightarrow x \in \text{thunks } \Gamma$

**by** (*induction*  $\Gamma$ ) (*auto* *simp* *add*: *thunks-def*)

**lemma** *thunksE*[*intro*]:  $x \in \text{thunks } \Gamma \Longrightarrow \text{map-of } \Gamma \text{ } x = \text{Some } e \Longrightarrow \neg \text{isVal } e$

by (induction  $\Gamma$ ) (auto simp add: thunks-def )

**lemma** *thunks-cong*: map-of  $\Gamma = \text{map-of } \Delta \implies \text{thunks } \Gamma = \text{thunks } \Delta$   
by (simp add: thunks-def)

**lemma** *thunks-eqvt*[*eqvt*]:  
 $\pi \cdot \text{thunks } \Gamma = \text{thunks } (\pi \cdot \Gamma)$   
unfolding *thunks-def*  
by *perm-simp rule*

### 3.2.10 Non-recursive Let bindings

**definition** *nonrec* :: heap  $\implies$  bool **where**  
*nonrec*  $\Gamma = (\exists x e. \Gamma = [(x,e)] \wedge x \notin \text{fv } e)$

**lemma** *nonrecE*:  
assumes *nonrec*  $\Gamma$   
obtains *x e* **where**  $\Gamma = [(x,e)]$  **and**  $x \notin \text{fv } e$   
using *assms*  
unfolding *nonrec-def*  
by *blast*

**lemma** *nonrec-eqvt*[*eqvt*]:  
*nonrec*  $\Gamma \implies \text{nonrec } (\pi \cdot \Gamma)$   
apply (erule *nonrecE*)  
apply (auto simp add: *nonrec-def* *fv-def* *fresh-def* )  
apply (metis *fresh-at-base-permute-iff* *fresh-def*)  
done

**lemma** *exp-induct-rec*[*case-names* *Var* *App* *Let* *Let-nonrec* *Lam* *Bool* *IfThenElse*]:  
assumes  $\bigwedge \text{var}. P (\text{Var } \text{var})$   
assumes  $\bigwedge \text{exp } \text{var}. P \text{exp} \implies P (\text{App } \text{exp } \text{var})$   
assumes  $\bigwedge \Gamma \text{exp}. \neg \text{nonrec } \Gamma \implies (\bigwedge x. x \in \text{domA } \Gamma \implies P (\text{the } (\text{map-of } \Gamma x))) \implies P \text{exp}$   
 $\implies P (\text{Let } \Gamma \text{exp})$   
assumes  $\bigwedge x e \text{exp}. x \notin \text{fv } e \implies P e \implies P \text{exp} \implies P (\text{let } x \text{ be } e \text{ in } \text{exp})$   
assumes  $\bigwedge \text{var } \text{exp}. P \text{exp} \implies P (\text{Lam } [\text{var}]. \text{exp})$   
assumes  $\bigwedge b. P (\text{Bool } b)$   
assumes  $\bigwedge \text{scrut } e1 e2. P \text{scrut} \implies P e1 \implies P e2 \implies P (\text{scrut } ? e1 : e2)$   
shows *P exp*  
apply (rule *exp-induct*[of *P*])  
apply (metis *assms*(1))  
apply (metis *assms*(2))  
apply (case-tac *nonrec*  $\Gamma$ )  
apply (erule *nonrecE*)  
apply *simp*  
apply (metis *assms*(4))  
apply (metis *assms*(3))  
apply (metis *assms*(5))

**apply** (*metis assms(6)*)  
**apply** (*metis assms(7)*)  
**done**

**lemma** *exp-strong-induct-rec*[*case-names Var App Let Let-nonrec Lam Bool IfThenElse*]:

**assumes**  $\bigwedge var\ c. P\ c\ (Var\ var)$   
**assumes**  $\bigwedge exp\ var\ c. (\bigwedge c. P\ c\ exp) \implies P\ c\ (App\ exp\ var)$   
**assumes**  $\bigwedge \Gamma\ exp\ c.$   
 $atom\ 'domA\ \Gamma\ \#\#*\ c \implies \neg\ nonrec\ \Gamma \implies (\bigwedge c\ x. x \in domA\ \Gamma \implies P\ c\ (the\ (map-of\ \Gamma\ x)))$   
 $\implies (\bigwedge c. P\ c\ exp) \implies P\ c\ (Let\ \Gamma\ exp)$   
**assumes**  $\bigwedge x\ e\ exp\ c. \{atom\ x\}\ \#\#*\ c \implies x \notin fv\ e \implies (\bigwedge c. P\ c\ e) \implies (\bigwedge c. P\ c\ exp) \implies P\ c\ (let\ x\ be\ e\ in\ exp)$   
**assumes**  $\bigwedge var\ exp\ c. \{atom\ var\}\ \#\#*\ c \implies (\bigwedge c. P\ c\ exp) \implies P\ c\ (Lam\ [var].\ exp)$   
**assumes**  $\bigwedge b\ c. P\ c\ (Bool\ b)$   
**assumes**  $\bigwedge scrut\ e1\ e2\ c. (\bigwedge c. P\ c\ scrut) \implies (\bigwedge c. P\ c\ e1) \implies (\bigwedge c. P\ c\ e2) \implies P\ c\ (scrut\ ?\ e1 : e2)$   
**shows**  $P\ (c::'a::fs)\ exp$   
**apply** (*rule exp-strong-induct[of P]*)  
**apply** (*metis assms(1)*)  
**apply** (*metis assms(2)*)  
**apply** (*case-tac nonrec Γ*)  
**apply** (*erule nonrecE*)  
**apply** *simp*  
**apply** (*metis assms(4)*)  
**apply** (*metis assms(3)*)  
**apply** (*metis assms(5)*)  
**apply** (*metis assms(6)*)  
**apply** (*metis assms(7)*)  
**done**

**lemma** *exp-strong-induct-rec-set*[*case-names Var App Let Let-nonrec Lam Bool IfThenElse*]:

**assumes**  $\bigwedge var\ c. P\ c\ (Var\ var)$   
**assumes**  $\bigwedge exp\ var\ c. (\bigwedge c. P\ c\ exp) \implies P\ c\ (App\ exp\ var)$   
**assumes**  $\bigwedge \Gamma\ exp\ c.$   
 $atom\ 'domA\ \Gamma\ \#\#*\ c \implies \neg\ nonrec\ \Gamma \implies (\bigwedge c\ x\ e. (x,e) \in set\ \Gamma \implies P\ c\ e) \implies (\bigwedge c. P\ c\ exp) \implies P\ c\ (Let\ \Gamma\ exp)$   
**assumes**  $\bigwedge x\ e\ exp\ c. \{atom\ x\}\ \#\#*\ c \implies x \notin fv\ e \implies (\bigwedge c. P\ c\ e) \implies (\bigwedge c. P\ c\ exp) \implies P\ c\ (let\ x\ be\ e\ in\ exp)$   
**assumes**  $\bigwedge var\ exp\ c. \{atom\ var\}\ \#\#*\ c \implies (\bigwedge c. P\ c\ exp) \implies P\ c\ (Lam\ [var].\ exp)$   
**assumes**  $\bigwedge b\ c. P\ c\ (Bool\ b)$   
**assumes**  $\bigwedge scrut\ e1\ e2\ c. (\bigwedge c. P\ c\ scrut) \implies (\bigwedge c. P\ c\ e1) \implies (\bigwedge c. P\ c\ e2) \implies P\ c\ (scrut\ ?\ e1 : e2)$   
**shows**  $P\ (c::'a::fs)\ exp$   
**apply** (*rule exp-strong-induct-set(1)[of P]*)  
**apply** (*metis assms(1)*)  
**apply** (*metis assms(2)*)  
**apply** (*case-tac nonrec Γ*)  
**apply** (*erule nonrecE*)  
**apply** *simp*

```

apply (metis assms(4))
apply (metis assms(3))
apply (metis assms(5))
apply (metis assms(6))
apply (metis assms(7))
done

```

### 3.2.11 Renaming a lambda-bound variable

**lemma** *change-Lam-Variable*:

```

assumes  $y' \neq y \implies \text{atom } y' \# (e, y)$ 
shows  $\text{Lam } [y]. e = \text{Lam } [y']. ((y \leftrightarrow y') \cdot e)$ 
proof (cases  $y' = y$ )
case True thus ?thesis by simp
next
case False
from assms[OF this]
have  $(y \leftrightarrow y') \cdot (\text{Lam } [y]. e) = \text{Lam } [y]. e$ 
by -(rule flip-fresh-fresh, (simp add: fresh-Pair)+)
moreover
have  $(y \leftrightarrow y') \cdot (\text{Lam } [y]. e) = \text{Lam } [y']. ((y \leftrightarrow y') \cdot e)$ 
by simp
ultimately
show  $\text{Lam } [y]. e = \text{Lam } [y']. ((y \leftrightarrow y') \cdot e)$  by (simp add: fresh-Pair)
qed

```

**end**

## 3.3 Substitution

```

theory Substitution
imports Terms
begin

```

Defining a substitution function on terms turned out to be slightly tricky.

```

fun
  subst-var :: var  $\Rightarrow$  var  $\Rightarrow$  var  $\Rightarrow$  var ( $\langle$ -[::v=-] $\rangle$  [1000,100,100] 1000)
where  $x[y ::v= z] = (\text{if } x = y \text{ then } z \text{ else } x)$ 

nominal-function (default case-sum ( $\lambda x. \text{Inl undefined}$ ) ( $\lambda x. \text{Inr undefined}$ ),
  invariant  $\lambda a r . (\forall \Gamma y z . ((a = \text{Inr } (\Gamma, y, z) \wedge \text{atom } \text{' domA } \Gamma \#* (y, z)) \longrightarrow$ 
  map ( $\lambda x . \text{atom } (\text{fst } x)$ ) (Sum-Type.proj r) = map ( $\lambda x . \text{atom } (\text{fst } x)$ )  $\Gamma$ ))
  subst :: exp  $\Rightarrow$  var  $\Rightarrow$  var  $\Rightarrow$  exp ( $\langle$ -[::=-] $\rangle$  [1000,100,100] 1000)
and
  subst-heap :: heap  $\Rightarrow$  var  $\Rightarrow$  var  $\Rightarrow$  heap ( $\langle$ -[::h=-] $\rangle$  [1000,100,100] 1000)
where
  (Var  $x$ )[ $y ::= z$ ] = Var ( $x[y ::v= z]$ )

```

```

| (App e v)[y ::= z] = App (e[y ::= z]) (v[y ::= v= z])
| atom ‘ domA Γ #* (y,z) ==>
  (Let Γ body)[y ::= z] = Let (Γ[y ::h= z]) (body[y ::= z])
| atom x # (y,z) ==> (Lam [x].e)[y ::= z] = Lam [x].(e[y::=z])
| (Bool b)[y ::= z] = Bool b
| (scrut ? e1 : e2)[y ::= z] = (scrut[y ::= z] ? e1[y ::= z] : e2[y ::= z])
| [][y ::h= z] = []
| ((v,e)# Γ)[y ::h= z] = (v, e[y ::= z])# (Γ[y ::h= z])

```

**proof** goal-cases

**have** eqvt-at-subst:  $\bigwedge e y z . eqvt\text{-at}\ subst\text{-subst-heap-sum}C (Inl (e, y, z)) \implies eqvt\text{-at} (\lambda(a, b, c). subst\ a\ b\ c) (e, y, z)$

```

apply(simp add: eqvt-at-def subst-def)
apply(rule)
apply(subst Projl-permute)
apply(thin-tac -)+
apply (simp add: subst-subst-heap-sumC-def)
apply (simp add: THE-default-def)
apply (case-tac Ex1 (subst-subst-heap-graph (Inl (e, y, z))))
apply(simp)
apply(auto)[1]
apply (erule-tac x=x in allE)
apply simp
apply(cases rule: subst-subst-heap-graph.cases)
apply(assumption)
apply(rule-tac x=Sum-Type.projl x in exI)
apply(clarify)
apply (rule the1-equality)
apply blast
apply(simp (no-asm) only: sum.sel)
apply(rule-tac x=Sum-Type.projl x in exI)
apply(clarify)
apply (rule the1-equality)
apply blast
apply(simp (no-asm) only: sum.sel)
apply(rule-tac x=Sum-Type.projl x in exI)
apply(clarify)
apply (rule the1-equality)
apply blast
apply(simp (no-asm) only: sum.sel)
apply(rule-tac x=Sum-Type.projl x in exI)
apply(clarify)
apply (rule the1-equality)
apply blast
apply(simp (no-asm) only: sum.sel)
apply(rule-tac x=Sum-Type.projl x in exI)
apply(clarify)
apply (rule the1-equality)
apply blast

```

```

apply(simp (no-asm) only: sum.sel)
apply(rule-tac x=Sum-Type.proj1 x in exI)
apply(clarify)
apply (rule the1-equality)
apply blast
apply(simp (no-asm) only: sum.sel)
apply (metis Inr-not-Inl)
apply (metis Inr-not-Inl)
apply(simp)
apply(perm-simp)
apply(simp)
done

have eqvt-at-subst-heap:  $\bigwedge \Gamma y z . eqvt-at\ subst-subst-heap-sumC (Inr (\Gamma, y, z)) \implies eqvt-at$ 
( $\lambda(a, b, c). subst-heap\ a\ b\ c$ ) ( $\Gamma, y, z$ )
  apply(simp add: eqvt-at-def subst-heap-def)
  apply(rule)
  apply(subst Projr-permute)
  apply(thin-tac -)+
  apply (simp add: subst-subst-heap-sumC-def)
  apply (simp add: THE-default-def)
  apply (case-tac Ex1 (subst-subst-heap-graph (Inr (\Gamma, y, z))))
  apply(simp)
  apply(auto)[1]
  apply (erule-tac x=x in allE)
  apply simp
  apply(cases rule: subst-subst-heap-graph.cases)
  apply(assumption)
  apply (metis (mono-tags) Inr-not-Inl)+
  apply(rule-tac x=Sum-Type.proj1 x in exI)
  apply(clarify)
  apply (rule the1-equality)
  apply auto[1]
  apply(simp (no-asm) only: sum.sel)

  apply(rule-tac x=Sum-Type.proj1 x in exI)
  apply(clarify)
  apply (rule the1-equality)
  apply auto[1]
  apply(simp (no-asm) only: sum.sel)

  apply(simp)
  apply(perm-simp)
  apply(simp)
done

{

case 1 thus ?case

```

**unfolding** *eqvt-def subst-subst-heap-graph-aux-def*  
**by** *simp*

**next case** 2 **thus** ?*case*  
**by** (*induct rule: subst-subst-heap-graph.induct*)(*auto simp add: exp-assn.bn-defs fresh-star-insert*)

**next case** *prems: (3 P x)* **show** ?*case*  
**proof**(*cases x*)  
**case** (*Inl a*) **thus** *P*  
**proof**(*cases a*)  
**case** (*fields a1 a2 a3*)  
**thus** *P* **using** *Inl prems*  
**apply** (*rule-tac y =a1 and c =(a2, a3) in exp-strong-exhaust*)  
**apply** (*auto simp add: fresh-star-def*)  
**done**  
**qed**  
**next**  
**case** (*Inr a*) **thus** *P*  
**proof** (*cases a*)  
**case** (*fields a1 a2 a3*)  
**thus** *P* **using** *Inr prems*  
**by** (*metis heapToAssn.cases*)  
**qed**  
**qed**

**next case** (19 *e y2 z2 Γ e2 y z as2*) **thus** ?*case*  
**apply** –  
**apply** (*drule eqvt-at-subst*)  
**apply** (*drule eqvt-at-subst-heap*)  
**apply** (*simp only: meta-eq-to-obj-eq[OF subst-def, symmetric, unfolded fun-eq-iff]*  
*meta-eq-to-obj-eq[OF subst-heap-def, symmetric, unfolded fun-eq-iff]*)  
**apply** (*auto simp add: Abs-fresh-iff*)  
**apply** (*drule-tac*  
*c = (y, z) and*  
*as = (map (λx. atom (fst x)) e) and*  
*bs = (map (λx. atom (fst x)) e2) and*  
*f = λ a b c . [a]lst. (subst (fst b) y z, subst-heap (snd b) y z ) in Abs-lst-fcb2*)  
**apply** (*simp add: perm-supp-eq fresh-Pair fresh-star-def Abs-fresh-iff*)  
**apply** (*metis domA-def image-image image-set*)  
**apply** (*metis domA-def image-image image-set*)  
**apply** (*simp add: eqvt-at-def, simp add: fresh-star-Pair perm-supp-eq*)  
**apply** (*simp add: eqvt-at-def, simp add: fresh-star-Pair perm-supp-eq*)  
**apply** (*simp add: eqvt-at-def*)  
**done**

**next case** (25 *x2 y2 z2 e2 x y z e*) **thus** ?*case*

```

apply –
apply (drule eqvt-at-subst) +
apply (simp only: Abs-fresh-iff meta-eq-to-obj-eq[OF subst-def, symmetric, unfolded fun-eq-iff])

apply (simp add: eqvt-at-def)
apply rule
apply (erule-tac x = (x2 ↔ c) in allE)
apply (erule-tac x = (x ↔ c) in allE)
apply auto
done
}
qed(auto)

```

**nominal-termination** (*eqvt*) **by** *lexicographic-order*

**lemma shows**

```

  True and bn-subst[simp]: domA (subst-heap Γ y z) = domA Γ
by(induct rule:subst-subst-heap.induct)
  (auto simp add: exp-assn.bn-defs fresh-star-insert)

```

**lemma** *subst-noop[simp]:*

```

shows  $e[y ::= y] = e$  and  $\Gamma[y::h=y] = \Gamma$ 
by(induct e y y and Γ y y rule:subst-subst-heap.induct)
  (auto simp add: fresh-star-Pair exp-assn.bn-defs)

```

**lemma** *subst-is-fresh[simp]:*

```

assumes atom y # z
shows
  atom y # e[y ::= z]
and
  atom ‘ domA Γ #* y ⇒ atom y # Γ[y::h=z]
using assms
by(induct e y z and Γ y z rule:subst-subst-heap.induct)
  (auto simp add: fresh-at-base fresh-star-Pair fresh-star-insert fresh-Nil fresh-Cons pure-fresh)

```

**lemma**

```

  subst-pres-fresh: atom x # e ∨ x = y ⇒ atom x # z ⇒ atom x # e[y ::= z]
and
  atom x # Γ ∨ x = y ⇒ atom x # z ⇒ x ∉ domA Γ ⇒ atom x # (Γ[y::h= z])
by(induct e y z and Γ y z rule:subst-subst-heap.induct)
  (auto simp add: fresh-star-Pair exp-assn.bn-defs fresh-Cons fresh-Nil pure-fresh)

```

**lemma** *subst-fresh-noop: atom x # e ⇒ e[x ::= y] = e*

```

  and subst-heap-fresh-noop: atom x # Γ ⇒ Γ[x::h= y] = Γ
by (nominal-induct e and Γ avoiding: x y rule:exp-heap-strong-induct)
  (auto simp add: fresh-star-def fresh-Pair fresh-at-base fresh-Cons simp del: exp-assn.eq-iff)

```

**lemma** *supp-subst-eq: supp (e[y::=x]) = (supp e – {atom y}) ∪ (if atom y ∈ supp e then {atom*

$x\}$  else  $\{\}$ )  
**and**  $\text{atom } \text{'domA } \Gamma \#* y \implies \text{supp } (\Gamma[y::h=x]) = (\text{supp } \Gamma - \{\text{atom } y\}) \cup (\text{if } \text{atom } y \in \text{supp } \Gamma \text{ then } \{\text{atom } x\} \text{ else } \{\})$   
**by** (*nominal-induct e and  $\Gamma$  avoiding: x y rule:exp-heap-strong-induct*)  
*(auto simp add: fresh-star-def fresh-Pair supp-Nil supp-Cons supp-Pair fresh-Cons exp-assn.supp Let-supp supp-at-base pure-supp simp del: exp-assn.eq-iff)*

**lemma** *supp-subst*:  $\text{supp } (e[y::=x]) \subseteq (\text{supp } e - \{\text{atom } y\}) \cup \{\text{atom } x\}$   
**using** *supp-subst-eq* **by** *auto*

**lemma** *fv-subst-eq*:  $\text{fv } (e[y::=x]) = (\text{fv } e - \{y\}) \cup (\text{if } y \in \text{fv } e \text{ then } \{x\} \text{ else } \{\})$   
**and**  $\text{atom } \text{'domA } \Gamma \#* y \implies \text{fv } (\Gamma[y::h=x]) = (\text{fv } \Gamma - \{y\}) \cup (\text{if } y \in \text{fv } \Gamma \text{ then } \{x\} \text{ else } \{\})$   
**by** (*nominal-induct e and  $\Gamma$  avoiding: x y rule:exp-heap-strong-induct*)  
*(auto simp add: fresh-star-def fresh-Pair supp-Nil supp-Cons supp-Pair fresh-Cons exp-assn.supp Let-supp supp-at-base simp del: exp-assn.eq-iff)*

**lemma** *fv-subst-subset*:  $\text{fv } (e[y ::= x]) \subseteq (\text{fv } e - \{y\}) \cup \{x\}$   
**using** *fv-subst-eq* **by** *auto*

**lemma** *fv-subst-int*:  $x \notin S \implies y \notin S \implies \text{fv } (e[y ::= x]) \cap S = \text{fv } e \cap S$   
**by** (*auto simp add: fv-subst-eq*)

**lemma** *fv-subst-int2*:  $x \notin S \implies y \notin S \implies S \cap \text{fv } (e[y ::= x]) = S \cap \text{fv } e$   
**by** (*auto simp add: fv-subst-eq*)

**lemma** *subst-swap-same*:  $\text{atom } x \# e \implies (x \leftrightarrow y) \cdot e = e[y ::= x]$   
**and**  $\text{atom } x \# \Gamma \implies \text{atom } \text{'domA } \Gamma \#* y \implies (x \leftrightarrow y) \cdot \Gamma = \Gamma[y ::= h=x]$   
**by** (*nominal-induct e and  $\Gamma$  avoiding: x y rule:exp-heap-strong-induct*)  
*(auto simp add: fresh-star-Pair fresh-star-at-base fresh-Cons pure-fresh permute-pure simp del: exp-assn.eq-iff)*

**lemma** *subst-subst-back*:  $\text{atom } x \# e \implies e[y::=x][x::=y] = e$   
**and**  $\text{atom } x \# \Gamma \implies \text{atom } \text{'domA } \Gamma \#* y \implies \Gamma[y::h=x][x::h=y] = \Gamma$   
**by**(*nominal-induct e and  $\Gamma$  avoiding: x y rule:exp-heap-strong-induct*)  
*(auto simp add: fresh-star-Pair fresh-star-at-base fresh-star-Cons fresh-Cons exp-assn.bn-defs simp del: exp-assn.eq-iff)*

**lemma** *subst-heap-delete[simp]*:  $(\text{delete } x \Gamma)[y ::= h= z] = \text{delete } x (\Gamma[y ::= h= z])$   
**by** (*induction  $\Gamma$* ) *auto*

**lemma** *subst-nil-iff[simp]*:  $\Gamma[x ::= h= z] = [] \longleftrightarrow \Gamma = []$   
**by** (*cases  $\Gamma$* ) *auto*

**lemma** *subst-SmartLet[simp]*:  
 $\text{atom } \text{'domA } \Gamma \#* (y, z) \implies (\text{SmartLet } \Gamma \text{ body})[y ::= z] = \text{SmartLet } (\Gamma[y ::= h= z]) (\text{body}[y ::= z])$   
**unfolding** *SmartLet-def* **by** *auto*

**lemma** *subst-let-be*[simp]:  
 $atom\ x' \# y \implies atom\ x' \# x \implies (let\ x'\ be\ e\ in\ exp)[y::=x] = (let\ x'\ be\ e[y::=x]\ in\ exp[y::=x])$   
**by** (*simp add: fresh-star-def fresh-Pair*)

**lemma** *isLam-subst*[simp]:  $isLam\ e[x::=y] = isLam\ e$   
**by** (*nominal-induct e avoiding: x y rule: exp-strong-induct*)  
*(auto simp add: fresh-star-Pair)*

**lemma** *isVal-subst*[simp]:  $isVal\ e[x::=y] = isVal\ e$   
**by** (*nominal-induct e avoiding: x y rule: exp-strong-induct*)  
*(auto simp add: fresh-star-Pair)*

**lemma** *thunks-subst*[simp]:  
 $thunks\ \Gamma[y::h=x] = thunks\ \Gamma$   
**by** (*induction \Gamma*) *(auto simp add: thunks-Cons)*

**lemma** *map-of-subst*:  
 $map-of\ (\Gamma[x::h=y])\ k = map-option\ (\lambda\ e.\ e[x::=y])\ (map-of\ \Gamma\ k)$   
**by** (*induction \Gamma*) *auto*

**lemma** *mapCollect-subst*[simp]:  
 $\{e\ k\ v \mid k \mapsto v \in map-of\ \Gamma[x::h=y]\} = \{e\ k\ v[x::=y] \mid k \mapsto v \in map-of\ \Gamma\}$   
**by** (*auto simp add: map-of-subst*)

**lemma** *subst-eq-Cons*:  
 $\Gamma[x::h=y] = (x', e) \# \Delta \iff (\exists\ e'\ \Gamma'.\ \Gamma = (x', e') \# \Gamma' \wedge e'[x::=y] = e \wedge \Gamma'[x::h=y] = \Delta)$   
**by** (*cases \Gamma*) *auto*

**lemma** *nonrec-subst*:  
 $atom\ 'domA\ \Gamma \#* x \implies atom\ 'domA\ \Gamma \#* y \implies nonrec\ \Gamma[x::h=y] \iff nonrec\ \Gamma$   
**by** (*auto simp add: nonrec-def fresh-star-def subst-eq-Cons fv-subst-eq*)

**end**

## 3.4 Launchbury

**theory** *Launchbury*  
**imports** *Terms Substitution*  
**begin**

### 3.4.1 The natural semantics

This is the semantics as in [Lau93], with two differences:

- Explicit freshness requirements for bound variables in the application and the Let rule.
- An additional parameter that stores variables that have to be avoided, but do not occur in the judgement otherwise, following [Ses97].

**inductive**

$reds :: heap \Rightarrow exp \Rightarrow var\ list \Rightarrow heap \Rightarrow exp \Rightarrow bool$   
 $(\langle - : - \Downarrow - : - \rangle [50,50,50,50] 50)$

**where***Lambda:*
 $\Gamma : (Lam\ [x].\ e) \Downarrow_L \Gamma : (Lam\ [x].\ e)$ 
*Application:*  $\llbracket$ 
 $atom\ y \# (\Gamma, e, x, L, \Delta, \Theta, z) ;$ 
 $\Gamma : e \Downarrow_L \Delta : (Lam\ [y].\ e')$ 
 $\Delta : e'[y ::= x] \Downarrow_L \Theta : z$ 
 $\rrbracket \Rightarrow$ 
 $\Gamma : App\ e\ x \Downarrow_L \Theta : z$ 
*Variable:*  $\llbracket$ 
 $map-of\ \Gamma\ x = Some\ e; delete\ x\ \Gamma : e \Downarrow_{x\#L} \Delta : z$ 
 $\rrbracket \Rightarrow$ 
 $\Gamma : Var\ x \Downarrow_L (x, z) \# \Delta : z$ 
*Let:*  $\llbracket$ 
 $atom\ 'domA\ \Delta \#* (\Gamma, L);$ 
 $\Delta @ \Gamma : body \Downarrow_L \Theta : z$ 
 $\rrbracket \Rightarrow$ 
 $\Gamma : Let\ \Delta\ body \Downarrow_L \Theta : z$ 
*Bool:*
 $\Gamma : Bool\ b \Downarrow_L \Gamma : Bool\ b$ 
*IfThenElse:*  $\llbracket$ 
 $\Gamma : scrut \Downarrow_L \Delta : (Bool\ b);$ 
 $\Delta : (if\ b\ then\ e_1\ else\ e_2) \Downarrow_L \Theta : z$ 
 $\rrbracket \Rightarrow$ 
 $\Gamma : (scrut\ ?\ e_1 : e_2) \Downarrow_L \Theta : z$ 
**equivariance reds****nominal-inductive reds****avoids** *Application:*  $y$ **by** (*auto simp add: fresh-star-def fresh-Pair*)**3.4.2 Example evaluations****lemma** *eval-test:*
 $\llbracket : (Let\ [(x, Lam\ [y].\ Var\ y)]\ (Var\ x)) \Downarrow_{\llbracket} [(x, Lam\ [y].\ Var\ y)] : (Lam\ [y].\ Var\ y)$ 
**apply**(*auto intro!: Lambda Application Variable Let**simp add: fresh-Pair fresh-Cons fresh-Nil fresh-star-def*)**done****lemma** *eval-test2:*
 $y \neq x \Rightarrow n \neq y \Rightarrow n \neq x \Rightarrow \llbracket : (Let\ [(x, Lam\ [y].\ Var\ y)]\ (App\ (Var\ x)\ x)) \Downarrow_{\llbracket} [(x, Lam\ [y].\ Var\ y)] : (Lam\ [y].\ Var\ y)$ 
**by** (*auto intro!: Lambda Application Variable Let simp add: fresh-Pair fresh-at-base fresh-Cons fresh-Nil fresh-star-def pure-fresh*)

### 3.4.3 Better introduction rules

This variant do not require freshness.

