Pointer Analysis and Separation Logic.

Elodie-Jane Sims

▶ To cite this version:

Elodie-Jane Sims. Pointer Analysis and Separation Logic.. Computer Science [cs]. Ecole Polytechnique X, 2007. English. NNT : . pastel-00003506

THÈSE

présentée à

l'ÉCOLE POLYTECHNIQUE

pour l'obtention du titre de

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE EN INFORMATIQUE

Élodie-Jane SIMS

1er décembre 2007

Analyses de Pointeurs et Logique de Séparation

Pointer Analysis and Separation Logic

Président:	Reinhard WILHELM *Professeur, Universität des Saarlandes, Saarbrücken, Allemagne*
Rapporteurs:	Hongseok YANG *Queen Mary, University of London, London, Royaume Uni*
	Reinhard WILHELM *Professeur, Universität des Saarlandes, Saarbrücken, Allemagne*
Examinateurs:	Roberto GIACOBAZZI *Professeur, Universita' degli Studi di Verona, Italie*
	David SCHMIDT *Professeur, Kansas State University, Manhattan Kansas, USA*
Directrice de thèse:	Radhia COUSOT *Directeur de recherche au CNRS, École Normale Supérieure, Paris*

École Polytechnique

Copyright

Résumé

Le cadre de cette thèse est l'analyse statique modulaire par interprétation abstraite de logiciels en vue de leur vérification automatique. Nous nous intéressons en particulier aux programmes comportant des objets alloués dynamiquement sur un tas et repérés par des pointeurs. Le but final étant de trouver des erreurs dans un programme (problèmes de déréférencements et d'alias) ou de prouver qu'un programme est correct (relativement à ces problèmes) de façon automatique.

Isthiaq, Pym, O'Hearn et Reynolds ont développé récemment des logiques de fragmentation (*separation logics*) qui sont des logiques de Hoare avec un langage d'assertions/de prédicats permettant de démontrer qu'un programme manipulant des pointeurs sur un tas est correct. La sémantique des triplets de la logique ($\{P\}C\{P'\}$) est définie par des transformateurs de prédicats de style plus faible pré-condition.

Nous avons exprimé et prouvé la correction de ces plus faibles pré-conditions (wlp) et plus fortes post-conditions (sp), en particulier dans le cas de la commande *while*. L'avantage par rapport à ce qui est fait dans la communauté est que les wlp et sp sont définis pour toute formule alors que certaines des règles existantes avaient des restrictions syntaxiques.

Nous avons rajouté des points fixes à la logique ainsi qu'une substitution retardée permettant d'exprimer des formules récursives. Nous avons exprimé les wlp et sp dans cette logique avec points fixes et prouvé leur correction. La substitution retardée a une utilité directe pour l'expression de formules récursives. Par exemple, $\mathtt{nclist}(x) = \mu X_v.\ (x = \mathtt{nil}) \vee \exists x_1, x_2.(\mathtt{isval}(x_1) \wedge (x \mapsto x_1, x_2\ *\ X_v[x_2/x]))$ décrit l'ensemble des mémoires où x pointe vers une liste d'entiers.

Le but ensuite était d'utiliser cette logique de fragmentation avec points fixes comme langage d'interface pour des analyses de pointeurs. Il s'agit de formuler une traduction

des domaines de ces analyses en formules de la logique (et inversement) et d'en prouver la correction. On peut également parfois utiliser ces traductions pour prouver la correction de ces analyses.

Nous avions déjà illustré cette approche pour une analyse très simple de partitionnement des pointeurs. Nous avons traduit les formules de la logique dans un nouveau langage abstrait permettant de décrire le type des valeurs associées aux variables dans la mémoire (nil, entier, bool, pointeur vers une paire de tel type, etc.) ainsi que les relations d'aliasing et non-aliasing entre variables et entre points de la mémoire. Le principal apport est la définition de ce langage et de sa sa sémantique dans le domaine concret qui est celui utilisé pour la sémantique des formules. En particulier, les variables auxiliaires dont la sémantique est habituellement une question d'implémentation font ici explicitement part du langage et de sa sémantique. Ce langage est un produit cartesien de plusieurs sous domaines et peut être paramétré par les domaines numériques existants. Nous avons créé un sous-domaine qui est un tableau permettant de compenser le manque de précision dû à l'utilisation de graphes d'ensembles au lieu d'ensembles de graphes. Nous avons exprimé et prouvé les traductions des formules dans ce langage abstrait.

Abstract

We are interested in modular *static analysis* to analyse softwares automatically. We focus on programs with data structures, and in particular, programs with pointers. The final goal is to find errors in a program (problems of dereferencing, aliasing, etc) or to prove that a program is correct (regarding those problems) in an automatic way.

Isthiaq, Pym, O'Hearn and Reynolds have recently developed *separation logics*, which are Hoare logics with assertions and predicates language that allow to prove the correctness of programs that manipulate pointers. The semantics of the logic's triples ($\{P\}C\{P'\}$) is defined by predicate transformers in the style of weakest preconditions.

We expressed and proved the correctness of those weakest preconditions (wlp) and strongest postconditions (sp), in particular in the case of *while*-loops. The advance from the existing work is that wlp and sp are defined for *any* formula, while previously existing rules had syntactic restrictions.

We added fixpoints to the logic as well as a postponed substitution which then allow to express recursive formulae. We expressed wlp and sp in the extended logic and proved their correctness. The postponed substitution is directly useful to express recursive formulae. For example, $\texttt{nclist}(x) = \mu X_v.\ (x = \texttt{nil}) \vee \exists x_1, x_2.(\texttt{isval}(x_1) \wedge (x \mapsto x_1, x_2\ *\ X_v[x_2/x]))$ describes the set of memory where x points to a list of integers.

Next, the goal was to use separation logic with fixpoints as an interface language for pointer analysis. That is, translating the domains of those analyses into formulae of the logic (and conversely) and to prove their correctness. One might also use the translations to prove the correctness of the pointer analysis itself.

We illustrate this approach with a simple pointers-partitioning analysis. We translate the logic formulae into an abstract language we designed which allows us to describe the

type of values registered in the memory (nil, integer, booleans, pointers to pairs of some types, etc.) as well as the aliasing and non-aliasing relations between variables and locations in the memory. The main contribution is the definition of the abstract language and its semantics in a concrete domain which is the same as the one for the semantics of formulae. In particular, the semantics of the auxiliary variables, which is usually a question of implementation, is explicit in our language and its semantics. The abstract language is a partially reduced product of several subdomains and can be parametrised with existing numerical domains. We created a subdomain which is a tabular data structure to cope with the imprecision from not having sets of graphs. We expressed and proved the translations of formulae into this abstract language.

Contents

List of Figures

Acknowledgments

I would like to warmly thank all those I will not mention here and were there during this heavy period as a PhD student and in particular all those who went all the way up hill in the maze to attend my defense !

I am very proud that Reinhard Wilhelm and Hongseok Yang accepted to be my reviewers and grateful for the work they have done. I have been impressed by the interest and kindness they shown and the quality of their remarks. I would like to thank Roberto Giacobazzi for accepting to be a member of my jury, it was an honor.

The most important person involved in a PhD, after the student is its adviser. I was glad to be granted of two advisers, Radhia Cousot and David Schmidt. They both were very helpful, and supporting, in their own domains of action. I obviously wouldn't have started this thesis without them, but I also wouldn't have finished it ! My biggest thanks go for them.

I naturally come to thank Patrick Cousot after them, he is the one who introduced me to them both, and to my area of work. He also was a great scientific help and trigger during my DEA's internship and at the very beginning of my thesis.

I was surprised to get so many and great gifts after my defense, but the best one was three big brothers: Charles Hymans, Georg Jung and Francesco Logozzo. I haven't been as supportive to the youngest as I received, but I was very happy to be with Guillaume Capron and Pietro Ferrara. I believe co-PhDs remain co-PhDs after their defenses !

I was so lucky to have great colleagues and would like to thank: Anindya Banerjee, Julien Bertrane, Bruno Blanchet, Simon Bliudze, Gilles Dowek, Jérôme Feret, Bertrand Jeannet, Gurvan Le Guernic, Mathieu Martel, Isabella Mastroeni, Laurent Mauborgne, Antoine Miné, Alexander Serebrenik, Axel Simon, Allen Stoughton, Xavier Rival, Robby and Sarah Zennou.

I would like to thank Kansas State University and in particular CIS department, for their kindness when I was there and all the administrative organisation.

I am sincerely glad my family is there for me, in particular my american family who is so supportive from far away. Je tiens à remercier mon grand-père André Pichard qui fut mon modèle. Sans ma formidable grand-mère Jeanne Pichard, j'aurais fini cette thèse transformée en pizza ! Je remercie ma grand-tante Paulette Côme pour son écoute et ses encouragements. Je remercie ma mère Anne-Lise Pichard pour le soutien qu'elle a su m'apporter. Je tiens à remercier mon affreuse frangine, Anne-Orlis Pigneur (et son mari Thomas), pour mon adorable nièce Mathilde, bon je plaisante, je la remercie aussi pour bien bien d'autres choses.

Last but not least. Je tiens à remercier mes amis, mon plus important soutien, tout particulièrement ma meilleure amie Marion Martin dont le téléphone a supporté mes lamentations journalières ! Je ne peux remercier ici que ceux qui m'ont promis des représailles en cas d'oubli ;-) : Noémie Atlan, Térence Bayen, Anne-Marie Bleau, Denis Conduché, Florian Douetteau, Clémentine Doumenc, Alain Frisch, Céline Georgin, Jean-Marc Léoni, Melanie Pfeiffer, Benoît Piedallu, Andrea Platz, Héloïse Viard et Alexandra Vidalenc. I omit now unforgettable thanks.

Chapter 1

Introduction

1.1 Motivations

Programs (also called *software*) are used in many places in our environment and a big part of the effort of programing is used for the so called *debugging* part (finding errors). One massively used technique for debugging is *testing* (trying to run the program in various situations). Through testing, one can test only some values, thus testing never insures the program has no error. You can not try all integers before answering that the program works for all integers; you can not run the program forever before answering the program will run without error forever. When programs are used in critical situations (for example, in spaceships, public transportations, power plants, banking), it is then very important to insure their safety, and it is even sometimes mandatory by laws. Formal methods try to address the problem by providing mathematically sound techniques that guarantee a full coverage of all program behaviours.

We are interested in doing *static analysis*. This consists of proving a program's correctness or other properties on the program without running the program. Running a program uses a domain called the *operational domain* (like values assigned to variables, having memory locations, ...) and applies changes to elements of the domain following the instructions given by the program. Static analysis uses a domain which abstracts ("represents") the operational domain and applies changes to elements on this abstracting domain. The changes

in the abstracting domain are related to the changes the program would operate on the operational domain so that at the end, knowing the changes in the abstracting domain gives us informations about how the program behaves on the operational domain.

Abstract interpretation[1–3] provides a framework which first helps proving that some properties on the abstracting domain (there called the *abstract domain*) implies some properties on the operational domain; secondly, it helps designing abstracting domains.

Here, we give an example for non computer scientists. Imagine a program that tries to compute an integer (a number). The program gives an error if it attempts to do a division by 0 since the result would be infinity, which is usually not represented in the machine. As said about *testing*, we can not try all integers before answering there is no division by 0. Here, the integers would be the *operational domain*. Now, we want an abstracting domain, which would allow us to give an answer in a finite time (and preferably not in a century), but which would also give a safe answer. We allow the analysis to say it cannot answer, but if it says the program has no division by 0, we want to trust it. One simple *abstract domain* is the domain of the signs: $POSITIVE$, $NEGATIVE$, $ZERO$, $DONT\_KNOW$ and $ERROR$. Then, we change the program on integers into a program on this abstract domain, replacing the usual operation on integers $(+, -,...)$ by the sign rules ($POSITIVE + POSITIVE = POSITIVE$, $NEGATIVE + NEGATIVE = NEGATIVE$, $POSITIVE + NEGATIVE = DONT\_KNOW$,..., $POSITIVE/POSITIVE = POSITIVE$, $POSITIVE/DONT\_KNOW = DONT\_KNOW$, $POSITIVE/ZERO = ERROR$,...). If the result at the end is $POSITIVE$, $NEGATIVE$ or $ZERO$, we know that the program ran without the error of division by 0 (we even know a little more). If the result is $DONT\_KNOW$, we indeed don't know anything about the program. (This kind of result would in practice lead to the search for a more precise *abstract domain*). If the result is $ERROR$, then we know for sure

2

that the program will reach an error of division by 0.

In this thesis, we focus on programs with data structures, and in particular programs with pointers. Data structures could be "lists", "trees" or others and are structures commonly used by programmers. Programs that use data structures are a major field of research since many real programs errors come from the use of pointers. Errors can come from aliasing, two variables pointing to the same value, then when a variable changes its value, it also changes what is pointed by the second one without mentioning its name. There are other errors like trying to dereference a `nil` or trying to access a part of the memory which has been freed earlier.

Now, we go back to the reader who does not know about pointers. You can imagine that a program consists in putting things in boxes, moving things from one box to another and so on. Then, this being not convenient enough, the programmers also have a big board of mailboxes like in a post office. Then, instead of putting things/mail directly in the first boxes, they will put in the boxes the number of the mailbox, and to get the mail, they will find the number of the mailbox in that box and get the mail in the mailbox. Things get a little more complicated since the mailboxes can also contain mailboxes's numbers instead of mail and then one can go through several mailboxes before reaching their mail. This structure is quite convenient for programming, but it brings some errors not obvious for the programmers. First, two boxes can contain the same mailbox address, then if you want to change the mail for one box, you will go change the mail in its mailbox which will automatically change the mail of the other box. Now, you have to realise that those mailboxes are something real in your computer called *memory*. You don't have infinite memory, like you don't have infinite space in a post office. So when some mailboxes are no more used, they will be removed so that someone else can install its own mailboxes. This

leads to errors, when you forget that someone was using a mailbox, you remove
that mailbox and this someone now try to get its mail and cannot.

Analysing programs dealing with pointers is an old challenge but still alive. There
are many pointer analyses but none of them would be elected as the one which is precise
enough, efficient and entirely formally proved. A widespread approach is so called *shape
analysis*[4,5]; those analyses are using as abstracting domains some sort of "graphs" which
represent the data structures implemented in the program. Nodes represent locations of the
memory (mailboxes) and edges represents the pointing relations (the mailboxes represented
by the origin of the arrow contain the addresses of the mailboxes represented by the node
pointed by the goal of the arrow). This approach is related to the programmers habits,
programmers have a big memory available, but when they program, they are thinking that
they are building lists, trees, or other peculiar data structures.

In the graph built by shape analyses, nodes represent one or more locations of the mem-
ory. The principal idea is that you do not record the whole memory, but only the part of
the memory which is pertinent for the analysis. Shape analysis abstraction is usually not
only forgetting about part of the memory, but also having nodes of the abstract graph rep-
resenting several locations of the memory (called *summary nodes*). For example, a memory
with a list from x with a next field called n could be represented like $\boxed{x} \longrightarrow \bullet \overset{n}{\circlearrowright} \overset{n}{\longrightarrow} \texttt{nil}$
instead of $\boxed{x} \longrightarrow \bullet \overset{n}{\longrightarrow} ... \bullet \overset{n}{\longrightarrow} \texttt{nil}$
There are many shape analyses, with different kind of additional informations (for example,
modality saying that an edge "must" or "may" exist), and different operations on how to
modify the graphs. The major default of existing shape analyses is that they use rules for
modifying graphs and for merging two graphs for calls of functions, and those rules are hard
to prove and lead to tricky and hard to imagine errors. Scalability (the analysis should not
take too much time when the programs are getting too big) is also a major concern.

When I started this thesis, *separation logic*[6] appeared as a promising approach to de-
scribing memory properties and has been used to do proofs on program, but only manually.

The idea behind separation logic is that there are spatial connectives which allow one to speak about disjoints part of the memory. The logic appears to permit to describe the memory in a very natural way. People do manual proofs using a set of rules, and the most interesting for them was that, with some restrictions, they could prove some properties of a program running on a memory and this would imply properties of that program on an extended memory. The principal lack of this approach is that it requires to find a loop invariant for analyzing a while-loop, and this step can never be fully automatic.

Following, in Section 1.2, we will give a detailed introduction about separation logic. In Section 1.3, we will describe what was our project and what we achieved.

1.2   Introduction to separation logic

We first present what is a logic: a logic has *symbols*, a *syntax* for *formulae*, at least one *model* and at least one *semantics* per model.

The *symbols* of the logic are usually infinitely many *variables* (for example x, y, z,...) (in our case, some of them are program variables) and *connectives* (for example $=$, $\wedge$, $\vee$,...). We call *syntax* a set of rules which combine the symbols of the logic to build *formulae* (for example $x = 3 \wedge x = y$). A rule of the syntax could be "If P and Q are two formulae then $P \wedge Q$ is also a formula."

A program has an *operational domain* on which it runs. Similarly, a logic has a *model* (for example, mappings from variables to integer values).

For a program, a *semantics* is related to an *operational domain* and describes how a command transforms an element of this domain (often called a *memory*).

For a logic, a *semantics* is related to a *model* and gives the "meaning" of a formula of the logic in term of elements of this *model* (for example $x = 3 \wedge y = x$ means that x has the value 3 and y has the same value).

This relation of "meaning" between a formula and an element of its *model* is called *satisfaction* and is usually written $\models$ (for example, we could write $[x \mapsto 3 \mid y \mapsto 3] \models (x = 3 \wedge y =$

5

x)).

So the *semantics* for a *model* M would often be written as a set of rules like "For any memory m element of M, if $m \models P$ and $m \models Q$ then $m \models P \wedge Q$."

Here, we give a short example to explain why people decided to use logics to analyse programs. Take a short program:

```
x := 3; y := x;
```

You can take as the *operational domain* the mapping of x and y to integers.

Then, you can imagine that you start with both x and y assigned to 0 and run the program:

$$[x \mapsto 0 \mid y \mapsto 0] \xrightarrow{\texttt{x:=3;}} [x \mapsto 3 \mid y \mapsto 0] \xrightarrow{\texttt{y:=x;}} [x \mapsto 3 \mid y \mapsto 3]$$

But if you had started with some other integer values, you could have for example

$$[x \mapsto 5 \mid y \mapsto 2] \xrightarrow{\texttt{x:=3;}} [x \mapsto 3 \mid y \mapsto 2] \xrightarrow{\texttt{y:=x;}} [x \mapsto 3 \mid y \mapsto 3]$$

If you want to analyse the program and prove that at the end both x and y are assigned to 3, as explained earlier, you can not try all possible integers for the starting values of x and y.

Logics appeared to be natural domains to express properties of the memory, and in particular Hoare logic[7], whose central feature is the *Hoare triple* $\{P\}C\{Q\}$ where P and Q are formulae and C a program. The meaning of such a Hoare triple is that: if we run the program C on a memory which satisfies the formula P then if the program terminates without error, it terminates on a memory which satisfies the formula Q.

For our example, we could write the triple

$$\{\texttt{true}\} \texttt{ x:=3; y:=x; } \{x = 3 \wedge y = x\}$$

which means that in any case, if the program terminates at the end x has the value 3 and y has the same value.

Then, people wanted those formulae to be automatically found (or as much automatically as possible), so they wrote rules, for example, in fact we had

$$\{\texttt{true}\}\ \texttt{x:=3;}\ \{x = 3\}$$

and

$$\{x = 3\}\ \texttt{y:=x;}\ \{x = 3 \wedge y = x\}$$

and the results come from the use of the rule that for any formulae P, Q, R and any programs C_1 and C_2 we have

$$\text{If } \{P\}C_1\{Q\} \text{ and } \{Q\}C_2\{R\} \text{ then } \{P\}C_1; C_2\{R\}$$

People also wanted the result to be as precise as possible. If you take the example, we could also have written

$$\{x = 5\}\ \texttt{x:=3;}\ \texttt{y:=x;}\ \{x = 3 \wedge y = x\}$$

which means that if at the beginning x has the value 5, then at the end x has the value 3 and y has the same value.

This leads to the problem: for a formula Q, and a program C what is the least restrictive formula P such that $\{P\}C\{Q\}$ is correct ? This formula P is called the *weakest precondition* and we would write rules of the form

$$\{wp(C, Q)\}C\{Q\}$$

Such rules can be found in Hoare logic[7] and Dijkstra-style weakest-precondition logics[8].

Symmetrically, if you take the example, we could also have written

$$\{\texttt{true}\}\ \texttt{x:=3;}\ \texttt{y:=x;}\ \{x = 3\}$$

That is correct, it means that in any case at the end x has value 3. But obviously it is imprecise since it does not tell us information about y being equal to x.

This leads to the problem: for a formula, P, and a program, C, what is the most precise formula Q such that $\{P\}C\{Q\}$ is correct ? This formula Q is called the *strongest postcondition*, and we would write rules of the form

$$\{P\}C\{sp(C,P)\}$$

Now, remember that we are interested in programs which manipulate data structures and in particular in programs with pointers. So the *operational domain* would not be simple mappings of variables but also location memory (what we explained in terms of mailboxes).

To be precise, we are interested in programs whose *operational domain* could be (we say "could be", because having locations mapping to pairs instead of other representations like for example the ones using pointer arithmetic was just a choice and is not a limitation.) a pair with a *stack* and a *heap*. The stack is a mapping from variables to values, like in the previous examples, but the values could also be locations and the heap is a mapping from locations to pairs of values.

Then, naturally, to express properties on this domain, we need a logic which has a semantics in this domain and allows us to express properties of the memory.

One of the rules of Hoare logic is $\{P[E/x]\}x := E\{P\}$, this means that if the memory satisfies the property P about x (in fact, x does not need to be mentioned by P) after assigning the value of E to the variable x, then we know that at the beginning the memory had the same property P but for E and not x. For example, we have $\{(x = y+1)[3/x]\}x := 3\{x = y + 1\}$ which is $\{3 = y + 1\}x := 3\{x = y + 1\}$ or simply $\{y = 2\}x := 3\{x = y + 1\}$, this means that if we assign 3 to x and get that x is equal to $y + 1$, then we know that before y was equal to 2.
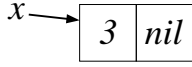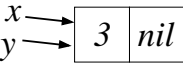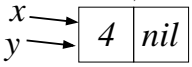
When reasoning about programs that manipulate pointers or heap storage, Hoare logic[7] and Dijkstra-style weakest-precondition logics[8] appeared to be failing because the logics require that each program variable names a distinct storage location.

If you consider the program

```
(1)    x := new_pair (3, nil);
```

```
(2)    y := x;

(3)    y.first = 4;
```

after the command (1), the memory is such that $x \longrightarrow \boxed{3 \mid nil}$, after the command (2) it is like $x, y \longrightarrow \boxed{3 \mid nil}$, after the command (3) it is like $x, y \longrightarrow \boxed{4 \mid nil}$.

If we try to analyse it using the rule $\{P[E/x]\}x := E\{P\}$, you can prove that

$$
\begin{aligned}
\{4 > 3\} \Rightarrow\quad & \{4 > new\_pair(3, \texttt{nil}).first\} \\
& \texttt{x := new\_pair (3, nil);} \\
& \{4 > x.first\} \\
& \texttt{y := x;} \\
& \{4 > x.first\} \\
& \texttt{y.first := 4;} \\
& \{y.first > x.first\}
\end{aligned}
$$

This is false, or as we say *unsound*, because at the end, as you can see in the drawing, y.first and x.first are both equal to 4, so we do not have $y.first > x.first$.