**lemma** *reds-ApplicationI*:

**assumes**  $\Gamma : e \Downarrow_L \Delta : Lam [y]. e'$

**assumes**  $\Delta : e'[y::=x] \Downarrow_L \Theta : z$

**shows**  $\Gamma : App e x \Downarrow_L \Theta : z$

**proof** –

**obtain**  $y' :: var$  **where**  $atom\ y' \# (\Gamma, e, x, L, \Delta, \Theta, z, e')$  **by** (*rule obtain-fresh*)

**have**  $a: Lam [y']. ((y' \leftrightarrow y) \cdot e') = Lam [y]. e'$

**using**  $\langle atom\ y' \# \rightarrow \rangle$

**by** (*auto simp add: Abs1-eq-iff fresh-Pair fresh-at-base*)

**have**  $b: ((y' \leftrightarrow y) \cdot e')[y'::=x] = e'[y::=x]$

**proof**(*cases x = y*)

**case** *True*

**have**  $atom\ y' \# e'$  **using**  $\langle atom\ y' \# \rightarrow \rangle$  **by** *simp*

**thus** *?thesis*

**by** (*simp add: True subst-swap-same subst-sbst-back*)

**next**

**case** *False*

**hence**  $atom\ y \# x$  **by** *simp*

**have**  $[simp]: (y' \leftrightarrow y) \cdot x = x$  **using**  $\langle atom\ y \# \rightarrow \rangle \langle atom\ y' \# \rightarrow \rangle$

**by** (*simp add: flip-fresh-fresh fresh-Pair fresh-at-base*)

**have**  $((y' \leftrightarrow y) \cdot e')[y'::=x] = (y' \leftrightarrow y) \cdot (e'[y::=x])$  **by** *simp*

**also have**  $\dots = e'[y::=x]$

**using**  $\langle atom\ y \# \rightarrow \rangle \langle atom\ y' \# \rightarrow \rangle$

**by** (*simp add: flip-fresh-fresh fresh-Pair fresh-at-base subst-pres-fresh*)

**finally**

**show** *?thesis*.

**qed**

**have**  $atom\ y' \# (\Gamma, e, x, L, \Delta, \Theta, z)$  **using**  $\langle atom\ y' \# \rightarrow \rangle$  **by** (*simp add: fresh-Pair*)

**from** *this assms[folded a b]*

**show** *?thesis ..*

**qed**

**lemma** *reds-SmartLet*:  $\llbracket$

$atom\ 'domA\ \Delta \#* (\Gamma, L);$

$\Delta @ \Gamma : body \Downarrow_L \Theta : z$

$\rrbracket \implies$

$\Gamma : SmartLet\ \Delta\ body \Downarrow_L \Theta : z$

**unfolding** *SmartLet-def*

**by** (*auto intro: reds.Let*)

A single rule for values

**lemma** *reds-isValI*:  
 $isVal\ z \implies \Gamma : z \Downarrow_L \Gamma : z$   
**by** (*cases z rule:isVal.cases*) (*auto intro: reds.intros*)

### 3.4.4 Properties of the semantics

Heap entries are never removed.

**lemma** *reds-doesnt-forget*:  
 $\Gamma : e \Downarrow_L \Delta : z \implies domA\ \Gamma \subseteq domA\ \Delta$   
**by**(*induct rule: reds.induct*) *auto*

Live variables are not added to the heap.

**lemma** *reds-avoids-live'*:  
**assumes**  $\Gamma : e \Downarrow_L \Delta : z$   
**shows**  $(domA\ \Delta - domA\ \Gamma) \cap set\ L = \{\}$   
**using** *assms*  
**by**(*induct rule:reds.induct*)  
(*auto dest: map-of-domAD fresh-distinct-list simp add: fresh-star-Pair*)

**lemma** *reds-avoids-live*:  
 $\llbracket \Gamma : e \Downarrow_L \Delta : z;$   
 $x \in set\ L;$   
 $x \notin domA\ \Gamma$   
 $\rrbracket \implies x \notin domA\ \Delta$   
**using** *reds-avoids-live'* **by** *blast*

Fresh variables either stay fresh or are added to the heap.

**lemma** *reds-fresh*:  $\llbracket \Gamma : e \Downarrow_L \Delta : z;$   
 $atom\ (x::var)\ \#\ (\Gamma, e)$   
 $\rrbracket \implies atom\ x\ \#\ (\Delta, z) \vee x \in (domA\ \Delta - set\ L)$   
**proof**(*induct rule: reds.induct*)  
**case** (*Lambda*  $\Gamma\ x\ e$ ) **thus** *?case by auto*  
**next**  
**case** (*Application*  $y\ \Gamma\ e\ x'\ L\ \Delta\ \Theta\ z\ e'$ )  
**hence**  $atom\ x\ \#\ (\Delta, Lam\ [y].\ e') \vee x \in domA\ \Delta - set\ (x'\ \#\ L)$  **by** (*auto simp add: fresh-Pair*)

**thus** *?case*

**proof**

**assume**  $atom\ x\ \#\ (\Delta, Lam\ [y].\ e')$   
**hence**  $atom\ x\ \#\ e'[y ::= x']$   
**using** *Application.prem*  
**by** (*auto intro: subst-pres-fresh simp add: fresh-Pair*)  
**thus** *?thesis using Application.hyps(5) <atom x # (Δ, Lam [y]. e')>* **by** *auto*  
**next**  
**assume**  $x \in domA\ \Delta - set\ (x'\ \#\ L)$   
**thus** *?thesis using reds-doesnt-forget[OF Application.hyps(4)]* **by** *auto*  
**qed**

next

```
case (Variable  $\Gamma v e L \Delta z$ )
  have atom  $x \# \Gamma$  and atom  $x \# v$  using Variable.prem $s(1)$  by (auto simp add: fresh-Pair)
  from fresh-delete[OF this(1)]
  have atom  $x \# delete v \Gamma$ .
  moreover
  have  $v \in domA \Gamma$  using Variable.hyps(1) by (metis domA-from-set map-of-SomeD)
  from fresh-map-of[OF this  $\langle atom x \# \Gamma \rangle$ ]
  have atom  $x \# the (map-of \Gamma v)$ .
  hence atom  $x \# e$  using  $\langle map-of \Gamma v = Some e \rangle$  by simp
  ultimately
  have atom  $x \# (\Delta, z) \vee x \in domA \Delta - set (v \# L)$  using Variable.hyps(3) by (auto simp
  add: fresh-Pair)
  thus ?case using  $\langle atom x \# v \rangle$  by (auto simp add: fresh-Pair fresh-Cons fresh-at-base)
next
```

```
case (Bool  $\Gamma b L$ )
  thus ?case by auto
next
```

```
case (IfThenElse  $\Gamma scrut L \Delta b e_1 e_2 \Theta z$ )
  from  $\langle atom x \# (\Gamma, scrut ? e_1 : e_2) \rangle$ 
  have atom  $x \# (\Gamma, scrut)$  and atom  $x \# (e_1, e_2)$  by (auto simp add: fresh-Pair)
  from IfThenElse.hyps(2)[OF this(1)]
  show ?case
  proof
    assume atom  $x \# (\Delta, Bool b)$  with  $\langle atom x \# (e_1, e_2) \rangle$ 
    have atom  $x \# (\Delta, if b then e_1 else e_2)$  by auto
    from IfThenElse.hyps(4)[OF this]
    show ?thesis.
  next
    assume  $x \in domA \Delta - set L$ 
    with reds-doesnt-forget[OF  $\langle \Delta : (if b then e_1 else e_2) \Downarrow_L \Theta : z \rangle$ ]
    show ?thesis by auto
  qed
next
```

```
case (Let  $\Delta \Gamma L body \Theta z$ )
  show ?case
  proof (cases  $x \in domA \Delta$ )
    case False
      hence atom  $x \# \Delta$  using Let.prem $s$  by (auto simp add: fresh-Pair)
      show ?thesis
        apply (rule Let.hyps(3))
        using Let.prem $s$   $\langle atom x \# \Delta \rangle$  False
        by (auto simp add: fresh-Pair fresh-append)
    next
      case True
```

```

hence  $x \notin \text{set } L$ 
using Let(1)
by (metis fresh-PairD(2) fresh-star-def image-eqI set-not-fresh)
with True
show ?thesis
using reds-doesnt-forget[OF Let.hyps(2)] by auto
qed
qed

```

```

lemma reds-fresh-fv:  $\llbracket \Gamma : e \Downarrow_L \Delta : z;$ 
 $x \in \text{fv}(\Delta, z) \wedge (x \notin \text{dom}A \Delta \vee x \in \text{set } L)$ 
 $\rrbracket \implies x \in \text{fv}(\Gamma, e)$ 
using reds-fresh
unfolding fv-def fresh-def
by blast

```

```

lemma new-free-vars-on-heap:
assumes  $\Gamma : e \Downarrow_L \Delta : z$ 
shows  $\text{fv}(\Delta, z) - \text{dom}A \Delta \subseteq \text{fv}(\Gamma, e) - \text{dom}A \Gamma$ 
using reds-fresh-fv[OF assms(1)] reds-doesnt-forget[OF assms(1)] by auto

```

```

lemma reds-pres-closed:
assumes  $\Gamma : e \Downarrow_L \Delta : z$ 
and  $\text{fv}(\Gamma, e) \subseteq \text{set } L \cup \text{dom}A \Gamma$ 
shows  $\text{fv}(\Delta, z) \subseteq \text{set } L \cup \text{dom}A \Delta$ 
using new-free-vars-on-heap[OF assms(1)] assms(2) by auto

```

Reducing the set of variables to avoid is always possible.

```

lemma reds-smaller-L:  $\llbracket \Gamma : e \Downarrow_L \Delta : z;$ 
 $\text{set } L' \subseteq \text{set } L$ 
 $\rrbracket \implies \Gamma : e \Downarrow_{L'} \Delta : z$ 
proof(nominal-induct avoiding : L' rule: reds.strong-induct)
case (Lambda  $\Gamma x e L L'$ )
show ?case
by (rule reds.Lambda)
next
case (Application  $y \Gamma e xa L \Delta \Theta z e' L'$ )
from Application.hyps(10)[OF Application.prem] Application.hyps(12)[OF Application.prem]
show ?case
by (rule reds-ApplicationI)
next
case (Variable  $\Gamma xa e L \Delta z L'$ )
have  $\text{set}(xa \# L') \subseteq \text{set}(xa \# L)$ 
using Variable.prem by auto
thus ?case
by (rule reds.Variable[OF Variable(1) Variable.hyps(3)])
next
case (Bool  $b$ )
show ?case..

```

```

next
case (IfThenElse  $\Gamma$  scrut  $L \Delta b e_1 e_2 \Theta z L'$ )
  thus ?case by (metis reds.IfThenElse)
next
case (Let  $\Delta \Gamma L$  body  $\Theta z L'$ )
  have atom ' domA  $\Delta \#*$  ( $\Gamma, L'$ )
  using Let(1-3) Let.premis
  by (auto simp add: fresh-star-Pair fresh-star-set-subset)
  thus ?case
  by (rule reds.Let[OF - Let.hyps(4)][OF Let.premis])
qed

```

Things are evaluated to a lambda expression, and the variable can be freely chose.

**lemma** *result-evaluated*:

```

 $\Gamma : e \Downarrow_L \Delta : z \implies isVal z$ 
by (induct  $\Gamma e L \Delta z$  rule:reds.induct) (auto dest: reds-doesnt-forget)

```

**lemma** *result-evaluated-fresh*:

```

assumes  $\Gamma : e \Downarrow_L \Delta : z$ 
obtains  $y e'$ 
where  $z = (Lam [y]. e')$  and  $atom y \# (c::'a::fs) \mid b$  where  $z = Bool b$ 
proof -
from assms
have isVal  $z$  by (rule result-evaluated)
hence  $(\exists y e'. z = Lam [y]. e' \wedge atom y \# c) \vee (\exists b. z = Bool b)$ 
by (nominal-induct  $z$  avoiding:  $c$  rule:exp-strong-induct) auto
thus thesis using that by blast
qed

```

end

## 4 Denotational domain

### 4.1 Value

```

theory Value
imports HOLCF
begin

```

#### 4.1.1 The semantic domain for values and environments

```

domain Value = Fn (lazy Value  $\rightarrow$  Value) | B (lazy bool discr)

```

```

fixrec Fn-project :: Value  $\rightarrow$  Value  $\rightarrow$  Value
where Fn-project.(Fn.f) = f

```

**abbreviation** *F<sub>n</sub>-project-abbr* (**infix**  $\langle \downarrow F_n \rangle$  55)  
**where**  $f \downarrow F_n v \equiv F_n\text{-project} \cdot f \cdot v$

**lemma** [*simp*]:  
 $\perp \downarrow F_n x = \perp$   
 $(B \cdot b) \downarrow F_n x = \perp$   
**by** (*fixrec-simp*)<sup>+</sup>

**fixrec** *B-project* :: *Value*  $\rightarrow$  *Value*  $\rightarrow$  *Value*  $\rightarrow$  *Value* **where**  
*B-project*  $\cdot (B \cdot db) \cdot v_1 \cdot v_2 = (\text{if } \text{undiscr } db \text{ then } v_1 \text{ else } v_2)$

**lemma** [*simp*]:  
*B-project*  $\cdot (B \cdot (\text{Discr } b)) \cdot v_1 \cdot v_2 = (\text{if } b \text{ then } v_1 \text{ else } v_2)$   
*B-project*  $\cdot \perp \cdot v_1 \cdot v_2 = \perp$   
*B-project*  $\cdot (F_n \cdot f) \cdot v_1 \cdot v_2 = \perp$   
**by** *fixrec-simp*<sup>+</sup>

A chain in the domain *Value* is either always bottom, or eventually *F<sub>n</sub>* of another chain

**lemma** *Value-chainE*[*consumes 1, case-names bot B F<sub>n</sub>*]:

**assumes** *chain Y*  
**obtains**  $Y = (\lambda \cdot \cdot \perp) \mid$   
 $n \ b \ \text{where } Y = (\lambda m. (\text{if } m < n \text{ then } \perp \text{ else } B \cdot b)) \mid$   
 $n \ Y' \ \text{where } Y = (\lambda m. (\text{if } m < n \text{ then } \perp \text{ else } F_n \cdot (Y' (m - n)))) \ \text{chain } Y'$   
**proof**(*cases Y = (\lambda \cdot \cdot \perp)*)  
**case** *True*  
**thus** *?thesis* **by** (*rule that(1)*)  
**next**  
**case** *False*  
**hence**  $\exists i. Y \ i \neq \perp$  **by** *auto*  
**hence**  $\exists n. Y \ n \neq \perp \wedge (\forall m. Y \ m \neq \perp \longrightarrow m \geq n)$   
**by** (*rule exE*)(*rule ex-has-least-nat*)  
**then obtain**  $n$  **where**  $Y \ n \neq \perp$  **and**  $\forall m. m < n \longrightarrow Y \ m = \perp$  **by** *fastforce*  
**hence**  $(\exists f. Y \ n = F_n \cdot f) \vee (\exists b. Y \ n = B \cdot b)$  **by** (*metis Value.exhaust*)  
**thus** *?thesis*  
**proof**  
**assume**  $(\exists f. Y \ n = F_n \cdot f)$   
**then obtain**  $f$  **where**  $Y \ n = F_n \cdot f$  **by** *blast*  
**{**  
**fix**  $i$   
**from**  $\langle \text{chain } Y \rangle$  **have**  $Y \ n \sqsubseteq Y \ (i + n)$  **by** (*metis chain-mono le-add2*)  
**with**  $\langle Y \ n = \cdot \rangle$   
**have**  $\exists g. (Y \ (i + n) = F_n \cdot g)$   
**by** (*metis Value.dist-les(1) Value.exhaust below-bottom-iff*)  
**}**  
**then obtain**  $Y'$  **where**  $Y': \bigwedge i. Y \ (i + n) = F_n \cdot (Y' \ i)$  **by** *metis*  
  
**have**  $Y = (\lambda m. \text{if } m < n \text{ then } \perp \text{ else } F_n \cdot (Y' (m - n)))$   
**using**  $\langle \forall m. \cdot \rightarrow Y' \ \text{by } (\text{metis add-diff-inverse add commute})$   
**moreover**

```

havechain  $Y'$  using  $\langle \text{chain } Y \rangle$ 
  by (auto intro! : chainI elim: chainE simp add: Value.inverts[symmetric] Y'[symmetric]
simp del: Value.inverts)
  ultimately
  show ?thesis by (rule that(3))
next
  assume  $(\exists b. Y\ n = B \cdot b)$ 
  then obtain  $b$  where  $Y\ n = B \cdot b$  by blast
  {
    fix  $i$ 
    from  $\langle \text{chain } Y \rangle$  have  $Y\ n \sqsubseteq Y\ (i+n)$  by (metis chain-mono le-add2)
    with  $\langle Y\ n = \cdot \rangle$ 
    have  $Y\ (i+n) = B \cdot b$ 
    by (metis Value.dist-les(2) Value.exhaust Value.inverts(2) below-bottom-iff discrete-cpo)
  }
  hence  $Y' : \bigwedge i. Y\ (i + n) = B \cdot b$  by metis

  have  $Y = (\lambda m. \text{if } m < n \text{ then } \perp \text{ else } B \cdot b)$ 
    using  $\langle \forall m. \rightarrow Y' \rangle$  by (metis add-diff-inverse add commute)
  thus ?thesis by (rule that(2))
qed
qed

end

```

## 4.2 Value-Nominal

```

theory Value-Nominal
imports Value Nominal-Utills Nominal-HOLCF
begin

```

Values are pure, i.e. contain no variables.

```

instantiation Value :: pure
begin
  definition  $p \cdot (v :: \text{Value}) = v$ 
instance
  apply standard
  apply (auto simp add: permute-Value-def)
  done
end

```

```

instance Value :: pcpo-pt
  by standard (simp add: pure-permute-id)

end

```

## 5 Denotational semantics

### 5.1 Iterative

```
theory Iterative
imports Env-HOLCF
begin
```

A setup for defining a fixed point of mutual recursive environments iteratively

```
locale iterative =
  fixes  $\rho :: 'a::type \Rightarrow 'b::pcpo$ 
  and  $e1 :: ('a \Rightarrow 'b) \rightarrow ('a \Rightarrow 'b)$ 
  and  $e2 :: ('a \Rightarrow 'b) \rightarrow 'b$ 
  and  $S :: 'a \text{ set}$  and  $x :: 'a$ 
  assumes  $ne: x \notin S$ 
begin
  abbreviation  $L == (\Lambda \rho'. (\rho \text{ ++}_S e1 \cdot \rho')(x := e2 \cdot \rho'))$ 
  abbreviation  $H == (\lambda \rho'. \Lambda \rho''. \rho' \text{ ++}_S e1 \cdot \rho'')$ 
  abbreviation  $R == (\Lambda \rho'. (\rho \text{ ++}_S (\text{fix} \cdot (H \rho')))(x := e2 \cdot \rho'))$ 
  abbreviation  $R' == (\Lambda \rho'. (\rho \text{ ++}_S (\text{fix} \cdot (H \rho')))(x := e2 \cdot (\text{fix} \cdot (H \rho'))))$ 
```

**lemma** *split-x*:

```
  fixes  $y$ 
  obtains  $y = x$  and  $y \notin S \mid y \in S$  and  $y \neq x \mid y \notin S$  and  $y \neq x$  using  $ne$  by blast
  lemmas below = fun-belowI[OF split-x, where  $y1 = \lambda x. x$ ]
  lemmas eq = ext[OF split-x, where  $y1 = \lambda x. x$ ]
```

**lemma** *lookup-fix[simp]*:

```
  fixes  $y$  and  $F :: ('a \Rightarrow 'b) \rightarrow ('a \Rightarrow 'b)$ 
  shows  $(\text{fix} \cdot F) y = (F \cdot (\text{fix} \cdot F)) y$ 
  by (subst fix-eq, rule)
```

**lemma** *R-S*:  $\bigwedge y. y \in S \Longrightarrow (\text{fix} \cdot R) y = (e1 \cdot (\text{fix} \cdot (H (\text{fix} \cdot R)))) y$   
 by (case-tac  $y$  rule: split-x) simp-all

**lemma** *R'-S*:  $\bigwedge y. y \in S \Longrightarrow (\text{fix} \cdot R') y = (e1 \cdot (\text{fix} \cdot (H (\text{fix} \cdot R')))) y$   
 by (case-tac  $y$  rule: split-x) simp-all

**lemma** *HR-is-R[simp]*:  $\text{fix} \cdot (H (\text{fix} \cdot R)) = \text{fix} \cdot R$   
 by (rule eq) simp-all

**lemma** *HR'-is-R'[simp]*:  $\text{fix} \cdot (H (\text{fix} \cdot R')) = \text{fix} \cdot R'$   
 by (rule eq) simp-all

**lemma** *H-noop*:

```
  fixes  $\rho' \rho''$ 
  assumes  $\bigwedge y. y \in S \Longrightarrow y \neq x \Longrightarrow (e1 \cdot \rho'') y \sqsubseteq \rho' y$ 
  shows  $H \rho' \cdot \rho'' \sqsubseteq \rho'$ 
  using assms
```

by  $-(rule\ below, simp\ all)$

**lemma** *HL-is-L[simp]*:  $fix \cdot (H (fix \cdot L)) = fix \cdot L$

**proof** (*rule below-antisym*)

show  $fix \cdot (H (fix \cdot L)) \sqsubseteq fix \cdot L$

by (*rule fix-least-below[OF H-noop]*) *simp*

hence \*:  $e2 \cdot (fix \cdot (H (fix \cdot L))) \sqsubseteq e2 \cdot (fix \cdot L)$  **by** (*rule monofun-cfun-arg*)

show  $fix \cdot L \sqsubseteq fix \cdot (H (fix \cdot L))$

by (*rule fix-least-below[OF below]*) (*simp-all add: ne \**)

qed

**lemma** *iterative-override-on*:

shows  $fix \cdot L = fix \cdot R$

**proof**(*rule below-antisym*)

show  $fix \cdot R \sqsubseteq fix \cdot L$

by (*rule fix-least-below[OF below]*) *simp-all*

show  $fix \cdot L \sqsubseteq fix \cdot R$

apply (*rule fix-least-below[OF below]*)

apply *simp*

apply (*simp del: lookup-fix add: R-S*)

apply *simp*

done

qed

**lemma** *iterative-override-on'*:

shows  $fix \cdot L = fix \cdot R'$

**proof**(*rule below-antisym*)

show  $fix \cdot R' \sqsubseteq fix \cdot L$

by (*rule fix-least-below[OF below]*) *simp-all*

show  $fix \cdot L \sqsubseteq fix \cdot R'$

apply (*rule fix-least-below[OF below]*)

apply *simp*

apply (*simp del: lookup-fix add: R'-S*)

apply *simp*

done

qed

end

end

## 5.2 HasESem

**theory** *HasESem*

**imports** *Nominal-HOLCF Env-HOLCF*

**begin**

A locale to work abstract in the expression type and semantics.

```

locale has-ESem =
  fixes ESem :: 'exp::pt  $\Rightarrow$  ('var::at-base  $\Rightarrow$  'value)  $\rightarrow$  'value::{pure,pcpo}
begin
  abbreviation ESem-syn ( $\langle \llbracket - \rrbracket \cdot \rangle$  [0,0] 110) where  $\llbracket e \rrbracket_{\varrho} \equiv ESem\ e \cdot \varrho$ 
end

```

```

locale has-ignore-fresh-ESem = has-ESem +
  assumes fv-supp: supp e = atom ' (fv e :: 'b set)
  assumes ESem-considers-fv:  $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho} f | ' (fv\ e)$ 

```

**end**

### 5.3 HeapSemantics

**theory** *HeapSemantics*

```

imports EvalHeap AList-Utills-Nominal HasESem Iterative Env-Nominal
begin

```

#### 5.3.1 A locale for heap semantics, abstract in the expression semantics

```

context has-ESem
begin

```

```

abbreviation EvalHeapSem-syn ( $\langle \llbracket - \rrbracket \cdot \rangle$  [0,0] 110)
  where EvalHeapSem-syn  $\Gamma\ \varrho \equiv evalHeap\ \Gamma\ (\lambda\ e.\ \llbracket e \rrbracket_{\varrho})$ 

```

**definition**

```

HSem :: ('var  $\times$  'exp) list  $\Rightarrow$  ('var  $\Rightarrow$  'value)  $\rightarrow$  ('var  $\Rightarrow$  'value)
  where HSem  $\Gamma = (\Lambda\ \varrho \cdot (\mu\ \varrho'.\ \varrho\ ++\ domA\ \Gamma\ \llbracket \Gamma \rrbracket_{\varrho'}))$ 

```

```

abbreviation HSem-syn ( $\langle \{\Gamma\} \cdot \rangle$  [0,60] 60)
  where  $\{\Gamma\}_{\varrho} \equiv HSem\ \Gamma \cdot \varrho$ 

```

```

lemma HSem-def':  $\{\Gamma\}_{\varrho} = (\mu\ \varrho'.\ \varrho\ ++\ domA\ \Gamma\ \llbracket \Gamma \rrbracket_{\varrho'})$ 
  unfolding HSem-def by simp

```

#### 5.3.2 Induction and other lemmas about *HSem*

**lemma** *HSem-ind*:

```

assumes adm P
assumes P  $\perp$ 
assumes step:  $\bigwedge\ \varrho'.\ P\ \varrho' \Longrightarrow P\ (\varrho\ ++\ domA\ \Gamma\ \llbracket \Gamma \rrbracket_{\varrho'})$ 
shows P ( $\{\Gamma\}_{\varrho}$ )
unfolding HSem-def'
apply (rule fix-ind[OF assms(1), OF assms(2)])
using step by simp

```

**lemma** *HSem-below*:

**assumes**  $\rho$ :  $\bigwedge x. x \notin \text{dom} A \ h \implies \varrho \ x \sqsubseteq r \ x$

**assumes**  $h$ :  $\bigwedge x. x \in \text{dom} A \ h \implies \llbracket \text{the } (\text{map-of } h \ x) \rrbracket_r \sqsubseteq r \ x$

**shows**  $\{\!\{h\}\!\}_\varrho \sqsubseteq r$

**proof** (*rule HSem-ind, goal-cases*)

**case 1 show** *?case* **by** (*auto*)

**next**

**case 2 show** *?case* **by** (*rule minimal*)

**next**

**case** (*3*  $\varrho'$ )

**show** *?case*

**by** (*rule override-on-belowI*)

(*auto simp add: lookupEvalHeap below-trans[OF monofun-cfun-arg[OF  $\langle \varrho' \sqsubseteq r \rangle$ ] h] rho*)

**qed**

**lemma** *HSem-bot-below*:

**assumes**  $h$ :  $\bigwedge x. x \in \text{dom} A \ h \implies \llbracket \text{the } (\text{map-of } h \ x) \rrbracket_r \sqsubseteq r \ x$

**shows**  $\{\!\{h\}\!\}_\perp \sqsubseteq r$

**using** *assms*

**by** (*metis HSem-below fun-belowD minimal*)

**lemma** *HSem-bot-ind*:

**assumes** *adm*  $P$

**assumes**  $P \perp$

**assumes** *step*:  $\bigwedge \varrho'. P \ \varrho' \implies P \ (\llbracket \Gamma \rrbracket_{\varrho'})$

**shows**  $P \ (\{\!\{\Gamma\}\!\}_\perp)$

**apply** (*rule HSem-ind[OF assms(1,2)]*)

**apply** (*drule assms(3)*)

**apply** *simp*

**done**

**lemma** *parallel-HSem-ind*:

**assumes** *adm*  $(\lambda \varrho'. P \ (\text{fst } \varrho') \ (\text{snd } \varrho'))$

**assumes**  $P \perp \perp$

**assumes** *step*:  $\bigwedge y \ z. P \ y \ z \implies$

$P \ (\varrho_1 \ ++_{\text{dom} A \ \Gamma_1} \llbracket \Gamma_1 \rrbracket y) \ (\varrho_2 \ ++_{\text{dom} A \ \Gamma_2} \llbracket \Gamma_2 \rrbracket z)$

**shows**  $P \ (\{\!\{\Gamma_1\}\!\}_{\varrho_1}) \ (\{\!\{\Gamma_2\}\!\}_{\varrho_2})$

**unfolding** *HSem-def'*

**apply** (*rule parallel-fix-ind[OF assms(1), OF assms(2)]*)

**using** *step* **by** *simp*

**lemma** *HSem-eq*:

**shows**  $\{\!\{\Gamma\}\!\}_\varrho = \varrho \ ++_{\text{dom} A \ \Gamma} \llbracket \Gamma \rrbracket_{\{\!\{\Gamma\}\!\}_\varrho}$

**unfolding** *HSem-def'*

**by** (*subst fix-eq*) *simp*

**lemma** *HSem-bot-eq*:

**shows**  $\{\!\{\Gamma\}\!\}_\perp = \llbracket \Gamma \rrbracket_{\{\!\{\Gamma\}\!\}_\perp}$

**by** (*subst HSem-eq*) *simp*

**lemma** *lookup-HSem-other*:

**assumes**  $y \notin \text{dom}A \ h$   
**shows**  $\{\!|h|\!\}_\varrho \ y = \varrho \ y$   
**apply** (*subst HSem-eq*)  
**using** *assms* **by** *simp*

**lemma** *lookup-HSem-heap*:

**assumes**  $y \in \text{dom}A \ h$   
**shows**  $\{\!|h|\!\}_\varrho \ y = \llbracket \text{the } (\text{map-of } h \ y) \rrbracket \{\!|h|\!\}_\varrho$   
**apply** (*subst HSem-eq*)  
**using** *assms* **by** (*simp add: lookupEvalHeap*)

**lemma** *HSem-edom-subset*:  $\text{edom } (\{\!\Gamma\!\}_\varrho) \subseteq \text{edom } \varrho \cup \text{dom}A \ \Gamma$

**apply** *rule*  
**unfolding** *edomIff*  
**apply** (*case-tac x \in \text{dom}A \ \Gamma*)  
**apply** (*auto simp add: lookup-HSem-other*)  
**done**

**lemma** (**in**  $-$ ) *env-restr-override-onI*:  $-S2 \subseteq S \implies \text{env-restr } S \ \varrho1 \ ++_{S2} \ \varrho2 = \varrho1 \ ++_{S2} \ \varrho2$   
**by** (*rule ext*) (*auto simp add: lookup-override-on-eq*)

**lemma** *HSem-restr*:

$\{\!|h|\!\}_\varrho \ f \!|' \ (- \ \text{dom}A \ h) = \{\!|h|\!\}_\varrho$   
**apply** (*rule parallel-HSem-ind*)  
**apply** *simp*  
**apply** *auto[1]*  
**apply** (*subst env-restr-override-onI*)  
**apply** *simp-all*  
**done**

**lemma** *HSem-restr-cong*:

**assumes**  $\varrho \ f \!|' \ (- \ \text{dom}A \ h) = \varrho' \ f \!|' \ (- \ \text{dom}A \ h)$   
**shows**  $\{\!|h|\!\}_\varrho = \{\!|h|\!\}_{\varrho'}$   
**apply** (*subst (1 2) HSem-restr[symmetric]*)  
**by** (*simp add: assms*)

**lemma** *HSem-restr-cong-below*:

**assumes**  $\varrho \ f \!|' \ (- \ \text{dom}A \ h) \sqsubseteq \varrho' \ f \!|' \ (- \ \text{dom}A \ h)$   
**shows**  $\{\!|h|\!\}_\varrho \sqsubseteq \{\!|h|\!\}_{\varrho'}$   
**by** (*subst (1 2) HSem-restr[symmetric]*) (*rule monofun-cfun-arg[OF assms]*)

**lemma** *HSem-reorder*:

**assumes**  $\text{map-of } \Gamma = \text{map-of } \Delta$   
**shows**  $\{\!\Gamma\!\}_\varrho = \{\!\Delta\!\}_\varrho$   
**by** (*simp add: HSem-def' evalHeap-reorder[OF assms] assms dom-map-of-conv-domA[symmetric]*)

**lemma** *HSem-reorder-head*:

**assumes**  $x \neq y$   
**shows**  $\{\!(x, e1)\#(y, e2)\#\Gamma\}\varrho = \{\!(y, e2)\#(x, e1)\#\Gamma\}\varrho$   
**proof**–  
**have**  $set((x, e1)\#(y, e2)\#\Gamma) = set((y, e2)\#(x, e1)\#\Gamma)$   
**by** *auto*  
**thus** *?thesis*  
**unfolding** *HSem-def evalHeap-reorder-head[OF assms]*  
**by** (*simp add: domA-def*)  
**qed**

**lemma** *HSem-reorder-head-append*:  
**assumes**  $x \notin domA \Gamma$   
**shows**  $\{\!(x, e)\#\Gamma @ \Delta\}\varrho = \{\!\Gamma @ ((x, e)\#\Delta)\}\varrho$   
**proof**–  
**have**  $set((x, e)\#\Gamma @ \Delta) = set(\Gamma @ ((x, e)\#\Delta))$  **by** *auto*  
**thus** *?thesis*  
**unfolding** *HSem-def evalHeap-reorder-head-append[OF assms]*  
**by** *simp*  
**qed**

**lemma** *env-restr-HSem*:  
**assumes**  $domA \Gamma \cap S = \{\}$   
**shows**  $(\{\!\Gamma\}\varrho) f |' S = \varrho f |' S$   
**proof** (*rule env-restr-eqI*)  
**fix**  $x$   
**assume**  $x \in S$   
**hence**  $x \notin domA \Gamma$  **using** *assms* **by** *auto*  
**thus**  $(\{\!\Gamma\}\varrho) x = \varrho x$   
**by** (*rule lookup-HSem-other*)  
**qed**

**lemma** *env-restr-HSem-noop*:  
**assumes**  $domA \Gamma \cap edom \varrho = \{\}$   
**shows**  $(\{\!\Gamma\}\varrho) f |' edom \varrho = \varrho$   
**by** (*simp add: env-restr-HSem[OF assms] env-restr-useless*)

**lemma** *HSem-Nil[simp]*:  $\{\!\{\}\}\varrho = \varrho$   
**by** (*subst HSem-eq, simp*)

### 5.3.3 Substitution

**lemma** *HSem-subst-exp*:  
**assumes**  $\bigwedge \varrho'. \llbracket e \rrbracket_{\varrho'} = \llbracket e' \rrbracket_{\varrho'}$   
**shows**  $\{\!(x, e)\#\Gamma\}\varrho = \{\!(x, e')\#\Gamma\}\varrho$   
**by** (*rule parallel-HSem-ind*) (*auto simp add: assms evalHeap-subst-exp*)

**lemma** *HSem-subst-expr-below*:  
**assumes** *below*:  $\llbracket e1 \rrbracket \{\!(x, e2)\#\Gamma\}\varrho \sqsubseteq \llbracket e2 \rrbracket \{\!(x, e2)\#\Gamma\}\varrho$   
**shows**  $\{\!(x, e1)\#\Gamma\}\varrho \sqsubseteq \{\!(x, e2)\#\Gamma\}\varrho$

by (rule *HSem-below*) (auto simp add: lookup-*HSem-heap* below lookup-*HSem-other*)

**lemma** *HSem-subst-expr*:

**assumes** *below1*:  $\llbracket e1 \rrbracket_{\{\!(x, e2) \#\ \Gamma\}}_{\varrho} \sqsubseteq \llbracket e2 \rrbracket_{\{\!(x, e2) \#\ \Gamma\}}_{\varrho}$

**assumes** *below2*:  $\llbracket e2 \rrbracket_{\{\!(x, e1) \#\ \Gamma\}}_{\varrho} \sqsubseteq \llbracket e1 \rrbracket_{\{\!(x, e1) \#\ \Gamma\}}_{\varrho}$

**shows**  $\{\!(x, e1) \#\ \Gamma\}_{\varrho} = \{\!(x, e2) \#\ \Gamma\}_{\varrho}$

**by** (*metis* *assms HSem-subst-expr-below below-antisym*)

### 5.3.4 Re-calculating the semantics of the heap is idempotent

**lemma** *HSem-redo*:

**shows**  $\{\!\Gamma\}\{\!\Gamma @ \Delta\}_{\varrho} f \mid' (edom \varrho \cup domA \Delta) = \{\!\Gamma @ \Delta\}_{\varrho}$  (is ?*LHS* = ?*RHS*)

**proof** (*rule below-antisym*)

**show** ?*LHS*  $\sqsubseteq$  ?*RHS*

**by** (*rule HSem-below*)

(*auto simp add: lookup-HSem-heap fun-belowD[OF env-restr-below-itself]*)

**show** ?*RHS*  $\sqsubseteq$  ?*LHS*

**proof**(*rule HSem-below, goal-cases*)

**case** (1 *x*)

**thus** ?*case*

**by** (*cases x*  $\notin$  *edom*  $\varrho$ ) (*auto simp add: lookup-HSem-other dest:lookup-not-edom*)

**next**

**case** *prems*: (2 *x*)

**thus** ?*case*

**proof**(*cases x*  $\in$  *domA*  $\Gamma$ )

**case** *True*

**thus** ?*thesis* **by** (*auto simp add: lookup-HSem-heap*)

**next**

**case** *False*

**hence** *delta*:  $x \in domA \Delta$  **using** *prems* **by** *auto*

**with** *False*  $\langle ?LHS \sqsubseteq ?RHS \rangle$

**show** ?*thesis* **by** (*auto simp add: lookup-HSem-other lookup-HSem-heap monofun-cfun-arg*)

**qed**

**qed**

**qed**

### 5.3.5 Iterative definition of the heap semantics

**lemma** *iterative-HSem*:

**assumes**  $x \notin domA \Gamma$

**shows**  $\{\!(x, e) \#\ \Gamma\}_{\varrho} = (\mu \varrho'. (\varrho ++_{domA \Gamma} (\{\!\Gamma\}\varrho')))(x := \llbracket e \rrbracket_{\varrho'})$

**proof**–

**from** *assms*

**interpret** *iterative*

**where**  $e1 = \Lambda \varrho'. \llbracket \Gamma \rrbracket_{\varrho'}$

**and**  $e2 = \Lambda \varrho'. \llbracket e \rrbracket_{\varrho'}$

**and**  $S = domA \Gamma$

**and**  $x = x$  **by** *unfold-locales*

**have**  $\llbracket (x, e) \# \Gamma \rrbracket_{\varrho} = \text{fix} \cdot L$   
**by** (*simp add: HSem-def' override-on-upd ne*)  
**also have**  $\dots = \text{fix} \cdot R$   
**by** (*rule iterative-override-on*)  
**also have**  $\dots = (\mu \varrho'. (\varrho \text{ ++ }_{\text{dom}A \Gamma} (\llbracket \Gamma \rrbracket_{\varrho'})) (x := \llbracket e \rrbracket_{\varrho'}))$   
**by** (*simp add: HSem-def'*)  
**finally show** *?thesis.*  
**qed**

**lemma** *iterative-HSem'*:  
**assumes**  $x \notin \text{dom}A \Gamma$   
**shows**  $(\mu \varrho'. (\varrho \text{ ++ }_{\text{dom}A \Gamma} (\llbracket \Gamma \rrbracket_{\varrho'})) (x := \llbracket e \rrbracket_{\varrho'}))$   
 $= (\mu \varrho'. (\varrho \text{ ++ }_{\text{dom}A \Gamma} (\llbracket \Gamma \rrbracket_{\varrho'})) (x := \llbracket e \rrbracket_{\llbracket \Gamma \rrbracket_{\varrho'}}))$

**proof**–  
**from** *assms*  
**interpret** *iterative*  
**where**  $e1 = \Lambda \varrho'. \llbracket \Gamma \rrbracket_{\varrho'}$   
**and**  $e2 = \Lambda \varrho'. \llbracket e \rrbracket_{\varrho'}$   
**and**  $S = \text{dom}A \Gamma$   
**and**  $x = x$  **by** *unfold-locales*

**have**  $(\mu \varrho'. (\varrho \text{ ++ }_{\text{dom}A \Gamma} (\llbracket \Gamma \rrbracket_{\varrho'})) (x := \llbracket e \rrbracket_{\varrho'})) = \text{fix} \cdot R$   
**by** (*simp add: HSem-def'*)  
**also have**  $\dots = \text{fix} \cdot L$   
**by** (*rule iterative-override-on[symmetric]*)  
**also have**  $\dots = \text{fix} \cdot R'$   
**by** (*rule iterative-override-on'*)  
**also have**  $\dots = (\mu \varrho'. (\varrho \text{ ++ }_{\text{dom}A \Gamma} (\llbracket \Gamma \rrbracket_{\varrho'})) (x := \llbracket e \rrbracket_{\llbracket \Gamma \rrbracket_{\varrho'}}))$   
**by** (*simp add: HSem-def'*)  
**finally**  
**show** *?thesis.*

**qed**

### 5.3.6 Fresh variables on the heap are irrelevant

**lemma** *HSem-ignores-fresh-restr'*:  
**assumes**  $\text{fv } \Gamma \subseteq S$   
**assumes**  $\bigwedge x \varrho. x \in \text{dom}A \Gamma \implies \llbracket \text{the } (\text{map-of } \Gamma x) \rrbracket_{\varrho} = \llbracket \text{the } (\text{map-of } \Gamma x) \rrbracket_{\varrho} f|' (\text{fv } (\text{the } (\text{map-of } \Gamma x)))$   
**shows**  $(\llbracket \Gamma \rrbracket_{\varrho}) f|' S = \llbracket \Gamma \rrbracket_{\varrho} f|' S$   
**proof**(*induction rule: parallel-HSem-ind[case-names adm base step]*)  
**case** *adm* **thus** *?case* **by** *simp*  
**next**  
**case** *base*  
**show** *?case* **by** *simp*  
**next**  
**case** (*step*  $y z$ )  
**have**  $\llbracket \Gamma \rrbracket_y = \llbracket \Gamma \rrbracket_z$

```

proof(rule evalHeap-cong^)
  fix x
  assume  $x \in \text{dom}A \Gamma$ 
  hence  $\text{fv } (the \ (map-of \ \Gamma \ x)) \subseteq \text{fv } \Gamma$  by (rule map-of-fv-subset)
  with assms(1)
  have  $\text{fv } (the \ (map-of \ \Gamma \ x)) \cap S = \text{fv } (the \ (map-of \ \Gamma \ x))$  by auto
  with step
  have  $y \ f|' \ \text{fv } (the \ (map-of \ \Gamma \ x)) = z \ f|' \ \text{fv } (the \ (map-of \ \Gamma \ x))$  by auto
  with  $\langle x \in \text{dom}A \ \Gamma \rangle$ 
  show  $\llbracket the \ (map-of \ \Gamma \ x) \rrbracket_y = \llbracket the \ (map-of \ \Gamma \ x) \rrbracket_z$ 
    by (subst (1 2) assms(2)[OF  $\langle x \in \text{dom}A \ \Gamma \rangle$ ]) simp
qed
moreover
have  $\text{dom}A \ \Gamma \subseteq S$  using domA-fv-subset assms(1) by auto
ultimately
show ?case by (simp add: env-restr-add env-restr-evalHeap-noop)
qed
end

```

### 5.3.7 Freshness

**context** *has-ignore-fresh-ESem* **begin**

**lemma** *ESem-fresh-cong*:

**assumes**  $\varrho \ f|' \ (fv \ e) = \varrho' \ f|' \ (fv \ e)$

**shows**  $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho'}$

**by** (*metis* *assms* *ESem-considers-fv*)

**lemma** *ESem-fresh-cong-subset*:

**assumes**  $\text{fv } e \subseteq S$

**assumes**  $\varrho \ f|' \ S = \varrho' \ f|' \ S$

**shows**  $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho'}$

**by** (*rule* *ESem-fresh-cong*[*OF* *env-restr-eq-subset*[*OF* *assms*]])

**lemma** *ESem-fresh-cong-below*:

**assumes**  $\varrho \ f|' \ (fv \ e) \sqsubseteq \varrho' \ f|' \ (fv \ e)$

**shows**  $\llbracket e \rrbracket_{\varrho} \sqsubseteq \llbracket e \rrbracket_{\varrho'}$

**by** (*metis* *assms* *ESem-considers-fv* *monofun-cfun-arg*)

**lemma** *ESem-fresh-cong-below-subset*:

**assumes**  $\text{fv } e \subseteq S$

**assumes**  $\varrho \ f|' \ S \sqsubseteq \varrho' \ f|' \ S$

**shows**  $\llbracket e \rrbracket_{\varrho} \sqsubseteq \llbracket e \rrbracket_{\varrho'}$

**by** (*rule* *ESem-fresh-cong-below*[*OF* *env-restr-below-subset*[*OF* *assms*]])

**lemma** *ESem-ignores-fresh-restr*:

**assumes**  $\text{atom } 'S \ \sharp^* \ e$

**shows**  $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho \ f|' \ (- \ S)}$

**proof**–

**have**  $fv\ e \cap -\ S = fv\ e$  **using** *assms* **by** (*auto simp add: fresh-def fresh-star-def fv-supp*)  
**thus** *?thesis* **by** (*subst (1 2) ESem-considers-fv*) *simp*  
**qed**

**lemma** *ESem-ignores-fresh-restr'*:  
**assumes**  $atom\ 'e\ (edom\ \varrho - S) \#* e$   
**shows**  $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho} f|' S$

**proof** –

**have**  $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho} f|' (- (edom\ \varrho - S))$   
**by** (*rule ESem-ignores-fresh-restr'[OF assms]*)  
**also have**  $\varrho f|' (- (edom\ \varrho - S)) = \varrho f|' S$   
**by** (*rule ext*) (*auto simp add: lookup-env-restr-eq dest: lookup-not-edom*)  
**finally show** *?thesis*.

**qed**

**lemma** *HSem-ignores-fresh-restr''*:

**assumes**  $fv\ \Gamma \subseteq S$   
**shows**  $(\{\Gamma\}_{\varrho}) f|' S = \{\Gamma\}_{\varrho} f|' S$

**by** (*rule HSem-ignores-fresh-restr''[OF assms(1) ESem-considers-fv]*)

**lemma** *HSem-ignores-fresh-restr*:

**assumes**  $atom\ 'S\ \#* \Gamma$   
**shows**  $(\{\Gamma\}_{\varrho}) f|' (- S) = \{\Gamma\}_{\varrho} f|' (- S)$

**proof** –

**from** *assms* **have**  $fv\ \Gamma \subseteq - S$  **by** (*auto simp add: fv-def fresh-star-def fresh-def*)  
**thus** *?thesis* **by** (*rule HSem-ignores-fresh-restr''*)

**qed**

**lemma** *HSem-fresh-cong-below*:

**assumes**  $\varrho f|' ((S \cup fv\ \Gamma) - domA\ \Gamma) \sqsubseteq \varrho' f|' ((S \cup fv\ \Gamma) - domA\ \Gamma)$   
**shows**  $(\{\Gamma\}_{\varrho}) f|' S \sqsubseteq (\{\Gamma\}_{\varrho'}) f|' S$

**proof** –

**from** *assms*  
**have**  $\{\Gamma\}_{\varrho} f|' (S \cup fv\ \Gamma) \sqsubseteq \{\Gamma\}_{\varrho'} f|' (S \cup fv\ \Gamma)$   
**by** (*auto intro: HSem-restr-cong-below simp add: Diff-eq inf-commute*)  
**hence**  $(\{\Gamma\}_{\varrho}) f|' (S \cup fv\ \Gamma) \sqsubseteq (\{\Gamma\}_{\varrho'}) f|' (S \cup fv\ \Gamma)$   
**by** (*subst (1 2) HSem-ignores-fresh-restr''*) *simp-all*  
**thus** *?thesis*  
**by** (*rule env-restr-below-subset[OF Un-upper1]*)

**qed**

**lemma** *HSem-fresh-cong*:

**assumes**  $\varrho f|' ((S \cup fv\ \Gamma) - domA\ \Gamma) = \varrho' f|' ((S \cup fv\ \Gamma) - domA\ \Gamma)$   
**shows**  $(\{\Gamma\}_{\varrho}) f|' S = (\{\Gamma\}_{\varrho'}) f|' S$

**apply** (*rule below-antisym*)

**apply** (*rule HSem-fresh-cong-below[OF eq-imp-below[OF assms]]*)

**apply** (*rule HSem-fresh-cong-below[OF eq-imp-below[OF assms[symmetric]]]*)

**done**

### 5.3.8 Adding a fresh variable to a heap does not affect its semantics

**lemma** *HSem-add-fresh'*:

**assumes** *fresh*:  $\text{atom } x \# \Gamma$

**assumes**  $x \notin \text{edom } \varrho$

**assumes** *step*:  $\bigwedge e \varrho'. e \in \text{snd } \langle \text{set } \Gamma \rangle \implies \llbracket e \rrbracket_{\varrho'} = \llbracket e \rrbracket_{\text{env-delete } x \varrho'}$

**shows**  $\text{env-delete } x (\llbracket (x, e) \# \Gamma \rrbracket \varrho) = \llbracket \Gamma \rrbracket \varrho$

**proof** (*rule parallel-HSem-ind, goal-cases*)

**case 1 show ?case by simp**

**next**

**case 2 show ?case by auto**

**next**

**case prems**:  $(\exists y z)$

**have**  $\text{env-delete } x \varrho = \varrho$  **using**  $\langle x \notin \text{edom } \varrho \rangle$  **by** (*rule env-delete-noop*)

**moreover**

**from** *fresh* **have**  $x \notin \text{domA } \Gamma$  **by** (*metis domA-not-fresh*)

**hence**  $\text{env-delete } x (\llbracket (x, e) \# \Gamma \rrbracket y) = \llbracket \Gamma \rrbracket y$

**by** (*auto intro: env-delete-noop dest: subsetD[OF edom-evalHeap-subset]*)

**moreover**

**have**  $\dots = \llbracket \Gamma \rrbracket z$

**apply** (*rule evalHeap-cong[OF refl]*)

**apply** (*subst (1) step, assumption*)

**using** *prems(1)* **apply** *auto*

**done**

**ultimately**

**show** *?case* **using**  $\langle x \notin \text{domA } \Gamma \rangle$

**by** (*simp add: env-delete-add*)

**qed**

**lemma** *HSem-add-fresh*:

**assumes**  $\text{atom } x \# \Gamma$

**assumes**  $x \notin \text{edom } \varrho$

**shows**  $\text{env-delete } x (\llbracket (x, e) \# \Gamma \rrbracket \varrho) = \llbracket \Gamma \rrbracket \varrho$

**proof**(*rule HSem-add-fresh'[OF assms], goal-cases*)

**case**  $(1 e \varrho')$

**assume**  $e \in \text{snd } \langle \text{set } \Gamma \rangle$

**hence**  $\text{atom } x \# e$  **by** (*metis assms(1) fresh-heap-expr'*)

**hence**  $x \notin \text{fv } e$  **by** (*simp add: fv-def fresh-def*)

**thus** *?case*

**by** (*rule ESem-fresh-cong[OF env-restr-env-delete-other[symmetric]]*)

**qed**

### 5.3.9 Mutual recursion with fresh variables

**lemma** *HSem-subset-below*:

**assumes** *fresh*:  $\text{atom } \langle \text{domA } \Gamma \# * \Delta \rangle$

**shows**  $\llbracket \Delta \rrbracket (\varrho f | \langle - \text{domA } \Gamma \rangle) \sqsubseteq (\llbracket \Delta @ \Gamma \rrbracket \varrho) f | \langle - \text{domA } \Gamma \rangle$

**proof**(*rule HSem-below*)

**fix**  $x$

**assume** [*simp*]:  $x \in \text{domA } \Delta$

**with** *assms* **have** \*: *atom* ‘ *domA*  $\Gamma$   $\sharp$ \* *the* (*map-of*  $\Delta$   $x$ ) **by** (*metis* *fresh-star-map-of*)  
**hence** [*simp*]:  $x \notin \text{domA } \Gamma$  **using** *fresh* ‘ $x \in \text{domA } \Delta$ ’ **by** (*metis* *fresh-star-def* *domA-not-fresh* *image-eqI*)  
**show**  $\llbracket \text{the } (\text{map-of } \Delta \ x) \rrbracket (\{\Delta @ \Gamma\}_\rho) f | ‘ (- \text{domA } \Gamma) \sqsubseteq ((\{\Delta @ \Gamma\}_\rho) f | ‘ (- \text{domA } \Gamma)) \ x$   
**by** (*simp* *add: lookup-HSem-heap ESem-ignores-fresh-restr*[*OF* \*, *symmetric*])  
**qed** (*simp* *add: lookup-HSem-other lookup-env-restr-eq*)

In the following lemma we show that the semantics of fresh variables can be calculated together with the presently bound variables, or separately.

**lemma** *HSem-merge*:

**assumes** *fresh*: *atom* ‘ *domA*  $\Gamma$   $\sharp$ \*  $\Delta$   
**shows**  $\{\Gamma\}\{\Delta\}_\rho = \{\Gamma @ \Delta\}_\rho$   
**proof**(*rule* *below-antisym*)  
**have** *map-of-eq*: *map-of* ( $\Delta @ \Gamma$ ) = *map-of* ( $\Gamma @ \Delta$ )  
**proof**  
**fix**  $x$   
**show** *map-of* ( $\Delta @ \Gamma$ )  $x$  = *map-of* ( $\Gamma @ \Delta$ )  $x$   
**proof** (*cases*  $x \in \text{domA } \Gamma$ )  
**case** *True*  
**hence**  $x \notin \text{domA } \Delta$  **by** (*metis* *fresh-distinct* *fresh IntI* *equals0D*)  
**thus** *map-of* ( $\Delta @ \Gamma$ )  $x$  = *map-of* ( $\Gamma @ \Delta$ )  $x$   
**by** (*simp* *add: map-add-dom-app-simps dom-map-of-conv-domA*)  
**next**  
**case** *False*  
**thus** *map-of* ( $\Delta @ \Gamma$ )  $x$  = *map-of* ( $\Gamma @ \Delta$ )  $x$   
**by** (*simp* *add: map-add-dom-app-simps dom-map-of-conv-domA*)  
**qed**  
**qed**

**show**  $\{\Gamma\}\{\Delta\}_\rho \sqsubseteq \{\Gamma @ \Delta\}_\rho$

**proof**(*rule* *HSem-below*)

**fix**  $x$   
**assume** [*simp*]:  $x \notin \text{domA } \Gamma$   
  
**have**  $(\{\Delta\}_\rho) \ x = ((\{\Delta\}_\rho) f | ‘ (- \text{domA } \Gamma)) \ x$  **by** *simp*  
**also** **have**  $\dots = (\{\Delta\}_\rho) f | ‘ (- \text{domA } \Gamma)) \ x$   
**by** (*rule* *arg-cong*[*OF* *HSem-ignores-fresh-restr*[*OF* *fresh*]])  
**also** **have**  $\dots \sqsubseteq ((\{\Delta @ \Gamma\}_\rho) f | ‘ (- \text{domA } \Gamma)) \ x$   
**by** (*rule* *fun-belowD*[*OF* *HSem-subset-below*[*OF* *fresh*]])  
**also** **have**  $\dots = (\{\Delta @ \Gamma\}_\rho) \ x$  **by** *simp*  
**also** **have**  $\dots = (\{\Gamma @ \Delta\}_\rho) \ x$  **by** (*rule* *arg-cong*[*OF* *HSem-reorder*[*OF* *map-of-eq*]])  
**finally**  
**show**  $(\{\Delta\}_\rho) \ x \sqsubseteq (\{\Gamma @ \Delta\}_\rho) \ x$ .  
**qed** (*auto* *simp* *add: lookup-HSem-heap lookup-env-restr-eq*)

**have** \*:  $\bigwedge x. x \in \text{domA } \Delta \implies x \notin \text{domA } \Gamma$   
**using** *fresh* **by** (*auto* *simp* *add: fresh-Pair* *fresh-star-def* *domA-not-fresh*)

**have** *foo*:  $\text{edom } \rho \cup \text{domA } \Delta \cup \text{domA } \Gamma - (\text{edom } \rho \cup \text{domA } \Delta \cup \text{domA } \Gamma) \cap - \text{domA } \Gamma =$

```

domA  $\Gamma$  by auto
have foo2:(edom  $\varrho \cup \text{domA } \Delta - (\text{edom } \varrho \cup \text{domA } \Delta) \cap - \text{domA } \Gamma) \subseteq \text{domA } \Gamma$  by auto

{ fix x
  assume  $x \in \text{domA } \Delta$ 
  hence *: atom ' domA  $\Gamma$  #* the (map-of  $\Delta$  x)
    by (rule fresh-star-map-of[OF - fresh])

  have [ the (map-of  $\Delta$  x) ]_{\{\Gamma\}\{\Delta\}\varrho} = [ the (map-of  $\Delta$  x) ]_{(\{\Gamma\}\{\Delta\}\varrho) f|' (- \text{domA } \Gamma)}
    by (rule ESem-ignores-fresh-restr[OF *])
  also have (\{\Gamma\}\{\Delta\}\varrho) f|' (- \text{domA } \Gamma) = (\{\Delta\}\varrho) f|' (- \text{domA } \Gamma)
    by (simp add: env-restr-HSem)
  also have [ the (map-of  $\Delta$  x) ]_{\dots} = [ the (map-of  $\Delta$  x) ]_{\{\Delta\}\varrho}
    by (rule ESem-ignores-fresh-restr[symmetric, OF *])
  finally
  have [ the (map-of  $\Delta$  x) ]_{\{\Gamma\}\{\Delta\}\varrho} = [ the (map-of  $\Delta$  x) ]_{\{\Delta\}\varrho}.
}
}
thus \{\Gamma@\Delta\}\varrho \sqsubseteq \{\Gamma\}\{\Delta\}\varrho
  by -(rule HSem-below, auto simp add: lookup-HSem-other lookup-HSem-heap *)
qed
end

```

### 5.3.10 Parallel induction

```

lemma parallel-HSem-ind-different-ESem:
  assumes adm ( $\lambda \varrho'. P$  (fst  $\varrho'$ ) (snd  $\varrho'$ ))
  assumes  $P \perp \perp$ 
  assumes  $\bigwedge y z. P$   $y$   $z \implies P$  ( $\varrho ++_{\text{domA } h}$  evalHeap  $h$  ( $\lambda e. \text{ESem1 } e \cdot y$ )) ( $\varrho' ++_{\text{domA } h}$  evalHeap  $h$  ( $\lambda e. \text{ESem2 } e \cdot z$ ))
  shows  $P$  (has-ESem.HSem ESem1  $h \cdot \varrho$ ) (has-ESem.HSem ESem2  $h$   $\cdot \varrho'$ )
proof -
  interpret HSem1: has-ESem ESem1.
  interpret HSem2: has-ESem ESem2.

  show ?thesis
    unfolding HSem1.HSem-def' HSem2.HSem-def'
    apply (rule parallel-fix-ind[OF assms(1)])
    apply (rule assms(2))
    apply simp
    apply (erule assms(3))
    done
qed

```

### 5.3.11 Congruence rule

```

lemma HSem-cong[fundef-cong]:
  [ ( $\bigwedge e. e \in \text{snd}$  ' set heap2  $\implies \text{ESem1 } e = \text{ESem2 } e$ ); heap1 = heap2 ]
   $\implies \text{has-ESem.HSem ESem1 heap1} = \text{has-ESem.HSem ESem2 heap2}$ 
  unfolding has-ESem.HSem-def

```

by (auto cong:evalHeap-cong)

### 5.3.12 Equivariance of the heap semantics

**lemma** *HSem-eqvt[eqvt]*:

$\pi \cdot \text{has-ESem.HSem ESem } \Gamma = \text{has-ESem.HSem } (\pi \cdot \text{ESem}) (\pi \cdot \Gamma)$

**proof** –

**show** *?thesis*

**unfolding** *has-ESem.HSem-def*

**apply** (*subst permute-Lam, simp*)

**apply** (*subst eqvt-lambda*)

**apply** (*simp add: Cfun-app-eqvt permute-Lam*)

**done**

**qed**

**end**

## 5.4 AbstractDenotational

**theory** *AbstractDenotational*

**imports** *HeapSemantics Terms*

**begin**

### 5.4.1 The denotational semantics for expressions

Because we need to define two semantics later on, we are abstract in the actual domain.

**locale** *semantic-domain* =

**fixes** *Fn* :: ('Value → 'Value) → ('Value::{pcpo-pt,pure})

**fixes** *Fn-project* :: 'Value → ('Value → 'Value)

**fixes** *B* :: bool *discr* → 'Value

**fixes** *B-project* :: 'Value → 'Value → 'Value → 'Value

**fixes** *tick* :: 'Value → 'Value

**begin**

**nominal-function**

*ESem* :: *exp* ⇒ (*var* ⇒ 'Value) → 'Value

**where**

*ESem* (*Lam* [*x*]. *e*) = (Λ *ρ*. *tick*·(*Fn*·(Λ *v*. *ESem* *e*·((*ρ* *f*|<sup>‘</sup> *fv* (*Lam* [*x*]. *e*))(*x* := *v*))))))

| *ESem* (*App* *e* *x*) = (Λ *ρ*. *tick*·(*Fn-project*·(*ESem* *e*·*ρ*)·(*ρ* *x*)))

| *ESem* (*Var* *x*) = (Λ *ρ*. *tick*·(*ρ* *x*))

| *ESem* (*Let* *as* *body*) = (Λ *ρ*. *tick*·(*ESem* *body*·(*has-ESem.HSem* *ESem* *as*·(*ρ* *f*|<sup>‘</sup> *fv* (*Let* *as* *body*))))))

| *ESem* (*Bool* *b*) = (Λ *ρ*. *tick*·(*B*·(*Discr* *b*)))

| *ESem* (*scrut* ? *e1* : *e2*) = (Λ *ρ*. *tick*·((*B-project*·(*ESem* *scrut*·*ρ*)·(*ESem* *e1*·*ρ*)·(*ESem* *e2*·*ρ*)))

**proof** *goal-cases*

The following proofs discharge technical obligations generated by the Nominal package.