Through a series of papers[6,9,10], Reynolds and O'Hearn have addressed this foundationally difficult issue of designing a logic for reasoning about programs that manipulate pointers or heap storage. Their key insight is that a command executes within a *region* of heap storage: they write

$$s, h \models \phi$$

to denote that property ϕ holds true within heap subregion h and local-variable stack s. One could also say that a formula describes a property of the states it represents. For example, ϕ might be:

emp   which means that the heap is empty

$E \mapsto a, b$ which means there is exactly one "cons" cell in the heap, containing the values of a and b as its "head" and "tail" values and that E points to it.

$E \hookrightarrow a, b$ which is the same as the previous example, except that the heap can contain additional cells

With the assistance of a new connective, the "separating conjunction", denoted $*$, Reynolds and O'Hearn write

$$s, h_1 \cdot h_2 \models \phi_1 * \phi_2$$
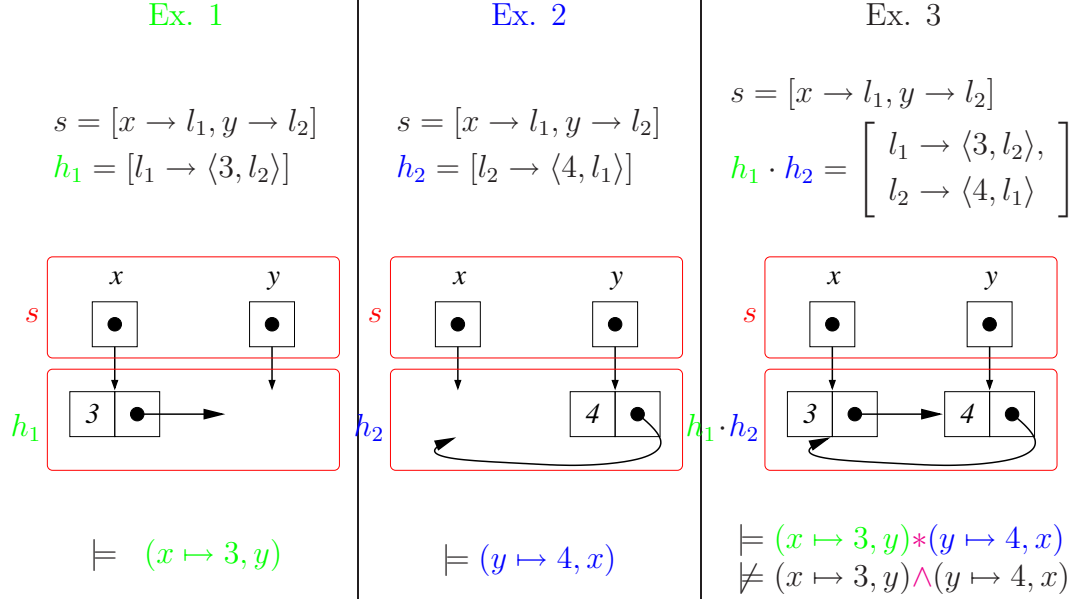
Figure 1.1: *Examples of separation formulae and memory satisfying them*

to assert that both ϕ_1 and ϕ_2 use *disjoint* heap subregions, h_1 and h_2, to justify the truth of ϕ_1 and ϕ_2 respectively. There is no aliasing between the variables mentioned in ϕ_1 and ϕ_2.

For example, as shown in Figure 1.1, if $s = \begin{bmatrix} x \to l_1 \\ y \to l_2 \end{bmatrix}$ and $h = \begin{bmatrix} l_1 \to \langle 3, l_2 \rangle \\ l_2 \to \langle 4, l_1 \rangle \end{bmatrix}$, then $s, h \models (x \mapsto 3, y) * (y \mapsto 4, x)$ (Fig. 1.1, ex. 3) because if $h_1 = [l_1 \to \langle 3, l_2 \rangle]$ and $h_2 = [l_2 \to \langle 4, l_1 \rangle]$ we have $s, h_1 \models x \mapsto 3, y$ and $s, h_2 \models y \mapsto 4, x$ (Fig. 1.1, ex. 1 and 2).

We also have $s, h \models (x \hookrightarrow 3, y)$ but $s, h \not\models (x \mapsto 3, y)$. If $s = \begin{bmatrix} x \to l_1 \\ y \to l_1 \end{bmatrix}$ and $h = [l_1 \to \langle 3, 4 \rangle]$, then $s, h \models (x \mapsto 3, 4) \wedge (y \mapsto 3, 4)$ but $s, h \not\models (x \mapsto 3, 4) * (y \mapsto 3, 4)$.

Adjoint to the separating conjunction is a "separating implication,"

$$s, h \models \phi_1 \mathbin{-\!\!*} \phi_2$$

which asserts, "if heap region h is augmented by h' such that $s, h' \models \phi_1$, then $s, h \cdot h' \models \phi_2$". For example, if $s = \begin{bmatrix} x \to l_1 \\ y \to l_2 \end{bmatrix}$ and $h_1 = [l_1 \to \langle 3, l_2 \rangle]$, then $s, h_1 \models (y \mapsto 4, x) \mathbin{-\!\!*} ((x \mapsto 3, y) * (y \mapsto 4, x))$, because $\forall h'.(l_1 \notin dom(h') \wedge s, h' \models y \mapsto 4, x)$ implies that $h' = [l_2 \to \langle 4, l_1 \rangle]$ which is $h' = h_2$ and $s, h_1 \cdot h_2 \models (x \mapsto 3, y) * (y \mapsto 4, x)$.

10

Ishtiaq and O'Hearn[10] showed how to add the separating connectives to a classical logic, producing *separation logic* in which Hoare-logic-style reasoning can be conducted on while-programs that manipulate temporary-variable stacks and heaps.

A Hoare triple, $\{\phi_1\}C\{\phi_2\}$, uses assertions ϕ_i, written in separation logic; the semantics of the triple is stated with respect to a stack-heap storage model.

Finally, there is an additional inference rule, the *frame rule*, which formalizes compositional reasoning based on disjoint heap regions:

$$\frac{\{\phi_1\}C\{\phi_2\}}{\{\phi_1 * \phi'\}C\{\phi_2 * \phi'\}}$$

where ϕ''s variables are not modified by C.

In Figure 1.2 is the example of the analysis of a piece of program and in Figure 1.3 is an example of how the *frame rule* can be applied using the previous example.

The reader interested in the *set-of-inference-rules* approach for separation logic is invited to read[10], and also[11] for details on the frame rule. The rules can also be found in the survey on separation logics[6]. We do not present the rule set here since we were not interested in them during the project.

1.3   History of the project and contributions

We now present a short history of our project and finish this section with a list of contributions of our work. When we started, separation logic appeared as a promising approach to describe memory properties. It was applied to do proofs of program correctness, but only manually.

Separation logic is a very natural language to describe memory, and it allows to speak of only part of the memory and does not require to define the data structures (like lists, trees) to describe properties of the memory. One can write assertions like:

- x points to a list of [1;2;3]

  $\exists x_2, x_3. \, (x \hookrightarrow 1, x_2) * (x_2 \hookrightarrow 2, x_3) * (x_3 \hookrightarrow 3, \texttt{nil})$

11

Figure 1.2: *Example of separation logic formulae which caracterize a piece of code inserting a cell in a linked list, on the right is a graphical view of memory satifying the formulae*
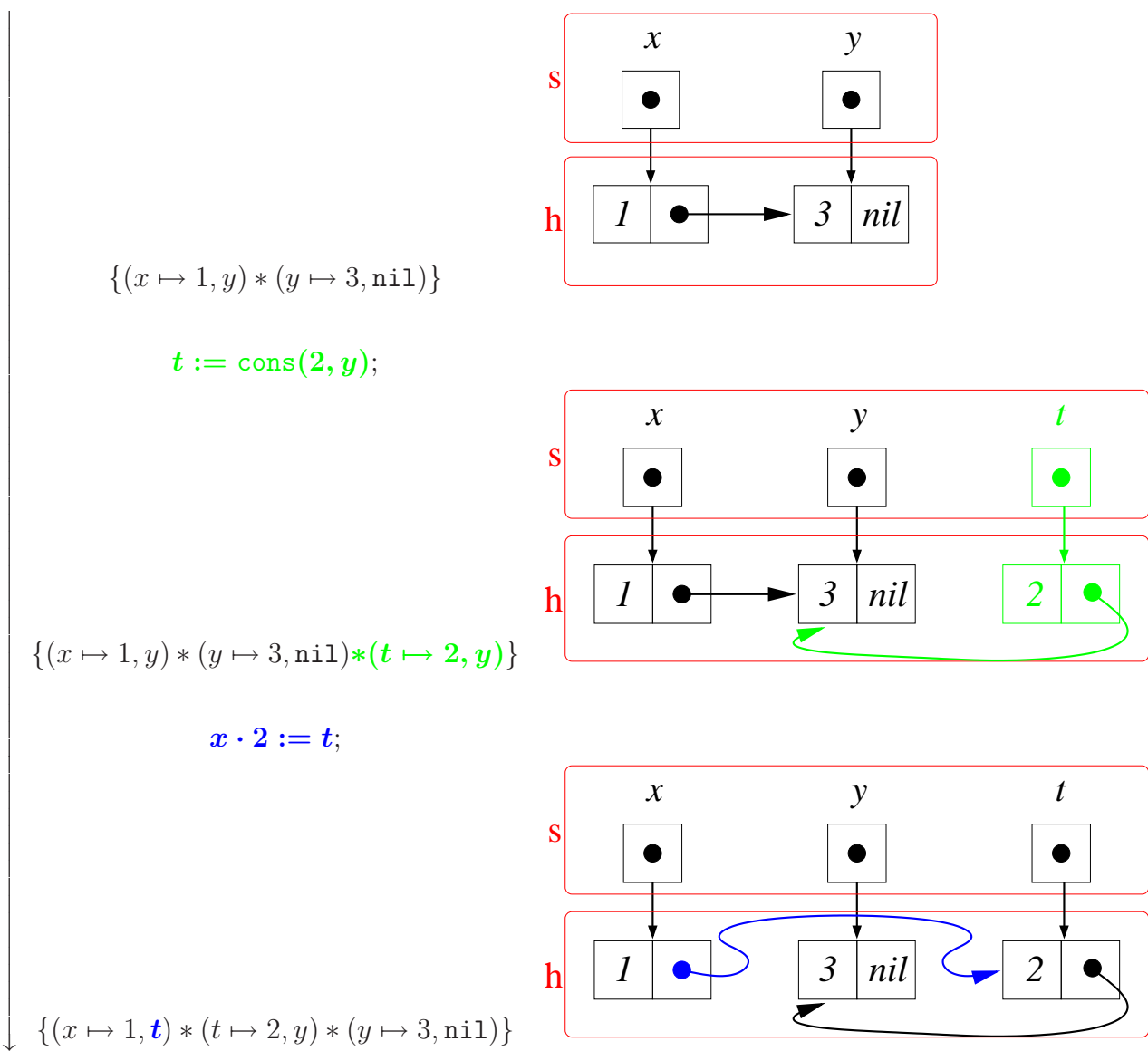
Figure 1.3: *Local reasoning: exemple of Fig. 1.2 for an extended heap*

13
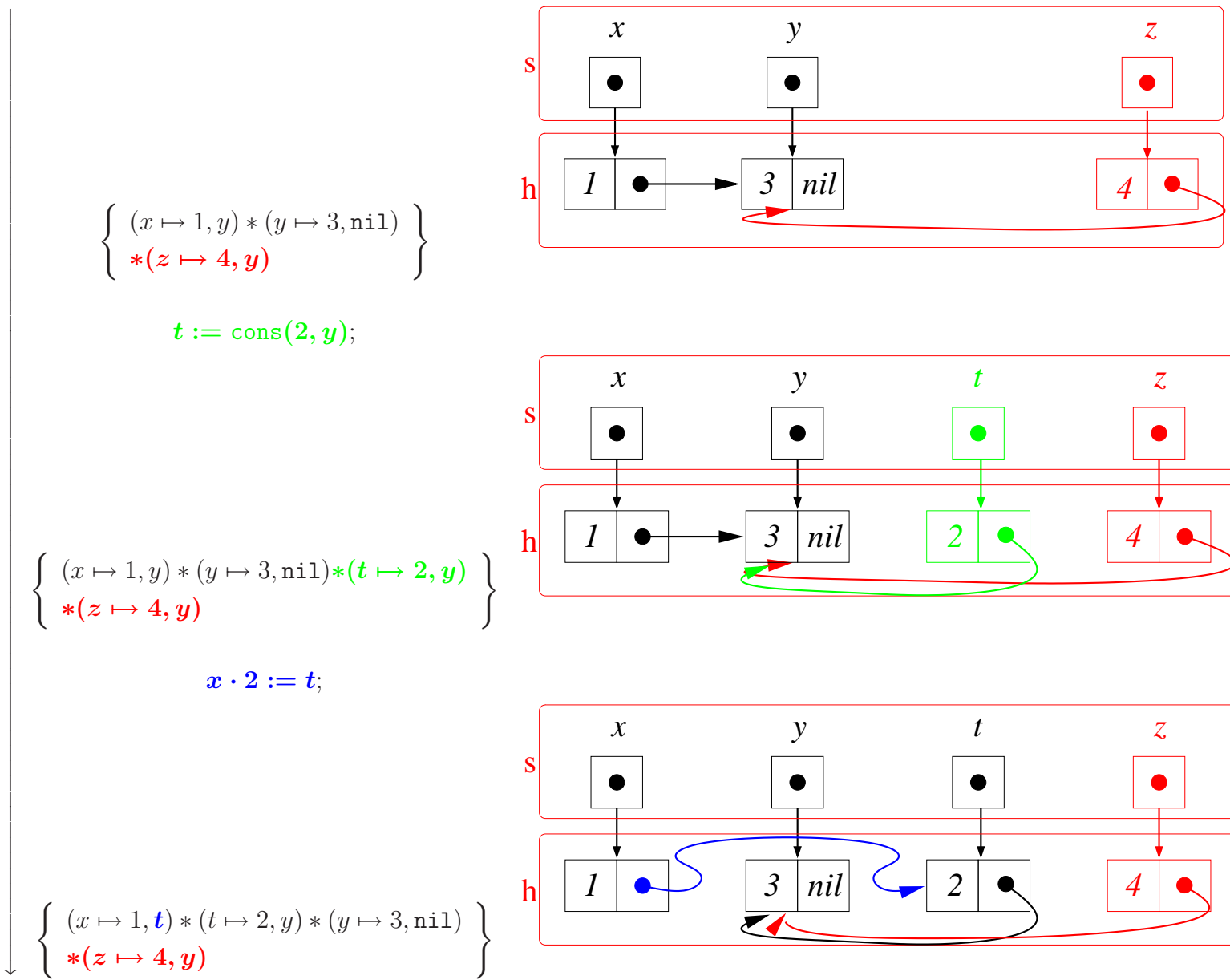
- x and y are aliased pointers

  $x = y \wedge \exists x_1, x_2. \ (x \hookrightarrow x_1, x_2)$

- Partitioning: x and y belong to two disjoint parts of the heap which have no pointers from one to the other.

We wanted to use this logic as an interface language for modular analysis, where the formula of the logic would be used to characterise the pre- and post-conditions of a library function, F, and would be used during the analysis of a call to F from another analysis.
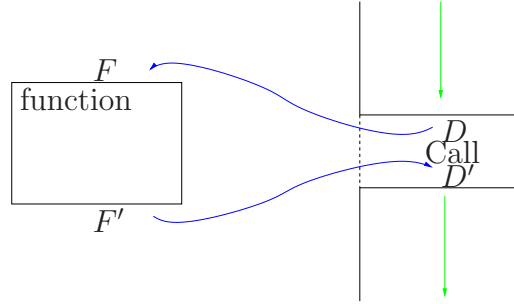
Our first step was to transform the logic. As we said earlier, most of the precondition formulae were already expressible in the existing logic, but for while-loops, there were none because these require *recursively defined* assertions, which were not in the logic's reach. So people would have to find the loop invariant manually, which can not be automated in all cases.

One primary accomplishment of this thesis was to add least- and greatest-fixed-point operators to separation logic, so that pre- and post-condition semantics for the while-language can be wholly expressed within the logic. As a pleasant consequence of the addition of fixpoints, it becomes possible to formalize recursively defined properties on inductively (and co-inductively) defined data structures. In the past, people were forced to write formulae about data structures in a recursive way without having a formal definition and use them to instantiate rules which were only proved for non-recursive formulae. We made it possible

to express, for example,

$$\texttt{nonCircularList}(x) \triangleq$$
$$\mu X_v. \, (x = \texttt{nil}) \lor \exists x_1, x_2.(\texttt{isval}(x_1) \land (x \mapsto x_1, x_2 \; * \; X_v[x_2/x]))$$

which asserts that x is a linear, non-circular list, where $\texttt{isval}(x_1)$ ensures that x_1 is a value (this predicate is defined shortly) and $[\,/\,]$ denotes postponed substitution. The recursion variable, X_v, is subscripted by v for emphasis.

Another example is the definition of non-circular binary tree:

$$\texttt{nctree}(x) \triangleq$$
$$\mu X_v.(x = \texttt{nil}) \lor \quad \exists x_v, x_l, x_r, x'.\texttt{isval}(x_v) \land$$
$$(x \mapsto x_v, x' \; * \; x' \mapsto x_l, x_r \; * \; X_v[x_l/x] \; * \; X_v[x_r/x])$$

We wish to advise the reader who is surprised by the postponed substitution (the connective, $[\,/\,]$) to read the formula, $P[E/x]$, as (Moggi's [12]) call-by-value let-expression, "$\texttt{let } x = E \texttt{ in } P$". A precise semantics is given later in Chapter 2.

The addition of the recursion operators comes with a price: the usual definition of syntactic substitution and the classic substitution laws become more complex; the reasons are related to the semantics of stack- and heap-storage as well as to the inclusion of the recursion operators. We proved several key properties about substitutions, variable renaming, and unfolding (for example: $\mu X_v. \, P \equiv P\{(\mu X_v. \, P)/X_v\}$).

Now we have all the pre- and post-conditions for all commands, including the while-loop. But not surprisingly, when computing the pre- or post-conditions, we obtained formulae that were so complicated to read that their meanings are too difficult to understand. And, as we wanted to use the logic for modularity with other analysis domains, we decided to create an intermediate language for separation-logic formulae such that:

- it resembles the existing shape/alias analysis domains to allow translations from/to those domains

- it comes with a concrete semantics in term of sets of states which is the same domain as for the formulae's semantics

- we can translate the formulae into our domain

- our domain is a partially reduced product of different subdomains so that we can cheaply tune the precision depending on the needs. For example, the domain is parametrised by a numerical domain which can be forgotten if we do not care about numericals.

At very beginning, we tried to implement a translation to a very simple language which would only say if a variable is `nil` or not. If we think of the earlier example of trying to find division by 0 errors, if you start with a domain which would only say if the variable is 0 or not, soon you will find that your analysis can rarely answer and you will think of adding $POSITIVE$ and $NEGATIVE$ to the domain. Then, you will want to increase the precision and define a numerical domain, and so on, getting a more precise analysis while keeping it efficient.

As for the abstract language we designed, the process was the same: we started with a very simple domain, `nil` or not, then we added graphs that were quite similar to shape graphs. Then, we needed auxiliary variables to say "$x = y$" and then say "$x = $ `nil`" but yet not have to re-check which values equal to x when refining the informations about "x" and its aliases: $\boxed{x} \longrightarrow \boxed{\alpha} \longrightarrow \top$ then $\boxed{x} \longrightarrow \boxed{\alpha} \longrightarrow \top$ `nil` .
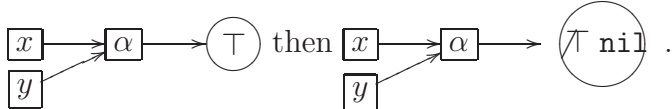
We designed a semantics for the language in the same domain as the model of the logic and itself. This was important in particular for auxiliary variables. First, we wanted that the semantics of a graph would be the intersection of the semantics of its arrows. Secondly, because it permits to write precise proofs about the functions which manipulates elements of the language , the functions that translate formulae while manipulating auxiliary variables are usually presented in papers as part of the implementation and not precisely described. We made this precise and explicit.

To resemble existing shape graphs, we naturally added *summary nodes*, nodes which represent several locations, to the graphs to bring some abstraction, but we generalised

them so that they would represent not only multiple locations but they could also represent any kind of values (numerical, boolean, etc.).

At this point, we did not have information about numericals and as a result most of the time, our result would say "the result is this but may be there is a numerical error" while in fact no numerical would be involved or there were no errors of numericals. So, we parametrised the domain by a numerical domain.

We did not want to record sets of graphs, but only one graph with sets of arrows, but this would imply imprecision during the union, so we created a new subdomain which is table-based, which allows us to keep additional information when we want to make a precise union.

Lastly, the arrows of the graphs include separation information.

Since we can express all pre- and post-conditions in our logic with fixpoints, we can already get information about the program by expressing the pre- and post-conditions and then translating them into our abstract language.

The principal place of abstraction comes when translating fixpoints and building summary nodes. This is interesting because it can provide useful information without knowing in advance the form of data structures used by a program.

If we may summarize the main accomplishments of the thesis, these would be:

- adding fixpoints to separation logic, which provides a way to express recursive formulae, expressing preconditions for while-loops, and expressing all post-conditions, letting us prove useful properties about the extended logic

- giving a precise semantics of the abstract domain of separation formulae in terms of sets of memory

- designing the abstract language as a partially reduced product of subdomains

- giving a semantics to auxiliary variables and not leaving this as an implementation design question

17

- combining the domain's heap analysis with a numerical domain which could be chosen from existing ones (e.g. polyhedra, octogons)

- designing the novel tabular data structure which allows extra precision by using a graph of sets instead of sets of graphs

1.4 Structure of the manuscript

Chapter 2 presents the extension with fixpoints we added to separation logic. It includes the syntax and semantics of the programs we analyse; the syntax, semantics and some properties of the separation logic with fixpoints; the pre- and post-conditions of programs in the extended logic for any formula and any command including *while*-loops and their proofs.

Chapter 3 presents the abstract language we have designed. It gives some examples, a precise semantics, and some useful operations on the domain like extension, union, merging nodes, stabilization, the function *ast* created for translating the connective $*$. The chapter gives the translation of formulae of separation logic with fixpoints into elements of the abstract language and their proofs.

Chapter 4 talks about the implementation of the analysis and in particular the data-structures adopted.

In Chapter 5, we gives comparison to related work. The chapter focuses on two active lines of work which have been operating for several years with many people: "smallfoot", the work from London, which is the most involved with separation logic, and "TVLA", a well established work for analysing storage structure.

We conclude in Chapter 6.

Chapter 2

Separation logic with fixpoints

In this chapter [1], we extend the separation logic, presented in the introduction, with fixpoint connectives to define recursive properties and to express the axiomatic semantics of a `while` statement. We present forward and backward analyses (*sp (strongest postcondition)*, *wlp (weakest liberal precondition)* expressed for all statements and all formulae.

In Sect. 2.1, we describe the command language we analyze and in Sect. 2.2, we present our logic $BI^{\mu\nu}$. In Sect. 2.3, we provide a backward analysis with $BI^{\mu\nu}$ in terms of "weakest liberal preconditions". We express the *wlp* for the composition, $\mathrm{if-then-else}$ and `while` commands. In Sect. 2.4, we provide a forward analysis with $BI^{\mu\nu}$ in terms of "strongest postconditions". In Sect. 2.5, we discuss another possibility for adding fixpoints to separation logic and other works proposed afterward.

2.1   Commands and basic domains

We consider a simple "while"-language with Lisp-like expressions for accessing and creating `cons` cells.

2.1.1   Command syntax

The commands we consider are as follows.

---

[1]most of this chapter has been plublished in [13] and [14]

$$
\begin{aligned}
C \quad ::= & \quad x := E \mid x := E.i \mid E.i := E' \mid x := \texttt{cons}(E_1, E_2) \mid \texttt{dispose}(E) \\
& \quad \mid C_1; C_2 \mid \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 \mid \texttt{skip} \mid \texttt{while } E \texttt{ do } C_1 \\
i \quad ::= & \quad 1 \mid 2 \\
E \quad ::= & \quad x \mid n \mid \texttt{nil} \mid True \mid False \mid E_1 \ op \ E_2
\end{aligned}
$$

An expression can denote an integer, an atom, or a heap-location. Here x is a variable in Var, n an integer and op is an operator in $(Val \times Val) \rightarrow Val$ such as $+ : (Int \times Int) \rightarrow Int$, $\vee : (Bool \times Bool) \rightarrow Bool$ (for Var and Val, see Sect. 2.1.2).

The second and third assignment statements read and update the heap, respectively. The fourth creates a new cons cell in the heap and places a pointer to it in x.

Notice that in our language we do not handle more than one dereferencings in one statement (no $x.i.j$, no $x.i := y.j$); this restriction is for simplicity and does not limit the expressivity of the language, requiring merely the addition of intermediate variables.

2.1.2 Semantic domains

$$
\begin{aligned}
Val &= Int \cup Bool \cup Atoms \cup Loc \\
S &= Var \rightharpoonup Val \\
H &= Loc \rightharpoonup Val \times Val
\end{aligned}
$$

Here, $Loc = \{l_1, l_2, ...\}$ is an infinite set of locations, $Var = \{x, y, ...\}$ is an infinite set of variables, $Atoms = \{\texttt{nil}, a, ...\}$ is a set of atoms, and $\rightharpoonup$ is for partial functions. We call an element $s \in S$ a stack and $h \in H$ a heap. We also call the pair $(s, h) \in S \times H$ a *state*.

We use $dom(h)$ to denote the domain of definition of a heap $h \in H$, and $dom(s)$ to denote the domain of a stack $s \in S$. Notice that we allow $dom(h)$ to be infinite.

An expression is interpreted as a heap-independent value: $[\![E]\!]^s \in Val$. For example, $[\![x]\!]^s = s(x)$, $[\![n]\!]^s = n$, $[\![\texttt{true}]\!]^s = true$, $[\![E_1 + E_2]\!]^s = [\![E_1]\!]^s + [\![E_2]\!]^s$.

Since domain S allows partial functions, $[\![\ ]\!]^s$ is also partial. Thus $[\![E_1 = E_2]\!]^s$ means $[\![E_1]\!]^s$ and $[\![E_2]\!]^s$ are defined and equal. From here on, when we write a formula of the form $\cdots [\![E]\!]^s \cdots$, we are also asserting that $[\![E]\!]^s$ is defined.

$$\frac{[\![E]\!]^s = v}{x := E, s, h \rightsquigarrow [s|x \rightarrow v], h} \qquad \frac{[\![E]\!]^s = l \quad h(l) = r}{x := E.i, s, h \rightsquigarrow [s|x \rightarrow \pi_i r], h}$$

$$\frac{[\![E]\!]^s = l \quad h(l) = r \quad [\![E']\!]^s = v'}{E.i = E', s, h \rightsquigarrow s, [h|l \rightarrow (r|i \rightarrow v')]} \qquad \frac{l \in dom(h) \quad [\![E]\!]^s = l}{\texttt{dispose}(E), s, h \rightsquigarrow s, (h - l)}$$

$$\frac{l \in Loc \quad l \notin dom(h) \quad [\![E_1]\!]^s = v_1, [\![E_2]\!]^s = v_2}{x := \texttt{cons}(E_1, E_2), s, h \rightsquigarrow [s|x \rightarrow l], [h|l \rightarrow \langle v_1, v_2 \rangle]}$$

$$\frac{C_1, s, h \rightsquigarrow C', s', h'}{C_1; C_2, s, h \rightsquigarrow C'; C_2, s', h'} \qquad \frac{C_1, s, h \rightsquigarrow s', h'}{C_1; C_2, s, h \rightsquigarrow C_2, s', h'} \qquad \texttt{skip}, s, h \rightsquigarrow s, h$$

$$\frac{[\![E]\!]^s = True}{\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2, s, h \rightsquigarrow C_1, s, h} \qquad \frac{[\![E]\!]^s = False}{\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2, s, h \rightsquigarrow C_2, s, h}$$

$$\frac{[\![E]\!]^s = False}{\texttt{while } E \texttt{ do } C, s, h \rightsquigarrow s, h} \qquad \frac{[\![E]\!]^s = True}{\texttt{while } E \texttt{ do } C, s, h \rightsquigarrow C; \texttt{while } E \texttt{ do } C, s, h}$$

Figure 2.1: *Operational small-step semantics of the commands*

2.1.3 Small-step semantics

The semantics of statements, C, are given small-step semantics defined by the relation $\rightsquigarrow$ on configurations. The configurations include triples C, s, h and terminal configurations s, h for $s \in S$ and $h \in H$. The rules are given in Fig. 2.1.

In the rules, we use r for elements of $Val \times Val$; $\pi_i r$ with $i \in \{1, 2\}$ for the first or second projection; $(r|i \rightarrow v)$ for the pair like r except that the i'th component is replaced with v; and $[s \mid x \rightarrow v]$ for the stack like s except that it maps x to v, $(h - l)$ for $h_{\lceil dom(h) \setminus \{l\}}$.

The location l in the `cons` case is not specified uniquely, so a new location is chosen non-deterministically.

Let the set of error configurations be: $\Omega = \{C, s, h \mid \nexists K.\ C, s, h \rightsquigarrow K\}$.

We say that:

- "C, s, h is safe" if and only if $\forall K.\ (C, s, h \rightsquigarrow^* K \implies K \notin \Omega)$

- "C, s, h is stuck" if and only if $C, s, h \in \Omega$

For instance, an error state can be reached by an attempt to dereference `nil` or an integer. Note also that the semantics allows dangling references, as in stack $[x \rightarrow l]$ with

empty heap [].

The definition of safety is formulated with partial correctness in mind: with loops, C, s, h could fail to converge to a terminal configuration but not get stuck.

We define the weakest liberal precondition in the operational domain:

Definition 2.1. *For* $\Delta \subseteq S \times H$, $wlp_o(\Delta, C) = \{s, h \mid (C, s, h \leadsto^* s', h' \Rightarrow s', h' \in \Delta) \wedge C, s, h$ *is safe*$\}$

We define the strongest postcondition similarly:

Definition 2.2. $sp_o(\Delta, C) = \{s', h' \mid \exists s, h \in \Delta. \; C, s, h \leadsto^* s', h'\}$

2.2 $BI^{\mu\nu}$

In this section, we present the logic $BI^{\mu\nu}$. It is designed to describe properties of the state. Typically, for analysis it will be used in Hoare triples of the form $\{P\}C\{Q\}$ with P and Q formulae of the logic and C a command.

We present in Sect. 2.2.1 the syntax of the logic and in Sect. 2.2.2 its formal semantics. In Sect. 2.2.3, we give the definition of a *true triple* $\{P\}C\{Q\}$. In Sect. 2.2.4, we discuss the additions to separation logic (fixpoints and postponed substitution).

2.2.1 Syntax of formulae

P, Q, R	$::=$	$E = E'$	Equality	$\mid$	$E \mapsto E_1, E_2$	Points to
	$\mid$	`false`	Falsity	$\mid$	$P \Rightarrow Q$	Classical Imp.
	$\mid$	$\exists x.P$	Existential Quant.	$\mid$	`emp`	Empty Heap
	$\mid$	$P * Q$	Spatial Conj.	$\mid$	$P \mathbin{-\!\!*} Q$	Spatial Imp.
	$\mid$	X_v	Formula Variable	$\mid$	$P[E/x]$	Postponed Substitution
	$\mid$	$\nu X_v.P$	Greatest Fixpoint	$\mid$	$\mu X_v.P$	Least Fixpoint

We have an infinite set of variables, Var_v, used for the variables bound by μ and ν and disjoint from the set Var. They range over sets of states, the others $(x,y,...)$ are variables which range over values. For emphasis, uppercase variables subscripted by $_v$ are used to define recursive formulae. We use the term "closed" for the usual notion of closure of

variables in *Var* (closed by $\exists$ or $\forall$) and the term "v-closed" for closure of variables in Var_v (v-closed by μ or ν).

Our additions to Reynolds and O'Hearn's separation logic are the fixed-point operators $\mu X_v.\ P$ and $\nu X_v.\ P$ and the substitution construction $P[E/x]$.

We can define various other connectives as usual, rather than taking them as primitives:

$$\neg P \quad \triangleq \quad P \Rightarrow \texttt{false} \qquad \texttt{true} \quad \triangleq \quad \neg(\texttt{false})$$
$$P \vee Q \quad \triangleq \quad (\neg P) \Rightarrow Q \qquad P \wedge Q \quad \triangleq \quad \neg(\neg P \vee \neg Q)$$
$$\forall x.P \quad \triangleq \quad \neg(\exists x.\neg P) \qquad E \hookrightarrow a, b \quad \triangleq \quad \texttt{true} * (E \mapsto a, b)$$
$$x = E.i \triangleq \exists x_1, x_2.\ (E \hookrightarrow x_1, x_2) \wedge (x = x_i)$$

We could have only one fixpoint connective in the syntax, since the usual equivalences, $\mu X_v.\ P \equiv \neg \nu X_v.\neg(P\{\neg X_v/X_v\})$ and $\nu X_v.\ P \equiv \neg \mu X_v.\neg(P\{\neg X_v/X_v\})$, hold (proofs in Sect. 2.6.7).

The set $FV(P)$ of free variables of a formula is defined as usual. The set $Var(P)$ of variables of a formula is defined as usual with $Var(P[E/x]) = Var(P) \cup Var(E) \cup \{x\}$. (see definitions of FV and Var in Sect. 2.6.1)

2.2.2   Semantics of formulae

The semantics of the logic is given in Fig. 2.2.

We use the following notations in formulating the semantics:

- $h \sharp h'$ indicates that the domains of heaps h and h' are disjoint;

- $h \cdot h'$ denotes the union of disjoint heaps (i.e., the union of functions with disjoint domains).

We express the semantics of the formulae in an environment ρ mapping formula variables to set of states: $\rho : Var_v \rightharpoonup \mathcal{P}(S \times H)$. The semantics of a formula in an environment ρ is the set of states which satisfy it, and is expressed by: $[\![\cdot]\!]_\rho : BI^{\mu\nu} \rightharpoonup \mathcal{P}(S \times H)$

We call $[\![P]\!]$ the semantics of a formula P in an empty environment $[\![P]\!] = [\![P]\!]_\emptyset$. We also define a forcing relation of the form:

$$s, h \models P \text{ if and only if } s, h \in [\![P]\!]$$

$$
\begin{aligned}
[\![E = E']\!]_\rho &= \{s, h \mid [\![E]\!]^s = [\![E']\!]^s\} \\
[\![E \mapsto E_1, E_2]\!]_\rho &= \{s, h \mid dom(h) = \{[\![E]\!]^s\} \\
&\quad\text{and } h([\![E]\!]^s) = \langle [\![E_1]\!]^s, [\![E_2]\!]^s \rangle\} \\
[\![\texttt{false}]\!]_\rho &= \emptyset \\
[\![P \Rightarrow Q]\!]_\rho &= ((S \times H) \setminus [\![P]\!]_\rho) \cup [\![Q]\!]_\rho \\
[\![\exists x.P]\!]_\rho &= \{s, h \mid \exists v \in \mathit{Val}.[s|x \to v], h \in [\![P]\!]_\rho\} \\
[\![\texttt{emp}]\!]_\rho &= \{s, h \mid h = [\,]\} \\
[\![P * Q]\!]_\rho &= \{s, h \mid \exists h_0, h_1.\ h_0 \sharp h_1,\ h = h_0 \cdot h_1 \\
&\quad s, h_0 \in [\![P]\!]_\rho \text{ and } s, h_1 \in [\![Q]\!]_\rho\} \\
[\![P \mathbin{-\!*} Q]\!]_\rho &= \{s, h \mid \forall h'.\ \text{if } h \sharp h' \text{ and } s, h' \in [\![P]\!]_\rho \text{ then} \\
&\quad s, h \cdot h' \in [\![Q]\!]_\rho\} \\
[\![X_v]\!]_\rho &= \rho(X_v),\ \text{if } X_v \in dom(\rho) \\
[\![\mu X_v . P]\!]_\rho &= \mathrm{lfp}_\emptyset^{\subseteq} \lambda X.\ [\![P]\!]_{[\rho|X_v \to X]} \\
[\![\nu X_v . P]\!]_\rho &= \mathrm{gfp}_\emptyset^{\subseteq} \lambda X.\ [\![P]\!]_{[\rho|X_v \to X]} \\
[\![P[E/x]]\!]_\rho &= \{s, h \mid [s \mid x \to [\![E]\!]^s], h \in [\![P]\!]_\rho\}
\end{aligned}
$$

Figure 2.2: *Semantics of $BI^{\mu\nu}$*

and an equivalence:

$P \equiv Q$ if and only if $\forall \rho.([\![P]\!]_\rho = [\![Q]\!]_\rho)$ or ($[\![P]\!]_\rho$ and $[\![Q]\!]_\rho$ both do not exist).

In both cases μ and ν, the X in λX is a fresh variable over sets of elements in $S \times H$ which does not already occur in ρ.

Notice that $[\![\cdot]\!]_\rho$ is only a partial function. In definitions above, $\mathrm{lfp}_\emptyset^{\subseteq} \lambda X.\ [\![P]\!]_{[\rho|X_v \to X]}$ ($\mathrm{gfp}_\emptyset^{\subseteq} \lambda X.\ [\![P]\!]_{[\rho|X_v \to X]}$) is the least fixpoint *(greatest fixpoint)* of the function $\lambda X.\ [\![P]\!]_{[\rho|X_v \to X]}$ on the poset $\langle \mathcal{P}(S \times H), \subseteq \rangle$, if it exists. Otherwise, $[\![\mu X_v.P]\!]_\rho$ ($[\![\nu X_v.P]\!]_\rho$) is not defined. For example, this is the case for $\mu X_v . (X_v \Rightarrow \texttt{false})$.

The syntactical criterions for formulae with defined semantics (like parity of negation under a fixpoint, etc.) are the usual ones knowing that in terms of monotonicity, $\mathbin{-\!*}$ acts like $\Rightarrow$, $*$ acts like $\wedge$, and $[\ /\ ]$ does not interfere. The fixpoint theory gives us criteria (using Tarski's fixpoint theorem) for the existence of $[\![P]\!]_\rho$, but no criteria for nonexistence. Nonetheless, we have these facts:

- if P is $E = E$ or $E \mapsto E_1, E_2$ or $\texttt{false}$ or $\texttt{emp}$, then $[\![P]\!]_\rho$ always exists; $\lambda X.[\![P]\!]_{[\rho|X_v \to X]}$

is monotonic and antitonic.

- $[\![X_v]\!]_\rho$ exists if and only if $X_v \in dom(\rho)$; $\lambda X.[\![X_v]\!]_{[\rho|Y_v \to X]}$ is monotonic and not antitonic.

- If P is $Q \Rightarrow R$ or $Q \mathbin{-\!\!*} R$, then $[\![P]\!]_\rho$ exists if and only if $[\![Q]\!]_\rho$ and $[\![R]\!]_\rho$ exist; $\lambda X.[\![P]\!]_{[\rho|X_v \to X]}$ is monotonic if and only if $\lambda X.[\![R]\!]_{[\rho|X_v \to X]}$ is monotonic and $\lambda X.[\![Q]\!]_{[\rho|X_v \to X]}$ is antitonic; $\lambda X.[\![P]\!]_{[\rho|X_v \to X]}$ is antitonic if and only if $\lambda X.[\![R]\!]_{[\rho|X_v \to X]}$ is antitonic and $\lambda X.[\![Q]\!]_{[\rho|X_v \to X]}$ is monotonic.

- $[\![Q * R]\!]_\rho$ exists if and only if $[\![Q]\!]_\rho$ and $[\![R]\!]_\rho$ exist; $\lambda X.[\![Q * R]\!]_{[\rho|X_v \to X]}$ is monotonic/antitonic if and only if $\lambda X.[\![R]\!]_{[\rho|X_v \to X]}$ and $\lambda X.[\![Q]\!]_{[\rho|X_v \to X]}$ are monotonic/antitonic.

- If P is $\exists x.\ Q$ or $Q[E/x]$, then $[\![P]\!]_\rho$ exists if and only if $[\![Q]\!]_\rho$ exists; $\lambda X.[\![P]\!]_{[\rho|X_v \to X]}$ is monotonic/antitonic if and only if $\lambda X.[\![Q]\!]_{[\rho|X_v \to X]}$ is monotonic/antitonic.

- If $\mu\nu \in \{\mu, \nu\}$ and if $\lambda X.[\![P]\!]_{[\rho|X_v \to X]}$ exists and is monotonic, then $[\![\mu\nu X_v.\ P]\!]_\rho$ exists and $\lambda X.[\![\mu\nu X_v.\ P]\!]_{[\rho|X_v \to X]}$ is monotonic and antitonic.

- If $\mu\nu \in \{\mu, \nu\}$ and if $\lambda X.[\![P]\!]_{[\rho|X_v \to X|Y_v \to Y]}$ is monotonic/antitonic, and $\lambda Y.[\![P]\!]_{[\rho|X_v \to X|Y_v \to Y]}$ exists and is monotonic, then $\lambda X.[\![\mu\nu Y_v.\ P]\!]_{[\rho|X_v \to X]}$ is monotonic/antitonic.

2.2.3 Interpretation of Triples

Hoare triples are of the form $\{P\}C\{Q\}$, where P and Q are assertions in $BI^{\mu\nu}$ and C is a command. The interpretation ensures that well-specified commands do not get stuck. (In this, it differs from the usual interpretation of Hoare triples[15].)

Definition 2.3. $\{P\}C\{Q\}$ *is a* true triple *if and only if* $\forall s, h$, *if* $s, h \models P$ *and* $FV(Q) \subseteq dom(s)$, *then*

- C, s, h *is safe*

- *if $C, s, h \leadsto^* s', h'$, then $s', h' \models Q$.*

This is a partial correctness interpretation; with looping, it does not guarantee termination. This is the reason for expressing "weakest liberal preconditions" for our backward analysis and not "weakest preconditions". However, the safety requirement rules out certain runtime errors and, as a result, we do not have that $\{\texttt{true}\}C\{\texttt{true}\}$ holds for all commands. For example, $\{\texttt{true}\}x := \texttt{nil}; x.1 := 3\{\texttt{true}\}$ is not a true triple.

2.2.4 Fixpoints and postponed substitution

In this section, we discuss our motivations for adding fixpoints and postponed substitution to separation logic. We show that the postponed substitution connective, [/], is not classical substitution, { / }, and that the usual variable renaming theorem does not hold for { / }. We develop the needed concepts in a series of vignettes:

First motivation

Our initial motivation for adding fixpoint operators to separation logic came from the habit of the separation logic community of informally defining recursive formulae and using them in proofs of correctness.

Since we have added fixed-point operators to the logic, we can formally and correctly express, for example, that x is a non-cyclic finite linear list as

$$\texttt{nclist}(x) = \mu X_v. \ (x = \texttt{nil}) \vee \exists x_1, x_2.(\texttt{isval}(x_1) \wedge (x \mapsto x_1, x_2 \ * \ X_v[x_2/x]))$$

and that x is non-cyclic finite or infinite list

$$\texttt{nclist}(x) = \nu X_v. \ (x = \texttt{nil}) \vee \exists x_1, x_2.(\texttt{isval}(x_1) \wedge (x \mapsto x_1, x_2 \ * \ X_v[x_2/x]))$$

where $\texttt{isval}(x) = (x = \texttt{true}) \vee (x = \texttt{false}) \vee (\exists n.n = x + 1)$

In earlier papers[16], Reynolds and O'Hearn use the definition,

$$\texttt{nclist}(x) = (x = \texttt{nil}) \vee \exists x_1, x_2.(\texttt{isval}(x_1) \wedge (x \mapsto x_1, x_2 \ * \ \texttt{nclist}(x_2)))$$

which is not within the syntax of separation logic.

Second motivation

The second motivation was the formulations of the *wlp* ({ ? }$C\{P\}$) and *sp* ({$P\}C\{$? })
in the case of `while` commands, which was not possible earlier. This problem is nontrivial:
For separation logic without fixed-points, we might express *sp* as

$$sp(P, \texttt{while } E \texttt{ do } C) = (\text{lfp}\,{}^{\models}_{\texttt{false}}\ \lambda X.sp(X \wedge E = \texttt{true}, C) \vee P) \wedge (E = \texttt{false})$$

with lfp $\,{}^{\models}_{\texttt{false}}\ \lambda X.F(X)$ defined, if it exists, as a formula P which satisfies:

- $P \equiv F(P)$

- for any formula Q, $(Q \equiv F(Q)$ implies $P \models Q$)

where

- $Q \models P$ if and only if $[\![Q]\!] \subseteq [\![P]\!]$ or $[\![Q]\!]$ and $[\![P]\!]$ are both not defined;

- $P \equiv Q$ if and only if $P \models Q$ and $Q \models P$.

This implies that during the computation of the *sp*, each time a `while` loop occurs, we must
find a formula in existing separation logic that was provably the fixpoint, so that we could
continue the computation of the *sp*. In another sense, this "work" could be seen as the
"work" of finding the strongest loop invariant in the application of the usual rule for `while`
loop.

Our addition of fixpoints (and the related postponed substitution) allows us to express
the *sp* directly within the logic:

$$sp(P, \texttt{while } E \texttt{ do } C) = (\mu X_v.sp(X_v \wedge E = \texttt{true}, C) \vee P) \wedge (E = \texttt{false}).$$

Although the definitions of the *wlp* and *sp* for the `while` loop are simple and elegant,
the "work" of finding loop invariants is not skipped, however it is now postponed for when
we have a specific proof to undertake. In the following chapters, we will present translations
of formulae into an other domain, and we have to find an approximation of the translation

of fixpoints which is precise and not too expensive to compute. The advantage here is that this work of building the translation is done once and for all, then the analysis can be fully automated while the methodology of a proof system and finding loop invariant implies hand work.

$[\,/\,]$ is not $\{\,/\,\}$

We use the notation $P\{E/x\}$ for capture-avoiding syntactical substitution (that is, the usual substitution of variables). Recall that $[\,/\,]$ is a connective of the logic (called *postponed substitution*) and is not equivalent to $\{\,/\,\}$. It might be helpful for the reader to understand $[\,/\,]$ to look at the formula $P[E/x]$ as (Moggi's[12]) call-by-value, let $x = E$ in P.

The distinction between $[\,/\,]$ and $\{\,/\,\}$ can be viewed in this example, where the command will be stuck in any state that has no value in its stack for y:

$$\{\texttt{true}\}x := y\{\texttt{true}\} \text{ is false}$$

This implies that the classical axiom for assignment, $\{P\{y/x\}\}x := y\{P\}$, is *unsound*.

In other versions of separation logic[6], $\{P\{y/x\}\}x := y\{P\}$ was sound, since the definition of a true triple required $FV(C,Q) \subseteq dom(s)$ and not merely $FV(Q) \subseteq dom(s)$, as here, *and also because there was no recursion.*

Our definition along with the choice to allow stacks to be partial functions does not require variables of the program to have a default value in the stack and it checks whether a variable has been assigned before we try to access its value. But the addition of fixpoints does not require stacks to be partial functions. (Indeed, if stacks were total functions, then more laws would hold for $[\,/\,]$, but the latter's definition would remain different from $\{\,/\,\}$'s.)