**case 1 thus** *?case*

```

unfolding eqvt-def ESem-graph-aux-def
apply rule
apply (perm-simp)
apply (simp add: Abs-cfun-eqvt)
apply (simp add: unpermute-def permute-pure)
done
next
case ( $\beta P x$ )
  thus ?case by (metis (poly-guards-query) exp-strong-exhaust)
next

case prems: ( $\lambda x e x' e'$ )
  from prems(5)
  show ?case
  proof (rule eqvt-lam-case)
    fix  $\pi :: \text{perm}$ 
    assume *:  $\text{supp } (-\pi) \#* (\text{fv } (\text{Lam } [x]. e) :: \text{var set})$ 
    { fix  $\varrho v$ 
      have  $\text{ESem-sumC } (\pi \cdot e) \cdot ((\varrho f |' \text{fv } (\text{Lam } [x]. e)) (\pi \cdot x := v)) = - \pi \cdot \text{ESem-sumC } (\pi \cdot e) \cdot ((\varrho f |' \text{fv } (\text{Lam } [x]. e)) (\pi \cdot x := v))$ 
      by (simp add: permute-pure)
      also have  $\dots = \text{ESem-sumC } e \cdot ((-\pi \cdot (\varrho f |' \text{fv } (\text{Lam } [x]. e))) (x := v))$  by (simp add: pemute-minus-self eqvt-at-apply[OF prems(1)])
      also have  $-\pi \cdot (\varrho f |' \text{fv } (\text{Lam } [x]. e)) = (\varrho f |' \text{fv } (\text{Lam } [x]. e))$  by (rule env-restr-perm'[OF *]) auto
      finally have  $\text{ESem-sumC } (\pi \cdot e) \cdot ((\varrho f |' \text{fv } (\text{Lam } [x]. e)) (\pi \cdot x := v)) = \text{ESem-sumC } e \cdot ((\varrho f |' \text{fv } (\text{Lam } [x]. e)) (x := v))$ .
    }
    thus  $(\Lambda \varrho. \text{tick} \cdot (\text{Fn} \cdot (\Lambda v. \text{ESem-sumC } (\pi \cdot e) \cdot ((\varrho f |' \text{fv } (\text{Lam } [x]. e)) (\pi \cdot x := v)))) = (\Lambda \varrho. \text{tick} \cdot (\text{Fn} \cdot (\Lambda v. \text{ESem-sumC } e \cdot ((\varrho f |' \text{fv } (\text{Lam } [x]. e)) (x := v))))$  by simp
  qed
next

case prems: ( $\lambda as \text{body } as' \text{body}'$ )
  from prems(9)
  show ?case
  proof (rule eqvt-let-case)
    fix  $\pi :: \text{perm}$ 
    assume *:  $\text{supp } (-\pi) \#* (\text{fv } (\text{Terms.Let } as \text{body}) :: \text{var set})$ 

    { fix  $\varrho$ 
      have  $\text{ESem-sumC } (\pi \cdot \text{body}) \cdot (\text{has-ESem.HSem } \text{ESem-sumC } (\pi \cdot as) \cdot (\varrho f |' \text{fv } (\text{Terms.Let } as \text{body})))$ 
       $= - \pi \cdot \text{ESem-sumC } (\pi \cdot \text{body}) \cdot (\text{has-ESem.HSem } \text{ESem-sumC } (\pi \cdot as) \cdot (\varrho f |' \text{fv } (\text{Terms.Let } as \text{body})))$ 
      by (rule permute-pure[symmetric])
      also have  $\dots = (- \pi \cdot \text{ESem-sumC } \text{body}) \cdot (\text{has-ESem.HSem } (- \pi \cdot \text{ESem-sumC } as) \cdot (- \pi \cdot \varrho f |' \text{fv } (\text{Terms.Let } as \text{body})))$ 
      by (simp add: pemute-minus-self)
    }
  
```

```

also have ( $-\pi \cdot ESem\text{-}sumC$ )  $body = ESem\text{-}sumC\ body$ 
  by (rule eqvt-at-apply[OF  $\langle eqvt\text{-}at\ ESem\text{-}sumC\ body \rangle$ ])
also have  $has\text{-}ESem.HSem\ (-\pi \cdot ESem\text{-}sumC)\ as = has\text{-}ESem.HSem\ ESem\text{-}sumC\ as$ 
  by (rule HSem-cong[OF  $eqvt\text{-}at\text{-}apply[OF\ prems(2)]\ refl$ ])
also have  $-\pi \cdot \varrho\ f|^{\prime}\ fv\ (Let\ as\ body) = \varrho\ f|^{\prime}\ fv\ (Let\ as\ body)$ 
  by (rule env-restr-perm'[OF  $*$ ], simp)
finally have  $ESem\text{-}sumC\ (\pi \cdot body) \cdot (has\text{-}ESem.HSem\ ESem\text{-}sumC\ (\pi \cdot as) \cdot (\varrho\ f|^{\prime}\ fv\ (Let\ as\ body))) = ESem\text{-}sumC\ body \cdot (has\text{-}ESem.HSem\ ESem\text{-}sumC\ as \cdot (\varrho\ f|^{\prime}\ fv\ (Let\ as\ body)))$ .
}
thus ( $\Lambda\ \varrho.\ tick \cdot (ESem\text{-}sumC\ (\pi \cdot body) \cdot (has\text{-}ESem.HSem\ ESem\text{-}sumC\ (\pi \cdot as) \cdot (\varrho\ f|^{\prime}\ fv\ (Let\ as\ body)))) =$ 
  ( $\Lambda\ \varrho.\ tick \cdot (ESem\text{-}sumC\ body \cdot (has\text{-}ESem.HSem\ ESem\text{-}sumC\ as \cdot (\varrho\ f|^{\prime}\ fv\ (Let\ as\ body))))$ ))
by (simp only:)
  qed
qed auto

```

**nominal-termination** (in *semantic-domain*) (*no-eqvt*) **by** *lexicographic-order*

**sublocale**  $has\text{-}ESem\ ESem$ .

**notation**  $ESem\text{-}syn\ (\langle \llbracket - \rrbracket \rangle \rightarrow [60,60]\ 60)$

**notation**  $EvalHeapSem\text{-}syn\ (\langle \llbracket - \rrbracket \rangle \rightarrow [0,0]\ 110)$

**notation**  $HSem\text{-}syn\ (\langle \{\!\!-\!\!\} \rangle \rightarrow [60,60]\ 60)$

**abbreviation**  $AHSem\text{-}bot\ (\langle \{\!\!-\!\!\} \rangle \rightarrow [60]\ 60)$  **where**  $\{\!\!\Gamma\!\!\} \equiv \{\!\!\Gamma\!\!\} \perp$

**end**

**end**

## 5.5 Abstract-Denotational-Props

**theory** *Abstract-Denotational-Props*

**imports** *AbstractDenotational Substitution*

**begin**

**context** *semantic-domain*

**begin**

### 5.5.1 The semantics ignores fresh variables

**lemma**  $ESem\text{-}considers\text{-}fv'$ :  $\llbracket e \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho\ f|^{\prime}}\ (fv\ e)$

**proof** (*induct e arbitrary:  $\varrho$  rule:exp-induct*)

**case** *Var*

**show** *?case by simp*

**next**

**have** [*simp*]:  $\bigwedge S\ x.\ S \cap insert\ x\ S = S$  **by** *auto*

**case** *App*

**show** *?case*

**by** (*simp, subst (1 2) App, simp*)

**next**

```

  case (Lam x e)
  show ?case by simp
next
  case (IfThenElse scrut e1 e2)
  have [simp]: (fv scrut  $\cap$  (fv scrut  $\cup$  fv e1  $\cup$  fv e2)) = fv scrut by auto
  have [simp]: (fv e1  $\cap$  (fv scrut  $\cup$  fv e1  $\cup$  fv e2)) = fv e1 by auto
  have [simp]: (fv e2  $\cap$  (fv scrut  $\cup$  fv e1  $\cup$  fv e2)) = fv e2 by auto
  show ?case
  apply simp
  apply (subst (1 2) IfThenElse(1))
  apply (subst (1 2) IfThenElse(2))
  apply (subst (1 2) IfThenElse(3))
  apply (simp)
  done
next
  case (Let as e)

  have  $\llbracket e \rrbracket_{\{\text{as}\}\varrho} = \llbracket e \rrbracket_{(\{\text{as}\}\varrho) f|' (fv \text{as} \cup fv e)}$ 
  apply (subst (1 2) Let(2))
  apply simp
  done
  also
  have  $fv \text{as} \subseteq fv \text{as} \cup fv e$  by (rule inf-sup-ord(3))
  hence  $(\{\text{as}\}\varrho) f|' (fv \text{as} \cup fv e) = \{\text{as}\}\varrho f|' (fv \text{as} \cup fv e)$ 
  by (rule HSem-ignores-fresh-restr'[OF - Let(1)])
  also
  have  $\{\text{as}\}\varrho f|' (fv \text{as} \cup fv e) = \{\text{as}\}\varrho f|' (fv \text{as} \cup fv e - \text{dom}A \text{as})$ 
  by (rule HSem-restr-cong) (auto simp add: lookup-env-restr-eq)
  finally
  show ?case by simp
qed auto

sublocale has-ignore-fresh-ESem ESem
  by standard (rule fv-supp-exp, rule ESem-considers-fv')

```

### 5.5.2 Nicer equations for ESem, without freshness requirements

```

lemma ESem-Lam[simp]:  $\llbracket Lam [x]. e \rrbracket_{\varrho} = tick \cdot (Fn \cdot (\Lambda v. \llbracket e \rrbracket_{\varrho(x := v)}))$ 
proof-
  have  $*$ :  $\bigwedge v. ((\varrho f|' (fv e - \{x\}))(x := v)) f|' fv e = (\varrho(x := v)) f|' fv e$ 
  by (rule ext) (auto simp add: lookup-env-restr-eq)

  have  $\llbracket Lam [x]. e \rrbracket_{\varrho} = \llbracket Lam [x]. e \rrbracket_{env-delete x \varrho}$ 
  by (rule ESem-fresh-cong) simp
  also have  $\dots = tick \cdot (Fn \cdot (\Lambda v. \llbracket e \rrbracket_{(\varrho f|' (fv e - \{x\}))(x := v)}))$ 
  by simp
  also have  $\dots = tick \cdot (Fn \cdot (\Lambda v. \llbracket e \rrbracket_{((\varrho f|' (fv e - \{x\}))(x := v)) f|' fv e}))$ 
  by (subst ESem-considers-fv, rule)
  also have  $\dots = tick \cdot (Fn \cdot (\Lambda v. \llbracket e \rrbracket_{\varrho(x := v) f|' fv e}))$ 

```

```

  unfolding *..
  also have ... = tick · (Fn · (Λ v. ⟦ e ⟧ρ(x := v)))
  unfolding ESem-considers-fv[symmetric]..
  finally show ?thesis.
qed
declare ESem.simps(1)[simp del]

lemma ESem-Let[simp]: ⟦Let as body⟧ρ = tick · (⟦body⟧as ρ)
proof –
  have ⟦Let as body⟧ρ = tick · (⟦body⟧as (ρ f |' fv (Let as body)))
  by simp
  also have ⟦as⟧ (ρ f |' fv (Let as body)) = ⟦as⟧ (ρ f |' (fv as ∪ fv body))
  by (rule HSem-restr-cong) (auto simp add: lookup-env-restr-eq)
  also have ... = (⟦as⟧ ρ) f |' (fv as ∪ fv body)
  by (rule HSem-ignores-fresh-restr'[symmetric, OF - ESem-considers-fv]) simp
  also have ⟦body⟧... = ⟦body⟧as ρ
  by (rule ESem-fresh-cong) (auto simp add: lookup-env-restr-eq)
  finally show ?thesis.
qed
declare ESem.simps(4)[simp del]

```

### 5.5.3 Denotation of Substitution

```

lemma ESem-subst-same: ρ x = ρ y ⟹ ⟦ e ⟧ρ = ⟦ e[x ::= y] ⟧ρ
and
  ρ x = ρ y ⟹ (⟦ as ⟧ρ) = ⟦ as[x ::= h=y] ⟧ρ
proof (nominal-induct e and as avoiding: x y arbitrary: ρ and ρ rule:exp-heap-strong-induct)
case Var thus ?case by auto
next
case App
  from App(1)[OF App(2)] App(2)
  show ?case by auto
next
case (Let as exp x y ρ)
  from ⟨atom ' domA as #* x⟩ ⟨atom ' domA as #* y⟩
  have x ∉ domA as y ∉ domA as
  by (metis fresh-star-at-base imageI)+
  hence [simp]: domA (as[x ::= h=y]) = domA as
  by (metis bn-subst)

  from ⟨ρ x = ρ y⟩
  have (⟦as⟧ ρ) x = (⟦as⟧ ρ) y
  using ⟨x ∉ domA as⟩ ⟨y ∉ domA as⟩
  by (simp add: lookup-HSem-other)
  hence ⟦exp⟧as ρ = ⟦exp[x ::= y]⟧as ρ
  by (rule Let)
  moreover
  from ⟨ρ x = ρ y⟩
  have ⟦as⟧ ρ = ⟦as[x ::= h=y]⟧ ρ and (⟦as⟧ ρ) x = (⟦as[x ::= h=y]⟧ ρ) y

```

```

apply (induction rule: parallel-HSem-ind)
apply (intro adm-lemmas cont2cont cont2cont-fun)
apply simp
apply simp
apply simp
apply (erule arg-cong[OF Let(3)])
using  $\langle x \notin \text{dom}A \text{ as} \rangle \langle y \notin \text{dom}A \text{ as} \rangle$ 
apply simp
done
ultimately
show ?case using Let(1,2,3) by (simp add: fresh-star-Pair)
next
case (Lam var exp x y ρ)
  from  $\langle \rho x = \rho y \rangle$ 
  have  $\bigwedge v. (\rho(\text{var} := v)) x = (\rho(\text{var} := v)) y$ 
    using Lam(1,2) by (simp add: fresh-at-base)
  hence  $\bigwedge v. \llbracket \text{exp} \rrbracket_{\rho(\text{var} := v)} = \llbracket \text{exp}[x::=y] \rrbracket_{\rho(\text{var} := v)}$ 
    by (rule Lam)
  thus ?case using Lam(1,2) by simp
next
case IfThenElse
  from IfThenElse(1)[OF IfThenElse(4)] IfThenElse(2)[OF IfThenElse(4)] IfThenElse(3)[OF IfThenElse(4)]
  show ?case
    by simp
next
case Nil thus ?case by auto
next
case Cons
  from Cons(1,2)[OF Cons(3)] Cons(3)
  show ?case by auto
qed auto

lemma ESem-subst:
  shows  $\llbracket e \rrbracket_{\sigma(x := \sigma y)} = \llbracket e[x::=y] \rrbracket_{\sigma}$ 
proof (cases x = y)
  case False
    hence [simp]:  $x \notin \text{fv } e[x::=y]$  by (auto simp add: fv-def supp-subst supp-at-base dest: subsetD[OF supp-subst])

    have  $\llbracket e \rrbracket_{\sigma(x := \sigma y)} = \llbracket e[x::=y] \rrbracket_{\sigma(x := \sigma y)}$ 
      by (rule ESem-subst-same) simp
    also have  $\dots = \llbracket e[x::=y] \rrbracket_{\sigma}$ 
      by (rule ESem-fresh-cong) simp
    finally
      show ?thesis.
  next
  case True
    thus ?thesis by simp

```

qed

end

end

## 5.6 Denotational

theory *Denotational*

imports *Abstract-Denotational-Props Value-Nominal*

begin

This is the actual denotational semantics as found in [Lau93].

interpretation *semantic-domain Fn Fn-project B B-project* ( $\Lambda x. x$ ).

notation *ESem-syn* ( $\langle \llbracket - \rrbracket \rangle$  [60,60] 60)

notation *EvalHeapSem-syn* ( $\langle \llbracket - \rrbracket \rangle$  [0,0] 110)

notation *HSem-syn* ( $\langle \{\!\!-\!\!\} \rangle$  [60,60] 60)

notation *AHSem-bot* ( $\langle \{\!\!-\!\!\} \rangle$  [60] 60)

lemma *ESem-simps-as-defined*:

$\llbracket \text{Lam } [x]. e \rrbracket_{\varrho} = \text{Fn} \cdot (\Lambda v. \llbracket e \rrbracket_{(\varrho f | \cdot (fv \text{ (Lam } [x]. e)))(x := v)})$

$\llbracket \text{App } e x \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho} \downarrow \text{Fn } \varrho x$

$\llbracket \text{Var } x \rrbracket_{\varrho} = \varrho x$

$\llbracket \text{Bool } b \rrbracket_{\varrho} = B \cdot (\text{Discr } b)$

$\llbracket (\text{scrut } ? e_1 : e_2) \rrbracket_{\varrho} = B\text{-project} \cdot (\llbracket \text{scrut} \rrbracket_{\varrho}) \cdot (\llbracket e_1 \rrbracket_{\varrho}) \cdot (\llbracket e_2 \rrbracket_{\varrho})$

$\llbracket \text{Let } \Gamma \text{ body} \rrbracket_{\varrho} = \llbracket \text{body} \rrbracket_{\{\!\! \Gamma \!\!\} \varrho f | \cdot (fv \text{ (Let } \Gamma \text{ body)})}$

by (*simp-all del: ESem-Lam ESem-Let add: ESem.simps(1,4)*)

lemma *ESem-simps*:

$\llbracket \text{Lam } [x]. e \rrbracket_{\varrho} = \text{Fn} \cdot (\Lambda v. \llbracket e \rrbracket_{\varrho(x := v)})$

$\llbracket \text{App } e x \rrbracket_{\varrho} = \llbracket e \rrbracket_{\varrho} \downarrow \text{Fn } \varrho x$

$\llbracket \text{Var } x \rrbracket_{\varrho} = \varrho x$

$\llbracket \text{Bool } b \rrbracket_{\varrho} = B \cdot (\text{Discr } b)$

$\llbracket (\text{scrut } ? e_1 : e_2) \rrbracket_{\varrho} = B\text{-project} \cdot (\llbracket \text{scrut} \rrbracket_{\varrho}) \cdot (\llbracket e_1 \rrbracket_{\varrho}) \cdot (\llbracket e_2 \rrbracket_{\varrho})$

$\llbracket \text{Let } \Gamma \text{ body} \rrbracket_{\varrho} = \llbracket \text{body} \rrbracket_{\{\!\! \Gamma \!\!\} \varrho}$

by *simp-all*

end

## 6 Resourced denotational domain

### 6.1 C

```
theory C
imports HOLCF Mono-Nat-Fun
begin
```

```
default-sort cpo
```

The initial solution to the domain equation  $C = C_{\perp}$ , i.e. the completion of the natural numbers.

```
domain C = C (lazy C)
```

```
lemma below-C:  $x \sqsubseteq C \cdot x$ 
  by (induct x) auto
```

```
definition Cinf ( $\langle C^{\infty} \rangle$ ) where  $C^{\infty} = \text{fix} \cdot C$ 
```

```
lemma C-Cinf[simp]:  $C \cdot C^{\infty} = C^{\infty}$  unfolding Cinf-def by (rule fix-eq[symmetric])
```

```
abbreviation Cpow ( $\langle C^{\cdot} \rangle$ ) where  $C^n \equiv \text{iterate } n \cdot C \cdot \perp$ 
```

```
lemma C-below-C[simp]:  $(C^i \sqsubseteq C^j) \longleftrightarrow i \leq j$ 
  apply (induction i arbitrary: j)
  apply simp
  apply (case-tac j, auto)
  done
```

```
lemma below-Cinf[simp]:  $r \sqsubseteq C^{\infty}$ 
  apply (induct r)
  apply simp-all[2]
  apply (metis (full-types) C-Cinf monofun-cfun-arg)
  done
```

```
lemma C-eq-Cinf[simp]:  $C^i \neq C^{\infty}$ 
  by (metis C-below-C Suc-n-not-le-n below-Cinf)
```

```
lemma Cinf-eq-C[simp]:  $C^{\infty} = C \cdot r \longleftrightarrow C^{\infty} = r$ 
  by (metis C.injects C-Cinf)
```

```
lemma C-eq-C[simp]:  $(C^i = C^j) \longleftrightarrow i = j$ 
  by (metis C-below-C le-antisym le-refl)
```

```
lemma case-of-C-below:  $(\text{case } r \text{ of } C \cdot y \Rightarrow x) \sqsubseteq x$ 
  by (cases r) auto
```

```
lemma C-case-below:  $C \cdot \text{case} \cdot f \sqsubseteq f$ 
```

by (metis cfun-belowI C.case-rews(2) below-C monofun-cfun-arg)

**lemma** C-case-bot[simp]: C-case ·  $\perp$  =  $\perp$   
**apply** (subst eq-bottom-iff)  
**apply** (rule C-case-below)  
**done**

**lemma** C-case-cong:  
**assumes**  $\bigwedge r'. r = C.r' \implies f.r' = g.r'$   
**shows** C-case.f.r = C-case.g.r  
**using** assms **by** (cases r) auto

**lemma** C-cases:

**obtains** n **where**  $r = C^n \mid r = C^\infty$

**proof**–

{ **fix** m  
**have**  $\exists n. C\text{-take } m \cdot r = C^n$   
**proof** (rule C.finite-induct)  
**have**  $\perp = C^0$  **by** simp  
**thus**  $\exists n. \perp = C^n$ ..  
**next**  
**fix** r  
**show**  $\exists n. r = C^n \implies \exists n. C.r = C^n$   
**by** (auto simp del: iterate-Suc simp add: iterate-Suc[symmetric])  
**qed**

}

**then obtain** f **where** take:  $\bigwedge m. C\text{-take } m \cdot r = C^f m$  **by** metis

**have** chain ( $\lambda m. C^f m$ ) **using** ch2ch-Rep-cfunL[OF C.chain-take, **where**  $x=r$ , unfolded take].

**hence** mono f **by** (auto simp add: mono-iff-le-Suc chain-def elim!:chainE)

**have** r:  $r = (\bigsqcup m. C^f m)$  **by** (metis (lifting) take C.reach lub-eq)

**from**  $\langle \text{mono } f \rangle$

**show** thesis

**proof**(rule nat-mono-characterization)

**fix** n  
**assume** n:  $\bigwedge m. n \leq m \implies f n = f m$   
**have** max-in-chain n ( $\lambda m. C^f m$ )  
**apply** (rule max-in-chainI)  
**apply** simp  
**apply** (erule n)  
**done**

**hence**  $(\bigsqcup m. C^f m) = C^f n$  **unfolding** maxinch-is-thelub[OF  $\langle \text{chain } \rightarrow \rangle$ ].

**thus** ?thesis **using** that **unfolding** r **by** blast

**next**

**assume**  $\bigwedge m. \exists n. m \leq f n$

**hence**  $\bigwedge n. C^n \sqsubseteq r$  **unfolding** r **by** (fastforce intro: below-lub[OF  $\langle \text{chain } \rightarrow \rangle$ ])

**hence**  $(\bigsqcup n. C^n) \sqsubseteq r$

**by** (rule lub-below[OF chain-iterate])

```

  hence  $C^\infty \sqsubseteq r$  unfolding Cinf-def fix-def2.
  hence  $C^\infty = r$  using below-Cinf by (metis below-antisym)
  thus thesis using that by blast
qed
qed

```

```

lemma C-case-Cinf[simp]:  $C\text{-case} \cdot f \cdot C^\infty = f \cdot C^\infty$ 
  unfolding Cinf-def
  by (subst fix-eq) simp

```

end

## 6.2 C-Meet

```

theory C-Meet
imports C HOLCF-Meet
begin

```

```

instantiation C :: Finite-Meet-cpo begin
  fixrec C-meet ::  $C \rightarrow C \rightarrow C$ 
    where  $C\text{-meet} \cdot (C \cdot a) \cdot (C \cdot b) = C \cdot (C\text{-meet} \cdot a \cdot b)$ 

```

```

lemma[simp]:  $C\text{-meet} \cdot \perp \cdot y = \perp$   $C\text{-meet} \cdot x \cdot \perp = \perp$  by (fixrec-simp, cases x, fixrec-simp+)

```

```

instance
  apply standard
  proof(intro exI conjI strip)
    fix  $x y$ 
    {
      fix  $t$ 
      have  $(t \sqsubseteq C\text{-meet} \cdot x \cdot y) = (t \sqsubseteq x \wedge t \sqsubseteq y)$ 
      proof (induct t rule:C.take-induct)
        fix  $n$ 
        show  $(C\text{-take } n \cdot t \sqsubseteq C\text{-meet} \cdot x \cdot y) = (C\text{-take } n \cdot t \sqsubseteq x \wedge C\text{-take } n \cdot t \sqsubseteq y)$ 
        proof (induct n arbitrary: t x y rule:nat-induct)
          case 0 thus ?case by auto
          next
          case (Suc n t x y)
            with C.nchotomy[of t] C.nchotomy[of x] C.nchotomy[of y]
            show ?case by fastforce
        qed
      qed auto
    } note * = this
    show  $C\text{-meet} \cdot x \cdot y \sqsubseteq x$  using * by auto
    show  $C\text{-meet} \cdot x \cdot y \sqsubseteq y$  using * by auto
    fix  $z$ 
    assume  $z \sqsubseteq x$  and  $z \sqsubseteq y$ 
    thus  $z \sqsubseteq C\text{-meet} \cdot x \cdot y$  using * by auto

```

qed  
end

**lemma** *C-meet-is-meet*:  $(z \sqsubseteq C\text{-meet}\cdot x\cdot y) = (z \sqsubseteq x \wedge z \sqsubseteq y)$

**proof** (*induct z rule:C.take-induct*)

fix *n*

show  $(C\text{-take } n\cdot z \sqsubseteq C\text{-meet}\cdot x\cdot y) = (C\text{-take } n\cdot z \sqsubseteq x \wedge C\text{-take } n\cdot z \sqsubseteq y)$

**proof** (*induct n arbitrary: z x y rule:nat-induct*)

case 0 **thus** ?*case* **by** *auto*

next

case (*Suc n z x y*) **thus** ?*case*

apply –

apply (*cases z, simp*)

apply (*cases x, simp*)

apply (*cases y, simp*)

apply (*fastforce simp add: cfun-below-iff*)

done

qed

qed *auto*

**instance** *C* :: *cont-binary-meet*

**proof** (*standard, goal-cases*)

have [*simp*]:  $\bigwedge x y. x \sqcap y = C\text{-meet}\cdot x\cdot y$

using *C-meet-is-meet*

by (*blast intro: is-meetI*)

case 1 **thus** ?*case*

by (*simp add: ch2ch-Rep-cfunR contlub-cfun-arg contlub-cfun-fun*)

qed

**lemma** [*simp*]:  $C\cdot r \sqcap r = r$

by (*auto intro: is-meetI simp add: below-C*)

**lemma** [*simp*]:  $r \sqcap C\cdot r = r$

by (*auto intro: is-meetI simp add: below-C*)

**lemma** [*simp*]:  $C\cdot r \sqcap C\cdot r' = C\cdot (r \sqcap r')$

apply (*rule is-meetI*)

apply (*metis below-refl meet-below1 monofun-cfun-arg*)

apply (*metis below-refl meet-below2 monofun-cfun-arg*)

apply (*case-tac a*)

apply *auto*

by (*metis meet-above-iff*)

end

### 6.3 C-restr

**theory** *C-restr*

**imports** *C C-Meet HOLCF-Utills*

begin

### 6.3.1 The demand of a $C$ -function

The demand is the least amount of resources required to produce a non-bottom element, if at all.

**definition**  $demand :: (C \rightarrow 'a::pcpo) \Rightarrow C$  **where**  
 $demand\ f = (if\ f \cdot C^\infty \neq \perp\ then\ C^{(LEAST\ n.\ f \cdot C^n \neq \perp)}\ else\ C^\infty)$

Because of continuity, a non-bottom value can always be obtained with finite resources.

**lemma** *finite-resources-suffice*:

**assumes**  $f \cdot C^\infty \neq \perp$

**obtains**  $n$  **where**  $f \cdot C^n \neq \perp$

**proof**–

{

**assume**  $\forall n.\ f \cdot (C^n) = \perp$

**hence**  $f \cdot C^\infty \sqsubseteq \perp$

**by** (*auto intro: lub-below[OF ch2ch-Rep-cfunR[OF chain-iterate]]*  
*simp add: Cinf-def fix-def2 contlub-cfun-arg[OF chain-iterate]*)

**with** *assms* **have** *False* **by** *simp*

}

**thus** *?thesis* **using** *that* **by** *blast*

**qed**

Because of monotonicity, a non-bottom value can always be obtained with more resources.

**lemma** *more-resources-suffice*:

**assumes**  $f \cdot r \neq \perp$  **and**  $r \sqsubseteq r'$

**shows**  $f \cdot r' \neq \perp$

**using** *assms(1) monofun-cfun-arg[OF assms(2)]*, **where**  $f = f$

**by** *auto*

**lemma** *infinite-resources-suffice*:

**shows**  $f \cdot r \neq \perp \implies f \cdot C^\infty \neq \perp$

**by** (*erule more-resources-suffice[OF - below-Cinf]*)

**lemma** *demand-suffices*:

**assumes**  $f \cdot C^\infty \neq \perp$

**shows**  $f \cdot (demand\ f) \neq \perp$

**apply** (*simp add: assms demand-def*)

**apply** (*rule finite-resources-suffice[OF assms]*)

**apply** (*rule LeastI*)

**apply** *assumption*

**done**

**lemma** *not-bot-demand*:

$f \cdot r \neq \perp \iff demand\ f \neq C^\infty \wedge demand\ f \sqsubseteq r$

**proof**(*intro iffI*)  
**assume**  $f \cdot r \neq \perp$   
**thus**  $\text{demand } f \neq C^\infty \wedge \text{demand } f \sqsubseteq r$   
**apply** (*cases r rule:C-cases*)  
**apply** (*auto intro: Least-le simp add: demand-def dest: infinite-resources-suffice*)  
**done**  
**next**  
**assume**  $*$ :  $\text{demand } f \neq C^\infty \wedge \text{demand } f \sqsubseteq r$   
**then have**  $f \cdot C^\infty \neq \perp$  **by** (*auto intro: Least-le simp add: demand-def dest: infinite-resources-suffice*)  
**hence**  $f \cdot (\text{demand } f) \neq \perp$  **by** (*rule demand-suffices*)  
**moreover from**  $*$  **have**  $\text{demand } f \sqsubseteq r..$   
**ultimately**  
**show**  $f \cdot r \neq \perp$  **by** (*rule more-resources-suffice*)  
**qed**

**lemma** *infinity-bot-demand*:  
 $f \cdot C^\infty = \perp \iff \text{demand } f = C^\infty$   
**by** (*metis below-Cinf not-bot-demand*)

**lemma** *demand-suffices'*:  
**assumes**  $\text{demand } f = C^n$   
**shows**  $f \cdot (\text{demand } f) \neq \perp$   
**by** (*metis C-eq-Cinf assms demand-suffices infinity-bot-demand*)

**lemma** *demand-Suc-Least*:  
**assumes** [*simp*]:  $f \cdot \perp = \perp$   
**assumes**  $\text{demand } f \neq C^\infty$   
**shows**  $\text{demand } f = C^{(\text{Suc } (\text{LEAST } n. f \cdot C^{\text{Suc } n} \neq \perp))}$

**proof** –  
**from** *assms*  
**have**  $\text{demand } f = C^{(\text{LEAST } n. f \cdot C^n \neq \perp)}$  **by** (*auto simp add: demand-def*)  
**also**  
**then obtain**  $n$  **where**  $f \cdot C^n \neq \perp$  **by** (*metis demand-suffices'*)  
**hence**  $(\text{LEAST } n. f \cdot C^n \neq \perp) = \text{Suc } (\text{LEAST } n. f \cdot C^{\text{Suc } n} \neq \perp)$   
**apply** (*rule Least-Suc*) **by** *simp*  
**finally show** *?thesis*.  
**qed**

**lemma** *demand-C-case[simp]*:  $\text{demand } (C\text{-case}\cdot f) = C \cdot (\text{demand } f)$   
**proof**(*cases demand (C-case\cdot f) = C^\infty*)

**case** *True*  
**then have**  $C\text{-case}\cdot f \cdot C^\infty = \perp$   
**by** (*metis infinity-bot-demand*)  
**with** *True*  
**show** *?thesis* **apply** *auto* **by** (*metis infinity-bot-demand*)

**next**  
**case** *False*  
**hence**  $\text{demand } (C\text{-case}\cdot f) = C^{\text{Suc } (\text{LEAST } n. (C\text{-case}\cdot f) \cdot C^{\text{Suc } n} \neq \perp)}$   
**by** (*rule demand-Suc-Least[OF C.case-rews(1)]*)

also have  $\dots = C.C^{LEAST} n. f.C^n \neq \perp$  **by** *simp*  
 also have  $\dots = C.(demand\ f)$   
 using *False unfolding demand-def* **by** *auto*  
 finally show *?thesis*.  
**qed**

**lemma** *demand-contravariant*:

assumes  $f \sqsubseteq g$

shows  $demand\ g \sqsubseteq demand\ f$

**proof**(*cases demand f rule:C-cases*)

fix  $n$

assume  $demand\ f = C^n$

hence  $f.(demand\ f) \neq \perp$  **by** (*metis demand-suffices'*)

also note *monofun-cfun-fun[OF assms]*

finally have  $g.(demand\ f) \neq \perp$  **by** *this (intro cont2cont)*

thus  $demand\ g \sqsubseteq demand\ f$  **unfolding** *not-bot-demand* **by** *auto*

**qed** *auto*

### 6.3.2 Restricting functions with domain C

**fixrec** *C-restr* ::  $C \rightarrow (C \rightarrow 'a::pcpo) \rightarrow (C \rightarrow 'a)$

where  $C-restr.r.f.r' = (f.(r \sqcap r'))$

**abbreviation** *C-restr-syn* ::  $(C \rightarrow 'a::pcpo) \Rightarrow C \Rightarrow (C \rightarrow 'a)$  ( $\langle \cdot \rangle$  [111,110] 110)

where  $f|_r \equiv C-restr.r.f$

**lemma** [*simp*]:  $\perp|_r = \perp$  **by** *fixrec-simp*

**lemma** [*simp*]:  $f.\perp = \perp \implies f|_{\perp} = \perp$  **by** *fixrec-simp*

**lemma** *C-restr-C-restr[simp]*:  $(v|_{r'})|_r = v|_{(r' \sqcap r)}$

**by** (*rule cfun-eqI*) *simp*

**lemma** *C-restr-eqD*:

assumes  $f|_r = g|_r$

assumes  $r' \sqsubseteq r$

shows  $f.r' = g.r'$

**by** (*metis C-restr.simps assms below-refl is-meetI*)

**lemma** *C-restr-eq-lower*:

assumes  $f|_r = g|_r$

assumes  $r' \sqsubseteq r$

shows  $f|_{r'} = g|_{r'}$

**by** (*metis C-restr-C-restr assms below-refl is-meetI*)

**lemma** *C-restr-below[intro, simp]*:

$x|_r \sqsubseteq x$

**apply** (*rule cfun-belowI*)

**apply** *simp*

**by** (*metis below-refl meet-below2 monofun-cfun-arg*)

**lemma** *C-restr-below-cong*:

$(\bigwedge r'. r' \sqsubseteq r \implies f \cdot r' \sqsubseteq g \cdot r') \implies f|_r \sqsubseteq g|_r$

**apply** (*rule cfun-belowI*)

**apply** *simp*

**by** (*metis below-refl meet-below1*)

**lemma** *C-restr-cong*:

$(\bigwedge r'. r' \sqsubseteq r \implies f \cdot r' = g \cdot r') \implies f|_r = g|_r$

**apply** (*intro below-antisym C-restr-below-cong*)

**by** (*metis below-refl*)<sup>+</sup>

**lemma** *C-restr-C-cong*:

$(\bigwedge r'. r' \sqsubseteq r \implies f \cdot (C \cdot r') = g \cdot (C \cdot r')) \implies f \cdot \perp = g \cdot \perp \implies f|_{C \cdot r} = g|_{C \cdot r}$

**apply** (*rule C-restr-cong*)

**by** (*case-tac r'*, *auto*)

**lemma** *C-restr-C-case[simp]*:

$(C \cdot \text{case} \cdot f)|_{C \cdot r} = C \cdot \text{case} \cdot (f|_r)$

**apply** (*rule cfun-eqI*)

**apply** *simp*

**apply** (*case-tac x*)

**apply** *simp*

**apply** *simp*

**done**

**lemma** *C-restr-bot-demand*:

**assumes**  $C \cdot r \sqsubseteq \text{demand } f$

**shows**  $f|_r = \perp$

**proof** (*rule cfun-eqI*)

**fix**  $r'$

**have**  $f \cdot (r \sqcap r') = \perp$

**proof** (*rule classical*)

**have**  $r \sqsubseteq C \cdot r$  **by** (*rule below-C*)

**also**

**note** *assms*

**also**

**assume**  $*$ :  $f \cdot (r \sqcap r') \neq \perp$

**hence**  $\text{demand } f \sqsubseteq (r \sqcap r')$  **unfolding** *not-bot-demand* **by** *auto*

**hence**  $\text{demand } f \sqsubseteq r$  **by** (*metis below-refl meet-below1 below-trans*)

**finally** (*below-antisym*) **have**  $r = \text{demand } f$  **by** *this* (*intro cont2cont*)

**with** *assms*

**have**  $\text{demand } f = C^\infty$  **by** (*cases demand f rule: C-cases*) (*auto simp add: iterate-Suc[symmetric]*)

*simp del: iterate-Suc*)

**thus**  $f \cdot (r \sqcap r') = \perp$  **by** (*metis not-bot-demand*)

**qed**

**thus**  $(f|_r) \cdot r' = \perp \cdot r'$  **by** *simp*

**qed**

### 6.3.3 Restricting maps of C-ranged functions

**definition**  $env\text{-}C\text{-restr} :: C \rightarrow ('var::type \Rightarrow (C \rightarrow 'a::pcpo)) \rightarrow ('var \Rightarrow (C \rightarrow 'a))$  **where**  
 $env\text{-}C\text{-restr} = (\Lambda r f. cfun\text{-}comp.(C\text{-restr}.r).f)$

**abbreviation**  $env\text{-}C\text{-restr}\text{-}syn :: ('var::type \Rightarrow (C \rightarrow 'a::pcpo)) \Rightarrow C \Rightarrow ('var \Rightarrow (C \rightarrow 'a))$  ( $\langle \cdot |^\circ \cdot \rangle$  [111,110] 110)  
**where**  $f|^\circ_r \equiv env\text{-}C\text{-restr}.r.f$

**lemma**  $env\text{-}C\text{-restr}\text{-}upd[simp]: (\varrho(x := v))|^\circ_r = (\varrho|^\circ_r)(x := v|_r)$   
**unfolding**  $env\text{-}C\text{-restr}\text{-}def$  **by**  $simp$

**lemma**  $env\text{-}C\text{-restr}\text{-}lookup[simp]: (\varrho|^\circ_r) v = \varrho v|_r$   
**unfolding**  $env\text{-}C\text{-restr}\text{-}def$  **by**  $simp$

**lemma**  $env\text{-}C\text{-restr}\text{-}bot[simp]: \perp|^\circ_r = \perp$   
**unfolding**  $env\text{-}C\text{-restr}\text{-}def$  **by**  $auto$

**lemma**  $env\text{-}C\text{-restr}\text{-}restr\text{-}below[intro]: \varrho|^\circ_r \sqsubseteq \varrho$   
**by** ( $auto$   $intro: fun\text{-}belowI$ )

**lemma**  $env\text{-}C\text{-restr}\text{-}env\text{-}C\text{-restr}[simp]: (v|^\circ_{r'})|^\circ_r = v|^\circ_{(r' \sqcap r)}$   
**unfolding**  $env\text{-}C\text{-restr}\text{-}def$  **by**  $auto$

**lemma**  $env\text{-}C\text{-restr}\text{-}cong:$   
 $(\Lambda x r'. r' \sqsubseteq r \implies f x \cdot r' = g x \cdot r') \implies f|^\circ_r = g|^\circ_r$   
**unfolding**  $env\text{-}C\text{-restr}\text{-}def$   
**by** ( $rule ext$ ) ( $auto$   $intro: C\text{-restr}\text{-}cong$ )

**end**

## 6.4 CValue

**theory**  $CValue$   
**imports**  $C$   
**begin**

**domain**  $CValue$   
 $= CFn$  (**lazy**  $(C \rightarrow CValue) \rightarrow (C \rightarrow CValue)$ )  
 $| CB$  (**lazy**  $bool$   $discr$ )

**fixrec**  $CFn\text{-}project :: CValue \rightarrow (C \rightarrow CValue) \rightarrow (C \rightarrow CValue)$   
**where**  $CFn\text{-}project.(CFn.f).v = f \cdot v$

**abbreviation**  $CFn\text{-}project\text{-}abbr$  (**infix**  $\langle \downarrow CFn \rangle$  55)  
**where**  $f \downarrow CFn v \equiv CFn\text{-}project.f.v$

**lemma**  $CFn\text{-}project\text{-}strict[simp]:$

$\perp \downarrow CFn\ v = \perp$   
 $CB \cdot b \downarrow CFn\ v = \perp$   
**by** (*fixrec-simp*)<sup>+</sup>

**lemma** *CB-below[simp]*:  $CB \cdot b \sqsubseteq v \longleftrightarrow v = CB \cdot b$   
**by** (*cases v*) *auto*

**fixrec** *CB-project* ::  $CValue \rightarrow CValue \rightarrow CValue \rightarrow CValue$  **where**  
 $CB\text{-project} \cdot (CB \cdot db) \cdot v_1 \cdot v_2 = (\text{if } \text{undiscr } db \text{ then } v_1 \text{ else } v_2)$

**lemma** [*simp*]:  
 $CB\text{-project} \cdot (CB \cdot (Discr\ b)) \cdot v_1 \cdot v_2 = (\text{if } b \text{ then } v_1 \text{ else } v_2)$   
 $CB\text{-project} \cdot \perp \cdot v_1 \cdot v_2 = \perp$   
 $CB\text{-project} \cdot (CFn \cdot f) \cdot v_1 \cdot v_2 = \perp$   
**by** *fixrec-simp*<sup>+</sup>

**lemma** *CB-project-not-bot*:  
 $CB\text{-project} \cdot scrut \cdot v_1 \cdot v_2 \neq \perp \longleftrightarrow (\exists b. scrut = CB \cdot (Discr\ b) \wedge (\text{if } b \text{ then } v_1 \text{ else } v_2) \neq \perp)$   
**apply** (*cases scrut*)  
**apply** *simp*  
**apply** *simp*  
**by** (*metis (poly-guards-query) CB-project.simps CValue.injects(2) discr.exhaust undiscr-Discr*)

HOLCF provides us  $CValue\text{-take}::nat \Rightarrow CValue \rightarrow CValue$ ; we want a similar function for  $C \rightarrow CValue$ .

**abbreviation** *C-to-CValue-take* ::  $nat \Rightarrow (C \rightarrow CValue) \rightarrow (C \rightarrow CValue)$   
**where**  $C\text{-to-}CValue\text{-take } n \equiv cfun\text{-map}\cdot ID \cdot (CValue\text{-take } n)$

**lemma** *C-to-CValue-chain-take*: *chain C-to-CValue-take*  
**by** (*auto intro: chainI cfun-belowI chainE[OF CValue.chain-take] monofun-cfun-fun*)

**lemma** *C-to-CValue-reach*:  $(\bigsqcup n. C\text{-to-}CValue\text{-take } n \cdot x) = x$   
**by** (*auto intro: cfun-eqI simp add: contlub-cfun-fun[OF ch2ch-Rep-cfunL[OF C-to-CValue-chain-take]] CValue.reach*)

**end**

## 6.5 CValue-Nominal

**theory** *CValue-Nominal*  
**imports** *CValue Nominal-Utills Nominal-HOLCF*  
**begin**

**instantiation** *C* :: *pure*

**begin**

**definition**  $p \cdot (c::C) = c$

**instance** **by** *standard (auto simp add: permute-C-def)*

**end**  
**instance**  $C :: \text{pcpo-pt}$   
 by *standard* (*simp add: pure-permute-id*)

**instantiation**  $CValue :: \text{pure}$   
**begin**  
**definition**  $p \cdot (v :: CValue) = v$   
**instance**  
**apply** *standard*  
**apply** (*auto simp add: permute-CValue-def*)  
**done**  
**end**

**instance**  $CValue :: \text{pcpo-pt}$   
 by *standard* (*simp add: pure-permute-id*)

**end**

## 6.6 ResourcedDenotational

**theory** *ResourcedDenotational*  
**imports** *Abstract-Denotational-Props CValue-Nominal C-restr*  
**begin**

**type-synonym**  $CEnv = \text{var} \Rightarrow (C \rightarrow CValue)$

**interpretation** *semantic-domain*  
 $\Lambda f . \Lambda r . CFn \cdot (\Lambda v . (f \cdot (v)) | r)$   
 $\Lambda x y . (\Lambda r . (x \cdot r \downarrow CFn y | r) \cdot r)$   
 $\Lambda b r . CB \cdot b$   
 $\Lambda scrut v1 v2 r . CB\text{-project} \cdot (scrut \cdot r) \cdot (v1 \cdot r) \cdot (v2 \cdot r)$   
*C-case.*

**notation** *ESem-syn* ( $\langle \mathcal{N} \llbracket - \rrbracket \cdot \rangle [60, 60] 60$ )  
**notation** *EvalHeapSem-syn* ( $\langle \mathcal{N} \llbracket - \rrbracket \cdot \rangle [0, 0] 110$ )  
**notation** *HSem-syn* ( $\langle \mathcal{N} \llbracket - \rrbracket \cdot \rangle [60, 60] 60$ )  
**notation** *AHSem-bot* ( $\langle \mathcal{N} \llbracket - \rrbracket \cdot \rangle [60] 60$ )

Here we re-state the simplification rules, cleaned up by beta-reducing the locale parameters.

**lemma** *CESem-simps:*

$\mathcal{N} \llbracket Lam [x]. e \rrbracket_{\varrho} = (\Lambda (C \cdot r) . CFn \cdot (\Lambda v . (\mathcal{N} \llbracket e \rrbracket_{\varrho} (x := v)) | r))$   
 $\mathcal{N} \llbracket App e x \rrbracket_{\varrho} = (\Lambda (C \cdot r) . ((\mathcal{N} \llbracket e \rrbracket_{\varrho}) \cdot r \downarrow CFn \varrho x | r) \cdot r)$   
 $\mathcal{N} \llbracket Var x \rrbracket_{\varrho} = (\Lambda (C \cdot r) . (\varrho x) \cdot r)$   
 $\mathcal{N} \llbracket Bool b \rrbracket_{\varrho} = (\Lambda (C \cdot r) . CB \cdot (Discr b))$   
 $\mathcal{N} \llbracket (scrut ? e_1 : e_2) \rrbracket_{\varrho} = (\Lambda (C \cdot r) . CB\text{-project} \cdot ((\mathcal{N} \llbracket scrut \rrbracket_{\varrho}) \cdot r) \cdot ((\mathcal{N} \llbracket e_1 \rrbracket_{\varrho}) \cdot r) \cdot ((\mathcal{N} \llbracket e_2 \rrbracket_{\varrho}) \cdot r))$

$\mathcal{N} \llbracket \text{Let as } body \rrbracket_{\varrho} = (\Lambda (C \cdot r) \cdot (\mathcal{N} \llbracket body \rrbracket_{\mathcal{N} \{as\}_{\varrho}}) \cdot r)$   
**by** (*auto simp add: eta-cfun*)

**lemma** *CESem-bot[simp]*:  $(\mathcal{N} \llbracket e \rrbracket_{\sigma}) \cdot \perp = \perp$   
**by** (*nominal-induct e arbitrary:  $\sigma$  rule: exp-strong-induct*) *auto*

**lemma** *CHSem-bot[simp]*:  $(\mathcal{N} \{ \Gamma \} x) \cdot \perp = \perp$   
**by** (*cases  $x \in \text{dom } A \Gamma$ ) (auto simp add: lookup-HSem-heap lookup-HSem-other)*)

Sometimes we do not care much about the resource usage and just want a simpler formula.

**lemma** *CESem-simps-no-tick*:  
 $(\mathcal{N} \llbracket Lam [x]. e \rrbracket_{\varrho}) \cdot r \sqsubseteq CFn \cdot (\Lambda v. (\mathcal{N} \llbracket e \rrbracket_{\varrho}(x := v)) | r)$   
 $(\mathcal{N} \llbracket App e x \rrbracket_{\varrho}) \cdot r \sqsubseteq ((\mathcal{N} \llbracket e \rrbracket_{\varrho}) \cdot r \downarrow CFn \varrho x | r) \cdot r$   
 $\mathcal{N} \llbracket Var x \rrbracket_{\varrho} \sqsubseteq \varrho x$   
 $(\mathcal{N} \llbracket scrut ? e_1 : e_2 \rrbracket_{\varrho}) \cdot r \sqsubseteq CB\text{-project} \cdot ((\mathcal{N} \llbracket scrut \rrbracket_{\varrho}) \cdot r) \cdot ((\mathcal{N} \llbracket e_1 \rrbracket_{\varrho}) \cdot r) \cdot ((\mathcal{N} \llbracket e_2 \rrbracket_{\varrho}) \cdot r)$   
 $\mathcal{N} \llbracket \text{Let as } body \rrbracket_{\varrho} \sqsubseteq \mathcal{N} \llbracket body \rrbracket_{\mathcal{N} \{as\}_{\varrho}}$   
**apply** –  
**apply** (*rule below-trans[OF monofun-cfun-arg[OF below-C]], simp*)  
**apply** (*rule below-trans[OF monofun-cfun-arg[OF below-C]], simp*)  
**apply** (*rule cfun-belowI, rule below-trans[OF monofun-cfun-arg[OF below-C]], simp*)  
**apply** (*rule below-trans[OF monofun-cfun-arg[OF below-C]], simp*)  
**apply** (*rule cfun-belowI, rule below-trans[OF monofun-cfun-arg[OF below-C]], simp*)  
**done**

**lemma** *CELam-no-restr*:  $(\mathcal{N} \llbracket Lam [x]. e \rrbracket_{\varrho}) \cdot r \sqsubseteq CFn \cdot (\Lambda v. (\mathcal{N} \llbracket e \rrbracket_{\varrho}(x := v)))$

**proof** –  
**have**  $(\mathcal{N} \llbracket Lam [x]. e \rrbracket_{\varrho}) \cdot r \sqsubseteq CFn \cdot (\Lambda v. (\mathcal{N} \llbracket e \rrbracket_{\varrho}(x := v)) | r)$  **by** (*rule CESem-simps-no-tick*)  
**also have**  $\dots \sqsubseteq CFn \cdot (\Lambda v. (\mathcal{N} \llbracket e \rrbracket_{\varrho}(x := v)))$   
**by** (*intro cont2cont monofun-LAM below-trans[OF C-restr-below] monofun-cfun-arg below-refl fun-upd-mono*)  
**finally show** *?thesis by this (intro cont2cont)*  
**qed**

**lemma** *CEApp-no-restr*:  $(\mathcal{N} \llbracket App e x \rrbracket_{\varrho}) \cdot r \sqsubseteq ((\mathcal{N} \llbracket e \rrbracket_{\varrho}) \cdot r \downarrow CFn \varrho x) \cdot r$

**proof** –  
**have**  $(\mathcal{N} \llbracket App e x \rrbracket_{\varrho}) \cdot r \sqsubseteq ((\mathcal{N} \llbracket e \rrbracket_{\varrho}) \cdot r \downarrow CFn \varrho x | r) \cdot r$  **by** (*rule CESem-simps-no-tick*)  
**also have**  $\varrho x | r \sqsubseteq \varrho x$  **by** (*rule C-restr-below*)  
**finally show** *?thesis by this (intro cont2cont)*  
**qed**

**end**

## 7 Correctness of the natural semantics

### 7.1 CorrectnessOriginal

```
theory CorrectnessOriginal
imports Denotational Launchbury
begin
```

This is the main correctness theorem, Theorem 2 from [Lau93].

```
theorem correctness:
  assumes  $\Gamma : e \Downarrow_L \Delta : v$ 
  and  $fv(\Gamma, e) \subseteq set L \cup domA \Gamma$ 
  shows  $\llbracket e \rrbracket_{\Gamma}^{\varrho} = \llbracket v \rrbracket_{\Delta}^{\varrho}$ 
  and  $(\llbracket \Gamma \rrbracket^{\varrho}) f|^{\cdot} domA \Gamma = (\llbracket \Delta \rrbracket^{\varrho}) f|^{\cdot} domA \Gamma$ 
  using assms
proof (nominal-induct arbitrary:  $\varrho$  rule:reds.strong-induct)
case Lambda
  case 1 show ?case..
  case 2 show ?case..
next
case (Application  $y \Gamma e x L \Delta \Theta v e'$ )
  have Gamma-subset:  $domA \Gamma \subseteq domA \Delta$ 
  by (rule reds-doesnt-forget[OF Application.hyps(8)])

  case 1
  hence prem1:  $fv(\Gamma, e) \subseteq set L \cup domA \Gamma$  and  $x \in set L \cup domA \Gamma$  by auto
  moreover
  note reds-pres-closed[OF Application.hyps(8) prem1]
  moreover
  note reds-doesnt-forget[OF Application.hyps(8)]
  moreover
  have  $fv(e'[y::=x]) \subseteq fv(Lam [y]. e') \cup \{x\}$ 
  by (auto simp add: fv-subst-eq)
  ultimately
  have prem2:  $fv(\Delta, e'[y::=x]) \subseteq set L \cup domA \Delta$  by auto

  have *:  $(\llbracket \Gamma \rrbracket^{\varrho}) x = (\llbracket \Delta \rrbracket^{\varrho}) x$ 
  proof (cases  $x \in domA \Gamma$ )
  case True
    from Application.hyps(10)[OF prem1, where  $\varrho = \varrho$ ]
    have  $((\llbracket \Gamma \rrbracket^{\varrho}) f|^{\cdot} domA \Gamma) x = ((\llbracket \Delta \rrbracket^{\varrho}) f|^{\cdot} domA \Gamma) x$  by simp
    with True show ?thesis by simp
  case False
    from False  $\langle x \in set L \cup domA \Gamma \rangle$  reds-avoids-live[OF Application.hyps(8)]
    show ?thesis by (auto simp add: lookup-HSem-other)
  qed
  have  $\llbracket App e x \rrbracket_{\Gamma}^{\varrho} = (\llbracket e \rrbracket_{\Gamma}^{\varrho}) \Downarrow_{Fn} (\llbracket \Gamma \rrbracket^{\varrho}) x$ 
  by simp
```

**also have** ... = ( $\llbracket \text{Lam } [y]. e' \rrbracket_{\{\Delta\}\varrho} \downarrow \text{Fn } (\{\Gamma\}\varrho) x$   
**using** *Application.hyps(9)[OF prem1]* **by** *simp*  
**also have** ... = ( $\llbracket \text{Lam } [y]. e' \rrbracket_{\{\Delta\}\varrho} \downarrow \text{Fn } (\{\Delta\}\varrho) x$   
**unfolding** \*..  
**also have** ... = ( $\text{Fn}(\Lambda z. \llbracket e' \rrbracket_{(\{\Delta\}\varrho)(y := z)}) \downarrow \text{Fn } (\{\Delta\}\varrho) x$   
**by** *simp*  
**also have** ... =  $\llbracket e' \rrbracket_{(\{\Delta\}\varrho)(y := (\{\Delta\}\varrho) x)}$   
**by** *simp*  
**also have** ... =  $\llbracket e'[y ::= x] \rrbracket_{\{\Delta\}\varrho}$   
**unfolding** *ESem-subst..*  
**also have** ... =  $\llbracket v \rrbracket_{\{\Theta\}\varrho}$   
**by** (*rule Application.hyps(12)[OF prem2]*)  
**finally**  
**show**  $\llbracket \text{App } e x \rrbracket_{\{\Gamma\}\varrho} = \llbracket v \rrbracket_{\{\Theta\}\varrho}$ . **show**  $(\{\Gamma\}\varrho) f|' \text{domA } \Gamma = (\{\Theta\}\varrho) f|' \text{domA } \Gamma$   
**using** *Application.hyps(10)[OF prem1]*  
*env-restr-eq-subset[OF Gamma-subset Application.hyps(13)[OF prem2]]*  
**by** (*rule trans*)  
**next**  
**case** (*Variable*  $\Gamma x e L \Delta v$ )  
**hence** [*simp*]:  $x \in \text{domA } \Gamma$  **by** (*metis domA-from-set map-of-SomeD*)  
  
**let**  $? \Gamma = \text{delete } x \Gamma$   
  
**case** 2  
**have**  $x \notin \text{domA } \Delta$   
**by** (*rule reds-avoids-live[OF Variable.hyps(2)], simp-all*)  
  
**have** *subset*:  $\text{domA } ? \Gamma \subseteq \text{domA } \Delta$   
**by** (*rule reds-doesnt-forget[OF Variable.hyps(2)]*)  
  
**let**  $? \text{new} = \text{domA } \Delta - \text{domA } \Gamma$   
**have**  $\text{fv } (? \Gamma, e) \cup \{x\} \subseteq \text{fv } (\Gamma, \text{Var } x)$   
**by** (*rule fv-delete-heap[OF map-of  $\Gamma x = \text{Some } e$ ]*)  
**hence** *prem*:  $\text{fv } (? \Gamma, e) \subseteq \text{set } (x \# L) \cup \text{domA } ? \Gamma$  **using** 2 **by** *auto*  
**hence** *fv-subset*:  $\text{fv } (? \Gamma, e) - \text{domA } ? \Gamma \subseteq - ? \text{new}$   
**using** *reds-avoids-live'[OF Variable.hyps(2)]* **by** *auto*  
  
**have**  $\text{domA } \Gamma \subseteq (- ? \text{new})$  **by** *auto*  
  
**have**  $\{\Gamma\}\varrho = \{(x, e) \# ? \Gamma\}\varrho$   
**by** (*rule HSem-reorder[OF map-of-delete-insert[symmetric, OF Variable(1)]]*)  
**also have** ... =  $(\mu \varrho'. (\varrho ++_{\text{domA } ? \Gamma} (\{\? \Gamma\}\varrho')) (x := \llbracket e \rrbracket_{\varrho'})$   
**by** (*rule iterative-HSem, simp*)  
**also have** ... =  $(\mu \varrho'. (\varrho ++_{\text{domA } ? \Gamma} (\{\? \Gamma\}\varrho')) (x := \llbracket e \rrbracket_{\{\? \Gamma\}\varrho'})$   
**by** (*rule iterative-HSem', simp*)  
**finally**  
**have**  $(\{\Gamma\}\varrho) f|' (- ? \text{new}) = (\dots) f|' (- ? \text{new})$  **by** *simp*  
**also have** ... =  $(\mu \varrho'. (\varrho ++_{\text{domA } \Delta} (\{\Delta\}\varrho')) (x := \llbracket v \rrbracket_{\{\Delta\}\varrho'}) f|' (- ? \text{new})$

```

proof (induction rule: parallel-fix-ind[where  $P = \lambda x y. x f|'(- ?new) = y f|'(- ?new)$ ])
  case 1 show ?case by simp
next
  case 2 show ?case ..
next
  case ( $\exists \sigma \sigma'$ )
  hence  $\llbracket e \rrbracket_{\Gamma} \sigma = \llbracket e \rrbracket_{\Gamma} \sigma'$ 
  and  $(\llbracket \Gamma \rrbracket \sigma) f|' \text{domA } \Gamma = (\llbracket \Gamma \rrbracket \sigma') f|' \text{domA } \Gamma$ 
  using fv-subset by (auto intro: ESem-fresh-cong HSem-fresh-cong env-restr-eq-subset[OF
- 3])
  from trans[OF this(1) Variable(3)[OF prem]] trans[OF this(2) Variable(4)[OF prem]]
  have  $\llbracket e \rrbracket_{\Gamma} \sigma = \llbracket v \rrbracket_{\Delta} \sigma'$ 
  and  $(\llbracket \Gamma \rrbracket \sigma) f|' \text{domA } \Gamma = (\llbracket \Delta \rrbracket \sigma') f|' \text{domA } \Gamma$ .
  thus ?case
  using subset
  by (fastforce simp add: lookup-override-on-eq lookup-env-restr-eq dest: env-restr-eqD )
qed
also have  $\dots = (\mu \rho'. (\rho ++ \text{domA } \Delta (\llbracket \Delta \rrbracket \rho')) (x := \llbracket v \rrbracket_{\rho'})) f|' (- ?new)$ 
  by (rule arg-cong[OF iterative-HSem[symmetric], OF  $\langle x \notin \text{domA } \Delta \rangle$ ])
also have  $\dots = (\llbracket (x, v) \# \Delta \rrbracket_{\rho}) f|' (- ?new)$ 
  by (rule arg-cong[OF iterative-HSem[symmetric], OF  $\langle x \notin \text{domA } \Delta \rangle$ ])
finally
show le: ?case by (rule env-restr-eq-subset[OF  $\langle \text{domA } \Gamma \subseteq (- ?new) \rangle$ ])

  have  $\llbracket \text{Var } x \rrbracket_{\Gamma} \rho = \llbracket \text{Var } x \rrbracket_{\llbracket (x, v) \# \Delta \rrbracket_{\rho}}$ 
  using env-restr-eqD[OF le, where  $x = x$ ]
  by simp
  also have  $\dots = \llbracket v \rrbracket_{\llbracket (x, v) \# \Delta \rrbracket_{\rho}}$ 
  by (auto simp add: lookup-HSem-heap)
  finally
  show  $\llbracket \text{Var } x \rrbracket_{\Gamma} \rho = \llbracket v \rrbracket_{\llbracket (x, v) \# \Delta \rrbracket_{\rho}}$ .
next
case (Bool b)
  case 1
  show ?case by simp
  case 2
  show ?case by simp
next
case (IfThenElse  $\Gamma$  scrut L  $\Delta$  b e1 e2  $\Theta$  v)
  have Gamma-subset:  $\text{domA } \Gamma \subseteq \text{domA } \Delta$ 
  by (rule reds-doesnt-forget[OF IfThenElse.hyps(1)])

  let ?e = if b then e1 else e2

  case 1

  hence prem1:  $\text{fv } (\Gamma, \text{scrut}) \subseteq \text{set } L \cup \text{domA } \Gamma$ 
  and prem2:  $\text{fv } (\Delta, ?e) \subseteq \text{set } L \cup \text{domA } \Delta$ 