Unfolding

As usual, we have $\mu X_v.P \equiv P\{\mu X_v.P/X_v\}$
and $\nu X_v.P \equiv P\{\nu X_v.P/X_v\}$

See theorems and proofs in Sect. 2.6.4.

{/}: No variable renaming

Surprisingly, we have $\exists y.P \not\equiv \exists z.P\{z/y\}$ with $z \notin Var(P)$ *(when $y \neq z$)*. Here are two counterexamples, which expose the difficulties:

Counterexample 1:

$$\llbracket \nu X_v.y = 3 \wedge \exists y.(X_v \wedge y = 5) \rrbracket \not\equiv \llbracket \nu X_v.y = 3 \wedge \exists z.(X_v \wedge z = 5) \rrbracket$$

The left-hand side denotes the empty set, while the right-hand side denotes $\llbracket y = 3 \rrbracket$. Here are the detailed calculations:

$$
\begin{aligned}
&\llbracket \nu X_v.y = 3 \wedge \exists \boldsymbol{y}.(X_v \wedge \boldsymbol{y} = 5) \rrbracket_\emptyset \\
=\ & \mathrm{gfp}_\emptyset^\subseteq \lambda Y.\ \llbracket y = 3 \wedge \exists \boldsymbol{y}.(X_v \wedge \boldsymbol{y} = 5) \rrbracket_{[X_v \to Y]} \\
=\ & \mathrm{gfp}_\emptyset^\subseteq \lambda Y.\ \llbracket y = 3 \rrbracket_{[X_v \to Y]} \cap \llbracket \exists \boldsymbol{y}.(X_v \wedge \boldsymbol{y} = 5) \rrbracket_{[X_v \to Y]} \\
=\ & \mathrm{gfp}_\emptyset^\subseteq \lambda Y.\ \{s,h \mid s(y) = 3\} \cap \{s,h \mid \exists v.[s \mid \boldsymbol{y} \to v], h \in \llbracket X_v \wedge \boldsymbol{y} = 5 \rrbracket_{[X_v \to Y]}\} \\
=\ & \mathrm{gfp}_\emptyset^\subseteq \lambda Y.\ \{s,h \mid s(y) = 3\} \cap \{s,h \mid \exists v.[s \mid \boldsymbol{y} \to v], h \in Y \wedge [s \mid \boldsymbol{y} \to v](\boldsymbol{y}) = 5\} \\
=\ & \mathrm{gfp}_\emptyset^\subseteq \lambda Y.\ \{s,h \mid s(y) = 3\} \cap \{s,h \mid [s \mid \boldsymbol{y} \to 5], h \in Y\} \\
=\ & \boldsymbol{\emptyset}
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \nu X_v.y = 3 \wedge \exists \boldsymbol{z}.(X_v \wedge \boldsymbol{z} = 5) \rrbracket_\emptyset \\
=\ & \mathrm{gfp}_\emptyset^\subseteq \lambda Y.\ \llbracket y = 3 \wedge \exists \boldsymbol{z}.(X_v \wedge \boldsymbol{z} = 5) \rrbracket_{[X_v \to Y]} \\
=\ & \mathrm{gfp}_\emptyset^\subseteq \lambda Y.\ \llbracket y = 3 \rrbracket_{[X_v \to Y]} \cap \llbracket \exists \boldsymbol{z}.(X_v \wedge \boldsymbol{z} = 5) \rrbracket_{[X_v \to Y]} \\
=\ & \mathrm{gfp}_\emptyset^\subseteq \lambda Y.\ \{s,h \mid s(y) = 3\} \cap \{s,h \mid \exists v.[s \mid \boldsymbol{z} \to v], h \in \llbracket X_v \wedge \boldsymbol{z} = 5 \rrbracket_{[X_v \to Y]}\} \\
=\ & \mathrm{gfp}_\emptyset^\subseteq \lambda Y.\ \{s,h \mid s(y) = 3\} \cap \{s,h \mid \exists v.[s \mid \boldsymbol{z} \to v], h \in Y \wedge [s \mid \boldsymbol{z} \to v](\boldsymbol{z}) = 5\} \\
=\ & \mathrm{gfp}_\emptyset^\subseteq \lambda Y.\ \{s,h \mid s(y) = 3\} \cap \{s,h \mid [s \mid \boldsymbol{z} \to 5], h \in Y\} \\
=\ & \boldsymbol{\{s, h \mid s(y) = 3\}}
\end{aligned}
$$

Here is some intuition: For the left-hand side, $y = 3$ says that all the states defined by the assertion must bind y to 3, and "$\exists y.X_v \wedge y = 5$" says that for all those states defined by the assertion, we can bind y such that it satisfies $y = 5$, *even as it satisfies $y = 3$, due to the recursion*, which is impossible, so we have $\emptyset$ as the denotation.

For the right-hand side, $y = 3$ asserts again that y binds to 3, and $\exists z.X_v \wedge z = 5$ says that for all states in the assertion's denotation, we bind 5 to z, which is indeed possible, so we have $\llbracket y = 3 \rrbracket$ as the denotation of the assertion.

Counterexample 1 shows that variable renaming has a special behavior when applied to a formula which is not v-closed.

Counterexample 2:

$$[\![\exists y.\nu X_v.y = 3 \wedge \exists y.(X_v \wedge y = 5)]\!] \not\equiv [\![\exists z.\nu X_v.z = 3 \wedge \exists y.(X_v \wedge y = 5)]\!]$$

The left-hand side denotes the empty set, while the right-hand side denotes $S \times H$.

To see this, note that the left-hand side's semantics is essentially the same as its counterpart in the first counterexample. As for the right-hand side, if we apply the semantics of the right-hand side of the first counterexample, we see that $[\![\nu X_v.z = 3 \wedge \exists y.(X_v \wedge y = 5)]\!] = [\![z = 3]\!]$, signifying that all the states are such that we bind 5 to z. So, we have $S \times H$ as the denotation of the right-hand side.

Counterexample 2 shows that variables occurring free in the bodies of fixed-point formulae are subject to *dynamic binding* with respect to unrolling the recursive formulae via postponed substitution.

Full substitution

The previous counterexample 2 leads to the definition of a new substitution:

Definition 2.4. *Let* $\{[~/~]\}$ *be a full syntactical variable substitution:* $P\{[z/y]\}$ *is* P *in which all* y *are replaced by* z *wherever they occur, for example:*
$(\exists y.P)\{[z/y]\} \triangleq \exists z.(P\{[z/y]\})$, $(P[E/x])\{[z/y]\} \triangleq (P\{[z/y]\})[E\{z/y\}/x\{z/y\}]$

The variable renaming theorem for $BI^{\mu\nu}$

We define $class(z, s, h)$ as the set of states containing the state, s, h, and all other states identical to s, h except for z:

Definition 2.5. $class(z, s, h) = \{s', h \mid for~all~v \in Val, [s' \mid z \to v] = [s \mid z \to v]\}$

Alternatively, we can say that $class(z, s, h)$ is that set which satisfies:

- $s_{\restriction dom(s) \setminus \{z\}}, h \in class(z, s, h)$

- $\forall v.[s \mid z \rightarrow v], h \in class(z, s, h)$

Definition 2.6. *For $z \in Var$, $X \in \mathcal{P}(S \times H)$, define*

$$nodep(z, X) \triangleq True \text{ iff } \forall s, h \in X.class(z, s, h) \subseteq X$$

We extend this definition to environments as well:

$$nodep(z, \rho) \triangleq True \text{ iff } (\forall X_v \in dom(\rho).nodep(z, \rho(X_v)))$$

Proposition 2.7. *If $nodep(z, \rho)$, $FV_v(P) \subseteq dom(\rho)$, $z \notin FV(P)$ and $[\![P]\!]_\rho$ exists, then $nodep(z, [\![P]\!]_\rho)$*

The proof is found in Sect. 2.6.2.

The idea is, if P is v-closed and z does not occur free in P, then $\forall v.\ (s, h \in [\![P]\!]$ iff $[s \mid z \rightarrow v], h \in [\![P]\!])$. Yet another phrasing goes, if z does not occur free in a v-closed formula, then the set of states satisfying the formula does not have any particular values for z.

Now, let

- $s_{y,z}^\bullet \triangleq \begin{cases} [s \mid y \rightarrow s(z)] & \text{if } z \in dom(s) \\ s & \text{if } z \notin dom(s) \end{cases}$

- $\rho_{y,z}^\bullet \triangleq [\forall X_v \in dom(\rho).\ X_v \rightarrow \{s, h \mid s_{y,z}^\bullet, h \in \rho(X_v)\}]$

Proposition 2.8. *If $nodep(z, \rho)$ and $z \notin Var(P)$, then $[\![P\{[z/y]\}]\!]_{\rho_{y,z}^\bullet} = \{s, h \mid s_{y,z}^\bullet, h \in [\![P]\!]_\rho\}$*

Theorem 2.9. *If P is v-closed, $z \notin Var(P)$ and $y \notin FV(P)$, then $P \equiv P\{[z/y]\}$. In particular, $\exists y.P \equiv \exists z.(P\{[z/y]\})$.*

You can see proofs and details in Sect. 2.6.3.

Equivalences on $[\,/\,]$

For proofs see Sect. 2.6.8.

We define $is(E) \triangleq E = E$, which is just a formula ensuring that E has a value in the current state. If we had chosen that stacks were only total functions, $is(E)$ would always be equivalent to `true` and there would be more simplifications. We have these facts:

- If P does not contain any v-variable or fixpoint or postponed substitution, then
  $$P[E/x] \equiv P\{E/x\} \wedge is(E).$$

- If P is v-closed and if $x_1 \notin Var(E)$ and $x_1 \neq x_2$, then:
  $$(\exists x_1.P)[E/x_2] \equiv \exists x_1.(P[E/x_2]).$$

- $(\exists x.P)[E/x] \equiv (\exists x.P) \wedge is(E).$

- $(A \vee C)[E/x] \equiv (A[E/x]) \vee (C[E/x]).$

- If $y \notin Var(P)$, then
  $$(\mu X_v.P)[y/x] \equiv (\mu X_v.P\{[y/x]\}) \wedge is(y)$$
  $$(\nu X_v.P)[y/x] \equiv (\nu X_v.P\{[y/x]\}) \wedge is(y).$$

Concerning the last item, one would want a similar equivalence for E instead of y, but this is not possible since $(P[E'/x])\{[E/x]\}$ is not defined because $P[E'/E]$ is not defined. (The last argument must be a variable.) This explains why $[\,/\,]$ must be a connective.

To understand the last equivalence, we must return to the programming point of view, seeing fixpoints as `while` loops and $[\,/\,]$ as assignments, so that the precondition for $x := w$; `while` $x = y$ `do` $x := x + 1$ is the same as the one for `while` $w = y$ `do` $w := w + 1$. (In Sect. 2.3, we will learn that this will be $(\nu X_v.(x \neq y) \vee ((x = y) \wedge X_v[x + 1/x]))[w/x] \equiv is(w) \wedge (\nu X_v.(w \neq y) \vee ((w = y) \wedge X_v[w + 1/w])).)$

Example of unfolding

Let `nclist42(x)` $\triangleq \mu X_v.(x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * X_v[x_2/x])$ with $x_2 \neq x$. Let's prove that $X_v[x_2/x]$ is equivalent to `nclist42(x_2)`.

$$
\begin{aligned}
\texttt{nclist42}(x) \quad &\triangleq \quad \mu X_v.(x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * X_v[x_2/x]) \\
(unfolding) \quad &= \quad (x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2)* \\
&\qquad ((\mu X_v.(x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * X_v[x_2/x]))[x_2/x])) \\
(\text{Th. } 2.9) \quad &= \quad (x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2)* \\
&\qquad ((\mu X_v.(x = \texttt{nil}) \vee \exists x_3.((x \mapsto 42, x_3) * X_v[x_3/x]))[x_2/x])) \\
(\text{simplify } [\ /\ ] \ \text{case } \mu) \quad &= \quad (x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2)* \\
&\qquad (\mu X_v.(x_2 = \texttt{nil}) \vee \exists x_3.((x_2 \mapsto 42, x_3) * X_v[x_3/x_2]))) \\
&\triangleq \quad (x = \texttt{nil}) \vee \exists x_2.((x \mapsto 42, x_2) * \texttt{nclist42}(x_2))
\end{aligned}
$$

So we have $\texttt{nclist42}(x_2) \equiv (\texttt{nclist42}(x))[x_2/x]$, as expected.

Why is $BI + \mu + \nu \neq BI^{\mu\nu}$?

Or, why do we need to add $[\ /\ ]$ to the syntax? Informally stated, one can view the fixpoint as a while loop and $[\ /\ ]$ as an assignment, then if we have a while loop followed by an assignment, we cannot include the assignment within the loop. So, if an analysis postponed the computation of while loop (fixpoint), then it also has to postpone the computation of assignment ($[\ /\ ]$).

The need for $[\ /\ ]$ is not surprising. In [15], de Bakker proved that for his simple logic with fixpoints, there was no sp for the while loop statements.

Indeed, for P without any μ, ν, X_v in it, we have $P[E/x] \equiv P\{E/x\} \wedge is(E)$. But for $BI^{\mu\nu}$ without the connective $[\ /\ ]$, there is no formula in the logic equivalent to $P[E/x]$, which means that $[\ /\ ]$ has to be in the logic syntax. For example, $(\exists y.P)[E/x] \not\equiv \exists y.(P[E/x])$ when $y \neq x$ but $y \in Var(E)$ but the renaming theorem: $\exists y.P \equiv \exists z.P\{z/y\}$ with $z \notin Var(P)$ does not hold, so the attempt to find an equivalent formula for $(\exists y.P)[E/x]$ will fail.

2.3 Backward analysis

We now define the *weakest liberal precondition* (*wlp*) semantics of the while-loop language with pointers; see Fig. 2.3. Most of the clauses are from Ishtiaq and O'Hearn[10], but our

definition for `while E do C` is new and crucial. We add to wlp a parameter $V \in \mathcal{P}(Var)$, such that when choosing fresh variables, they are not in V.

If we can establish $\{P\}C\{\texttt{true}\}$, then we will know that execution of C is safe in any state satisfying P. So for our backward analysis, in Fig. 2.3 we express wlp such that

Theorem 2.10. $\llbracket wlp(P,C) \rrbracket = wlp_o(\llbracket P \rrbracket, C)$.

Corollary 2.11. $\{wlp(P,C)\}C\{P\}$ *is true.*

Proof. To prove that our definition indeed defines wlp, we formally relate it to the inverse state-transition function wlp_o: The definition of a true triple implies that:

$$\{wlp(P,C)\}C\{P\} \text{ true}$$
$$\text{if and only if}$$
$$\left(\begin{array}{c} \llbracket wlp(P,C) \rrbracket \\ \cap\{s,h \mid FV(P) \subseteq dom(s)\} \end{array} \right) \subseteq wlp_o(\llbracket P \rrbracket, C)$$

$$
\begin{array}{ccc}
BI^{\mu\nu} & \xleftarrow{\;wlp\;} & \boxed{BI^{\mu\nu}} \\
\llbracket \cdot \rrbracket \downarrow & & \downarrow \llbracket \cdot \rrbracket \\
op \subseteq & \xleftarrow{\;wlp_o\;} & op
\end{array}
$$

To prove that our analysis is correct, we express wlp_o for each command, and prove by induction on the syntax of C that for each C and P, we have $\llbracket wlp(P,C) \rrbracket \subseteq wlp_o(\llbracket P \rrbracket, C)$. To prove that those preconditions are the weakest we establish that $\llbracket wlp(P,C) \rrbracket = wlp_o(\llbracket P \rrbracket, C)$. For details, see Sect. 2.6.10. $\qquad \square$

Example: $wlp(\texttt{true}, \texttt{while } i > 0 \texttt{ do } (x := x \cdot 2; i := i - 1)) \triangleq \nu X_v.((i \leq 0 \wedge \texttt{true}) \vee (i > 0 \wedge \exists x_1, x_2. \ (X_v[i-1/i][x_2/x]) \wedge (x \mapsto x_1, x_2)))$, which simplifies to $\nu X_v.i \leq 0 \vee (i > 0 \wedge \exists x_1, x_2. \ X_v[i-1/i][x_2/x] \wedge x \mapsto x_1, x_2)$.

2.4 Forward analysis

In the previous section, we defined wlp for C and P such that $\{wlp(P,C)\}C\{P\}$ is true. Unfortunately, the strongest postcondition semantics $sp(P,C)$ is not always defined — we can find C and P such that there exists no Q that makes $\{P\}C\{Q\}$ true. This is due to the fact that a true triple requires C to be executable from all states satisfying P *and also such*

$$
\begin{aligned}
wlp(P, C) &= wlp_\emptyset(P, C) \\
wlp_V(P, x := E) &= P[E/x] \\
wlp_V(P, x := E.i) &= \exists x_1 \exists x_2.(P[x_i/x] \wedge (E \hookrightarrow x_1, x_2)) \\
&\qquad \text{with } x_i \notin V \cup FV(E, P) \\
wlp_V(P, E.1 := E') &= (E' = E') \wedge \exists x_1 \exists x_2.(E \mapsto x_1, x_2) * ((E \mapsto E', x_2) \!\!-\!\!* P) \\
&\qquad \text{with } x_i \notin V \cup FV(E, E', P) \\
wlp_V(P, E.2 := E') &= (E' = E') \wedge \exists x_1 \exists x_2.(E \mapsto x_1, x_2) * ((E \mapsto x_1, E') \!\!-\!\!* P) \\
&\qquad \text{with } x_i \notin V \cup FV(E, E', P) \\
wlp_V(P, x := \mathtt{cons}(E_1, E_2)) &= (E_1 = E_1) \wedge (E_2 = E_2) \wedge \forall x'.(x' \mapsto E_1, E_2) \!\!-\!\!* P[x'/x] \\
&\qquad \text{with } x' \notin V \cup FV(E_1, E_2, P) \\
wlp_V(P, \mathtt{dispose}(E)) &= P * (\exists a \exists b.(E \mapsto a, b)) \\
&\qquad \text{with } a, b \notin V \cup FV(E) \\
wlp_V(P, C_1; C_2) &= wlp_V(wlp_V(P, C_2), C_1) \\
wlp_V(P, \mathtt{if}\ E\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2) &= (E = \mathtt{true} \wedge wlp_V(P, C_1)) \\
&\qquad \vee (E = \mathtt{false} \wedge wlp_V(P, C_2)) \\
wlp_V(P, \mathtt{skip}) &= P \\
wlp_V(P, \mathtt{while}\ E\ \mathtt{do}\ C_1) &= \nu X_v.((E = \mathtt{true} \wedge wlp_{V \cup Var(E,P)}(X_v, C_1)) \\
&\qquad\qquad \vee (E = \mathtt{false} \wedge P)) \\
&\qquad \text{with } X_v \text{ not in } P
\end{aligned}
$$

Figure 2.3: *Weakest liberal preconditions*

that $FV(Q) \subseteq dom(s)$ which is obviously *not* the case for some C and P. (For example, $\{\texttt{true}\}x := nil; y := x.1\{?\}$ has no solution, since all states satisfy P but the command can never be executed — $\texttt{nil}.1$ is not defined).

We therefore split the analysis into two steps. The first step checks whether C is executable from all states satisfying P or not. The second step gives $sp(P, C)$ that makes the triple $\{P\}C\{sp(P, C)\}$ true if C is executable from all states satisfying P.

Step 1 : $wlp(\texttt{true}, C)$**:**

$$(\forall s, h \in \llbracket P \rrbracket. \ C, s, h \text{ is safe}) \text{ if and only if } (P \models wlp(\texttt{true}, C))$$

The first step expresses the $wlp(\texttt{true}, C)$ formulae, which are the formulae given in Fig. 2.3, instantiated for $P = \texttt{true}$.

Step 2: $sp(P, C)$**:** This is given in Fig. 2.4.

In other words, our formula $sp(P, C)$ does not characterise all the states reached after an execution of C from a state satisfying P but only, if there is a state resulting of the execution of C from a state satisfying P, then this state satisfies $sp(P, C)$.

This gives us

Theorem 2.12. $sp_o(\llbracket P \rrbracket, C) = \llbracket sp(P, C) \rrbracket$.

Corollary 2.13. $\{P \land wlp(true, C)\}C\{sp(P \land wlp(true, C), C)\}$ *is always true.*
In case $P \models wlp(\texttt{true}, C)$ *this is equivalent to* $\{P\}C\{sp(P, C)\}$ *is true.*

Corollary 2.14. *If* $P \not\models wlp(\texttt{true}, C)$*, then* C *cannot be executable from all states satisfying* P*. But for those states from which* C *is executable, the final states satisfy* $sp(P \land wlp(\texttt{true}, C), C)$*.*

Our $sp(P, C)$ makes the triple $\{P\}C\{sp(P, C)\}$ always true in the usual definition of Hoare triples (which is $\{P\}C\{Q\}$ true iff $sp_o(\llbracket P \rrbracket, C) \subseteq \llbracket Q \rrbracket$).

Proof. To prove that our definition indeed defines sp, we formally relate it to the inverse state-transition function sp_o. The definition of a true triple implies that

$$\{P\}C\{Q\} \text{ if } P \models wlp(\texttt{true}, C) \wedge sp_o(\llbracket P \rrbracket, C) \subseteq \llbracket Q \rrbracket$$

(We could also have written

$$\{P\}C\{Q\}$$
$$\text{if and only if}$$
$$(\llbracket P \rrbracket \cap \{s, h \mid FV(P) \subseteq dom(s)\}) \subseteq \llbracket wlp(\texttt{true}, C) \rrbracket$$
$$\wedge\, (sp_o(\llbracket P \rrbracket, C) \cap \{s, h \mid FV(P) \subseteq dom(s)\}) \subseteq \llbracket Q \rrbracket$$

)

To prove that our analysis is correct, we expressed sp_o for each command, and proved by induction on the syntax of C that for each C, and P, we have

If $P \models wlp(\texttt{true}, C)$
then $sp_o(\llbracket P \rrbracket, C) \subseteq \llbracket sp(P, C) \rrbracket$

But since sp_o is defined such that it only collects the final states of successful computations, we must only prove that for each C and P: $sp_o(\llbracket P \rrbracket, C) \subseteq \llbracket sp(P, C) \rrbracket$.

Finally, to prove that those postconditions are the strongest we have established that $sp_o(\llbracket P \rrbracket, C) = \llbracket sp(P, C) \rrbracket$. For details see Sect. 2.6.9. $\qquad\square$

Example: $sp(\texttt{true}, i := 0; x := \texttt{nil}; \texttt{while } i \neq 5 \texttt{ do } x := \texttt{cons}(i, x); i := i + 1) \triangleq i = 5 \wedge \mu X_v.((\exists x'.(\exists i'.\texttt{true}[i'/i] \wedge i = 0)[x'/x] \wedge x = \texttt{nil}) \vee \exists i'.(\exists x'.(X_v \wedge i \neq 5)[x'/x] * (x \mapsto i, x'))[i'/i] \wedge i = i' + 1)$, which is after simplifications, $i = 5 \wedge \mu X_v.((i = 0 \wedge x = \texttt{nil}) \vee \exists x'.i'.i = i' + 1 \wedge i' \neq 5 \wedge (X_v[x'/x] * (x \mapsto i', x')))$

2.5 Variations of $BI^{\mu\nu}$

Our version of $BI^{\mu\nu}$ is not unique. One variant would preserve the usual renaming theorem but at the price of additional complexity in defining fixed-point formulae: the v-variables

$$
\begin{aligned}
sp(P, C) &= sp_\emptyset(P, C) \\
sp_V(P, x := E) &= \exists x'.\ P[x'/x] \wedge x = E\{x'/x\} \\
&\qquad \text{with } x' \notin V \cup FV(E, P) \\
sp_V(P, x := E.i) &= \exists x'.\ P[x'/x] \wedge x = (E\{x'/x\}).i \\
&\qquad \text{with } x' \notin V \cup FV(E, P) \\
sp_V(P, E.1 := E') &= \exists x_1 \exists x_2.(E \mapsto E', x_2) * ((E \mapsto x_1, x_2) \mathbin{-\!\!*} P) \\
&\qquad \text{with } x_i \notin V \cup FV(E, E', P) \\
sp_V(P, E.2 := E') &= \exists x_1 \exists x_2.(E \mapsto x_1, E') * ((E \mapsto x_1, x_2) \mathbin{-\!\!*} P) \\
&\qquad \text{with } x_i \notin V \cup FV(E, E', P) \\
sp_V(P, x := \mathtt{cons}(E_1, E_2)) &= \exists x'.(P[x'/x] * (x \mapsto E_1\{x'/x\}, E_2\{x'/x\})) \\
&\qquad \text{with } x' \notin V \cup FV(E_1, E_2, P) \\
sp_V(P, \mathtt{dispose}(E)) &= \exists x_1, x_2.\ \neg((E \mapsto x_1, x_2) \mathbin{-\!\!*} \neg P) \\
&\qquad \text{with } x_1, x_2 \notin V \cup FV(E, P) \\
sp_V(P, C_1; C_2) &= sp_V(sp_V(P, C_1), C_2) \\
sp_V(P, \mathtt{if}\ E\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2) &= sp_V(P \wedge E = \mathtt{true}, C_1) \\
&\quad \vee sp_V(P \wedge E = \mathtt{false}, C_2) \\
sp_V(P, \mathtt{skip}) &= P \\
sp_V(P, \mathtt{while}\ E\ \mathtt{do}\ C_1) &= (\mu X_v.sp_{V \cup Var(E, P)}(X_v \wedge E = \mathtt{true}, C_1) \vee P) \\
&\quad \wedge (E = \mathtt{false}) \\
&\qquad \text{with } X_v \text{ not in } P
\end{aligned}
$$

Figure 2.4: *Strongest postconditions*

become functions whose parameters are the free variables of the formula. Instead of having postponed substitution, one would have an application connective. The syntax reads

$$\text{nonCircularList}(x) =$$
$$\mu X_v(x).\ (x = \texttt{nil}) \lor \exists x_1, x_2.(\texttt{isval}(x_1) \land (x \mapsto x_1, x_2 \ * \ X_v(x_2)))$$

When considering our example to the renaming theorem, one states

$$\exists y.\nu X_v(y).y = 3 \land \exists y.(X_v(y) \land y = 5)$$

which becomes equivalent to

$$\exists z.\nu X_v(z).z = 3 \land \exists y.(X_v(y) \land y = 5)$$

Those formulae are not precisely stated; let us try to formalize. The changes are that $\rho : Var_v \rightharpoonup (\mathcal{P}(Var) \times \mathcal{P}(S \times H))$ and that $\llbracket \cdot \rrbracket_\rho \ : \ BI^{\mu\nu\Lambda} \rightharpoonup (\mathcal{P}(S \times H) \uplus (\mathcal{P}(Var) \times \mathcal{P}(S \times H)))$.

The semantics for fixpoints and postponed substitution would be:

$$
\begin{aligned}
\llbracket \mu X_v(x_1, ..., x_n)\ .\ P \rrbracket_\rho &= \text{lfp}_{\emptyset}^{\subseteq} \lambda X.\ \llbracket P \rrbracket_{[\rho|X_v \to ((x_1,...,x_n),X)]} \\
\llbracket \nu X_v(x_1, ..., x_n)\ .\ P \rrbracket_\rho &= \text{gfp}_{\emptyset}^{\subseteq} \lambda X.\ \llbracket P \rrbracket_{[\rho|X_v \to ((x_1,...,x_n),X)]} \\
\llbracket P(E_1, ..., E_n) \rrbracket_\rho &= \left\{ s, h \ \middle| \ \begin{array}{l} \exists x_1, ..., x_n, X.\ \llbracket P \rrbracket_\rho = ((x_1, ..., x_n), X) \\ \land [s \mid x_1 \to \llbracket E_1 \rrbracket^s \mid ... \mid x_n \to \llbracket E_n \rrbracket^s], h \in X \end{array} \right\}
\end{aligned}
$$

But this implies that to write $\mu X_v(x_1, ..., x_n)$. P, we must consider the free variables in P. (Maybe this would help the users of the logic!)

Another important point is that $\llbracket \nu X_v.X_v \rrbracket = S \times H$, while $\llbracket \nu X_v(x).X_v(x) \rrbracket = \{s, h \mid x \in dom(s)\}$. That is, if one allows partial functions for stacks (as we do), the meaning changes.

To update our definition, $wlp(P, x := E) = P[E/x]$, we require a function connective, and we write $wlp(P, x := E) = (\Lambda x.P)(E)$. (we use Λ instead of λ to avoid confusion between a real function and the new function connective.) And instead of writing $\mu X_v(x_1, ..., x_n).P$, we would write $\mu(\Lambda(x_1, ..., x_n).X_v).P$. The non-circular linear list example reads as follows:

$$\text{nonCircularList}(x) =$$
$$\mu X_v.\ (x = \texttt{nil}) \lor \exists x_1, x_2.(\texttt{isval}(x_1) \land (x \mapsto x_1, x_2 \ * \ (\Lambda x.X_v)(x_2)))$$

This implies a new semantics: First, we define a new recursive type, $res = \mathcal{P}(S \times H) \uplus (Var \times res)$. Next, we define

$$
\begin{aligned}
apply &: (Exp \times (Var \times res)) \to res \\
apply(E, (x, S)) &= \{s, h \mid [s \mid x \to [\![E]\!]^s], h \in S\} \quad \text{if } S \in \mathcal{P}(S \times H) \\
apply(E, (x, (y, S))) &= (y, apply(E, (x, S)))
\end{aligned}
$$

$$
\begin{aligned}
[\![\cdot]\!]_\rho &: BI^{\mu\nu\Lambda} \rightharpoonup res \\
&\quad \dots \\
[\![\Lambda x.P]\!]_\rho &= ((x), [\![P]\!]_\rho) \\
[\![\mu X_v . P]\!]_\rho &= \mathrm{lfp}_\emptyset^{\subseteq} \lambda X. \, [\![P]\!]_{[\rho \mid X_v \to X]} \\
[\![\nu X_v . P]\!]_\rho &= \mathrm{gfp}_\emptyset^{\subseteq} \lambda X. \, [\![P]\!]_{[\rho \mid X_v \to X]} \\
[\![P(E)]\!]_\rho &= apply(E, [\![P]\!]_\rho) \quad \text{if } \exists x, X. \, [\![P]\!]_\rho = (x, X)
\end{aligned}
$$

With this semantics, in wlp or sp, the only change is that wherever $P[E/x]$ appears, it should be replaced by $(\Lambda x.P)E$.

As for our counterexample of the renaming theorem,

$$
\exists y.\nu X_v.y = 3 \wedge \exists y.(((\Lambda y.X_v)y) \wedge y = 5)
$$

becomes equivalent to

$$
\exists z.\nu X_v.z = 3 \wedge \exists y.(((\Lambda z.X_v)y) \wedge y = 5)
$$

But again, this is not the usual renaming theorem.

What we propose now is a mix of the last two semantics, where the renaming theorem will not always hold, but if one wants it to hold, then one must verify, wherever there is a fixpoint, that the fixpoint should be written in the format, $\mu X_v(FV(P)).P$. The user must be aware that $(\Lambda x.P)(x)$ is not always equivalent to P (because $(\Lambda x.P)(x)$ implies that x can be evaluated in the actual context (i.e. $x \in dom(s)$)).

Now $\rho : Var_v \rightharpoonup res$, and the example becomes

$$
\exists y.\nu X_v(y).y = 3 \wedge \exists y.(X_v(y) \wedge y = 5)
$$

which is equivalent to

$$
\exists z.\nu X_v(z).z = 3 \wedge \exists y.(X_v(y) \wedge y = 5)
$$

The semantics goes as follows; omitted clauses are the same as those in Section 2.2.2.

$$
\begin{aligned}
\llbracket \cdot \rrbracket_\rho &: \quad BI^{\mu\nu\Lambda} \rightharpoonup res \\
&\quad ... \\
\llbracket \Lambda x.P \rrbracket_\rho &= \quad ((x), \llbracket P \rrbracket_\rho) \\
\llbracket \mu X_v \, . \, P \rrbracket_\rho &= \quad \text{lfp}_\emptyset^{\subseteq} \lambda X. \, \llbracket P \rrbracket_{[\rho | X_v \to X]} \\
\llbracket \mu X_v(x_1, ..., x_n) \, . \, P \rrbracket_\rho &= \quad \text{lfp}_\emptyset^{\subseteq} \lambda X. \, \llbracket P \rrbracket_{[\rho | X_v \to (x_1, (..., (x_n, X)))]} \\
\llbracket \nu X_v \, . \, P \rrbracket_\rho &= \quad \text{gfp}_\emptyset^{\subseteq} \lambda X. \, \llbracket P \rrbracket_{[\rho | X_v \to X]} \\
\llbracket \nu X_v(x_1, ..., x_n) \, . \, P \rrbracket_\rho &= \quad \text{gfp}_\emptyset^{\subseteq} \lambda X. \, \llbracket P \rrbracket_{[\rho | X_v \to (x_1, (..., (x_n, X)))]} \\
\llbracket P(E) \rrbracket_\rho &= \quad apply(E, \llbracket P \rrbracket_\rho) \text{ if } \exists x, X. \, \llbracket P \rrbracket_\rho = (x, X)
\end{aligned}
$$

(We gave the semantics for $\mu X_v.P$ and $\nu X_v.P$ separately but they are the nullary case of the other considering that $(x_1, (..., (x_0, X))) = X$).

This semantics preserves our *wlp* and *sp* formulae (switching $P[E/x]$ with $(\Lambda x.P)(E)$) and allows the user to have functions for v-variable and a restricted renaming theorem.

Later, some people added recursion to separation logic in other ways like the grammars of Yang and al in[17] and in Biering and al's work[18] they present a higher-order separation logic where the logic allows quantifiers over sets of states.

2.6 Appendix

2.6.1 Definitions

We recall the definitions of Var (Def. 2.15), FV (Def. 2.16), FV_v (Def. 2.17), v-closed (Def. 2.18), $\{[\,/\,]\}$ (Def. 2.19):

Definition 2.15.

$$
\begin{aligned}
Var(x) &= \{x\} \\
Var(42) &= \emptyset \\
Var(\texttt{nil}) &= \emptyset \\
Var(True) &= \emptyset \\
Var(False) &= \emptyset \\
Var(E_1 op E_2) &= Var(E_1) \cup Var(E_2) \\
&\dots \\
Var(E_1 = E_2) &= Var(E_1) \cup Var(E_2) \\
Var(E \mapsto E_1, E_2) &= Var(E) \cup Var(E_1) \cup Var(E_2) \\
Var(\texttt{false}) &= \emptyset \\
Var(P \Rightarrow Q) &= Var(P) \cup Var(Q) \\
Var(\exists x.P) &= Var(P) \cup \{x\} \\
Var(\texttt{emp}) &= \emptyset \\
Var(P * Q) &= Var(P) \cup Var(Q) \\
Var(P \mathbin{-\!\!*} Q) &= Var(P) \cup Var(Q) \\
Var(X_v) &= \emptyset \\
Var(\mu X_v.P) &= Var(P) \\
Var(\nu X_v.P) &= Var(P) \\
Var(P[E/x]) &= Var(P) \cup Var(E) \cup \{x\}
\end{aligned}
$$

Definition 2.16.

$$
\begin{aligned}
FV(E_1 = E_2) &= Var(E_1) \cup Var(E_2) \\
FV(E \mapsto E_1, E_2) &= Var(E) \cup Var(E_1) \cup Var(E_2) \\
FV(\texttt{false}) &= \emptyset \\
FV(P \Rightarrow Q) &= FV(P) \cup FV(Q) \\
FV(\exists x.P) &= FV(P) \setminus \{x\} \\
FV(\texttt{emp}) &= \emptyset \\
FV(P * Q) &= FV(P) \cup FV(Q) \\
FV(P \mathbin{-\!\!*} Q) &= FV(P) \cup FV(Q) \\
FV(X_v) &= \emptyset \\
FV(\mu X_v.P) &= FV(P) \\
FV(\nu X_v.P) &= FV(P) \\
FV(P[E/x]) &= FV(P) \cup FV(E) \cup \{x\}
\end{aligned}
$$

Definition 2.17.

$$
\begin{aligned}
FV_v(E_1 = E_2) &= \emptyset \\
FV_v(E \mapsto E_1, E_2) &= \emptyset \\
FV_v(\texttt{false}) &= \emptyset \\
FV_v(P \Rightarrow Q) &= FV_v(P) \cup FV_v(Q) \\
FV_v(\exists x.P) &= FV_v(P) \\
FV_v(\texttt{emp}) &= \emptyset \\
FV_v(P * Q) &= FV_v(P) \cup FV_v(Q) \\
FV_v(P \mathbin{-\!\!*} Q) &= FV_v(P) \cup FV_v(Q) \\
FV_v(X_v) &= \{X_v\} \\
FV_v(\mu X_v.P) &= FV_v(P) \setminus \{X_v\} \\
FV_v(\nu X_v.P) &= FV_v(P) \setminus \{X_v\} \\
FV_v(P[E/x]) &= FV_v(P)
\end{aligned}
$$

Definition 2.18. *P is v-closed iff* $FV_v(P) = \emptyset$.

Definition 2.19.

$$
\begin{aligned}
x\{z/y\} &= x \text{ if } y \neq x \\
y\{z/y\} &= z \\
42\{z/y\} &= 42 \\
\texttt{nil}\{z/y\} &= \texttt{nil} \\
True\{z/y\} &= True \\
False\{z/y\} &= False \\
(E_1 op E_2)\{z/y\} &= E_1\{z/y\} op E_2\{z/y\} \\
&\ldots \\
(E_1 = E_2)\{[z/y]\} &= E_1\{z/y\} = E_2\{z/y\} \\
(E \mapsto E_1, E_2)\{[z/y]\} &= E\{z/y\} \mapsto E_1\{z/y\}, E_2\{z/y\} \\
(\texttt{false})\{[z/y]\} &= \texttt{false} \\
(P \Rightarrow Q)\{[z/y]\} &= (P\{[z/y]\}) \Rightarrow (Q\{[z/y]\}) \\
(\exists x.P)\{[z/y]\} &= \exists(x\{z/y\}).(P\{[z/y]\}) \\
(\texttt{emp})\{[z/y]\} &= \texttt{emp} \\
(P * Q)\{[z/y]\} &= (P\{[z/y]\}) * (Q\{[z/y]\}) \\
(P \mathbin{-\!\!*} Q)\{[z/y]\} &= (P\{[z/y]\}) \mathbin{-\!\!*} (Q\{[z/y]\}) \\
(X_v)\{[z/y]\} &= X_v \\
(\mu X_v.P)\{[z/y]\} &= \mu X_v.(P\{[z/y]\}) \\
(\nu X_v.P)\{[z/y]\} &= \nu X_v.(P\{[z/y]\}) \\
(P[E/x])\{[z/y]\} &= (P\{[z/y]\})[E\{z/y\}/x\{z/y\}]
\end{aligned}
$$

2.6.2   Stack Extension Theorem

Definition 2.20.

$$classe(z, s, h) = \{s', h \mid [s' \mid z \rightarrow 42] = [s \mid z \rightarrow 42]\}$$

May be it's more clear to say that

$s_{\upharpoonright dom(s) \setminus \{z\}}, h \in classe(z, s, h)$

$\forall v.[s \mid z \rightarrow v], h \in classe(z, s, h)$.

This is the $classe(z, s, h)$ is the set of states containing s, h and all states similar to s, h for everything but for z.

Definition 2.6. *For $z \in Var$, $X \in \mathcal{P}(S \times H)$,*

$$nodep(z, X) \triangleq True \; iff \; \forall s, h \in X.classe(z, s, h) \subseteq X$$

We extend this definition to environments:

$$nodep(z, \rho) \triangleq True \; iff \; (\forall X_v \in dom(\rho).nodep(z, \rho(X_v)))$$

Notice that we have $\forall z.nodep(z, \emptyset)$

Theorem 2.7. If $\begin{array}{l} - \quad nodep(z, \rho) \\ - \quad FV_v(P) \subseteq dom(\rho) \\ - \quad z \notin FV(P) \\ - \quad \llbracket P \rrbracket_\rho \; exists \end{array}$ then $nodep(z, \llbracket P \rrbracket_\rho)$.

Corollary 2.21. If $\begin{array}{l} - \quad z \notin FV(P) \\ - \quad \llbracket P \rrbracket \; exists \end{array}$ then $nodep(z, \llbracket P \rrbracket)$.

The idea of the theorem would be, if P is v-closed, z does not occur free in P, then $\forall v. \; (s, h \in \llbracket P \rrbracket$ iff $[s \mid z \rightarrow v], h \in \llbracket P \rrbracket)$.

We could say it as, if z does not occur free in a v-closed formula, then set of states satisfying the formula does not have any particular values for z.

Cor. 2.21. Direct from Th. 2.7. □

Th. 2.7. THE PROOF IS ONLY MADE IF ALL THE lfp and gpf are for MONOTONIC FUNCTIONS. But this is the case for the *wlp* and *sp* formulae and for the example in the paper.

First notice that if $z \notin Var(E).\llbracket E \rrbracket s = \llbracket E \rrbracket^{[s|z \to v]}$.

$\llbracket x \rrbracket^s = s(x) = \llbracket x \rrbracket^{[s|z \to v]}$

$\llbracket 42 \rrbracket^s = 42 = \llbracket 42 \rrbracket^{[s|z \to v]}$

$\llbracket True \rrbracket^s = true = \llbracket x \rrbracket^{[s|z \to v]}$

$\llbracket E_1 op E_2 \rrbracket^s = \llbracket E_1 \rrbracket^s op \llbracket E_2 \rrbracket^s = \llbracket E_1 \rrbracket^{[s|z \to v]} op \llbracket E_2 \rrbracket^{[s|z \to v]} = \llbracket x \rrbracket^{[s|z \to v]}$

We proceed by induction on P. We do not write down when we use induction to have the conditions for $z \notin FV(...)$.

- $s, h \in \llbracket E_1 = E_2 \rrbracket$

    iff $\llbracket E_1 \rrbracket s = \llbracket E_2 \rrbracket s$

    iff $\llbracket E_1 \rrbracket^{[s|z \to v]} = \llbracket E_2 \rrbracket^{[s|z \to v]})\}$

    iff $[s \mid z \to v], h \in \llbracket E_1 = E_2 \rrbracket$

- $s, h \in \llbracket E \mapsto E_1, E_2 \rrbracket$

    iff $dom(h) = \{\llbracket E \rrbracket s\}$

    and $h(\llbracket E \rrbracket s) = \langle \llbracket E_1 \rrbracket s, \llbracket E_2 \rrbracket s \rangle$

    iff $dom(h) = \{\llbracket E \rrbracket^{[s|z \to v]}\}$

    and $h(\llbracket E \rrbracket^{[s|z \to v]}) = \langle \llbracket E_1 \rrbracket^{[s|z \to v]}, \llbracket E_2 \rrbracket^{[s|z \to v]} \rangle$

    iff $[s \mid z \to v], h \in \llbracket E \mapsto E_1, E_2 \rrbracket$

- $s, h \in \llbracket \texttt{false} \rrbracket$

    iff $s, h \in \emptyset$

    iff $[s \mid z \to v], h \in \emptyset$

    iff $[s \mid z \to v], h \in \llbracket \texttt{false} \rrbracket$

- $s, h \in \llbracket P \Rightarrow Q \rrbracket_\rho$

    iff $s, h \in (\top \setminus \llbracket P \rrbracket_\rho) \cup \llbracket Q \rrbracket_\rho$

    iff $[s \mid z \to v], h \in (\top \setminus \{s, h \mid \llbracket P \rrbracket_\rho) \cup \llbracket Q \rrbracket_\rho$ (ind.)

    iff $[s \mid z \to v], h \in \llbracket P \Rightarrow Q \rrbracket_\rho$

- $s, h \in \llbracket \exists x.P \rrbracket_\rho$ when $x \neq z$

45

iff $\exists v'.[s \mid x \mapsto v'], h \in \llbracket P \rrbracket_\rho$

   iff $\exists v'.[s \mid x \mapsto v' \mid z \mapsto v], h \in \llbracket P \rrbracket_\rho$ (ind.)

   iff $\exists v'.[s \mid z \mapsto v \mid x \mapsto v'], h \in \llbracket P \rrbracket_\rho$

   iff $[s \mid z \to v], h \in \llbracket \exists x.P \rrbracket_\rho$

- $s, h \in \llbracket \exists z.P \rrbracket_\rho$

   iff $\exists v'.[s \mid z \mapsto v'], h \in \llbracket P \rrbracket_\rho$

   iff $\exists v'.[s \mid z \mapsto v \mid z \mapsto v'], h \in \llbracket P \rrbracket_\rho$

   iff $[s \mid z \to v], h \in \llbracket \exists z.P \rrbracket_\rho$

- $s, h \in \llbracket \mathtt{emp} \rrbracket$

   iff $h = []$

   iff $[s \mid z \to v], h \in \llbracket \mathtt{emp} \rrbracket$

- $s, h \in \llbracket P * Q \rrbracket_\rho$

   iff $\exists h_0, h_1.h_0 \sharp h_1, h = h_0 \cdot h_1$
       $s, h_0 \in \llbracket P \rrbracket_\rho$ and $s, h_1 \llbracket Q \rrbracket_\rho$

   iff $\exists h_0, h_1.h_0 \sharp h_1, h = h_0 \cdot h_1$
       $[s \mid z \to v], h_0 \in \llbracket P \rrbracket_\rho$ and $[s \mid z \to v], h_1 \llbracket Q \rrbracket_\rho$ (ind.)

   iff $[s \mid z \to v], h \in \llbracket P * Q \rrbracket_\rho$

- $s, h \in \llbracket P \mathbin{-\!\!*} Q \rrbracket_\rho$

   iff $\forall h'., h_1.h' \sharp h.$ if $s, h \in \llbracket P \rrbracket_\rho$
       then $s, h \cdot h' \in \llbracket Q \rrbracket_\rho$

   iff $\forall h'., h_1.h' \sharp h.$ if $[s \mid z \to v], h \in \llbracket P \rrbracket_\rho$
       then $[s \mid z \to v], h \cdot h' \in \llbracket Q \rrbracket_\rho$ (ind.)

   iff $[s \mid z \to v], h \in \llbracket P \mathbin{-\!\!*} Q \rrbracket_\rho$

- $s, h \in \llbracket X_v \rrbracket_\rho$

   iff $s, h \in \rho(X_v)$

   iff $[s \mid z \mapsto v], h \in \rho(X_v)$ (hyp.)

   iff $[s \mid z \mapsto v], h \in \llbracket X_v \rrbracket_\rho$

- $s, h \in \llbracket P[E/x] \rrbracket_\rho$

   iff $[s \mid x \to \llbracket E \rrbracket^s], h \in \llbracket P \rrbracket_\rho$

   iff $[s \mid x \to \llbracket E \rrbracket^s \mid z \mapsto v], h \in \llbracket P \rrbracket_\rho$ (ind.)

   iff $[s \mid z \mapsto v \mid x \to \llbracket E \rrbracket^s], h \in \llbracket P \rrbracket_\rho$ (hyp. $z \neq x$)

   iff $[s \mid z \mapsto v \mid x \to \llbracket E \rrbracket^{[s|z \mapsto v]}], h \in \llbracket P \rrbracket_\rho$ (hyp. $z \notin Var(E)$)

   iff $[s \mid z \mapsto v], h \in \llbracket P[E/x] \rrbracket_\rho$

- $s, h \in \llbracket \mu X_v.P \rrbracket_\rho$

  iff $\quad s, h \in \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X.\ \llbracket P \rrbracket_{[\rho|X_v \to X]}$

  iff $\quad [s \mid z \mapsto v], h \in \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X.\ \llbracket P \rrbracket_{[\rho|X_v \to X]}$ $\quad$ (proof below)

  iff $\quad [s \mid z \mapsto v], h \in \llbracket \mu X_v.P \rrbracket_\rho$

  Let $A \triangleq \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X.\ \llbracket P \rrbracket_{[\rho|X_v \to X]}$, we want to prove $nodep(z, A)$, we proceed by contradiction. Let $B \triangleq \{s, h \in A \mid [s \mid z \mapsto v], h \notin A\} \cup \{[s \mid z \mapsto v], h \in A \mid s, h \notin A\}$,

  Let $C \triangleq A \setminus B$, by construction is the biggest set such that $nodep(z, C)$ and $C \subseteq A$

  since $nodep(z, \rho)$ we then have $nodep(z, [\rho \mid X_v \to C])$, then by induction we have $nodep(z, \llbracket P \rrbracket_{[\rho|X_v \to C]})$.