```

```

and  $fv\ ?e \subseteq domA\ \Gamma \cup set\ L$ 
using new-free-vars-on-heap[OF IfThenElse.hyps(1)] Gamma-subset by auto

have  $\llbracket (scrut\ ?\ e_1 : e_2) \rrbracket_{\{\Gamma\}\varrho} = B\text{-project} \cdot (\llbracket scrut \rrbracket_{\{\Gamma\}\varrho}) \cdot (\llbracket e_1 \rrbracket_{\{\Gamma\}\varrho}) \cdot (\llbracket e_2 \rrbracket_{\{\Gamma\}\varrho})$  by simp
also have  $\dots = B\text{-project} \cdot (\llbracket Bool\ b \rrbracket_{\{\Delta\}\varrho}) \cdot (\llbracket e_1 \rrbracket_{\{\Gamma\}\varrho}) \cdot (\llbracket e_2 \rrbracket_{\{\Gamma\}\varrho})$ 
  unfolding IfThenElse.hyps(2)[OF prem1]..
also have  $\dots = \llbracket ?e \rrbracket_{\{\Gamma\}\varrho}$  by simp
also have  $\dots = \llbracket ?e \rrbracket_{\{\Delta\}\varrho}$ 
proof(rule ESem-fresh-cong-subset[OF  $\langle fv\ ?e \subseteq domA\ \Gamma \cup set\ L \rangle$  env-restr-eqI])
  fix  $x$ 
  assume  $x \in domA\ \Gamma \cup set\ L$ 
  thus  $(\{\Gamma\}\varrho)\ x = (\{\Delta\}\varrho)\ x$ 
  proof(cases  $x \in domA\ \Gamma$ )
    assume  $x \in domA\ \Gamma$ 
    from IfThenElse.hyps(3)[OF prem1]
    have  $((\{\Gamma\}\varrho)\ f|' domA\ \Gamma)\ x = ((\{\Delta\}\varrho)\ f|' domA\ \Gamma)\ x$  by simp
    with  $\langle x \in domA\ \Gamma \rangle$  show ?thesis by simp
  next
  assume  $x \notin domA\ \Gamma$ 
  from this  $\langle x \in domA\ \Gamma \cup set\ L \rangle$  reds-avoids-live[OF IfThenElse.hyps(1)]
  show ?thesis
  by (simp add: lookup-HSem-other)
  qed
qed
also have  $\dots = \llbracket v \rrbracket_{\{\Theta\}\varrho}$ 
  unfolding IfThenElse.hyps(5)[OF prem2]..
finally
show ?case.
thm env-restr-eq-subset
show  $(\{\Gamma\}\varrho)\ f|' domA\ \Gamma = (\{\Theta\}\varrho)\ f|' domA\ \Gamma$ 
  using IfThenElse.hyps(3)[OF prem1]
  env-restr-eq-subset[OF Gamma-subset IfThenElse.hyps(6)[OF prem2]]
  by (rule trans)
next
case (Let as  $\Gamma\ L\ body\ \Delta\ v$ )
  case 1
  { fix  $a$ 
    assume  $a: a \in domA\ as$ 
    have atom  $a \notin \Gamma$ 
      by (rule Let(1)[unfolded fresh-star-def, rule-format, OF imageI[OF a]])
    hence  $a \notin domA\ \Gamma$ 
    by (metis domA-not-fresh)
  }
note  $*$  = this

have  $fv\ (as\ @\ \Gamma,\ body) - domA\ (as\ @\ \Gamma) \subseteq fv\ (\Gamma,\ Let\ as\ body) - domA\ \Gamma$ 
by auto

```

```

with 1 have prem: fv (as @ Γ, body) ⊆ set L ∪ domA (as @ Γ) by auto

have f1: atom ' domA as #* Γ
  using Let(1) by (simp add: set-bn-to-atom-domA)

have [ Let as body ]_{Γ} ρ = [ body ]_{as} _{Γ} ρ
  by (simp)
also have ... = [ body ]_{as @ Γ} ρ
  by (rule arg-cong[OF HSem-merge[OF f1]])
also have ... = [ v ]_{Δ} ρ
  by (rule Let.hyps(4)[OF prem])
finally
show ?case.

have (_{Γ} ρ) f|' (domA Γ) = (_{as} (_{Γ} ρ)) f|' (domA Γ)
  apply (rule ext)
  apply (case-tac x ∈ domA as)
  apply (auto simp add: lookup-HSem-other lookup-env-restr-eq *)
  done
also have ... = (_{as @ Γ} ρ) f|' (domA Γ)
  by (rule arg-cong[OF HSem-merge[OF f1]])
also have ... = (_{Δ} ρ) f|' (domA Γ)
  by (rule env-restr-eq-subset[OF - Let.hyps(5)[OF prem]]) simp
finally
show (_{Γ} ρ) f|' domA Γ = (_{Δ} ρ) f|' domA Γ.
qed

end



## 7.2 CorrectnessResourced


theory CorrectnessResourced
  imports ResourcedDenotational Launchbury
begin

theorem correctness:
  assumes Γ : e ↓L Δ : z
  and fv (Γ, e) ⊆ set L ∪ domA Γ
  shows N[e]_{N}_{Γ} ρ ⊆ N[z]_{N}_{Δ} ρ and (N_{Γ} ρ) f|' domA Γ ⊆ (N_{Δ} ρ) f|' domA Γ
  using assms
proof (nominal-induct arbitrary: ρ rule:reds.strong-induct)
case Lambda
  case 1 show ?case..
  case 2 show ?case..
next
case (Application y Γ e x L Δ Θ z e')
  have Gamma-subset: domA Γ ⊆ domA Δ
  by (rule reds-doesnt-forget[OF Application.hyps(8)])

```

**case 1**  
**hence**  $\text{prem1}: \text{fv}(\Gamma, e) \subseteq \text{set } L \cup \text{domA } \Gamma$  **and**  $x \in \text{set } L \cup \text{domA } \Gamma$  **by** *auto*  
**moreover**  
**note**  $\text{reds-pres-closed}[OF \text{ Application.hyps}(8) \text{ prem1}]$   
**moreover**  
**note**  $\text{reds-doesnt-forget}[OF \text{ Application.hyps}(8)]$   
**moreover**  
**have**  $\text{fv}(e'[y ::= x]) \subseteq \text{fv}(\text{Lam } [y]. e') \cup \{x\}$   
**by** *(auto simp add: fv-subst-eq)*  
**ultimately**  
**have**  $\text{prem2}: \text{fv}(\Delta, e'[y ::= x]) \subseteq \text{set } L \cup \text{domA } \Delta$  **by** *auto*

**have**  $*$ :  $(\mathcal{N}\{\Gamma\}\varrho) x \sqsubseteq (\mathcal{N}\{\Delta\}\varrho) x$   
**proof**(*cases*  $x \in \text{domA } \Gamma$ )  
**case** *True*  
**thus** *?thesis*  
**using**  $\text{fun-belowD}[OF \text{ Application.hyps}(10)[OF \text{ prem1}], \text{ where } \varrho1 = \varrho \text{ and } x = x]$   
**by** *simp*  
**next**  
**case** *False*  
**from**  $\text{False} \langle x \in \text{set } L \cup \text{domA } \Gamma \rangle$   $\text{reds-avoids-live}[OF \text{ Application.hyps}(8)]$   
**show** *?thesis* **by** *(auto simp add: lookup-HSem-other)*  
**qed**

{  
**fix**  $r$   
**have**  $(\mathcal{N}[ \text{App } e x ]_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r \sqsubseteq ((\mathcal{N}[ e ]_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r \downarrow \text{CFn } (\mathcal{N}\{\Gamma\}\varrho) x) \cdot r$   
**by** *(rule CEApp-no-restr)*  
**also have**  $((\mathcal{N}[ e ]_{\mathcal{N}\{\Gamma\}\varrho})) \sqsubseteq ((\mathcal{N}[ \text{Lam } [y]. e' ]_{\mathcal{N}\{\Delta\}\varrho}))$   
**using**  $\text{Application.hyps}(9)[OF \text{ prem1}]$ .  
**also note**  $\langle (\mathcal{N}\{\Gamma\}\varrho) x \sqsubseteq (\mathcal{N}\{\Delta\}\varrho) x \rangle$   
**also have**  $(\mathcal{N}[ \text{Lam } [y]. e' ]_{\mathcal{N}\{\Delta\}\varrho}) \cdot r \sqsubseteq (\text{CFn} \cdot (\Lambda v. (\mathcal{N}[ e' ]_{(\mathcal{N}\{\Delta\}\varrho)(y := v)})))$   
**by** *(rule CELam-no-restr)*  
**also have**  $\text{CFn} \cdot (\Lambda v. (\mathcal{N}[ e' ]_{(\mathcal{N}\{\Delta\}\varrho)(y := v)})) \downarrow \text{CFn } ((\mathcal{N}\{\Delta\}\varrho) x) = (\mathcal{N}[ e' ]_{(\mathcal{N}\{\Delta\}\varrho)(y := (\mathcal{N}\{\Delta\}\varrho) x)})$   
**by** *simp*  
**also have**  $\dots = (\mathcal{N}[ e'[y ::= x] ]_{(\mathcal{N}\{\Delta\}\varrho)})$   
**unfolding** *ESem-subst..*  
**also have**  $\dots \sqsubseteq \mathcal{N}[ z ]_{\mathcal{N}\{\Theta\}\varrho}$   
**using**  $\text{Application.hyps}(12)[OF \text{ prem2}]$ .  
**finally**  
**have**  $(\mathcal{N}[ \text{App } e x ]_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r \sqsubseteq (\mathcal{N}[ z ]_{\mathcal{N}\{\Theta\}\varrho}) \cdot r$  **by** *this (intro cont2cont)+*  
}  
**thus** *?case* **by** *(rule cfun-belowI)*

**show**  $(\mathcal{N}\{\Gamma\}\varrho) f|'(\text{domA } \Gamma) \sqsubseteq (\mathcal{N}\{\Theta\}\varrho) f|'(\text{domA } \Gamma)$   
**using**  $\text{Application.hyps}(10)[OF \text{ prem1}]$

```

      env-restr-below-subset[OF Gamma-subset Application.hyps(13)][OF prem2]]
    by (rule below-trans)
next
case (Variable  $\Gamma$   $x \in L$   $\Delta$   $z$ )
  hence [simp]:  $x \in \text{dom} A \ \Gamma$ 
    by (metis domA-from-set map-of-SomeD)

  case 2

  have  $x \notin \text{dom} A \ \Delta$ 
    by (rule reds-avoids-live[OF Variable.hyps(2)], simp-all)

  have subset:  $\text{dom} A \ (\text{delete } x \ \Gamma) \subseteq \text{dom} A \ \Delta$ 
    by (rule reds-doesnt-forget[OF Variable.hyps(2)])

  let ?new =  $\text{dom} A \ \Delta - \text{dom} A \ \Gamma$ 
  have fv ( $\text{delete } x \ \Gamma, e$ )  $\cup \{x\} \subseteq \text{fv} \ (\Gamma, \text{Var } x)$ 
    by (rule fv-delete-heap[OF map-of  $\Gamma$   $x = \text{Some } e$ ])
  hence prem:  $\text{fv} \ (\text{delete } x \ \Gamma, e) \subseteq \text{set} \ (x \# L) \cup \text{dom} A \ (\text{delete } x \ \Gamma)$  using 2 by auto
  hence fv-subset:  $\text{fv} \ (\text{delete } x \ \Gamma, e) - \text{dom} A \ (\text{delete } x \ \Gamma) \subseteq - \ ?new$ 
    using reds-avoids-live'[OF Variable.hyps(2)] by auto

  have  $\text{dom} A \ \Gamma \subseteq (- \ ?new)$  by auto

  have  $\mathcal{N}\{\Gamma\}_{\varrho} = \mathcal{N}\{(x,e) \# \text{delete } x \ \Gamma\}_{\varrho}$ 
    by (rule HSem-reorder[OF map-of-delete-insert[symmetric, OF Variable(1)]])
  also have ... =  $(\mu \ \varrho'. (\varrho \ ++_{\text{dom} A \ (\text{delete } x \ \Gamma)} (\mathcal{N}\{\text{delete } x \ \Gamma\}_{\varrho'})) (x := \mathcal{N}\llbracket e \rrbracket_{\varrho'}))$ 
    by (rule iterative-HSem, simp)
  also have ... =  $(\mu \ \varrho'. (\varrho \ ++_{\text{dom} A \ (\text{delete } x \ \Gamma)} (\mathcal{N}\{\text{delete } x \ \Gamma\}_{\varrho'})) (x := \mathcal{N}\llbracket e \rrbracket_{\mathcal{N}\{\text{delete } x \ \Gamma\}_{\varrho'}}))$ 
    by (rule iterative-HSem', simp)
  finally
  have  $(\mathcal{N}\{\Gamma\}_{\varrho})f|' \ (- \ ?new) \sqsubseteq (\dots) f|' \ (- \ ?new)$  by (rule ssubst) (rule below-refl)
  also have ...  $\sqsubseteq (\mu \ \varrho'. (\varrho \ ++_{\text{dom} A \ \Delta} (\mathcal{N}\{\Delta\}_{\varrho'})) (x := \mathcal{N}\llbracket z \rrbracket_{\mathcal{N}\{\Delta\}_{\varrho'}})) f|' \ (- \ ?new)$ 
  proof (induction rule: parallel-fix-ind[where  $P = \lambda x y. x f|' \ (- \ ?new) \sqsubseteq y f|' \ (- \ ?new)$ ])
    case 1 show ?case by simp
  next
    case 2 show ?case ..
  next
    case (3  $\sigma \ \sigma'$ )
    hence  $\mathcal{N}\llbracket e \rrbracket_{\mathcal{N}\{\text{delete } x \ \Gamma\}_{\sigma}} \sqsubseteq \mathcal{N}\llbracket e \rrbracket_{\mathcal{N}\{\text{delete } x \ \Gamma\}_{\sigma'}}$ 
      and  $(\mathcal{N}\{\text{delete } x \ \Gamma\}_{\sigma}) f|' \ \text{dom} A \ (\text{delete } x \ \Gamma) \sqsubseteq (\mathcal{N}\{\text{delete } x \ \Gamma\}_{\sigma'}) f|' \ \text{dom} A \ (\text{delete } x \ \Gamma)$ 
      using fv-subset by (auto intro: ESem-fresh-cong-below HSem-fresh-cong-below env-restr-below-subset[OF
- 3])
    from below-trans[OF this(1) Variable(3)][OF prem]] below-trans[OF this(2) Variable(4)][OF
pre]]

```

**have**  $\mathcal{N}\llbracket e \rrbracket_{\mathcal{N}\{\text{delete } x \Gamma\}\sigma} \sqsubseteq \mathcal{N}\llbracket z \rrbracket_{\mathcal{N}\{\Delta\}\sigma'}$   
**and**  $(\mathcal{N}\{\text{delete } x \Gamma\}\sigma) f|' \text{ dom}A (\text{delete } x \Gamma) \sqsubseteq (\mathcal{N}\{\Delta\}\sigma') f|' \text{ dom}A (\text{delete } x \Gamma)$ .  
**thus** *?case*  
**using** *subset*  
**by** (*auto intro!*: *fun-belowI simp add: lookup-override-on-eq lookup-env-restr-eq elim: env-restr-belowD*)  
**qed**  
**also have**  $\dots = (\mu \varrho'. (\varrho \text{ ++ }_{\text{dom}A} \Delta (\mathcal{N}\{\Delta\}\varrho')) (x := \mathcal{N}\llbracket z \rrbracket_{\varrho'})) f|' (-?new)$   
**by** (*rule arg-cong[OF iterative-HSem[symmetric], OF ‹x ∉ domA Δ›]*)  
**also have**  $\dots = (\mathcal{N}\{(x,z) \# \Delta\}\varrho) f|' (-?new)$   
**by** (*rule arg-cong[OF iterative-HSem[symmetric], OF ‹x ∉ domA Δ›]*)  
**finally**  
**show** *le*: *?case* **by** (*rule env-restr-below-subset[OF ‹domA Γ ⊆ (-?new)›]*) (*intro cont2cont*)+  
  
**have**  $\mathcal{N}\llbracket \text{Var } x \rrbracket_{\mathcal{N}\{\Gamma\}\varrho} \sqsubseteq (\mathcal{N}\{\Gamma\}\varrho) x$  **by** (*rule CESem-simps-no-tick*)  
**also have**  $\dots \sqsubseteq (\mathcal{N}\{(x,z) \# \Delta\}\varrho) x$   
**using** *fun-belowD[OF le, where x = x]* **by** *simp*  
**also have**  $\dots = \mathcal{N}\llbracket z \rrbracket_{\mathcal{N}\{(x,z) \# \Delta\}\varrho}$   
**by** (*simp add: lookup-HSem-heap*)  
**finally**  
**show**  $\mathcal{N}\llbracket \text{Var } x \rrbracket_{\mathcal{N}\{\Gamma\}\varrho} \sqsubseteq \mathcal{N}\llbracket z \rrbracket_{\mathcal{N}\{(x,z) \# \Delta\}\varrho}$  **by** *this* (*intro cont2cont*)+  
**next**  
**case** (*Bool b*)  
**case 1**  
**show** *?case* **by** *simp*  
**case 2**  
**show** *?case* **by** *simp*  
**next**  
**case** (*IfThenElse Γ scrut L Δ b e<sub>1</sub> e<sub>2</sub> Θ z*)  
**have** *Gamma-subset*:  $\text{dom}A \Gamma \subseteq \text{dom}A \Delta$   
**by** (*rule reds-doesnt-forget[OF IfThenElse.hyps(1)]*)  
  
**let** *?e* = *if b then e<sub>1</sub> else e<sub>2</sub>*  
  
**case 1**  
  
**hence** *prem1*:  $\text{fv}(\Gamma, \text{scrut}) \subseteq \text{set } L \cup \text{dom}A \Gamma$   
**and** *prem2*:  $\text{fv}(\Delta, ?e) \subseteq \text{set } L \cup \text{dom}A \Delta$   
**and**  $\text{fv } ?e \subseteq \text{dom}A \Gamma \cup \text{set } L$   
**using** *new-free-vars-on-heap[OF IfThenElse.hyps(1)] Gamma-subset* **by** *auto*  
  
**{**  
**fix** *r*  
**have**  $(\mathcal{N}\llbracket (\text{scrut } ?e_1 : e_2) \rrbracket_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r \sqsubseteq \text{CB-project} \cdot ((\mathcal{N}\llbracket \text{scrut} \rrbracket_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r) \cdot ((\mathcal{N}\llbracket e_1 \rrbracket_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r) \cdot ((\mathcal{N}\llbracket e_2 \rrbracket_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r)$   
**by** (*rule CESem-simps-no-tick*)  
**also have**  $\dots \sqsubseteq \text{CB-project} \cdot ((\mathcal{N}\llbracket \text{Bool } b \rrbracket_{\mathcal{N}\{\Delta\}\varrho}) \cdot r) \cdot ((\mathcal{N}\llbracket e_1 \rrbracket_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r) \cdot ((\mathcal{N}\llbracket e_2 \rrbracket_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r)$   
**by** (*intro monofun-cfun-fun monofun-cfun-arg IfThenElse.hyps(2)[OF prem1]*)  
**}**

**also have**  $\dots = (\mathcal{N}[\ ?e \ ]_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r$  **by** (*cases r simp-all*)  
**also have**  $\dots \sqsubseteq (\mathcal{N}[\ ?e \ ]_{\mathcal{N}\{\Delta\}\varrho}) \cdot r$   
**proof**(*rule monofun-cfun-fun[OF ESem-fresh-cong-below-subset[OF <fv ?e ⊆ domA Γ ∪ set L> Env.env-restr-belowI]]*)  
  **fix**  $x$   
  **assume**  $x \in \text{domA } \Gamma \cup \text{set } L$   
  **thus**  $(\mathcal{N}\{\Gamma\}\varrho) x \sqsubseteq (\mathcal{N}\{\Delta\}\varrho) x$   
  **proof**(*cases x ∈ domA Γ*)  
    **assume**  $x \in \text{domA } \Gamma$   
    **from** *IfThenElse.hyps(3)[OF prem1]*  
    **have**  $(\mathcal{N}\{\Gamma\}\varrho) f|' \text{ domA } \Gamma x \sqsubseteq ((\mathcal{N}\{\Delta\}\varrho) f|' \text{ domA } \Gamma) x$  **by** (*rule fun-belowD*)  
    **with**  $\langle x \in \text{domA } \Gamma \rangle$  **show** *?thesis* **by** *simp*  
  **next**  
  **assume**  $x \notin \text{domA } \Gamma$   
  **from** *this <x ∈ domA Γ ∪ set L> reds-avoids-live[OF IfThenElse.hyps(1)]*  
  **show** *?thesis*  
  **by** (*simp add: lookup-HSem-other*)  
  **qed**  
  **qed**  
**also have**  $\dots \sqsubseteq (\mathcal{N}[ z \ ]_{\mathcal{N}\{\Theta\}\varrho}) \cdot r$   
  **by** (*intro monofun-cfun-fun monofun-cfun-arg IfThenElse.hyps(5)[OF prem2]*)  
**finally**  
**have**  $(\mathcal{N}[\ (\text{scrut } ? e_1 : e_2) \ ]_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r \sqsubseteq (\mathcal{N}[ z \ ]_{\mathcal{N}\{\Theta\}\varrho}) \cdot r$  **by** *this (intro cont2cont)+*  
**}**  
**thus** *?case* **by** (*rule cfun-belowI*)  
  
**show**  $(\mathcal{N}\{\Gamma\}\varrho) f|' (\text{domA } \Gamma) \sqsubseteq (\mathcal{N}\{\Theta\}\varrho) f|' (\text{domA } \Gamma)$   
  **using** *IfThenElse.hyps(3)[OF prem1]*  
  *env-restr-below-subset[OF Gamma-subset IfThenElse.hyps(6)[OF prem2]]*  
  **by** (*rule below-trans*)  
**next**  
**case** (*Let as Γ L body Δ z*)  
  **case 1**  
  **have**  $*$ :  $\text{domA } as \cap \text{domA } \Gamma = \{\}$  **by** (*metis Let.hyps(1) fresh-distinct*)  
  
  **have**  $\text{fv } (as @ \Gamma, \text{body}) - \text{domA } (as @ \Gamma) \subseteq \text{fv } (\Gamma, \text{Let } as \text{ body}) - \text{domA } \Gamma$   
  **by** *auto*  
  **with 1** **have** *prem*:  $\text{fv } (as @ \Gamma, \text{body}) \subseteq \text{set } L \cup \text{domA } (as @ \Gamma)$  **by** *auto*  
  
  **have**  $f1$ :  $\text{atom } ' \text{domA } as \ \# * \ \Gamma$   
  **using** *Let(1)* **by** (*simp add: set-bn-to-atom-domA*)  
  
  **have**  $\mathcal{N}[\ \text{Let } as \ \text{body} \ ]_{\mathcal{N}\{\Gamma\}\varrho} \sqsubseteq \mathcal{N}[\ \text{body} \ ]_{\mathcal{N}\{as\}\mathcal{N}\{\Gamma\}\varrho}$   
  **by** (*rule CESem-simps-no-tick*)  
  **also have**  $\dots = \mathcal{N}[\ \text{body} \ ]_{\mathcal{N}\{as @ \Gamma\}\varrho}$   
  **by** (*rule arg-cong[OF HSem-merge[OF f1]]*)  
  **also have**  $\dots \sqsubseteq \mathcal{N}[ z \ ]_{\mathcal{N}\{\Delta\}\varrho}$   
  **by** (*rule Let.hyps(4)[OF prem]*)

**finally**  
**show**  $?case$  **by**  $this$   $(intro\ cont2cont)+$   
  
**have**  $(\mathcal{N}\{\Gamma\}\varrho) f|' (domA\ \Gamma) = (\mathcal{N}\{as\}(\mathcal{N}\{\Gamma\}\varrho)) f|' (domA\ \Gamma)$   
**unfolding**  $env-restr-HSem[OF\ *]..$   
**also have**  $\mathcal{N}\{as\}(\mathcal{N}\{\Gamma\}\varrho) = (\mathcal{N}\{as\ @\ \Gamma\}\varrho)$   
**by**  $(rule\ HSem-merge[OF\ f1])$   
**also have**  $\dots f|' domA\ \Gamma \sqsubseteq (\mathcal{N}\{\Delta\}\varrho) f|' domA\ \Gamma$   
**by**  $(rule\ env-restr-below-subset[OF\ -\ Let.hyps(5)[OF\ prem]])\ simp$   
**finally**  
**show**  $(\mathcal{N}\{\Gamma\}\varrho) f|' domA\ \Gamma \sqsubseteq (\mathcal{N}\{\Delta\}\varrho) f|' domA\ \Gamma.$   
**qed**

**corollary**  $correctness-empty-env:$

**assumes**  $\Gamma : e \Downarrow_L \Delta : z$   
**and**  $fv(\Gamma, e) \subseteq set\ L$   
**shows**  $\mathcal{N}[e]_{\mathcal{N}\{\Gamma\}} \sqsubseteq \mathcal{N}[z]_{\mathcal{N}\{\Delta\}}$  **and**  $\mathcal{N}\{\Gamma\} \sqsubseteq \mathcal{N}\{\Delta\}$

**proof**–

**from**  $assms(2)$  **have**  $fv(\Gamma, e) \subseteq set\ L \cup domA\ \Gamma$  **by**  $auto$   
**note**  $corr = correctness[OF\ assms(1)\ this, where\ \varrho = \perp]$

**show**  $\mathcal{N}[e]_{\mathcal{N}\{\Gamma\}} \sqsubseteq \mathcal{N}[z]_{\mathcal{N}\{\Delta\}}$  **using**  $corr(1).$

**have**  $\mathcal{N}\{\Gamma\} = (\mathcal{N}\{\Gamma\}) f|' domA\ \Gamma$   
**using**  $env-restr-useless[OF\ HSem-edom-subset, where\ \varrho1 = \perp]$  **by**  $simp$   
**also have**  $\dots \sqsubseteq (\mathcal{N}\{\Delta\}) f|' domA\ \Gamma$  **using**  $corr(2).$   
**also have**  $\dots \sqsubseteq \mathcal{N}\{\Delta\}$  **by**  $(rule\ env-restr-below-itself)$   
**finally show**  $\mathcal{N}\{\Gamma\} \sqsubseteq \mathcal{N}\{\Delta\}$  **by**  $this\ (intro\ cont2cont)+$

**qed**

**end**

## 8 Equivalence of the denotational semantics

### 8.1 ValueSimilarity

```
theory ValueSimilarity
imports Value CValue Pointwise
begin
```

This theory formalizes Section 3 of [SGHHOM11]. Their domain  $D$  is our type  $Value$ , their domain  $E$  is our type  $CValue$  and  $A$  corresponds to  $C \rightarrow CValue$ .

In our case, the construction of the domains was taken care of by the HOLCF package ([Huf12]), so where [SGHHOM11] refers to elements of the domain approximations  $D_n$  resp.  $E_n$ , these are just elements of  $Value$  resp.  $CValue$  here. Therefore the  $n$ -injection  $\phi_n^E: E_n \rightarrow E$  is the identity here.

The projections correspond to the take-functions generated by the HOLCF package:

$$\begin{aligned} \psi_n^E: E \rightarrow E_n & \text{ corresponds to } & CValue\text{-take}::nat \Rightarrow CValue \rightarrow CValue \\ \psi_n^A: A \rightarrow A_n & \text{ corresponds to } & C\text{-to-}CValue\text{-take}::nat \Rightarrow (C \rightarrow CValue) \rightarrow C \rightarrow CValue \\ \psi_n^D: D \rightarrow D_n & \text{ corresponds to } & Value\text{-take}::nat \Rightarrow Value \rightarrow Value. \end{aligned}$$

The syntactic overloading of  $e(a)(c)$  to mean either  $Ap_{E_n}^\perp$  or  $AP_E^\perp$  turns into our non-overloaded  $\downarrow CFn \text{ -}:: CValue \Rightarrow (C \rightarrow CValue) \Rightarrow C \rightarrow CValue$ .