  Let $F \triangleq \lambda X.\ \llbracket P \rrbracket_{[\rho|X_v \to X]}$, we have $nodep(z, F(C))$.

  ....

  If F is monotonic, then since $C \subseteq A$ we have $F(C) \subseteq F(A)$ and so $F(C) \subseteq A$, since by construction, C is the biggest set X such that $X \subseteq A$ and $nodep(z, X)$, we have $F(C) \subseteq C$, then since A is the $\mathrm{lfp}^{\subseteq}_{\emptyset} F$, by Tarsky $A = \sqcap\{X \mid F(X) \subseteq X\}$, so we have $A \subseteq C$ and so $A = C$ and then $nodep(z, A)$ as expected.

- $s, h \in \llbracket \nu X_v.P \rrbracket_\rho$

  iff $\quad s, h \in \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda X.\ \llbracket P \rrbracket_{[\rho|X_v \to X]}$

  iff $\quad [s \mid z \mapsto v], h \in \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda X.\ \llbracket P \rrbracket_{[\rho|X_v \to X]}$ $\quad$ (proof below)

  iff $\quad [s \mid z \mapsto v], h \in \llbracket \nu X_v.P \rrbracket_\rho$

  Let $A \triangleq \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda X.\ \llbracket P \rrbracket_{[\rho|X_v \to X]}$, we want to prove $nodep(z, A)$, we proceed by contradiction. Let $C \triangleq \{s, h \mid \exists s', h \in A.[s' \mid z \mapsto 42] = [s \mid z \mapsto 42]\}$, by construction C is the smallest set such that $nodep(z, C)$ and $A \subseteq C$

  since $nodep(z, \rho)$ we then have $nodep(z, [\rho \mid X_v \to C])$, then by induction we have $nodep(z, \llbracket P \rrbracket_{[\rho|X_v \to C]})$.

  Let $F \triangleq \lambda X.\ \llbracket P \rrbracket_{[\rho|X_v \to X]}$, we have $nodep(z, F(C))$.

  ....

  If F is monotonic, then since $A \subseteq C$ we have $F(A) \subseteq F(C)$ and so $A \subseteq F(C)$, since by construction, C is the smallest set X such that $X \subseteq A$ and $nodep(z, X)$, we have

$C \subseteq F(C)$, then since A is the $\text{gfp}^{\subseteq}_{\emptyset} F$, by Tarsky $A = \sqcup \{X \mid X \subseteq F(X)\}$, so we have $C \subseteq A$ and so $A = C$ and then $nodep(z, A)$ as expected.

$\square$

2.6.3 Variable Renaming Theorem for $BI^{\mu\nu}$

Theorem 2.9. If
$$
\begin{array}{l}
\text{-} \quad P \text{ is } v\text{-closed} \\
\text{-} \quad z \notin Var(P) \quad \text{then } P \equiv P\{[z/y]\} \ . \\
\text{-} \quad y \notin FV(P)
\end{array}
$$

Lemma 2.22. $[\![E\{E'/x\}]\!]^s = [\![E]\!]^{[s|x \to [\![E']\!]^s]}$ if $[\![E']\!]^s$ exists

Proof. Lemma 2.22

- $[\![x\{E'/x\}]\!]^s = [\![E']\!]^s = [\![x]\!]^{[s|x \to [\![E']\!]^s]}$

- $[\![y\{E'/x\}]\!]^s = [\![y]\!]^s = [\![y]\!]^{[s|x \to [\![E']\!]^s]}$

- $[\![True\{E'/x\}]\!]^s = [\![True]\!]^s = true = [\![True]\!]^{[s|x \to [\![E']\!]^s]}$

- $[\![False\{E'/x\}]\!]^s = [\![False]\!]^s = false = [\![False]\!]^{[s|x \to [\![E']\!]^s]}$

- $[\![42\{E'/x\}]\!]^s = [\![42]\!]^s = 42 = [\![42]\!]^{[s|x \to [\![E']\!]^s]}$

- $[\![(E_1 \ op \ E_2)\{E'/x\}]\!]^s = [\![E_1\{E'/x\} \ op \ E_2\{E'/x\}]\!]^s = [\![E_1\{E'/x\}]\!]^s \ op \ [\![E_2\{E'/x\}]\!]^s = [\![E_1]\!]^{[s|x \to [\![E']\!]^s]} \ op \ [\![E_2]\!]^{[s|x \to [\![E']\!]^s]} = [\![E_1 \ op \ E_2]\!]^{[s|x \to [\![E']\!]^s]}$

$\square$

Lemma 2.23. $[\![E\{z/y\}]\!]^s = [\![E]\!]^{[s|y \to s(z)]}$ if $z \in dom(s)$

Proof. Lemma 2.23 By Lemma 2.22. $\square$

Lemma 2.24. $[\![E\{E'/x\}]\!]^s = [\![E]\!]^s$ if $[\![E']\!]^s$ doesn't exists but $[\![E\{E'/x\}]\!]^s$ does

Proof. Lemma 2.24 $[\![x\{E'/x\}]\!]^s = [\![E']\!]^s$

So $x \notin Var(E)$ and we directly have $E\{E'/x\} = E$ $\square$

Lemma 2.25. $[\![E\{z/y\}]\!]^s = [\![E]\!]^s$ *if* $z \notin \in dom(s)$

Proof. Lemma 2.25 By Lemma 2.24. $\qquad\qquad\qquad\qquad\qquad\qquad\square$

Let $s^\bullet \triangleq \begin{bmatrix} [s \mid y \to s(z)] & \text{if} & z \in dom(s) \\ s & \text{if} & z \notin dom(s) \end{bmatrix}$

Let $\rho^\bullet$ be $[\forall X_v \in dom(\rho). \; X_v \to \{s, h \mid s^\bullet, h \in \rho(X_v)\}]$

Lemma 2.26. $[\![E\{z/y\}]\!]^s = [\![E]\!]^{s^\bullet}$

Proof. Lemma 2.26 By Lemma 2.23 and 2.25. $\qquad\qquad\qquad\qquad\qquad\square$

Remember that in case P is $E = E'$, $E \mapsto E_1, E_2$, `false` and `emp` we have $\forall \rho. [\![P]\!]_\rho = [\![P]\!]$ since they are v-closed, see Lemma 2.27.

Remember $P \equiv Q$ iff $\forall \rho$. either ($[\![P]\!]_\rho$ and $[\![Q]\!]_\rho$ do not exist) either $[\![P]\!]_\rho = [\![Q]\!]_\rho$.

Th. 2.9. By Th. 2.8, $[\![P\{[z/y]\}]\!]_{\rho^\bullet} = \{s, h \mid s^\bullet, h \in [\![P]\!]_\rho\}$ in case $nodep(z, \rho)$ and $z \notin Var(P)$, then $[\![P\{[z/y]\}]\!] = \{s, h \mid s^\bullet, h \in [\![P]\!]\}$ if $z \notin Var(P)$.

Which is if $z \notin dom(s)$ then $s, h \in [\![P\{[z/y]\}]\!]$ iff $s, h \in [\![P]\!]$ and if $z \in dom(s)$ we have $s, h \in [\![P\{[z/y]\}]\!]$ iff $[s \mid y \to s(z)], h \in [\![P]\!]$.

Then since $y \notin FV(P)$ with the stack extension theorem 2.21 we have $nodep(y, P)$ and so $s, h \in$ so $[s \mid y \to s(z)], h \in [\![P]\!]$ iff $s, h \in [\![P]\!]$.

We then have $[\![P\{[z/y]\}]\!] = [\![P]\!]$ which is what we wanted since P is v-closed (see Lemma 2.27). $\qquad\qquad\qquad\qquad\qquad\square$

Theorem 2.8. $\boxed{\begin{array}{l} If \quad \begin{array}{l} - \quad nodep(z, \rho) \\ - \quad z \notin Var(P) \end{array} \quad then \; [\![P\{[z/y]\}]\!]_{\rho^\bullet} = \{s, h \mid s^\bullet, h \in [\![P]\!]_\rho\} \end{array}}$

Th. 2.8. THE PROOF IS ONLY MADE IF ALL THE lfp and gpf are for MONOTONIC FUNCTIONS. But this is the case for the wlp and sp formulae and for the example in the paper.

We will prove by structural induction on P:

(*recall that for* $E1 = E2, E \mapsto E_1, E_2,$ `false`, `emp` $\forall \rho. [\![P]\!]_\rho = [\![P]\!]$)

- $\llbracket (E_1 = E_2)\{[z/y]\} \rrbracket$
  $= \llbracket (E_1\{z/y\} = E_2\{z/y\}) \rrbracket$
  $= \{s,h \mid \llbracket E_1\{z/y\} \rrbracket^s = \llbracket E_2\{z/y\} \rrbracket^s\}$
  $= \{s,h \mid \llbracket E_1 \rrbracket^{s^\bullet} = \llbracket E_2 \rrbracket^{s^\bullet}\}$
  $= \{s,h \mid s^\bullet, h \in \llbracket E_1 = E_2 \rrbracket\}$

- $\llbracket (E \mapsto E_1, E_2)\{[z/y]\} \rrbracket$
  $= \llbracket E\{z/y\} \mapsto E_1\{z/y\}, E_2\{z/y\} \rrbracket$
  $= \{s,h \mid dom(h) = \{\llbracket E\{z/y\} \rrbracket^s\}$ and $h(\llbracket E\{z/y\} \rrbracket^s) = \langle \llbracket E_1\{z/y\} \rrbracket^s, \llbracket E_2\{z/y\} \rrbracket^s \rangle$
  $= \{s,h \mid dom(h) = \{\llbracket E \rrbracket^{s^\bullet}\}$ and $h(\llbracket E \rrbracket^{s^\bullet}) = \langle \llbracket E_1 \rrbracket^{s^\bullet}, \llbracket E_2 \rrbracket^{s^\bullet} \rangle\}$
  $= \{s,h \mid s^\bullet, h \in \llbracket E \mapsto E_1, E_2 \rrbracket\}$

- $\llbracket \mathtt{false}\{[z/y]\} \rrbracket$
  $= \llbracket \mathtt{false} \rrbracket$
  $= \emptyset$
  $= \{s,h \mid s^\bullet, h \in \llbracket \mathtt{false} \rrbracket$

- If $\llbracket P \Rightarrow Q \rrbracket_\rho$ exists then

  $\llbracket (P \Rightarrow Q)\{[z/y]\} \rrbracket_{\rho^\bullet}$
  $= \llbracket P\{[z/y]\} \Rightarrow Q\{[z/y]\} \rrbracket_{\rho^\bullet}$
  $= (\top \setminus \llbracket P\{[z/y]\} \rrbracket_{\rho^\bullet}) \cup \llbracket Q\{[z/y]\} \rrbracket_{\rho^\bullet}$
  $= (\top \setminus \{s,h \mid s^\bullet, h \in \llbracket P \rrbracket_\rho\}) \cup \{s,h \mid s^\bullet \in \llbracket Q \rrbracket_\rho\}$ *(ind.hyp.)*
  $= \{s,h \mid s^\bullet, h \in (\top \setminus \llbracket P \rrbracket_\rho) \cup \llbracket Q \rrbracket_\rho\}$
  $= \{s,h \mid s^\bullet, h \in \llbracket P \Rightarrow Q \rrbracket_\rho\}$

- $\llbracket (\exists x.\ P)\{[z/y]\} \rrbracket_{\rho^\bullet}$ when $x \neq y$
  $= \llbracket \exists x.\ (P\{[z/y]\}) \rrbracket_{\rho^\bullet}$
  $= \{s,h \mid \exists v.[s \mid x \rightarrow v], h \in \llbracket P\{[z/y]\} \rrbracket_{\rho^\bullet}\}$
  $= \{s,h \mid \exists v.[s \mid x \rightarrow v], h \in \{s,h \mid s^\bullet, h \in \llbracket P \rrbracket_\rho\}\}$ (ind.)
  $= \{s,h \mid \exists v.[s \mid x \rightarrow v]^\bullet, h \in \llbracket P \rrbracket_\rho\}$
  $= \{s,h \mid \exists v.[s^\bullet \mid x \rightarrow v], h \in \llbracket P \rrbracket_\rho\}$ (since $x \neq y, z$)
  $= \{s,h \mid s^\bullet, h \in \llbracket \exists x.P \rrbracket_\rho\}$

- $\llbracket (\exists y.\ P)\{[z/y]\} \rrbracket_{\rho^\bullet}$

$= \quad [\![\exists z. \, (P\{[z/y]\})]\!]_{\rho\bullet}$

$= \quad \{s, h \mid \exists v.[s \mid z \to v], h \in [\![P\{[z/y]\}]\!]_{\rho\bullet}\}$

$= \quad \{s, h \mid \exists v.[s \mid z \to v], h \in \{s, h \mid s^\bullet, h \in [\![P]\!]_{\rho}\}\}$ (ind.)

$= \quad \{s, h \mid \exists v.[s \mid z \to v]^\bullet, h \in [\![P]\!]_{\rho}\}$

$= \quad \{s, h \mid \exists v.[s \mid z \to v \mid y \to v], h \in [\![P]\!]_{\rho}\}$

$= \quad \{s, h \mid \exists v.[s \mid y \to v \mid z \to v], h \in [\![P]\!]_{\rho}\}$ (since $z \neq y$)

$= \quad \{s, h \mid \exists v.[s \mid y \to v], h \in [\![P]\!]_{\rho}\}$ (from stack extension theorem)

$= \quad \{s, h \mid \exists v.[s^\bullet \mid y \to v], h \in [\![P]\!]_{\rho}\}$

$= \quad \{s, h \mid s^\bullet, h \in [\![\exists y.P]\!]_{\rho}\}$

- $[\![\text{emp}\{[z/y]\}]\!]$

  $= \quad [\![\text{emp}]\!]$

  $= \quad \{s, h \mid h = []\}$

  $= \quad \{s, h \mid s^\bullet, h \in [\![emp]\!]\}$

- $[\![(P * Q)\{[z/y]\}]\!]_{\rho\bullet}$

  $= \quad [\![(P\{[z/y]\} * Q\{[z/y]\})]\!]_{\rho\bullet}$

  $= \quad \{s, h \mid \exists h_0, h_1. \, h_0 \sharp h_1, h_0.h_1 = h, s, h_0 \in [\![P\{[z/y]\}]\!]_{\rho\bullet} \text{ and } s, h_1 \in [\![Q\{[z/y]\}]\!]_{\rho\bullet}\}$

  $= \quad \{s, h \mid \exists h_0, h_1. \, h_0 \sharp h_1, h_0.h_1 = h, s^\bullet, h_0 \in [\![P]\!]_{\rho} \text{ and } s^\bullet, h_1 \in [\![Q]\!]_{\rho}\}$ (ind. hyp.)

  $= \quad \{s, h \mid s^\bullet, h \in [\![P * Q]\!]_{\rho}\}$

- $[\![(P \mathbin{-\!\!*} Q)\{[z/y]\}]\!]_{\rho\bullet}$

  $= \quad [\![P\{[z/y]\} \mathbin{-\!\!*} Q\{[z/y]\}]\!]_{\rho\bullet}$

  $= \quad \{s, h \mid \forall h'. \text{ if } h' \sharp h \text{ and } s, h' \in [\![P\{[z/y]\}]\!]_{\rho\bullet} \text{ then } s, h.h' \in [\![Q\{[z/y]\}]\!]_{\rho\bullet}\}$

  $= \quad \{s, h \mid \forall h'. \text{ if } h' \sharp h \text{ and } s^\bullet, h' \in [\![P]\!]_{\rho} \text{ then } s^\bullet, h.h' \in [\![Q]\!]_{\rho}\}$ (ind. hyp.)

  $= \quad \{s, h \mid s^\bullet, h \in [\![P \mathbin{-\!\!*} Q]\!]_{\rho}\}$

- $[\![X_v\{[z/y]\}]\!]_{\rho\bullet}$

  $= \quad [\![X_v]\!]_{\rho\bullet}$

  $= \quad \rho^\bullet(X_v)$

  $= \quad \{s, h \mid s^\bullet, h \in \rho(X_v)\}$

  $= \quad \{s, h \mid s^\bullet, h \in [\![X_v]\!]_{\rho}\}$

- $[\![P[E/x]\{[z/y]\}]\!]_{\rho\bullet} \text{ if } x \neq y$

$$\begin{aligned}
&= & [\![(P\{[z/y]\})[E\{z/y\}/x]]\!]_{\rho^\bullet}\\
&= & \{s,h \mid [s \mid x \to [\![E\{[z/y]\}]\!]^s], h \in [\![P\{[z/y]\}]\!]_{\rho^\bullet}\}\\
&= & \{s,h \mid [s \mid x \to [\![E]\!]^{s^\bullet}], h \in [\![P\{[z/y]\}]\!]_{\rho^\bullet}\}\\
&= & \{s,h \mid [s \mid x \to [\![E]\!]^{s^\bullet}], h \in \{s,h \mid s^\bullet, h \in [\![P]\!]_\rho\}\} \quad \text{(ind.)}\\
&= & \{s,h \mid ([s \mid x \to [\![E]\!]^{s^\bullet}])^\bullet, h \in [\![P]\!]_\rho\}\\
&= & \{s,h \mid [s^\bullet \mid x \to [\![E]\!]^{s^\bullet}], h \in [\![P]\!]_\rho\} \quad\quad\quad \text{(since } x \neq y,z)\\
&= & \{s,h \mid s^\bullet, h \in [\![P[E/x]]\!]_\rho\}
\end{aligned}$$

- $[\![P[E/y]\{[z/y]\}]\!]_{\rho^\bullet}$

$$\begin{aligned}
&= & [\![(P\{[z/y]\})[E\{z/y\}/z]]\!]_{\rho^\bullet}\\
&= & \{s,h \mid [s \mid z \to [\![E\{[z/y]\}]\!]^s], h \in [\![P\{[z/y]\}]\!]_{\rho^\bullet}\}\\
&= & \{s,h \mid [s \mid z \to [\![E]\!]^{s^\bullet}], h \in [\![P\{[z/y]\}]\!]_{\rho^\bullet}\}\\
&= & \{s,h \mid [s \mid z \to [\![E]\!]^{s^\bullet}], h \in \{s,h \mid s^\bullet, h \in [\![P]\!]_\rho\}\} \quad \text{(ind.)}\\
&= & \{s,h \mid ([s \mid z \to [\![E]\!]^{s^\bullet}])^\bullet, h \in [\![P]\!]_\rho\}\\
&= & \{s,h \mid [s \mid z \to [\![E]\!]^{s^\bullet} \mid y \to [\![E]\!]^{s^\bullet}], h \in [\![P]\!]_\rho\}\\
&= & \{s,h \mid [s \mid y \to [\![E]\!]^{s^\bullet}], h \in [\![P]\!]_\rho\} \quad\quad\quad \textcolor{red}{\text{(from stack extension th.)}}\\
&= & \{s,h \mid [s^\bullet \mid y \to [\![E]\!]^{s^\bullet}], h \in [\![P]\!]_\rho\}\\
&= & \{s,h \mid s^\bullet, h \in [\![P[E/y]]\!]_\rho\}
\end{aligned}$$

- If $[\![\mu X.P]\!]_\rho$ exists:

$[\![(\mu X.P)\{[z/y]\}]\!]_{\rho^\bullet}$

$$\begin{aligned}
&= & [\![\mu X.P\{[z/y]\}]\!]_{\rho^\bullet}\\
&= & \mathrm{lfp}_\emptyset^\subseteq \lambda X.[\![P\{[z/y]\}]\!]_{[\rho^\bullet \mid X_v \to X]}\\
& & \text{see proof below}\\
&= & \{s,h \mid s^\bullet, h \in \mathrm{lfp}_\emptyset^\subseteq \lambda X.[\![P]\!]_{[\rho \mid X_v \to X]}\}\\
&= & \{s,h \mid s^\bullet, h \in [\![\mu X.P]\!]_\rho\}
\end{aligned}$$

Let
$$\begin{aligned}
F &\triangleq & \lambda X.[\![P\{[z/y]\}]\!]_{[\rho^\bullet \mid X_v \to X]}\\
A &\triangleq & \mathrm{lfp}_\emptyset^\subseteq F\\
G &\triangleq & \lambda X.[\![P]\!]_{[\rho \mid X_v \to X]}\\
B &\triangleq & \mathrm{lfp}_\emptyset^\subseteq G\\
C &\triangleq & \{s,h \mid s^\bullet, h \in B\}
\end{aligned}$$
We want then $A = C$.

We know that B exists.

Notice that $([\rho \mid X_v \to B])^\bullet = [\rho^\bullet \mid X_v \to C]$.

First we prove that $A \subseteq C$, we prove it by proving that $C = F(C)$.

Since $B = [\![\mu X.P]\!]_{[\rho|X_v \to X]}$, by the stack extension theorem, we have $nodep(z, B)$ so $nodep(z, [\rho \mid X_v \to B])$ and we can use the induction.

By definition, $B = G(B)$, so $B = [\![P]\!]_{[\rho|X_v \to B]}$

and then $C = \{s, h \mid s^\bullet, h \in [\![P]\!]_{[\rho|X_v \to B]}\}$

by induction we then have $C = [\![P\{z/y\}]\!]_{([\rho|X_v \to B])^\bullet}$, which is $C = [\![P\{z/y\}]\!]_{[\rho^\bullet|X_v \to C]}$,

we then have $C = F(C)$ and so A exists and $A \subseteq C$.

Now we want to prove that $C \subseteq A$.

Let D be the biggest set such that $nodep(z, D)$ and $\{s, h \mid s^\bullet, h \in D\} \subseteq A$. By the stack extension theorem, $nodep(z, [\![P]\!]_{[\rho|X_v \to D]})$ and we can use the induction.

Since we said we are working with monotonic functions, we have $F(\{s, h \mid s^\bullet, h \in D\}) \subseteq F(A)$

since A is a fix point we have then $F(\{s, h \mid s^\bullet, h \in D\}) \subseteq A$

which is $[\![P\{[z/y]\}]\!]_{[\rho^\bullet|X_v \to \{s,h|s^\bullet,h\in D\}]} \subseteq A$

by induction we have then $\{s, h \mid s^\bullet, h \in [\![P]\!]_{[\rho|X_v \to D]}\} \subseteq A$

which is $\{s, h \mid s^\bullet, h \in G(D)\} \subseteq A$

Then by construction of D as the biggest set we have $G(D) \subseteq D$

and then D is a postfixpoint of G and then $B \subseteq D$ and then $\{s, h \mid s^\bullet, h \in B\} \subseteq \{s, h \mid s^\bullet, h \in D\}$ which is $C \subseteq A$.

- If $[\![\nu X.P]\!]_\rho$ exists:

$[\![(\nu X.P)\{[z/y]\}]\!]_{\rho^\bullet}$
$= [\![\nu X.P\{[z/y]\}]\!]_{\rho^\bullet}$
$= \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda X.[\![P\{[z/y]\}]\!]_{[\rho^\bullet|X_v \to X]}$
  see proof below
$= \{s, h \mid s^\bullet, h \in \mathrm{gfp}^{\subseteq}_{\emptyset} \lambda X.[\![P]\!]_{[\rho|X_v \to X]}\}$
$= \{s, h \mid s^\bullet, h \in [\![\nu X.P]\!]_\rho\}$
Let

53

$$
\begin{aligned}
F &\triangleq \lambda X.[\![P\{[z/y]\}]\!]_{[\rho^\bullet | X_v \to X]} \\
A &\triangleq \mathrm{gfp}_\emptyset^\subseteq F \\
G &\triangleq \lambda X.[\![P]\!]_{[\rho | X_v \to X]} \\
B &\triangleq \mathrm{gfp}_\emptyset^\subseteq G \\
C &\triangleq \{s, h \mid s^\bullet, h \in B\}
\end{aligned}
$$

We want then $A = C$.

We know that B exists.

Notice that $([\rho \mid X_v B])^\bullet = [\rho^\bullet \mid X_v \to C]$.

First we prove that $A \supseteq C$, we prove it by proving that $C = F(C)$.

Since $B = [\![\nu X.P]\!]_{[\rho | X_v \to X]}$, by the stack extension theorem, we have $nodep(z, B)$ so $nodep(z, [\rho \mid X_v \to B])$ and we can use the induction.

By definition, $B = G(B)$, so $B = [\![P]\!]_{[\rho | X_v \to B]}$

and then $C = \{s, h \mid s^\bullet, h \in [\![P]\!]_{[\rho | X_v \to B]}\}$

by induction we then have $C = [\![P\{z/y\}]\!]_{([\rho | X_v \to B])^\bullet}$, which is $C = [\![P\{z/y\}]\!]_{[\rho^\bullet | X_v \to C]}$, we then have $C = F(C)$ and so A exists and $A \supseteq C$.

Now we want to prove that $C \supseteq A$.

Let D be the smallest set such that $nodep(z, D)$ and $\{s, h \mid s^\bullet, h \in D\} \supseteq A$. By the stack extension theorem, $nodep(z, [\![P]\!]_{[\rho | X_v \to D]})$ and we can use the induction.

Since we said we are working with monotonic functions, we have $F(\{s, h \mid s^\bullet, h \in D\}) \supseteq F(A)$

since A is a fix point we have then $F(\{s, h \mid s^\bullet, h \in D\}) \supseteq A$

which is $[\![P\{[z/y]\}]\!]_{[\rho^\bullet | X_v \to \{s,h|s^\bullet, h \in D\}]} \supseteq A$

by induction we have then $\{s, h \mid s^\bullet, h \in [\![P]\!]_{[\rho | X_v \to D]}\} \supseteq A$

which is $\{s, h \mid s^\bullet, h \in G(D)\} \supseteq A$

Then by construction of D as the smallest set we have $G(D) \supseteq D$

and then D is a prefixpoint of G and then $B \supseteq D$ and then $\{s, h \mid s^\bullet, h \in B\} \supseteq \{s, h \mid$

$s^\bullet, h \in D\}$ which is $C \supseteq A$.

$\square$

Lemma 2.27. *If P is v-closed formula:* $\forall \rho. [\![P]\!]_\rho = [\![P]\!]$.

Lemma 2.27. The simple case are direct from the definition of $[\![\cdot]\!]_\rho$, the others come by induction.
$\square$

2.6.4 Unfolding theorems

Theorem 2.28. $\boxed{If \quad \begin{array}{l} \text{-} \quad \mu X_v.\ P \text{ is v-closed} \\ \text{-} \quad [\![\mu X_v.\ P]\!] \text{ exists} \end{array} \quad then \ \mu X_v.\ P \equiv P\{(\mu X_v.\ P)/X_v\}\ .}$

Th. 2.28.

$$
\begin{aligned}
&[\![\mu X_v.P]\!]_\rho \\
={}& \mathrm{lfp}_\emptyset^\subseteq \lambda X.\ [\![P]\!]_{[\rho|X_v \to X]} \quad \text{def} \\
={}& [\![P]\!]_{[\rho|X_v \to ([\![\mu X_v.P]\!]_\rho)]} \quad \text{(since it's a fix point)} \\
={}& [\![P(\mu X_v.P)/X_v]\!]_\rho \quad \text{(from the substitution theorem for } BI^{\mu\nu} \text{ general Th. 2.32)}
\end{aligned}
$$

$\square$

Theorem 2.29. $\boxed{If \quad \begin{array}{l} \text{-} \quad \nu X_v.\ P \text{ is v-closed} \\ \text{-} \quad [\![\nu X_v.\ P]\!] \text{ exists} \end{array} \quad then \ \nu X_v.\ P \equiv P\{(\nu X_v.\ P)/X_v\}\ .}$

Th. 2.29.

$$
\begin{aligned}
&[\![\nu X_v.P]\!]_\rho \\
={}& \mathrm{gfp}_\emptyset^\subseteq \lambda X.\ [\![P]\!]_{[\rho|X_v \to X]} \quad \text{def} \\
={}& [\![P]\!]_{[\rho|X_v \to ([\![\nu X_v.P]\!]_\rho)]} \quad \text{(since it's a fix point)} \\
={}& [\![P(\nu X_v.P)/X_v]\!]_\rho \quad \text{(from the substitution theorem for } BI^{\mu\nu} \text{ general Th. 2.32)}
\end{aligned}
$$

$\square$

2.6.5 Substitution theorems for $BI^{\mu\nu}$

Theorem 2.30. $\boxed{\begin{array}{l} If\ [\![Y]\!]\ and\ [\![P]\!]_{\rho[X_v \to [\![Y]\!]]}\ exist \\[2mm] \qquad\qquad [\![P]\!]_{\rho[X_v \to [\![Y]\!]]} = [\![P\{Y/X_v\}]\!]_\rho \end{array}}$

Theorem 2.30. By induction on the formula P.

- Case P as the form $E = E'$, $E \mapsto E_1, e_2$, `false`, `emp`

  $P\{Y/X_v\} = P$ and $\forall \rho. \llbracket P \rrbracket_\rho = \llbracket P \rrbracket$ so

  $$
  \begin{aligned}
  & \llbracket P \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]} \\
  =\ & \llbracket P \rrbracket \\
  =\ & \llbracket P\{Y/X_v\} \rrbracket \\
  =\ & \llbracket P\{Y/X_v\} \rrbracket_\rho
  \end{aligned}
  $$

- $\llbracket P \Rightarrow Q \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]}$

  $$
  \begin{aligned}
  =\ & (\top \setminus \llbracket P \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]}) \cup \llbracket Q \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]} \\
  =\ & (\top \setminus \llbracket P\{Y/X_v\} \rrbracket_\rho) \cup \llbracket Q\{Y/X_v\} \rrbracket_\rho \quad \text{(ind. hyp.)} \\
  =\ & \llbracket P\{Y/X_v\} \Rightarrow Q\{Y/X_v\} \rrbracket_\rho \\
  =\ & \llbracket (P \Rightarrow Q)\{Y/X_v\} \rrbracket_\rho
  \end{aligned}
  $$

- $\llbracket \exists x. P \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]}$

  $$
  \begin{aligned}
  =\ & \{s, h \mid \exists v \in Val.[s \mid x \mapsto v], h \in \llbracket P \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]}\} \\
  =\ & \{s, h \mid \exists v \in Val.[s \mid x \mapsto v], h \in \llbracket P\{Y/X_v\} \rrbracket_\rho\} \quad \text{(ind.hyp)} \\
  =\ & \llbracket \exists x.(P\{Y/X_v\}) \rrbracket_\rho \\
  =\ & \llbracket (\exists x.P)\{Y/X_v\} \rrbracket_\rho
  \end{aligned}
  $$

- $\llbracket P * Q \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]}$

  $$
  \begin{aligned}
  =\ & \{s, h \mid \exists h_0, h_1\ h_0 \sharp h_1,\ h = h_0.h_1 \\
  & s, h_0 \in \llbracket P \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]} \text{ and } s, h_1 \in \llbracket Q \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]}\} \\
  =\ & \{s, h \mid \exists h_0, h_1\ h_0 \sharp h_1,\ h = h_0.h_1 \\
  & s, h_0 \in \llbracket P\{Y/X_v\} \rrbracket_\rho \text{ and } s, h_1 \in \llbracket Q\{Y/X_v\} \rrbracket_\rho\} \quad \text{(ind. hyp.)} \\
  =\ & \llbracket P\{Y/X_v\} * Q\{Y/X_v\} \rrbracket_\rho \\
  =\ & \llbracket (P * Q)\{Y/X_v\} \rrbracket_\rho
  \end{aligned}
  $$

- $\llbracket P \mathbin{-\!\!*} Q \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]}$

  $$
  \begin{aligned}
  =\ & \{s, h \mid \forall h'. \text{ if } h' \sharp h \text{ and } s, h' \in \llbracket P \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]} \text{ then} \\
  & s, h.h' \in \llbracket Q \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]}\} \\
  =\ & \{s, h \mid \forall h'. \text{ if } h' \sharp h \text{ and } s, h' \in \llbracket P\{Y/X_v\} \rrbracket_\rho \text{ then} \\
  & s, h.h' \in \llbracket Q\{Y/X_v\} \rrbracket_\rho\} \quad\quad\quad\quad \text{(ind. hyp.)} \\
  =\ & \llbracket P\{Y/X_v\} \mathbin{-\!\!*} Q\{Y/X_v\} \rrbracket_\rho \\
  =\ & \llbracket (P \mathbin{-\!\!*} Q)\{Y/X_v\} \rrbracket_\rho
  \end{aligned}
  $$

- $\llbracket X_v \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket]}$

$$= [\![Y]\!]$$
$$= [\![Y]\!]_\rho$$
$$= [\![X_v\{Y/X_v\}]\!]_\rho$$

- $[\![Z_v]\!]_{[\rho|X_v\mapsto[\![Y]\!]]}$ with $Z_v \neq X_v$

$$= [\![Z_v]\!]_\rho$$
$$= [\![Z_v\{Y/X_v\}]\!]_\rho$$

- $[\![\mu X_v.P]\!]_{[\rho|X_v\mapsto[\![Y]\!]]}$

$$= \mathrm{lfp}_\emptyset^{\subseteq} \lambda Z.[\![P]\!]_{[[\rho|X_v\mapsto[\![Y]\!]]|X_v\mapsto Z]}$$
$$= \mathrm{lfp}_\emptyset^{\subseteq} \lambda Z.[\![P]\!]_{[\rho|X_v\mapsto Z]}$$
$$= \mathrm{lfp}_\emptyset^{\subseteq} \lambda X.[\![P]\!]_{[\rho|X_v\mapsto X]}$$
$$= [\![\mu X_v.P]\!]_\rho$$
$$= [\![(\mu X_v.P)\{Y/X_v\}]\!]_\rho$$

- $[\![\mu Z_v.P]\!]_{[\rho|X_v\mapsto[\![Y]\!]]}$ with $X \neq Z$

$$= \mathrm{lfp}_\emptyset^{\subseteq} \lambda Z.[\![P]\!]_{[[\rho|X_v\mapsto[\![Y]\!]]|Z_v\mapsto Z]}$$
$$= \mathrm{lfp}_\emptyset^{\subseteq} \lambda Z.[\![P]\!]_{[[\rho|Z_v\mapsto Z]|X_v\mapsto[\![Y]\!]]}$$
$$= \mathrm{lfp}_\emptyset^{\subseteq} \lambda Z.[\![P\{Y/X_v\}]\!]_{[\rho|Z_v\mapsto Z]} \quad \text{(ind. hyp.)}$$
$$= [\![\mu Z_v.(P\{Y/X_v\})]\!]_\rho$$
$$= [\![(\mu Z_v.P)\{Y/X_v\}]\!]_\rho$$

- $[\![\nu X_v.P]\!]_{[\rho|X_v\mapsto[\![Y]\!]]}$

$$= \mathrm{gfp}_\emptyset^{\subseteq} \lambda Z.[\![P]\!]_{[[\rho|X_v\mapsto[\![Y]\!]]|X_v\mapsto Z]}$$
$$= \mathrm{gfp}_\emptyset^{\subseteq} \lambda Z.[\![P]\!]_{[\rho|X_v\mapsto Z]}$$
$$= \mathrm{gfp}_\emptyset^{\subseteq} \lambda X.[\![P]\!]_{[\rho|X_v\mapsto X]}$$
$$= [\![\nu X_v.P]\!]_\rho$$
$$= [\![(\nu X_v.P)\{Y/X_v\}]\!]_\rho$$

- $[\![\nu Z.P]\!]_{[\rho|X_v\mapsto[\![Y]\!]]}$ with $X \neq Z$

$$= \mathrm{gfp}_\emptyset^{\subseteq} \lambda Z.[\![P]\!]_{[[\rho|X_v\mapsto[\![Y]\!]]|Z_v\mapsto Z]}$$
$$= \mathrm{gfp}_\emptyset^{\subseteq} \lambda Z.[\![P]\!]_{[[\rho|Z_v\mapsto Z]|X_v\mapsto[\![Y]\!]]}$$
$$= \mathrm{gfp}_\emptyset^{\subseteq} \lambda Z.[\![P\{Y/X_v\}]\!]_{[\rho|Z_v\mapsto Z]} \quad \text{(ind. hyp.)}$$
$$= [\![\nu Z.(P\{Y/X_v\})]\!]_\rho$$
$$= [\![(\nu Z.P)\{Y/X_v\}]\!]_\rho$$

- $[\![P[E'/x]]\!]_{[\rho|X_v\mapsto[\![Y]\!]]}$

$$= \{s, h \mid [s \mid x \mapsto \llbracket E' \rrbracket s], h \in \llbracket P \rrbracket_{[\rho \mid X_v \mapsto \llbracket Y \rrbracket]}\}$$
$$= \{s, h \mid [s \mid x \mapsto \llbracket E' \rrbracket s], h \in \llbracket P\{Y/X_v\} \rrbracket_\rho\} \quad \text{(ind. hyp.)}$$
$$= \llbracket P\{Y/X_v\}[E'/x] \rrbracket_\rho$$
$$= \llbracket (P[E'/x])\{Y/X_v\} \rrbracket_\rho$$

$\square$

Lemma 2.31. *If $\llbracket Y \rrbracket$ exists*

$$sp(P, C)\{Y/X_v\} = sp(P\{Y/X_v\}, C)$$

Lemma 2.31. Proof by induction on C.

- Case $x := E$

$$(sp(P, C)\{Y/X_v\})$$
$$= (\exists x'.P[x'/x] \wedge x = E[x'/x])\{Y/X_v\}$$
$$= (\exists x'.(P\{Y/X_v\})[x'/x] \wedge x = E[x'/x])$$
$$= sp(P\{Y/X_v\}, C)$$

- Case $x := E.i$

$$(sp(P, C)\{Y/X_v\})$$
$$= (\exists x'.P[x'/x] \wedge x = (E[x'/x].i))\{Y/X_v\}$$
$$= (\exists x'.(P\{Y/X_v\})[x'/x] \wedge x = (E[x'/x].i))$$
$$= sp(P\{Y/X_v\}, C)$$

- Case $E.1 := E'$

$$(sp(P, C)\{Y/X_v\})$$
$$= (\exists x_1, x_2.(E \mapsto E', x_2) * ((E \mapsto x_1, x_2) \mathbin{-\!\!*} P)\{Y/X_v\}$$
$$= (\exists x_1, x_2.(E \mapsto E', x_2) * ((E \mapsto x_1, x_2) \mathbin{-\!\!*} (P\{Y/X_v\}))$$
$$= sp(P\{Y/X_v\}, C)$$

- Case $E.2 := E'$

$$(sp(P, C)\{Y/X_v\})$$
$$= (\exists x_1, x_2.(E \mapsto E', x_2) * ((E \mapsto x_1, x_2) \mathbin{-\!\!*} P)\{Y/X_v\}$$
$$= (\exists x_1, x_2.(E \mapsto x_1, E') * ((E \mapsto x_1, x_2) \mathbin{-\!\!*} (P\{Y/X_v\}))$$
$$= sp(P\{Y/X_v\}, C)$$

- Case $x := \mathtt{cons}(E_1, E_2)$

$$(sp(P, C)\{Y/X_v\})$$
$$= \exists x'.(P[x'/x] * (x \mapsto E_1[x'/x], E_2[x'/x]))\{Y/X_v\}$$
$$= \exists x'.((P\{Y/X_v\})[x'/x] * (x \mapsto E_1[x'/x], E_2[x'/x]))$$
$$= sp(P\{Y/X_v\}, C)$$

- Case $dispose(E)$

$$(sp(P, C)\{Y/X_v\})$$
$$= \exists x_1, x_2.\ ((E \mapsto x_1, x_2) \mathbin{-\!\!*} P)\{Y/X_v\}$$
$$= \exists x_1, x_2.\ ((E \mapsto x_1, x_2) \mathbin{-\!\!*} (P\{Y/X_v\}))$$
$$= sp(P\{Y/X_v\}, C)$$

- Case $C_1; C_2$

$$(sp(P, C)\{Y/X_v\})$$
$$= sp(sp(P, C_1), C_2)\{Y/X_v\}$$
$$= sp(sp(P, C_1)\{Y/X_v\}, C_2) \quad \text{(ind.hyp)}$$
$$= sp(sp(P\{Y/X_v\}, C_1), C_2) \quad \text{(ind.hyp)}$$
$$= sp(P\{Y/X_v\}, C)$$

- Case $if\ E\ then\ C_1\ else\ C_2$

$$(sp(P, C)\{Y/X_v\})$$
$$= (sp(P \wedge E = \texttt{true}, C_1) \vee sp(P \wedge E = \texttt{false}, C_2))\{Y/X_v\}$$
$$= sp(P \wedge E = \texttt{true}, C_1)\{Y/X_v\} \vee sp(P \wedge E = \texttt{false}, C_2)\{Y/X_v\}$$
$$= sp(P\{Y/X_v\} \wedge E = \texttt{true}, C_1) \vee sp(P\{Y/X_v\} \wedge E = \texttt{false}, C_2) \quad \text{(ind. hyp.)}$$
$$= sp(P\{Y/X_v\}, C)$$

- Case $\texttt{skip}$

$$(sp(P, C)\{Y/X_v\})$$
$$= P\{Y/X_v\}$$
$$= sp(P\{Y/X_v\}, C)$$

- Case $while\ E\ do\ C_1$

Case X_v not free in P

$$(sp(P, C)\{Y/X_v\})$$
$$= ((\mu X_v.sp(X_v \wedge E = \texttt{true}, C_1) \vee P) \wedge (E = \texttt{false}))\{Y/X_v\}$$
$$= ((\mu X_v.sp(X_v \wedge E = \texttt{true}, C_1) \vee P) \wedge (E = \texttt{false}))$$
$$= ((\mu X_v.sp(X_v \wedge E = \texttt{true}, C_1) \vee P\{Y/X_v\}) \wedge (E = \texttt{false}))$$
$$= sp(P\{Y/X_v\}, C)$$

Case X_v free in P

$$(sp(P,C)\{Y/X_v\})$$
$$= ((\mu Z_v.sp(Z_v \wedge E = \mathtt{true}, C_1) \vee P) \wedge (E = \mathtt{false}))\{Y/X_v\}$$
$$= ((\mu Z_v.sp(Z_v \wedge E = \mathtt{true}, C_1) \vee P\{Y/X_v\}) \wedge (E = \mathtt{false}))$$
$$= ((\mu W_v.sp(W_v \wedge E = \mathtt{true}, C_1) \vee P\{Y/X_v\}) \wedge (E = \mathtt{false}))$$
$$= sp(P\{Y/X_v\}, C)$$

We have the egality between the case Z_v and W_v since $[\![Y]\!]$ exists so there is no free formula variables in Y, so neither Z_v or W_v can become bounded in some Y placed in P.

$\square$

2.6.6 Substitution theorems for $BI^{\mu\nu}$ general

Theorem 2.32.

> If $[\![Y]\!]_\rho$ and $([\![P]\!]_{[\rho|X_v \to [\![Y]\!]_\rho]}$ or $[\![P\{Y/X_v\}]\!]_\rho$) exist then
>
> $$[\![P]\!]_{[\rho|X_v \to [\![Y]\!]_\rho]} = [\![P\{Y/X_v\}]\!]_\rho$$

Theorem 2.32. By induction on the formula P.

- Case P as the form $E = E'$, $E \mapsto E_1, e_2$, $\mathtt{false}$, $\mathtt{emp}$

  $P\{Y/X_v\} = P$ and $\forall \rho.[\![P]\!]_\rho = [\![P]\!]$ so
  $$[\![P]\!]_{[\rho|X_v \mapsto [\![Y]\!]_\rho]}$$
  $$= [\![P]\!]$$
  $$= [\![P\{Y/X_v\}]\!]$$
  $$= [\![P\{Y/X_v\}]\!]_\rho$$

- $[\![P \Rightarrow Q]\!]_{[\rho|X_v \mapsto [\![Y]\!]_\rho]}$
  $$= (\top \setminus [\![P]\!]_{[\rho|X_v \mapsto [\![Y]\!]_\rho]}) \cup [\![Q]\!]_{[\rho|X_v \mapsto [\![Y]\!]_\rho]}$$
  $$= (\top \setminus [\![P\{Y/X_v\}]\!]_\rho) \cup [\![Q\{Y/X_v\}]\!]_\rho \quad \text{(ind. hyp.)}$$
  $$= [\![P\{Y/X_v\} \Rightarrow Q\{Y/X_v\}]\!]_\rho$$
  $$= [\![(P \Rightarrow Q)\{Y/X_v\}]\!]_\rho$$

- $[\![\exists x.P]\!]_{[\rho|X_v \mapsto [\![Y]\!]_\rho]}$
  $$= \{s, h \mid \exists v \in Val.[s \mid x \mapsto v], h \in [\![P]\!]_{[\rho|X_v \mapsto [\![Y]\!]_\rho]}\}$$
  $$= \{s, h \mid \exists v \in Val.[s \mid x \mapsto v], h \in [\![P\{Y/X_v\}]\!]_\rho\} \quad \text{(ind.hyp)}$$
  $$= [\![\exists x.(P\{Y/X_v\})]\!]_\rho$$
  $$= [\![(\exists x.P)\{Y/X_v\}]\!]_\rho$$

- $\llbracket P * Q \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho]}$

  $= \{s, h \mid \exists h_0, h_1 \ h_0 \sharp h_1, \ h = h_0.h_1$
  $\quad s, h_0 \in \llbracket P \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho]} \text{ and } s, h_1 \in \llbracket Q \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho]}\}$

  $= \{s, h \mid \exists h_0, h_1 \ h_0 \sharp h_1, \ h = h_0.h_1$
  $\quad s, h_0 \in \llbracket P\{Y/X_v\} \rrbracket_\rho \text{ and } s, h_1 \in \llbracket Q\{Y/X_v\} \rrbracket_\rho\}$ \quad (ind. hyp.)