To have our presentation closer to [SGHHOM11], we introduce some notation:

```
notation Value-take ( $\psi^D \_ \_$ )
notation C-to-CValue-take ( $\psi^A \_ \_$ )
notation CValue-take ( $\psi^E \_ \_$ )
```

#### 8.1.1 A note about section 2.3

Section 2.3 of [SGHHOM11] contains equations (2) and (3) which do not hold in general. We demonstrate that fact here using our corresponding definition, but the counter-example carries over to the original formulation. Lemma (2) is a generalisation of (3) to the resourced semantics, so the counter-example for (3) is the simpler and more educating:

**lemma counter-example:**

```
assumes Equation (3):  $\bigwedge n d d'. \psi_n^D.(d \downarrow Fn d') = \psi_{Suc\ n}^D.d \downarrow Fn \psi_n^D.d'$ 
shows False
```

**proof** –

```
define n :: nat where n = 1
define d where d = Fn.( $\Lambda e. (e \downarrow Fn \perp)$ )
define d' where d' = Fn.( $\Lambda -. Fn.( \Lambda -. \perp)$ )
have Fn.( $\Lambda -. \perp$ ) =  $\psi_n^D.(d \downarrow Fn d')$ 
```

```

  by (simp add: d-def d'-def n-def cfun-map-def)
also
have ... =  $\psi^D_{Suc\ n} \cdot d \downarrow Fn \psi^D_n \cdot d'$ 
  using Equation (3).
also have ... =  $\perp$ 
  by (simp add: d-def d'-def n-def)
finally show False by simp
qed

```

For completeness, and to avoid making false assertions, the counter-example to equation (2):

**lemma** *counter-example2*:

```

  assumes Equation (2):  $\bigwedge n\ e\ a\ c. \psi^E_n \cdot ((e \downarrow CFn\ a) \cdot c) = (\psi^E_{Suc\ n} \cdot e \downarrow CFn\ \psi^A_n \cdot a) \cdot c$ 
  shows False

```

**proof** –

```

  define n :: nat where n = 1
  define e where e = CFn (λ e r. (e · r ↓ CFn ⊥) · r)
  define a :: C → CValue where a = (λ -. CFn (λ -. CFn (λ -. ⊥)))
  fix c :: C
  have CFn (λ -. ⊥) =  $\psi^E_n \cdot ((e \downarrow CFn\ a) \cdot c)$ 
    by (simp add: e-def a-def n-def cfun-map-def)
  also
  have ... =  $(\psi^E_{Suc\ n} \cdot e \downarrow CFn\ \psi^A_n \cdot a) \cdot c$ 
    using Equation (2).
  also have ... =  $\perp$ 
    by (simp add: e-def a-def n-def)
  finally show False by simp

```

**qed**

A suitable substitute for the lemma can be found in 4.3.5 (1) in [AO93], which in our setting becomes the following (note the extra invocation of  $\psi^D_n$  on the left hand side):

**lemma** *Abramsky 4,3,5 (1)*:

```

 $\psi^D_n \cdot (d \downarrow Fn \psi^D_n \cdot d') = \psi^D_{Suc\ n} \cdot d \downarrow Fn \psi^D_n \cdot d'$ 
  by (cases d) (auto simp add: Value.take-take)

```

The problematic equations are used in the proof of the only-if direction of proposition 9 in [SGHHOM11]. It can be fixed by applying take-induction, which inserts the extra call to  $\psi^D_n$  in the right spot.

### 8.1.2 Working with Value and CValue

Combined case distinguishing and induction rules.

**lemma** *value-CValue-cases*:

```

  obtains
  x =  $\perp$  y =  $\perp$  |
  f where x = Fn · f y =  $\perp$  |

```

$g$  where  $x = \perp$   $y = CFn.g$  |  
 $f g$  where  $x = Fn.f y = CFn \cdot g$  |  
 $b_1$  where  $x = B.(Discr b_1)$   $y = \perp$  |  
 $b_1 g$  where  $x = B.(Discr b_1)$   $y = CFn.g$  |  
 $b_1 b_2$  where  $x = B.(Discr b_1)$   $y = CB.(Discr b_2)$  |  
 $f b_2$  where  $x = Fn.f y = CB.(Discr b_2)$  |  
 $b_2$  where  $x = \perp$   $y = CB.(Discr b_2)$   
**by** (*metis CValue.exhaust Discr-undiscr Value.exhaust*)

**lemma** *Value-CValue-take-induct*:

**assumes** *adm* (*case-prod P*)  
**assumes**  $\bigwedge n. P (\psi^D n.x) (\psi^A n.y)$   
**shows**  $P x y$

**proof**–

**have** *case-prod P* ( $\bigsqcup n. (\psi^D n.x, \psi^A n.y)$ )

**by** (*rule admD[OF ‹adm (case-prod P)› ch2ch-Pair[OF ch2ch-Rep-cfunL[OF Value.chain-take] ch2ch-Rep-cfunL[OF C-to-CValue-chain-take]]]*)

(*simp add: assms(2)*)

**hence** *case-prod P* ( $x, y$ )

**by** (*simp add: lub-Pair[OF ch2ch-Rep-cfunL[OF Value.chain-take] ch2ch-Rep-cfunL[OF C-to-CValue-chain-take]]*)

*Value.reach C-to-CValue-reach*)

**thus** *?thesis* **by** *simp*

**qed**

### 8.1.3 Restricted similarity is defined recursively

The base case

**inductive** *similar'-base* ::  $Value \Rightarrow CValue \Rightarrow bool$  **where**  
*bot-similar'-base*[*simp,intro*]: *similar'-base*  $\perp \perp$

**inductive-cases** [*elim!*]:  
*similar'-base*  $x y$

The inductive case

**inductive** *similar'-step* ::  $(Value \Rightarrow CValue \Rightarrow bool) \Rightarrow Value \Rightarrow CValue \Rightarrow bool$  **for**  $s$  **where**  
*bot-similar'-step*[*intro!*]: *similar'-step*  $s \perp \perp$  |  
*bool-similar'-step*[*intro*]: *similar'-step*  $s (B \cdot b) (CB \cdot b)$  |  
*Fun-similar'-step*[*intro*]:  $(\bigwedge x y. s x (y \cdot C^\infty) \implies s (f \cdot x) (g \cdot y \cdot C^\infty)) \implies$  *similar'-step*  $s (Fn \cdot f)$   
 $(CFn \cdot g)$

**inductive-cases** [*elim!*]:  
*similar'-step*  $s x \perp$   
*similar'-step*  $s \perp y$   
*similar'-step*  $s (B \cdot f) (CB \cdot g)$   
*similar'-step*  $s (Fn \cdot f) (CFn \cdot g)$

We now create the restricted similarity relation, by primitive recursion over  $n$ .

This cannot be done using an inductive definition, as it would not be monotone.

**fun** *similar'* **where**

*similar'* 0 = *similar'*-base |  
*similar'* (Suc n) = *similar'*-step (*similar'* n)  
**declare** *similar'*.simps[simp del]

**abbreviation** *similar'*-syn ( $\langle - \rangle_n$  -> [50,50,50] 50)  
**where** *similar'*-syn  $x$   $n$   $y \equiv similar' n x y$

**lemma** *similar'*-botI[*intro!*,*simp*]:  $\perp \langle_n \perp$   
**by** (*cases* n) (*auto simp add: similar'.simps*)

**lemma** *similar'*-FnI[*intro!*]:  
**assumes**  $\bigwedge x y. x \langle_n y \cdot C^\infty \implies f \cdot x \langle_n g \cdot y \cdot C^\infty$   
**shows**  $Fn \cdot f \langle_{Suc\ n} CFn \cdot g$   
**using** *assms* **by** (*auto simp add: similar'.simps*)

**lemma** *similar'*-FnE[*elim!*]:  
**assumes**  $Fn \cdot f \langle_{Suc\ n} CFn \cdot g$   
**assumes**  $(\bigwedge x y. x \langle_n y \cdot C^\infty \implies f \cdot x \langle_n g \cdot y \cdot C^\infty) \implies P$   
**shows**  $P$   
**using** *assms* **by** (*auto simp add: similar'.simps*)

**lemma** *bot-or-not-bot'*:  
 $x \langle_n y \implies (x = \perp \longleftrightarrow y = \perp)$   
**by** (*cases* n) (*auto simp add: similar'.simps elim: similar'-base.cases similar'-step.cases*)

**lemma** *similar'*-bot[*elim-format*, *elim!*]:  
 $\perp \langle_n x \implies x = \perp$   
 $y \langle_n \perp \implies y = \perp$   
**by** (*metis bot-or-not-bot'*)**+**

**lemma** *similar'*-typed[*simp*]:  
 $\neg B \cdot b \langle_n CFn \cdot g$   
 $\neg Fn \cdot f \langle_n CB \cdot b$   
**by** (*cases* n, *auto simp add: similar'.simps elim: similar'-base.cases similar'-step.cases*)**+**

**lemma** *similar'*-bool[*simp*]:  
 $B \cdot b_1 \langle_{Suc\ n} CB \cdot b_2 \longleftrightarrow b_1 = b_2$   
**by** (*auto simp add: similar'.simps elim: similar'-base.cases similar'-step.cases*)

### 8.1.4 Moving up and down the similarity relations

These correspond to Lemma 7 in [SGHHOM11].

**lemma** *similar'*-down:  $d \langle_{Suc\ n} e \implies \psi^D_n \cdot d \langle_n \psi^E_n \cdot e$   
**and** *similar'*-up:  $d \langle_n e \implies \psi^D_n \cdot d \langle_{Suc\ n} \psi^E_n \cdot e$   
**proof** (*induction* n *arbitrary: d e*)  
**case** (Suc n) **case 1 with** *Suc*

```

show ?case
  by (cases d e rule:value-CValue-cases) auto
next
  case (Suc n) case 2 with Suc
  show ?case
  by (cases d e rule:value-CValue-cases) auto
qed auto

```

A generalisation of the above, doing multiple steps at once.

```

lemma similar'-up-le:  $n \leq m \implies \psi^D_n \cdot d \triangleleft_n \psi^E_n \cdot e \implies \psi^D_n \cdot d \triangleleft_m \psi^E_n \cdot e$ 
  by (induction rule: dec-induct )
  (auto dest: similar'-up simp add: Value.take-take CValue.take-take min-absorb2)

```

```

lemma similar'-down-le:  $n \leq m \implies \psi^D_m \cdot d \triangleleft_m \psi^E_m \cdot e \implies \psi^D_n \cdot d \triangleleft_n \psi^E_n \cdot e$ 
  by (induction rule: inc-induct )
  (auto dest: similar'-down simp add: Value.take-take CValue.take-take min-absorb1)

```

```

lemma similar'-take:  $d \triangleleft_n e \implies \psi^D_n \cdot d \triangleleft_n \psi^E_n \cdot e$ 
  apply (drule similar'-up)
  apply (drule similar'-down)
  apply (simp add: Value.take-take CValue.take-take)
done

```

### 8.1.5 Admissibility

A technical prerequisite for induction is admissibility of the predicate, i.e. that the predicate holds for the limit of a chain, given that it holds for all elements.

```

lemma similar'-base-adm: adm ( $\lambda x. \text{similar}'\text{-base } (fst\ x) (snd\ x)$ )
proof (rule admI, goal-cases)
  case (1 Y)
  then have  $Y = (\lambda \cdot . \perp)$  by (metis prod.exhaust fst-eqD inst-prod-pcpo similar'-base.simps snd-eqD)
  thus ?case by auto
qed

```

```

lemma similar'-step-adm:
  assumes adm ( $\lambda x. s\ (fst\ x)\ (snd\ x)$ )
  shows adm ( $\lambda x. \text{similar}'\text{-step } s\ (fst\ x)\ (snd\ x)$ )
proof (rule admI, goal-cases)
  case prems: (1 Y)
  from ⟨chain Y⟩
  have chain ( $\lambda i. \text{fst } (Y\ i)$ ) by (rule ch2ch-fst)
  thus ?case
  proof(cases rule: Value-chainE)
  case bot
  hence *:  $\bigwedge i. \text{fst } (Y\ i) = \perp$  by metis
  with prems(2)[unfolded split-beta]
  have  $\bigwedge i. \text{snd } (Y\ i) = \perp$  by auto

```

```

hence  $Y = (\lambda i. (\perp, \perp))$  using * by (metis surjective-pairing)
thus ?thesis by auto
next
case (B n b)
hence  $\forall i. \text{fst } (Y (i + n)) = B \cdot b$  by (metis add.commute not-add-less1)
with prems(2)
have  $\forall i. Y (i + n) = (B \cdot b, CB \cdot b)$ 
  apply auto
  apply (erule-tac  $x = i + n$  in allE)
  apply (erule-tac  $x = i$  in allE)
  apply (erule similar'-step.cases)
  apply auto
  by (metis fst-conv old.prod.exhaust snd-conv)
hence similar'-step s (fst ( $\sqcup i. Y (i + n)$ )) (snd ( $\sqcup i. Y (i + n)$ )) by auto
thus ?thesis
  by (simp add: lub-range-shift[OF ‹chain Y›])
next
fix n
fix Y'
assume chain Y' and ( $\lambda i. \text{fst } (Y i) = (\lambda m. (\text{if } m < n \text{ then } \perp \text{ else } \text{Fn} \cdot (Y' (m - n))))$ )
hence  $Y': \bigwedge i. \text{fst } (Y (i + n)) = \text{Fn} \cdot (Y' i)$  by (metis add-diff-cancel-right' not-add-less2)
with prems(2)[unfolded split-beta]
have  $\bigwedge i. \exists g'. \text{snd } (Y (i + n)) = C\text{Fn} \cdot g'$ 
  by -(erule-tac  $x = i + n$  in allE, auto elim!: similar'-step.cases)
then obtain  $Y''$  where  $Y'': \bigwedge i. \text{snd } (Y (i + n)) = C\text{Fn} \cdot (Y'' i)$  by metis
from prems(1) have  $\bigwedge i. Y i \sqsubseteq Y (\text{Suc } i)$ 
  by (simp add: po-class.chain-def)
then have *:  $\bigwedge i. Y (i + n) \sqsubseteq Y (\text{Suc } i + n)$ 
  by simp
have chain Y''
  apply (rule chainI)
  apply (rule iffD1[OF CValue.inverts(1)])
  apply (subst (1 2) Y''[symmetric])
  apply (rule snd-monofun)
  apply (rule *)
done

have similar'-step s (Fn · ( $\sqcup i. (Y' i)$ )) (C Fn · ( $\sqcup i. Y'' i$ ))
proof (rule Fun-similar'-step)
fix x y
from prems(2) Y' Y''
have  $\bigwedge i. \text{similar}'\text{-step } s (\text{Fn} \cdot (Y' i)) (C\text{Fn} \cdot (Y'' i))$  by metis
moreover
assume s x (y · C∞)
ultimately
have  $\bigwedge i. s (Y' i \cdot x) (Y'' i \cdot y \cdot C^\infty)$  by auto
hence case-prod s ( $\sqcup i. ((Y' i) \cdot x, (Y'' i) \cdot y \cdot C^\infty)$ )
  apply -
  apply (rule admD[OF adm-case-prod[where P = λ · . s, OF assms]])

```

```

apply (simp add: ‹chain Y'› ‹chain Y''›)
apply simp
done
thus s (( $\sqcup$  i. Y' i)·x) (( $\sqcup$  i. Y'' i)·y·C∞)
  by (simp add: lub-Pair ch2ch-Rep-cfunL contlub-cfun-fun ‹chain Y'› ‹chain Y''›)
qed
hence similar'-step s (fst ( $\sqcup$  i. Y (i+n))) (snd ( $\sqcup$  i. Y (i+n)))
  by (simp add: Y' Y''
    cont2contlubE[OF cont-fst chain-shift[OF prems(1)]] cont2contlubE[OF cont-snd
chain-shift[OF prems(1)]]
    contlub-cfun-arg[OF ‹chain Y''›] contlub-cfun-arg[OF ‹chain Y'›])
thus similar'-step s (fst ( $\sqcup$  i. Y i)) (snd ( $\sqcup$  i. Y i))
  by (simp add: lub-range-shift[OF ‹chain Y'›])
qed
qed

```

**lemma** similar'-adm: adm ( $\lambda x$ . fst x  $\triangleleft_n$  snd x)  
**by** (induct n) (auto simp add: similar'.simps intro: similar'-base-adm similar'-step-adm)

**lemma** similar'-admI: cont f  $\implies$  cont g  $\implies$  adm ( $\lambda x$ . f x  $\triangleleft_n$  g x)  
**by** (rule adm-subst[OF - similar'-adm, where t =  $\lambda x$ . (f x, g x), simplified]) auto

### 8.1.6 The real similarity relation

This is the goal of the theory: A relation between *Value* and *CValue*.

**definition** similar :: Value  $\Rightarrow$  CValue  $\Rightarrow$  bool (**infix** ‹ $\triangleleft$ › 50) **where**  
 $x \triangleleft y \iff (\forall n. \psi^D_n \cdot x \triangleleft_n \psi^E_n \cdot y)$

**lemma** similarI:  
 $(\bigwedge n. \psi^D_n \cdot x \triangleleft_n \psi^E_n \cdot y) \implies x \triangleleft y$   
**unfolding** similar-def **by** blast

**lemma** similarE:  
 $x \triangleleft y \implies \psi^D_n \cdot x \triangleleft_n \psi^E_n \cdot y$   
**unfolding** similar-def **by** blast

**lemma** similar-bot[simp]:  $\perp \triangleleft \perp$  **by** (auto intro: similarI)

**lemma** similar-bool[simp]:  $B \cdot b \triangleleft CB \cdot b$   
**by** (rule similarI, case-tac n, auto)

**lemma** [elim-format, elim!]:  $x \triangleleft \perp \implies x = \perp$   
**unfolding** similar-def  
**apply** (cases x)  
**apply** auto  
**apply** (erule-tac x = Suc 0 in allE, auto)+  
**done**

**lemma** [*elim-format, elim!*]:  $x \triangleleft CB \cdot b \implies x = B \cdot b$   
**unfolding** *similar-def*  
**apply** (*cases x*)  
**apply** *auto*  
**apply** (*erule-tac x = Suc 0 in allE, auto*)  
**done**

**lemma** [*elim-format, elim!*]:  $\perp \triangleleft y \implies y = \perp$   
**unfolding** *similar-def*  
**apply** (*cases y*)  
**apply** *auto*  
**apply** (*erule-tac x = Suc 0 in allE, auto*)  
**done**

**lemma** [*elim-format, elim!*]:  $B \cdot b \triangleleft y \implies y = CB \cdot b$   
**unfolding** *similar-def*  
**apply** (*cases y*)  
**apply** *auto*  
**apply** (*erule-tac x = Suc 0 in allE, auto*)  
**done**

**lemma** *take-similar'-similar*:

**assumes**  $x \triangleleft_n y$   
**shows**  $\psi^D_n \cdot x \triangleleft \psi^E_n \cdot y$   
**proof**(*rule similarI*)  
**fix**  $m$   
**from** *assms*  
**have**  $\psi^D_n \cdot x \triangleleft_n \psi^E_n \cdot y$  **by** (*rule similar'-take*)  
**moreover**  
**have**  $n \leq m \vee m \leq n$  **by** *auto*  
**ultimately**  
**show**  $\psi^D_m \cdot (\psi^D_n \cdot x) \triangleleft_m \psi^E_m \cdot (\psi^E_n \cdot y)$   
**by** (*auto elim: similar'-up-le similar'-down-le dest: similar'-take*  
*simp add: min-absorb2 min-absorb1 Value.take-take CValue.take-take*)  
**qed**

**lemma** *bot-or-not-bot*:

$x \triangleleft y \implies (x = \perp \longleftrightarrow y = \perp)$   
**by** (*cases x y rule:value-CValue-cases*) *auto*

**lemma** *bool-or-not-bool*:

$x \triangleleft y \implies (x = B \cdot b \longleftrightarrow (y = CB \cdot b))$   
**by** (*cases x y rule:value-CValue-cases*) *auto*

**lemma** *similar-bot-cases*[*consumes 1, case-names bot bool Fn*]:

**assumes**  $x \triangleleft y$   
**obtains**  $x = \perp \ y = \perp$  |  
**b where**  $x = B \cdot (Discr\ b) \ y = CB \cdot (Discr\ b)$  |

*f g* where  $x = Fn.f y = CFn \cdot g$   
**using** *assms*  
**by** (*metis CValue.exhaust Value.exhaust bool-or-not-bool bot-or-not-bot discr.exhaust*)

**lemma** *similar-adm*:  $adm (\lambda x. fst x \triangleleft snd x)$   
**unfolding** *similar-def*  
**by** (*intro adm-lemmas similar'-admI cont2cont*)

**lemma** *similar-admI*:  $cont f \implies cont g \implies adm (\lambda x. f x \triangleleft g x)$   
**by** (*rule adm-subst[OF - similar-adm, where t =  $\lambda x. (f x, g x)$ , simplified]*) *auto*

Having constructed the relation we can now show that it indeed is the desired relation, relating  $\perp$  with  $\perp$  and functions with functions, if they take related arguments to related values. This corresponds to Proposition 9 in [SGHHOM11].

**lemma** *similar-nice-def*:  $x \triangleleft y \iff (x = \perp \wedge y = \perp \vee (\exists b. x = B \cdot (Discr b) \wedge y = CB \cdot (Discr b)) \vee (\exists f g. x = Fn.f \wedge y = CFn.g \wedge (\forall a b. a \triangleleft b \cdot C^\infty \longrightarrow f \cdot a \triangleleft g \cdot b \cdot C^\infty)))$   
**(is ?L  $\iff$  ?R)**

**proof**

**assume** *?L*

**thus** *?R*

**proof** (*cases x y rule:similar-bot-cases*)

**case** *bot* **thus** *?thesis* **by** *simp*

**next**

**case** *bool* **thus** *?thesis* **by** *simp*

**next**

**case** (*Fn f g*)

**note**  $\langle ?L \rangle$  [*unfolded Fn*]

**have**  $\forall a b. a \triangleleft b \cdot C^\infty \longrightarrow f \cdot a \triangleleft g \cdot b \cdot C^\infty$

**proof** (*intro impI allI*)

**fix** *a b*

**assume**  $a \triangleleft b \cdot C^\infty$

**show**  $f \cdot a \triangleleft g \cdot b \cdot C^\infty$

**proof** (*rule similarI*)

**fix** *n*

**have**  $adm (\lambda (b, a). \psi^D_n \cdot (f \cdot b) \triangleleft_n \psi^E_n \cdot (g \cdot a \cdot C^\infty))$

**by** (*intro adm-case-prod similar'-admI cont2cont*)

**thus**  $\psi^D_n \cdot (f \cdot a) \triangleleft_n \psi^E_n \cdot (g \cdot b \cdot C^\infty)$

**proof** (*induct a b rule: Value-CValue-take-induct[consumes 1]*)

This take induction is required to avoid the wrong equation shown above.

**fix** *m*

**from**  $\langle a \triangleleft b \cdot C^\infty \rangle$

**have**  $\psi^D_m \cdot a \triangleleft_m \psi^E_m \cdot (b \cdot C^\infty)$  **by** (*rule similarE*)

**hence**  $\psi^D_m \cdot a \triangleleft_{\max m n} \psi^E_m \cdot (b \cdot C^\infty)$  **by** (*rule similar'-up-le[rotated]*) *auto*

**moreover**

**from**  $\langle Fn.f \triangleleft CFn.g \rangle$

```

      have  $\psi^D_{\text{Suc } (max\ m\ n)} \cdot (Fn \cdot f) \triangleleft_{\text{Suc } (max\ m\ n)} \psi^E_{\text{Suc } (max\ m\ n)} \cdot (CFn \cdot g)$  by (rule
similarE)
    ultimately
      have  $\psi^D_{max\ m\ n} \cdot (f \cdot (\psi^D_{max\ m\ n} \cdot (\psi^D_m \cdot a))) \triangleleft_{max\ m\ n} \psi^E_{max\ m\ n} \cdot (g \cdot (\psi^A_{max\ m\ n} \cdot (\psi^A_m \cdot b))) \cdot C^\infty$ 
      by auto
      hence  $\psi^D_{max\ m\ n} \cdot (f \cdot (\psi^D_m \cdot a)) \triangleleft_{max\ m\ n} \psi^E_{max\ m\ n} \cdot (g \cdot (\psi^A_m \cdot b)) \cdot C^\infty$ 
      by (simp add: Value.take-take cfun-map-map CValue.take-take ID-def eta-cfun
min-absorb2 min-absorb1)
      thus  $\psi^D_n \cdot (f \cdot (\psi^D_m \cdot a)) \triangleleft_n \psi^E_n \cdot (g \cdot (\psi^A_m \cdot b)) \cdot C^\infty$ 
      by (rule similar'-down-le[rotated]) auto
    qed
  qed
  qed
  thus ?thesis unfolding Fn by simp
qed
next
assume ?R
thus ?L
proof (elim conjE disjE exE ssubst)
  show  $\perp \triangleleft \perp$  by simp
next
  fix b
  show  $B \cdot (Discr\ b) \triangleleft CB \cdot (Discr\ b)$  by simp
next
  fix f g
  assume imp:  $\forall a\ b. a \triangleleft b \cdot C^\infty \longrightarrow f \cdot a \triangleleft g \cdot b \cdot C^\infty$ 
  show  $Fn \cdot f \triangleleft CFn \cdot g$ 
  proof (rule similarI)
    fix n
    show  $\psi^D_n \cdot (Fn \cdot f) \triangleleft_n \psi^E_n \cdot (CFn \cdot g)$ 
    proof (cases n)
      case 0 thus ?thesis by simp
    next
      case (Suc n)
      { fix x y
        assume  $x \triangleleft_n y \cdot C^\infty$ 
        hence  $\psi^D_n \cdot x \triangleleft \psi^E_n \cdot (y \cdot C^\infty)$  by (rule take-similar'-similar)
        hence  $f \cdot (\psi^D_n \cdot x) \triangleleft g \cdot (\psi^A_n \cdot y) \cdot C^\infty$  using imp by auto
        hence  $\psi^D_n \cdot (f \cdot (\psi^D_n \cdot x)) \triangleleft_n \psi^E_n \cdot (g \cdot (\psi^A_n \cdot y)) \cdot C^\infty$ 
        by (rule similarE)
      }
    with Suc
    show ?thesis by auto
  qed
  qed
  qed
  qed

```

lemma similar-FnI[intro]:

**assumes**  $\bigwedge x y. x \triangleleft y \cdot C^\infty \implies f \cdot x \triangleleft g \cdot y \cdot C^\infty$   
**shows**  $Fn \cdot f \triangleleft CFn \cdot g$   
**by** (*metis assms similar-nice-def*)

**lemma** *similar-FnD[elim!]*:  
**assumes**  $Fn \cdot f \triangleleft CFn \cdot g$   
**assumes**  $x \triangleleft y \cdot C^\infty$   
**shows**  $f \cdot x \triangleleft g \cdot y \cdot C^\infty$   
**using** *assms*  
**by** (*subst (asm) similar-nice-def*) *auto*

**lemma** *similar-FnE[elim!]*:  
**assumes**  $Fn \cdot f \triangleleft CFn \cdot g$   
**assumes**  $(\bigwedge x y. x \triangleleft y \cdot C^\infty \implies f \cdot x \triangleleft g \cdot y \cdot C^\infty) \implies P$   
**shows**  $P$   
**by** (*metis assms similar-FnD*)

### 8.1.7 The similarity relation lifted pointwise to functions.

**abbreviation** *fun-similar* ::  $('a :: \text{type} \Rightarrow \text{Value}) \Rightarrow ('a \Rightarrow (C \rightarrow C\text{Value})) \Rightarrow \text{bool}$  (**infix**  $\triangleleft^*$ , 50) **where**  
*fun-similar*  $\equiv$  *pointwise*  $(\lambda x y. x \triangleleft y \cdot C^\infty)$

**lemma** *fun-similar-fmap-bottom[simp]*:  $\perp \triangleleft^* \perp$   
**by** *auto*

**lemma** *fun-similarE[elim]*:  
**assumes**  $m \triangleleft^* m'$   
**assumes**  $(\bigwedge x. (m \ x) \triangleleft (m' \ x) \cdot C^\infty) \implies Q$   
**shows**  $Q$   
**using** *assms unfolding pointwise-def* **by** *blast*

**end**

## 8.2 Denotational-Related

**theory** *Denotational-Related*  
**imports** *Denotational ResourcedDenotational ValueSimilarity*  
**begin**

Given the similarity relation it is straight-forward to prove that the standard and the resourced denotational semantics produce similar results. (Theorem 10 in [SGHHOM11]).

**theorem** *denotational-semantics-similar*:  
**assumes**  $\varrho \triangleleft^* \sigma$   
**shows**  $\llbracket e \rrbracket_\varrho \triangleleft (\mathcal{N} \llbracket e \rrbracket_\sigma) \cdot C^\infty$   
**using** *assms*  
**proof** (*induct e arbitrary: \varrho \sigma rule:exp-induct*)  
**case** (*Var v*)

```

from Var have  $\varrho v \triangleleft (\sigma v) \cdot C^\infty$  by cases auto
thus ?case by simp
next
case (Lam v e)
{ fix x y
  assume  $x \triangleleft y \cdot C^\infty$ 
  with  $\langle \varrho \triangleleft^* \sigma \rangle$ 
  have  $\varrho(v := x) \triangleleft^* \sigma(v := y)$ 
    by (auto 1 4)
  hence  $\llbracket e \rrbracket_{\varrho(v := x)} \triangleleft (\mathcal{N} \llbracket e \rrbracket_{\sigma(v := y)}) \cdot C^\infty$ 
    by (rule Lam.hyps)
}
thus ?case by auto
next
case (App e v ρ σ)
hence App':  $\llbracket e \rrbracket_\varrho \triangleleft (\mathcal{N} \llbracket e \rrbracket_\sigma) \cdot C^\infty$  by auto
thus ?case
proof (cases rule: similar-bot-cases)
  case (Fn f g)
  from  $\langle \varrho \triangleleft^* \sigma \rangle$ 
  have  $\varrho v \triangleleft (\sigma v) \cdot C^\infty$  by auto
  thus ?thesis using Fn App' by auto
qed auto
next
case (Bool b)
thus  $\llbracket \text{Bool } b \rrbracket_\varrho \triangleleft (\mathcal{N} \llbracket \text{Bool } b \rrbracket_\sigma) \cdot C^\infty$  by auto
next
case (IfThenElse scrut e1 e2)
hence IfThenElse':
   $\llbracket \text{scrut} \rrbracket_\varrho \triangleleft (\mathcal{N} \llbracket \text{scrut} \rrbracket_\sigma) \cdot C^\infty$ 
   $\llbracket e_1 \rrbracket_\varrho \triangleleft (\mathcal{N} \llbracket e_1 \rrbracket_\sigma) \cdot C^\infty$ 
   $\llbracket e_2 \rrbracket_\varrho \triangleleft (\mathcal{N} \llbracket e_2 \rrbracket_\sigma) \cdot C^\infty$  by auto
from IfThenElse'(1)
show ?case
proof (cases rule: similar-bot-cases)
  case (bool b)
  thus ?thesis using IfThenElse' by auto
qed auto
next
case (Let as e ρ σ)
have  $\{as\}_\varrho \triangleleft^* \mathcal{N} \{as\}_\sigma$ 
proof (rule parallel-HSem-ind-different-ESem[OF pointwise-adm[OF similar-admI] fun-similar-fmap-bottom])
  fix  $\varrho' :: \text{var} \Rightarrow \text{Value}$  and  $\sigma' :: \text{var} \Rightarrow C \rightarrow C\text{Value}$ 
  assume  $\varrho' \triangleleft^* \sigma'$ 
  show  $\varrho ++ \text{domA } as \llbracket as \rrbracket_{\varrho'} \triangleleft^* \sigma ++ \text{domA } as \text{evalHeap } as (\lambda e. \mathcal{N} \llbracket e \rrbracket_{\sigma'})$ 
  proof(rule pointwiseI, goal-cases)
  case (1 x)
  show ?case using  $\langle \varrho \triangleleft^* \sigma \rangle$ 
    by (auto simp add: lookup-override-on-eq lookupEvalHeap elim: Let(1)[OF -  $\langle \varrho' \triangleleft^* \sigma' \rangle$ ])

```

)  
**qed**  
**qed auto**  
**hence**  $\llbracket e \rrbracket_{\{\text{as}\}_\rho} \triangleleft (\mathcal{N}[\llbracket e \rrbracket_{\mathcal{N}\{\text{as}\}_\sigma}] \cdot C^\infty)$  **by** (*rule Let(2)*)  
**thus** *?case by simp*  
**qed**

**corollary** *evalHeap-similar*:

$\bigwedge y z. y \triangleleft^* z \implies \llbracket \Gamma \rrbracket_y \triangleleft^* \mathcal{N}[\llbracket \Gamma \rrbracket_z]$

**by** (*rule pointwiseI*)

(*case-tac x ∈ dom A Γ, auto simp add: lookupEvalHeap denotational-semantics-similar*)

**theorem** *heaps-similar*:  $\{\Gamma\} \triangleleft^* \mathcal{N}\{\Gamma\}$

**by** (*rule parallel-HSem-ind-different-ESem[OF pointwise-adm[OF similar-admI]]*)

(*auto simp add: evalHeap-similar*)

**end**

## 9 Adequacy

### 9.1 ResourcedAdequacy

```
theory ResourcedAdequacy
imports ResourcedDenotational Launchbury AList–Utils CorrectnessResourced
begin
```

```
lemma demand-not-0: demand (N[e]ρ) ≠ ⊥
proof
  assume demand (N[e]ρ) = ⊥
  with demand-suffices'[where n = 0, simplified, OF this]
  have (N[e]ρ)·⊥ ≠ ⊥ by simp
  thus False by simp
qed
```

The semantics of an expression, given only  $r$  resources, will only use values from the environment with less resources.

```
lemma restr-can-restrict-env: (N[e]ρ)|C·r = (N[e]ρ|or)|C·r
proof(induction e arbitrary: ρ r rule: exp-induct)
  case (Var x)
  show ?case
  proof(rule C-restr-C-cong)
    fix r'
    assume r' ⊆ r
    have (N[Var x]ρ)·(C·r') = ρ x·r' by simp
    also have ... = ((ρ x)|r)·r' using ⟨r' ⊆ r⟩ by simp
    also have ... = (N[Var x]ρ|or)·(C·r') by simp
    finally show (N[Var x]ρ)·(C·r') = (N[Var x]ρ|or)·(C·r').
  qed simp
next
  case (Lam x e)
  show ?case
  proof(rule C-restr-C-cong)
    fix r'
    assume r' ⊆ r
    hence r' ⊆ C·r by (metis below-C below-trans)
    {
      fix v
      have ρ(x := v)|r = (ρ|or)(x := v)|r
      by simp
      hence (N[e]ρ(x := v))|r' = (N[e]ρ|or(x := v))|r'
      by (subst (1 2) C-restr-eq-lower[OF Lam ⟨r' ⊆ C·r⟩]) simp
    }
    thus (N[Lam [x]. e]ρ)·(C·r') = (N[Lam [x]. e]ρ|or)·(C·r')
    by simp
  qed simp
next
```

**case** (*App e x*)  
**show** ?*case*  
**proof** (*rule C-restr-C-cong*)  
    **fix**  $r'$   
    **assume**  $r' \sqsubseteq r$   
    **hence**  $r' \sqsubseteq C \cdot r$  **by** (*metis below-C below-trans*)  
    **hence**  $(\mathcal{N} \llbracket e \rrbracket_{\varrho}) \cdot r' = (\mathcal{N} \llbracket e \rrbracket_{\varrho|_r^\circ}) \cdot r'$   
        **by** (*rule C-restr-eqD[OF App]*)  
    **thus**  $(\mathcal{N} \llbracket \text{App } e \ x \rrbracket_{\varrho}) \cdot (C \cdot r') = (\mathcal{N} \llbracket \text{App } e \ x \rrbracket_{\varrho|_r^\circ}) \cdot (C \cdot r')$   
        **using**  $\langle r' \sqsubseteq r \rangle$  **by** *simp*  
    **qed** *simp*  
**next**  
**case** (*Bool b*)  
**show** ?*case* **by** *simp*  
**next**  
**case** (*IfThenElse scrut e<sub>1</sub> e<sub>2</sub>*)  
**show** ?*case*  
**proof** (*rule C-restr-C-cong*)  
    **fix**  $r'$   
    **assume**  $r' \sqsubseteq r$   
    **hence**  $r' \sqsubseteq C \cdot r$  **by** (*metis below-C below-trans*)  
  
    **have**  $(\mathcal{N} \llbracket \text{scrut} \rrbracket_{\varrho}) \cdot r' = (\mathcal{N} \llbracket \text{scrut} \rrbracket_{\varrho|_r^\circ}) \cdot r'$   
        **using**  $\langle r' \sqsubseteq C \cdot r \rangle$  **by** (*rule C-restr-eqD[OF IfThenElse(1)]*)  
    **moreover**  
    **have**  $(\mathcal{N} \llbracket e_1 \rrbracket_{\varrho}) \cdot r' = (\mathcal{N} \llbracket e_1 \rrbracket_{\varrho|_r^\circ}) \cdot r'$   
        **using**  $\langle r' \sqsubseteq C \cdot r \rangle$  **by** (*rule C-restr-eqD[OF IfThenElse(2)]*)  
    **moreover**  
    **have**  $(\mathcal{N} \llbracket e_2 \rrbracket_{\varrho}) \cdot r' = (\mathcal{N} \llbracket e_2 \rrbracket_{\varrho|_r^\circ}) \cdot r'$   
        **using**  $\langle r' \sqsubseteq C \cdot r \rangle$  **by** (*rule C-restr-eqD[OF IfThenElse(3)]*)  
    **ultimately**  
    **show**  $(\mathcal{N} \llbracket (\text{scrut } ? e_1 : e_2) \rrbracket_{\varrho}) \cdot (C \cdot r') = (\mathcal{N} \llbracket (\text{scrut } ? e_1 : e_2) \rrbracket_{\varrho|_r^\circ}) \cdot (C \cdot r')$   
        **using**  $\langle r' \sqsubseteq r \rangle$  **by** *simp*  
    **qed** *simp*  
**next**  
**case** (*Let  $\Gamma$  e*)

The lemma, lifted to heaps

**have** *restr-can-restrict-env-heap* :  $\bigwedge r. (\mathcal{N} \llbracket \Gamma \rrbracket_{\varrho})|_r^\circ = (\mathcal{N} \llbracket \Gamma \rrbracket_{\varrho|_r^\circ})|_r^\circ$   
**proof**(*rule has-ESem.parallel-HSem-ind*)  
    **fix**  $\varrho_1 \ \varrho_2 :: CEnv$  **and**  $r :: C$   
    **assume**  $\varrho_1|_r^\circ = \varrho_2|_r^\circ$   
  
    **show**  $(\varrho \ ++_{domA \ \Gamma} \mathcal{N} \llbracket \Gamma \rrbracket_{\varrho_1})|_r^\circ = (\varrho|_r^\circ \ ++_{domA \ \Gamma} \mathcal{N} \llbracket \Gamma \rrbracket_{\varrho_2})|_r^\circ$   
**proof**(*rule env-C-restr-cong*)  
    **fix**  $x$  **and**  $r'$   
    **assume**  $r' \sqsubseteq r$   
    **hence**  $r' \sqsubseteq C \cdot r$  **by** (*metis below-C below-trans*)

**show**  $(\varrho \text{ ++ } \text{dom}A \Gamma \mathcal{N}[\Gamma]_{\varrho_1}) x \cdot r' = (\varrho|^\circ_r \text{ ++ } \text{dom}A \Gamma \mathcal{N}[\Gamma]_{\varrho_2}) x \cdot r'$   
**proof** (*cases*  $x \in \text{dom}A \Gamma$ )  
  **case** *True*  
  **have**  $(\mathcal{N}[\text{the } (\text{map-of } \Gamma x)]_{\varrho_1}) \cdot r' = (\mathcal{N}[\text{the } (\text{map-of } \Gamma x)]_{\varrho_1|^\circ_r}) \cdot r'$   
  **by** (*rule C-restr-eqD[OF Let(1)[OF True] ⟨r' ⊆ C·r⟩*)  
  **also have**  $\dots = (\mathcal{N}[\text{the } (\text{map-of } \Gamma x)]_{\varrho_2|^\circ_r}) \cdot r'$   
  **unfolding**  $\langle \varrho_1|^\circ_r = \varrho_2|^\circ_r \rangle \dots$   
  **also have**  $\dots = (\mathcal{N}[\text{the } (\text{map-of } \Gamma x)]_{\varrho_2}) \cdot r'$   
  **by** (*rule C-restr-eqD[OF Let(1)[OF True] ⟨r' ⊆ C·r⟩, symmetric*)  
  **finally**  
  **show** *?thesis using True by (simp add: lookupEvalHeap)*  
**next**  
  **case** *False*  
  **with**  $\langle r' \subseteq r \rangle$   
  **show** *?thesis by simp*  
**qed**  
**qed**  
**qed** *simp-all*

**show** *?case*  
**proof** (*rule C-restr-C-cong*)  
  **fix**  $r'$   
  **assume**  $r' \subseteq r$   
  **hence**  $r' \subseteq C \cdot r$  **by** (*metis below-C below-trans*)

**have**  $(\mathcal{N}\{\Gamma\}\varrho)|^\circ_r = (\mathcal{N}\{\Gamma\}(\varrho|^\circ_r))|^\circ_r$   
  **by** (*rule restr-can-restrict-env-heap*)  
  **hence**  $(\mathcal{N}[e]_{\mathcal{N}\{\Gamma\}\varrho}) \cdot r' = (\mathcal{N}[e]_{\mathcal{N}\{\Gamma\}\varrho|^\circ_r}) \cdot r'$   
  **by** (*subst (1 2) C-restr-eqD[OF Let(2) ⟨r' ⊆ C·r⟩] simp*)

**thus**  $(\mathcal{N}[\text{Let } \Gamma e]_{\varrho}) \cdot (C \cdot r') = (\mathcal{N}[\text{Let } \Gamma e]_{\varrho|^\circ_r}) \cdot (C \cdot r')$   
  **using**  $\langle r' \subseteq r \rangle$  **by** *simp*  
**qed** *simp*  
**qed**

**lemma** *can-restrict-env*:

$(\mathcal{N}[e]_{\varrho}) \cdot (C \cdot r) = (\mathcal{N}[e]_{\varrho|^\circ_r}) \cdot (C \cdot r)$   
**by** (*rule C-restr-eqD[OF restr-can-restrict-env below-refl]*)

When an expression  $e$  terminates, then we can remove such an expression from the heap and it still terminates. This is the crucial trick to handle black-holing in the resourced semantics.

**lemma** *add-BH*:

**assumes** *map-of*  $\Gamma x = \text{Some } e$   
**assumes**  $(\mathcal{N}[e]_{\mathcal{N}\{\Gamma\}}) \cdot r' \neq \perp$   
**shows**  $(\mathcal{N}[e]_{\mathcal{N}\{\text{delete } x \Gamma\}}) \cdot r' \neq \perp$

**proof-**

**obtain**  $r$  **where**  $r: C \cdot r = \text{demand } (\mathcal{N} \llbracket e \rrbracket_{\mathcal{N}\{\Gamma\}})$   
**using** *demand-not-0* **by** (*cases demand*  $(\mathcal{N} \llbracket e \rrbracket_{\mathcal{N}\{\Gamma\}})$ ) *auto*

**from** *assms(2)*  
**have**  $C \cdot r \sqsubseteq r'$  **unfolding**  $r$  *not-bot-demand* **by** *simp*

**from** *assms(1)*  
**have** [*simp*]: *the*  $(\text{map-of } \Gamma \ x) = e$  **by** (*metis option.sel*)

**from** *assms(1)*  
**have** [*simp*]:  $x \in \text{domA } \Gamma$  **by** (*metis domIff dom-map-of-conv-domA not-Some-eq*)

**define**  $ub$  **where**  $ub = \mathcal{N}\{\Gamma\}$  — An upper bound for the induction

**have** *heaps*:  $(\mathcal{N}\{\Gamma\})|_r^\circ \sqsubseteq \mathcal{N}\{\text{delete } x \ \Gamma\}$  **and**  $\mathcal{N}\{\Gamma\} \sqsubseteq ub$

**proof** (*induction rule: HSem-bot-ind*)

**fix**  $\varrho$   
**assume**  $\varrho|_r^\circ \sqsubseteq \mathcal{N}\{\text{delete } x \ \Gamma\}$   
**assume**  $\varrho \sqsubseteq ub$

**show**  $(\mathcal{N} \llbracket \Gamma \rrbracket_{\varrho})|_r^\circ \sqsubseteq \mathcal{N}\{\text{delete } x \ \Gamma\}$

**proof** (*rule fun-belowI*)

**fix**  $y$   
**show**  $((\mathcal{N} \llbracket \Gamma \rrbracket_{\varrho})|_r^\circ) \ y \sqsubseteq (\mathcal{N}\{\text{delete } x \ \Gamma\}) \ y$

**proof** (*cases y = x*)

**case** *True*  
**have**  $((\mathcal{N} \llbracket \Gamma \rrbracket_{\varrho})|_r^\circ) \ x = (\mathcal{N} \llbracket e \rrbracket_{\varrho})|_r$   
**by** (*simp add: lookupEvalHeap*)  
**also have**  $\dots \sqsubseteq (\mathcal{N} \llbracket e \rrbracket_{ub})|_r$   
**using**  $\langle \varrho \sqsubseteq ub \rangle$  **by** (*intro monofun-cfun-arg*)  
**also have**  $\dots = (\mathcal{N} \llbracket e \rrbracket_{\mathcal{N}\{\Gamma\}})|_r$   
**unfolding** *ub-def..*  
**also have**  $\dots = \perp$   
**using**  $r$  **by** (*rule C-restr-bot-demand[OF eq-imp-below]*)  
**also have**  $\dots = (\mathcal{N}\{\text{delete } x \ \Gamma\}) \ x$   
**by** (*simp add: lookup-HSem-other*)  
**finally**  
**show** *?thesis* **unfolding** *True.*

**next**

**case** *False*  
**show** *?thesis*

**proof** (*cases y ∈ domA Γ*)

**case** *True*  
**have**  $(\mathcal{N} \llbracket \text{the } (\text{map-of } \Gamma \ y) \rrbracket_{\varrho})|_r = (\mathcal{N} \llbracket \text{the } (\text{map-of } \Gamma \ y) \rrbracket_{\varrho|_r^\circ})|_r$   
**by** (*rule C-restr-eq-lower[OF restr-can-restrict-env below-C]*)  
**also have**  $\dots \sqsubseteq \mathcal{N} \llbracket \text{the } (\text{map-of } \Gamma \ y) \rrbracket_{\varrho|_r^\circ}$   
**by** (*rule C-restr-below*)  
**also note**  $\langle \varrho|_r^\circ \sqsubseteq \mathcal{N}\{\text{delete } x \ \Gamma\} \rangle$

```

finally
show ?thesis
  using ⟨y ∈ domA Γ⟩ ⟨y ≠ x⟩
  by (simp add: lookupEvalHeap lookup-HSem-heap)
next
  case False
  thus ?thesis by simp
qed
qed
qed

from ⟨ρ ⊆ ub⟩
have (N[[Γ]]_ρ) ⊆ (N[[Γ]]_ub)
  by (rule cont2monofunE[rotated]) simp
also have ... = ub
  unfolding ub-def HSem-bot-eq[symmetric]..
finally
show (N[[Γ]]_ρ) ⊆ ub.
qed simp-all

from assms(2)
have (N[[e]]_N[[Γ]])·(C·r) ≠ ⊥
  unfolding r
  by (rule demand-suffices[OF infinite-resources-suffice])
also
have (N[[e]]_N[[Γ]])·(C·r) = (N[[e]]_(N[[Γ]]|°_r))·(C·r)
  by (rule can-restrict-env)
also
have ... ⊆ (N[[e]]_N[[delete x Γ]])·(C·r)
  by (intro monofun-cfun-arg monofun-cfun-fun heaps )
also
have ... ⊆ (N[[e]]_N[[delete x Γ]])·r'
  using ⟨C·r ⊆ r'⟩ by (rule monofun-cfun-arg)
finally
show ?thesis by this (intro cont2cont)+
qed

```

The semantics is continuous, so we can apply induction here:

```

lemma resourced-adequacy:
  assumes (N[[e]]_N[[Γ]])·r ≠ ⊥
  shows ∃ Δ v. Γ : e ↓S Δ : v
using assms
proof (induction r arbitrary: Γ e S rule: C.induct[case-names adm bot step])
  case adm show ?case by simp
next
  case bot
  hence False by auto
  thus ?case..

```

```

next
  case (step r)
  show ?case
  proof(cases e rule:exp-strong-exhaust(1)[where c = (Γ,S), case-names Var App Let Lam Bool
IfThenElse])
  case (Var x)
  let ?e = the (map-of Γ x)
  from step.prem[unfolded Var]
  have x ∈ domA Γ
  by (auto intro: ccontr simp add: lookup-HSem-other)
  hence map-of Γ x = Some ?e by (rule domA-map-of-Some-the)
  moreover
  from step.prem[unfolded Var] ⟨map-of Γ x = Some ?e⟩ ⟨x ∈ domA Γ⟩
  have (N[?e]N{Γ})·r ≠ ⊥ by (auto simp add: lookup-HSem-heap simp del: app-strict)
  hence (N[?e]N{delete x Γ})·r ≠ ⊥ by (rule add-BH[OF ⟨map-of Γ x = Some ?e⟩])
  from step.IH[OF this]
  obtain Δ v where delete x Γ : ?e ↓x # S Δ : v by blast
  ultimately
  have Γ : (Var x) ↓S (x,v) # Δ : v by (rule Variable)
  thus ?thesis using Var by auto
next
case (App e' x)
  have finite (set S ∪ fv (Γ, e')) by simp
  from finite-list[OF this]
  obtain S' where S': set S' = set S ∪ fv (Γ, e')..

  from step.prem[unfolded App]
  have prem: ((N[e']N{Γ})·r ↓CFn (N{Γ}) x|r)·r ≠ ⊥ by (auto simp del: app-strict)
  hence (N[e']N{Γ})·r ≠ ⊥ by auto
  from step.IH[OF this]
  obtain Δ v where lhs': Γ : e' ↓S' Δ : v by blast

  have fv (Γ, e') ⊆ set S' using S' by auto
  from correctness-empty-env[OF lhs' this]
  have correct1: N[e']N{Γ} ⊆ N[v]N{Δ} and N{Γ} ⊆ N{Δ} by auto

  from prem
  have ((N[v]N{Δ})·r ↓CFn (N{Γ}) x|r)·r ≠ ⊥
  by (rule not-bot-below-trans)(intro correct1 monofun-cfun-fun monofun-cfun-arg)
  with result-evaluated[OF lhs']
  have isLam v by (cases r, auto, cases v rule: isVal.cases, auto)
  then obtain y e'' where n': v = (Lam [y]. e'') by (rule isLam-obtain-fresh)
  with lhs'
  have lhs: Γ : e' ↓S' Δ : Lam [y]. e'' by simp

  have ((N[v]N{Δ})·r ↓CFn (N{Γ}) x|r)·r ≠ ⊥ by fact
  also have (N{Γ}) x|r ⊆ (N{Γ}) x by (rule C-restr-below)
  also note ⟨v = -⟩

```

**also note**  $\langle \mathcal{N}\{\Gamma\} \sqsubseteq \mathcal{N}\{\Delta\} \rangle$   
**also have**  $(\mathcal{N}\llbracket \text{Lam } [y]. e'' \rrbracket_{\mathcal{N}\{\Delta\}} \cdot r \sqsubseteq \text{CFn} \cdot (\Lambda v. \mathcal{N}\llbracket e'' \rrbracket_{\mathcal{N}\{\Delta\}}(y := v))$   
**by** (rule *CELam-no-restr*)  
**also have**  $(\dots \downarrow \text{CFn } (\mathcal{N}\{\Delta\}) x) \cdot r = (\mathcal{N}\llbracket e'' \rrbracket_{\mathcal{N}\{\Delta\}}(y := ((\mathcal{N}\{\Delta\}) x))) \cdot r$  **by** *simp*  
**also have**  $\dots = (\mathcal{N}\llbracket e''[y::=x] \rrbracket_{\mathcal{N}\{\Delta\}}) \cdot r$   
**unfolding** *ESem-subst..*  
**finally**  
**have**  $\dots \neq \perp$  **by** *this* (intro *cont2cont cont-fun*) +  
**then**  
**obtain**  $\Theta v'$  **where** *rhs*:  $\Delta : e''[y::=x] \downarrow_{S'} \Theta : v'$  **using** *step.IH* **by** *blast*

**have**  $\Gamma : \text{App } e' x \downarrow_{S'} \Theta : v'$   
**by** (rule *reds-ApplicationI[OF lhs rhs]*)  
**hence**  $\Gamma : \text{App } e' x \downarrow_{S'} \Theta : v'$   
**apply** (rule *reds-smaller-L*) **using** *S'* **by** *auto*  
**thus** *?thesis* **using** *App* **by** *auto*

**next**  
**case** (*Lam v e'*)  
**have**  $\Gamma : \text{Lam } [v]. e' \downarrow_{S'} \Gamma : \text{Lam } [v]. e' ..$   
**thus** *?thesis* **using** *Lam* **by** *blast*

**next**  
**case** (*Bool b*)  
**have**  $\Gamma : \text{Bool } b \downarrow_{S'} \Gamma : \text{Bool } b$  **by** *rule*  
**thus** *?thesis* **using** *Bool* **by** *blast*

**next**  
**case** (*IfThenElse scrut e1 e2*)

**from** *step.prem**s[unfolded IfThenElse]*  
**have** *prem*:  $\text{CB-project} \cdot ((\mathcal{N}\llbracket \text{scrut} \rrbracket_{\mathcal{N}\{\Gamma\}}) \cdot r) \cdot ((\mathcal{N}\llbracket e_1 \rrbracket_{\mathcal{N}\{\Gamma\}}) \cdot r) \cdot ((\mathcal{N}\llbracket e_2 \rrbracket_{\mathcal{N}\{\Gamma\}}) \cdot r) \neq \perp$  **by**  
*(auto simp del: app-strict)*  
**then obtain** *b* **where**  
*is-CB*:  $(\mathcal{N}\llbracket \text{scrut} \rrbracket_{\mathcal{N}\{\Gamma\}}) \cdot r = \text{CB} \cdot (\text{Discr } b)$   
**and** *not-bot2*:  $((\mathcal{N}\llbracket (\text{if } b \text{ then } e_1 \text{ else } e_2) \rrbracket_{\mathcal{N}\{\Gamma\}}) \cdot r) \neq \perp$   
**unfolding** *CB-project-not-bot* **by** *(auto split: if-splits)*

**have** *finite* (*set S*  $\cup$  *fv* ( $\Gamma$ , *scrut*)) **by** *simp*  
**from** *finite-list[OF this]*  
**obtain** *S'* **where** *S'*: *set S'* = *set S*  $\cup$  *fv* ( $\Gamma$ , *scrut*)..

**from** *is-CB* **have**  $(\mathcal{N}\llbracket \text{scrut} \rrbracket_{\mathcal{N}\{\Gamma\}}) \cdot r \neq \perp$  **by** *simp*  
**from** *step.IH[OF this]*  
**obtain**  $\Delta v$  **where** *lhs'*:  $\Gamma : \text{scrut} \downarrow_{S'} \Delta : v$  **by** *blast*  
**then have** *isVal v* **by** (rule *result-evaluated*)

**have** *fv* ( $\Gamma$ , *scrut*)  $\subseteq$  *set S'* **using** *S'* **by** *simp*  
**from** *correctness-empty-env[OF lhs' this]*  
**have** *correct1*:  $\mathcal{N}\llbracket \text{scrut} \rrbracket_{\mathcal{N}\{\Gamma\}} \sqsubseteq \mathcal{N}\llbracket v \rrbracket_{\mathcal{N}\{\Delta\}}$  **and** *correct2*:  $\mathcal{N}\{\Gamma\} \sqsubseteq \mathcal{N}\{\Delta\}$  **by** *auto*

```

from correct1
have  $(\mathcal{N}[\llbracket \text{scrut } \mathbb{1}_{\mathcal{N}\{\Gamma\}} \rrbracket \cdot r \sqsubseteq (\mathcal{N}[\llbracket v \mathbb{1}_{\mathcal{N}\{\Delta\}} \rrbracket \cdot r \text{ by (rule monofun-cfun-fun)}$ 
with is-CB
have  $(\mathcal{N}[\llbracket v \mathbb{1}_{\mathcal{N}\{\Delta\}} \rrbracket \cdot r = \text{CB} \cdot (\text{Discr } b) \text{ by simp}$ 
with  $\langle \text{isVal } v \rangle$ 
have  $v = \text{Bool } b \text{ by (cases } v \text{ rule: isVal.cases) (case-tac } r, \text{ auto})+$ 

from not-bot2  $\langle \mathcal{N}\{\Gamma\} \sqsubseteq \mathcal{N}\{\Delta\} \rangle$ 
have  $(\mathcal{N}[\llbracket (\text{if } b \text{ then } e_1 \text{ else } e_2) \mathbb{1}_{\mathcal{N}\{\Delta\}} \rrbracket \cdot r \neq \perp$ 
  by (rule not-bot-below-trans[OF - monofun-cfun-fun[OF monofun-cfun-arg]])
from step.IH[OF this]
obtain  $\Theta \ v' \text{ where rhs: } \Delta : (\text{if } b \text{ then } e_1 \text{ else } e_2) \Downarrow_{S'} \Theta : v' \text{ by blast}$ 

from lhs'[unfolded  $\langle v = - \rangle$ ] rhs
have  $\Gamma : (\text{scrut } ? e_1 : e_2) \Downarrow_{S'} \Theta : v' \text{ by rule}$ 
hence  $\Gamma : (\text{scrut } ? e_1 : e_2) \Downarrow_S \Theta : v'$ 
  apply (rule reds-smaller-L) using  $S'$  by auto
thus ?thesis unfolding IfThenElse by blast
next
case (Let  $\Delta \ e'$ )
  from step.prems[unfolded Let(2)]
  have prem:  $(\mathcal{N}[\llbracket e' \mathbb{1}_{\mathcal{N}\{\Delta\}} \mathbb{1}_{\mathcal{N}\{\Gamma\}} \rrbracket \cdot r \neq \perp$ 
    by (simp del: app-strict)
  also
    have atom  $\langle \text{dom } A \ \Delta \ \# * \ \Gamma \text{ using } \text{Let}(1) \text{ by (simp add: fresh-star-Pair)}$ 
    hence  $\mathcal{N}\{\Delta\} \mathcal{N}\{\Gamma\} = \mathcal{N}\{\Delta \ @ \ \Gamma\} \text{ by (rule } \text{HSem-merge})$ 
  finally
  have  $(\mathcal{N}[\llbracket e' \mathbb{1}_{\mathcal{N}\{\Delta \ @ \ \Gamma\}} \rrbracket \cdot r \neq \perp.$ 
  then
  obtain  $\Theta \ v \text{ where } \Delta \ @ \ \Gamma : e' \Downarrow_S \Theta : v \text{ using } \text{step.IH} \text{ by blast}$ 
  hence  $\Gamma : \text{Let } \Delta \ e' \Downarrow_S \Theta : v$ 
    by (rule reds.Let[OF Let(1)])
  thus ?thesis using Let by auto
qed
qed

end

```

## 9.2 Adequacy

```

theory Adequacy
imports ResourcedAdequacy Denotational-Related
begin

```

```

theorem adequacy:
  assumes  $\llbracket e \rrbracket_{\Gamma} \neq \perp$ 
  shows  $\exists \Delta \ v. \Gamma : e \Downarrow_S \Delta : v$ 

```

**proof–**  
**have**  $\{\Gamma\} \triangleleft^* \mathcal{N}\{\Gamma\}$  **by** (*rule heaps-similar*)  
**hence**  $\llbracket e \rrbracket_{\{\Gamma\}} \triangleleft (\mathcal{N}\llbracket e \rrbracket_{\mathcal{N}\{\Gamma\}}) \cdot C^\infty$  **by** (*rule denotational-antics-similar*)  
**from** *bot-or-not-bot[OF this] assms*  
**have**  $(\mathcal{N}\llbracket e \rrbracket_{\mathcal{N}\{\Gamma\}}) \cdot C^\infty \neq \perp$  **by** *metis*  
**thus** *?thesis* **by** (*rule resourced-adequacy*)  
**qed**  
  
**end**

## References

- [Abr90] Samson Abramsky, *The lazy lambda calculus*, Research topics in functional programming, 1990, pp. 65–116.
- [AO93] Samson Abramsky and Chih-Hao Luke Ong, *Full abstraction in the lazy lambda calculus*, Information and Computation **105** (1993), no. 2, 159 – 267.
- [Bre13] Joachim Breitner, *The correctness of launchbury’s natural semantics for lazy evaluation*, Archive of Formal Proofs (2013), <http://isa-afp.org/entries/Launchbury.shtml>, Formal proof development.
- [Huf12] Brian Huffman, *HOLCF ’11: A definitional domain theory for verifying functional programs*, Ph.D. thesis, Portland State University, 2012.
- [Lau93] John Launchbury, *A natural semantics for lazy evaluation*, POPL ’93, 1993, pp. 144–154.
- [Ses97] Peter Sestoft, *Deriving a lazy abstract machine*, Journal of Functional Programming **7** (1997), 231–264.
- [SGHHOM11] Lidia Sánchez-Gil, Mercedes Hidalgo-Herrero, and Yolanda Ortega-Mallén, *Relating function spaces to resourced function spaces*, SAC, 2011, pp. 1301–1308.
- [SGHHOM14] ———, *The role of indirections in lazy natural semantics*, PSI, 2014.
- [UK12] Christian Urban and Cezary Kaliszyk, *General bindings and alpha-equivalence in nominal isabelle*, Logical Methods in Computer Science **8** (2012), no. 2.