  $= \llbracket P\{Y/X_v\} * Q\{Y/X_v\} \rrbracket_\rho$

  $= \llbracket (P * Q)\{Y/X_v\} \rrbracket_\rho$

- $\llbracket P \mathbin{-\!\!*} Q \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho]}$

  $= \{s, h \mid \forall h'. \text{ if } h' \sharp h \text{ and } s, h' \in \llbracket P \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho]} \text{ then}$
  $\quad s, h.h' \in \llbracket Q \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho]}\}$

  $= \{s, h \mid \forall h'. \text{ if } h' \sharp h \text{ and } s, h' \in \llbracket P\{Y/X_v\} \rrbracket_\rho \text{ then}$
  $\quad s, h.h' \in \llbracket Q\{Y/X_v\} \rrbracket_\rho\}$ \quad (ind. hyp.)

  $= \llbracket P\{Y/X_v\} \mathbin{-\!\!*} Q\{Y/X_v\} \rrbracket_\rho$

  $= \llbracket (P \mathbin{-\!\!*} Q)\{Y/X_v\} \rrbracket_\rho$

- $\llbracket X_v \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho]}$

  $= \llbracket Y \rrbracket_\rho$

  $= \llbracket X_v\{Y/X_v\} \rrbracket_\rho$

- $\llbracket Z_v \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho]}$ with $Z_v \neq X_v$

  $= \llbracket Z_v \rrbracket_\rho$

  $= \llbracket Z_v\{Y/X_v\} \rrbracket_\rho$

- $\llbracket \mu X_v.P \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho]}$

  $= \mathrm{lfp}_\emptyset^\subseteq \lambda Z.\llbracket P \rrbracket_{[[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho] | X_v \mapsto Z]}$

  $= \mathrm{lfp}_\emptyset^\subseteq \lambda Z.\llbracket P \rrbracket_{[\rho | X_v \mapsto Z]}$

  $= \mathrm{lfp}_\emptyset^\subseteq \lambda X.\llbracket P \rrbracket_{[\rho | X_v \mapsto X]}$

  $= \llbracket \mu X_v.P \rrbracket_\rho$

  $= \llbracket (\mu X_v.P)\{Y/X_v\} \rrbracket_\rho$

- $\llbracket \mu Z_v.P \rrbracket_{[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho]}$ with $X_v \neq Z_v$

  $= \mathrm{lfp}_\emptyset^\subseteq \lambda Z.\llbracket P \rrbracket_{[[\rho | X_v \mapsto \llbracket Y \rrbracket_\rho] | Z_v \mapsto Z]}$

  $= \mathrm{lfp}_\emptyset^\subseteq \lambda Z.\llbracket P \rrbracket_{[[\rho | Z_v \mapsto Z] | X_v \mapsto \llbracket Y \rrbracket_\rho]}$

  $= \mathrm{lfp}_\emptyset^\subseteq \lambda Z.\llbracket P\{Y/X_v\} \rrbracket_{[\rho | Z_v \mapsto Z]}$ \quad (ind. hyp.)

  $= \llbracket \mu Z_v.(P\{Y/X_v\}) \rrbracket_\rho$

  $= \llbracket (\mu Z_v.P)\{Y/X_v\} \rrbracket_\rho$

61

- $\llbracket \nu X_v.P \rrbracket_{[\rho|X_v \mapsto \llbracket Y \rrbracket_\rho]}$

  $\begin{aligned}
  &= \quad \mathrm{gfp}_\emptyset^\subseteq \ \lambda Z.\llbracket P \rrbracket_{[[\rho|X_v \mapsto \llbracket Y \rrbracket_\rho]|X_v \mapsto Z]} \\
  &= \quad \mathrm{gfp}_\emptyset^\subseteq \ \lambda Z.\llbracket P \rrbracket_{[\rho|X_v \mapsto Z]} \\
  &= \quad \mathrm{gfp}_\emptyset^\subseteq \ \lambda X.\llbracket P \rrbracket_{[\rho|X_v \mapsto X]} \\
  &= \quad \llbracket \nu X_v.P \rrbracket_\rho \\
  &= \quad \llbracket (\nu X_v.P)\{Y/X_v\} \rrbracket_\rho
  \end{aligned}$

- $\llbracket \nu Z.P \rrbracket_{[\rho|X_v \mapsto \llbracket Y \rrbracket_\rho]}$ with $X \neq Z$

  $\begin{aligned}
  &= \quad \mathrm{gfp}_\emptyset^\subseteq \ \lambda Z.\llbracket P \rrbracket_{[[\rho|X_v \mapsto \llbracket Y \rrbracket_\rho]|Z_v \mapsto Z]} \\
  &= \quad \mathrm{gfp}_\emptyset^\subseteq \ \lambda Z.\llbracket P \rrbracket_{[[\rho|Z_v \mapsto Z]|X_v \mapsto \llbracket Y \rrbracket_\rho]} \\
  &= \quad \mathrm{gfp}_\emptyset^\subseteq \ \lambda Z.\llbracket P\{Y/X_v\} \rrbracket_{[\rho|Z_v \mapsto Z]} \quad \text{(ind. hyp.)} \\
  &= \quad \llbracket \nu Z.(P\{Y/X_v\}) \rrbracket_\rho \\
  &= \quad \llbracket (\nu Z.P)\{Y/X_v\} \rrbracket_\rho
  \end{aligned}$

- $\llbracket P[E'/x] \rrbracket_{[\rho|X_v \mapsto \llbracket Y \rrbracket_\rho]}$

  $\begin{aligned}
  &= \quad \{s,h \mid [s \mid x \mapsto \llbracket E' \rrbracket s], h \in \llbracket P \rrbracket_{[\rho|X_v \mapsto \llbracket Y \rrbracket_\rho]}\} \\
  &= \quad \{s,h \mid [s \mid x \mapsto \llbracket E' \rrbracket s], h \in \llbracket P\{Y/X_v\} \rrbracket_\rho\} \quad \text{(ind. hyp.)} \\
  &= \quad \llbracket P\{Y/X_v\}[E'/x] \rrbracket_\rho \\
  &= \quad \llbracket (P[E'/x])\{Y/X_v\} \rrbracket_\rho
  \end{aligned}$

$\square$

Lemma 2.33. *If* $\llbracket Y \rrbracket_\rho$ *exists*

$$sp(P,C)\{Y/X_v\} = sp(P\{Y/X_v\},C)$$

Lemma 2.33. Proof by induction on C.

- Case $x := E$

  $\begin{aligned}
  &\quad (sp(P,C)\{Y/X_v\}) \\
  &= \quad (\exists x'.P[x'/x] \wedge x = E[x'/x])\{Y/X_v\} \\
  &= \quad (\exists x'.(P\{Y/X_v\})[x'/x] \wedge x = E[x'/x]) \\
  &= \quad sp(P\{Y/X_v\},C)
  \end{aligned}$

- Case $x := E.i$

  $\begin{aligned}
  &\quad (sp(P,C)\{Y/X_v\}) \\
  &= \quad (\exists x'.P[x'/x] \wedge x = (E[x'/x].i))\{Y/X_v\} \\
  &= \quad (\exists x'.(P\{Y/X_v\})[x'/x] \wedge x = (E[x'/x].i)) \\
  &= \quad sp(P\{Y/X_v\},C)
  \end{aligned}$

- Case $E.1 := E'$

$$
\begin{aligned}
& (sp(P,C)\{Y/X_v\}) \\
=\ & (\exists x_1, x_2.(E \mapsto E', x_2) * ((E \mapsto x_1, x_2) \twoheadrightarrow P)\{Y/X_v\} \\
=\ & (\exists x_1, x_2.(E \mapsto E', x_2) * ((E \mapsto x_1, x_2) \twoheadrightarrow (P\{Y/X_v\})) \\
=\ & sp(P\{Y/X_v\}, C)
\end{aligned}
$$

- Case $E.2 := E'$

$$
\begin{aligned}
& (sp(P,C)\{Y/X_v\}) \\
=\ & (\exists x_1, x_2.(E \mapsto E', x_2) * ((E \mapsto x_1, x_2) \twoheadrightarrow P)\{Y/X_v\} \\
=\ & (\exists x_1, x_2.(E \mapsto x_1, E') * ((E \mapsto x_1, x_2) \twoheadrightarrow (P\{Y/X_v\})) \\
=\ & sp(P\{Y/X_v\}, C)
\end{aligned}
$$

- Case $x := \mathtt{cons}(E_1, E_2)$

$$
\begin{aligned}
& (sp(P,C)\{Y/X_v\}) \\
=\ & \exists x'.(P[x'/x] * (x \mapsto E_1[x'/x], E_2[x'/x]))\{Y/X_v\} \\
=\ & \exists x'.((P\{Y/X_v\})[x'/x] * (x \mapsto E_1[x'/x], E_2[x'/x])) \\
=\ & sp(P\{Y/X_v\}, C)
\end{aligned}
$$

- Case $dispose(E)$

$$
\begin{aligned}
& (sp(P,C)\{Y/X_v\}) \\
=\ & \exists x_1, x_2.\ ((E \mapsto x_1, x_2) \twoheadrightarrow P)\{Y/X_v\} \\
=\ & \exists x_1, x_2.\ ((E \mapsto x_1, x_2) \twoheadrightarrow (P\{Y/X_v\})) \\
=\ & sp(P\{Y/X_v\}, C)
\end{aligned}
$$

- Case $C_1; C_2$

$$
\begin{aligned}
& (sp(P,C)\{Y/X_v\}) \\
=\ & sp(sp(P,C_1), C_2)\{Y/X_v\} \\
=\ & sp(sp(P,C_1)\{Y/X_v\}, C_2) \quad \text{(ind.hyp)} \\
=\ & sp(sp(P\{Y/X_v\}, C_1), C_2) \quad \text{(ind.hyp)} \\
=\ & sp(P\{Y/X_v\}, C)
\end{aligned}
$$

- Case $if\ E\ then\ C_1\ else\ C_2$

$$
\begin{aligned}
& (sp(P,C)\{Y/X_v\}) \\
=\ & (sp(P \wedge E = \mathtt{true}, C_1) \vee sp(P \wedge E = \mathtt{false}, C_2))\{Y/X_v\} \\
=\ & sp(P \wedge E = \mathtt{true}, C_1)\{Y/X_v\} \vee sp(P \wedge E = \mathtt{false}, C_2)\{Y/X_v\} \\
=\ & sp(P\{Y/X_v\} \wedge E = \mathtt{true}, C_1) \vee sp(P\{Y/X_v\} \wedge E = \mathtt{false}, C_2) \quad \text{(ind. hyp.)} \\
=\ & sp(P\{Y/X_v\}, C)
\end{aligned}
$$

- Case $\mathtt{skip}$

$$(sp(P,C)\{Y/X_v\})$$
$$= P\{Y/X_v\}$$
$$= sp(P\{Y/X_v\},C)$$

- Case *while E do C_1*

  Case X_v not free in P
  $$(sp(P,C)\{Y/X_v\})$$
  $$= ((\mu X_v.sp(X_v \wedge E = \mathtt{true}, C_1) \vee P) \wedge (E = \mathtt{false}))\{Y/X_v\}$$
  $$= ((\mu X_v.sp(X_v \wedge E = \mathtt{true}, C_1) \vee P) \wedge (E = \mathtt{false}))$$
  $$= ((\mu X_v.sp(X_v \wedge E = \mathtt{true}, C_1) \vee P\{Y/X_v\}) \wedge (E = \mathtt{false}))$$
  $$= sp(P\{Y/X_v\},C)$$
  Case X_v free in P
  $$(sp(P,C)\{Y/X_v\})$$
  $$= ((\mu Z_v.sp(Z_v \wedge E = \mathtt{true}, C_1) \vee P) \wedge (E = \mathtt{false}))\{Y/X_v\}$$
  $$= ((\mu Z_v.sp(Z_v \wedge E = \mathtt{true}, C_1) \vee P\{Y/X_v\}) \wedge (E = \mathtt{false}))$$
  $$= ((\mu W_v.sp(W_v \wedge E = \mathtt{true}, C_1) \vee P\{Y/X_v\}) \wedge (E = \mathtt{false}))$$
  $$= sp(P\{Y/X_v\},C)$$
  We have the egality between the case Z_v and W_v since $[\![Y]\!]_\rho$ exists so there is no free

  formula variables in Y, so neither Z_v or W_v can become bounded in some Y placed in

  P.

  $\square$

2.6.7 μ and ν coincide

Theorem 2.34.
$$\boxed{\begin{array}{l} \mu X_v.\ P \equiv \neg\nu X_v.\neg(P\{\neg X_v\}) \\ \nu X_v.\ P \equiv \neg\mu X_v.\neg(P\{\neg X_v\}) \end{array}}$$

Theorem 2.34. First we recall the definition of equivalence between formulae $P \equiv Q$ iff

$\forall \rho.([\![P]\!]_\rho = [\![Q]\!]_\rho) \vee ([\![P]\!]_\rho$ and $[\![Q]\!]_\rho)$ both do not exist.

In this proof, we write $\top$ for $S \times H$, we sometime write C^c for $\top \setminus C$.

Let $B = [\![\nu X_v.P]\!]_\rho$ and $A = [\![\mu X_v.\neg(P\{\neg X_v/X_v\})]\!]_\rho$, we want to prove that

(B exists $\Rightarrow A$ exists and $B^c = A$) , and (A exists $\Rightarrow B$ exists and $A^c = B$).

If B exists, then

- $B = \mathrm{gfp}_{\emptyset}^{\subseteq} \lambda X.\ [\![P]\!]_{[\rho|X_v\to X]}$, so

- B is the biggest such that $B = [\![P]\!]_{[\rho|X_v \to B]}$, so

- B^c is the smallest such that $\top \setminus B^c = [\![P]\!]_{[\rho|X_v \to \top \setminus B^c]}$, so

- B^c is the smallest such that $B^c = \top \setminus [\![P]\!]_{[\rho|X_v \to \top \setminus B^c]}$, so

- $B^c = \text{lfp}_{\emptyset}^{\subseteq} \lambda X.\ \top \setminus [\![P]\!]_{[\rho|X_v \to \top \setminus X]}$, then from Th. 2.32

- $B^c = \text{lfp}_{\emptyset}^{\subseteq} \lambda X.\ \top \setminus [\![P\{\neg X_v/X_v\}]\!]_{[\rho|X_v \to X]}$, which is

- $B^c = \text{lfp}_{\emptyset}^{\subseteq} \lambda X.\ [\![\neg P\{\neg X_v/X_v\}]\!]_{[\rho|X_v \to X]}$, so

- A exists and $B^c = A$

  If A exists, then

- $A = \text{lfp}_{\emptyset}^{\subseteq} \lambda X.\ [\![\neg P\{\neg X_v/X_v\}]\!]_{[\rho|X_v \to X]}$, so

- $A = \text{lfp}_{\emptyset}^{\subseteq} \lambda X.\ \top \setminus [\![P\{\neg X_v/X_v\}]\!]_{[\rho|X_v \to X]}$, then from Th. 2.32

- $A = \text{lfp}_{\emptyset}^{\subseteq} \lambda X.\ \top \setminus [\![P]\!]_{[\rho|X_v \to \top \setminus X]}$, which is

- A is the smallest such that $A = \top \setminus [\![P]\!]_{[\rho|X_v \to \top \setminus A]}$, so

- A is the smallest such that $\top \setminus A = [\![P]\!]_{[\rho|X_v \to \top \setminus A]}$, so

- A^c is the biggest such that $A^c = [\![P]\!]_{[\rho|X_v \to A^c]}$, so

- $A^c = \text{gfp}_{\emptyset}^{\subseteq} \lambda X.\ [\![P]\!]_{[\rho|X_v \to X]}$, so

- B exists and $A^c = B$

  For the case $\mu X_v.P \equiv \neg \nu. \neg(P\{\neg X_v/X_v\})$, we proceed the same way. $\qquad \square$

2.6.8 Simplifications on $[\ /\ ]$

Theorem 2.35. $\boxed{\text{If } \begin{array}{l} P \text{ is } v\text{-closed} \\ z \notin Var(P) \end{array} \text{ then } P[z/y] \equiv P\{[z/y]\} \wedge is(z)}$

Th. 2.35. From the generalized variable renaming theorem Th. 2.8, $[\![P\{[z/y]\}]\!] = \{s, h \mid s^\bullet, h \in [\![P]\!]\}$,

so $[\![P\{[z/y]\} \wedge is(z)]\!] = \{s, h \mid [s \mid y \to s(z)], h \in [\![P]\!]\}$,

which is $[\![P[z/y]]\!]$. $\qquad \square$

Some more simplifications:

Remember that we have in Lemma 2.22 $[\![E\{E'/x\}]\!]^s = [\![E]\!]^{[s|x \to [\![E']\!]s]}$ if $[\![E']\!]s$ exists.

- $\boxed{(E_1 = E_2)[E'/x] \equiv (E_1\{E'/x\} = E_2\{E'/x\}) \land is(E')}$:

$$
\begin{aligned}
& [\![(E_1\{E'/x\} = E_2\{E'/x\}) \land is(E')]\!] \\
= \ & \{s,h \mid [\![E_1\{E'/x\}]\!]^s = [\![E_2\{E'/x\}]\!]^s \text{ and} [\![E']\!]s \text{ exists}\} \\
= \ & \{s,h \mid [\![E_1]\!]^{[s|x\to[\![E']\!]s]} = [\![E_2]\!]^{[s|x\to[\![E']\!]s]}\} \\
= \ & \{s,h \mid [\![E_1]\!]^{[s|x\to[\![E']\!]s]} = [\![E_2]\!]^{[s|x\to[\![E']\!]s]}\} \\
= \ & \{s,h \mid [s \mid x \to [\![E']\!]s], h \in [\![E_1 = E_2]\!]\} \\
= \ & [\![(E_1 = E_2)[E'/x]]\!]
\end{aligned}
$$

- $\boxed{(E \mapsto E_1, E_2)[E'/x] \equiv (E\{E'/x\} \mapsto E_1\{E'/x\}, E_2\{E'/x\}) \land is(E')}$:

$$
\begin{aligned}
& [\![(E\{E'/x\} \mapsto E_1\{E'/x\}, E_2\{E'/x\}) \land is(E')]\!] \\
= \ & \{s,h \mid dom(h) = \{[\![E\{E'/x\}]\!]^s\} \text{ and } h([\![E\{E'/x\}]\!]^s) = \langle [\![E_1\{E'/x\}]\!]^s, [\![E_2\{E'/x\}]\!]^s \rangle \\
& \quad \text{and } [\![E']\!]s \text{ exists}\} \\
= \ & \{s,h \mid dom(h) = \{[\![E]\!]^{[s|x\mapsto[\![E']\!]s]}\} \\
& \quad \text{and } h([\![E]\!]^{[s|x\mapsto[\![E']\!]s]}) = \langle [\![E_1]\!]^{[s|x\mapsto[\![E']\!]s]}, [\![E_2]\!]^{[s|x\mapsto[\![E']\!]s]} \rangle \} \\
= \ & \{s,h \mid [s \mid x \mapsto [\![E']\!]s], h \in [\![E \mapsto E_1, E_2]\!]\} \\
= \ & [\![(E \mapsto E_1, E_2)[E'/x]]\!]
\end{aligned}
$$

- $\boxed{\texttt{false}[E'/x] \equiv \texttt{false}}$:

$$
\begin{aligned}
& [\![\texttt{false}[E'/x]]\!] \\
= \ & \{s,h \mid [s \mid x \to [\![E']\!]s], h \in [\![\texttt{false}]\!]\} \\
= \ & \emptyset \\
= \ & [\![\texttt{false}]\!]
\end{aligned}
$$

- $\boxed{(P \Rightarrow Q)[E'/x] \equiv P[E'/x] \Rightarrow Q[E'/x]}$:

$$
\begin{aligned}
& [\![(P \to Q)[E'/x]]\!]_\rho \\
= \ & \{s,h \mid [s \mid x \to [\![E']\!]s], h \in [\![P \Rightarrow Q]\!]_\rho\} \\
= \ & \{s,h \mid [s \mid x \to [\![E']\!]s], h \in ((\top \setminus [\![P]\!]_\rho) \cup [\![Q]\!]_\rho)\} \\
= \ & ((\top \setminus \{s,h \mid [s \mid x \to [\![E']\!]s], h \in [\![P]\!]_\rho\}) \cup \{s,h \mid [s \mid x \to [\![E']\!]s], h \in [\![Q]\!]_\rho\}) \\
= \ & ((\top \setminus [\![P[E'/x]]\!]_\rho) \cup [\![Q[E'/x]]\!]_\rho) \\
= \ & [\![P[E'/x] \Rightarrow Q[E'/x]]\!]_\rho
\end{aligned}
$$

- $\boxed{(\exists x.P)[E/x] \equiv (\exists x.P) \wedge is(E)}$:

$$\begin{aligned}
&\llbracket (\exists x.P)[E/x] \rrbracket_\rho \\
=\ & \{s, h \mid [s \mid x \to \llbracket E \rrbracket s], h \in \llbracket \exists x.P \rrbracket_\rho \} \\
=\ & \{s, h \mid \exists v.[s \mid x \to \llbracket E \rrbracket s \mid x \to v], h \in \llbracket P \rrbracket_\rho \} \\
=\ & \{s, h \mid \exists v.[s \mid x \to v], h \in \llbracket P \rrbracket_\rho \text{ and } \llbracket E \rrbracket s \text{ exists} \} \\
=\ & \llbracket (\exists x.P) \wedge is(E) \rrbracket_\rho
\end{aligned}$$

- $\boxed{\text{If } \begin{array}{l} y \notin Var(E) \\ x \neq y \end{array} \text{ then } (\exists y.P)[E/x] \equiv \exists y.(P[E/x])}$:

First, notice that if $y \notin Var(E)$, then $\llbracket E \rrbracket s = \llbracket E \rrbracket^{[s \mid y \to v]}$.

$$\begin{aligned}
&\llbracket (\exists y.P)[E/x] \rrbracket_\rho \\
=\ & \{s, h \mid [s \mid x \to \llbracket E \rrbracket s], h \in \llbracket \exists x.P \rrbracket_\rho \} \\
=\ & \{s, h \mid \exists v.[s \mid x \to \llbracket E \rrbracket s \mid y \to v], h \in \llbracket P \rrbracket_\rho \} \\
=\ & \{s, h \mid \exists v.[s \mid y \to v \mid x \to \llbracket E \rrbracket s], h \in \llbracket P \rrbracket_\rho \} \\
=\ & \{s, h \mid \exists v.[s \mid y \to v \mid x \to \llbracket E \rrbracket^{[s \mid y \to v]}], h \in \llbracket P \rrbracket_\rho \} \\
=\ & \{s, h \mid \exists v.[s \mid y \to v], h \in \llbracket P[E/x] \rrbracket_\rho \} \\
=\ & \llbracket \exists y.(P[E/x]) \rrbracket_\rho
\end{aligned}$$

- $\boxed{\text{emp}[E'/x] \equiv \text{emp} \wedge is(E')}$:

$$\begin{aligned}
=\ & \llbracket \text{emp} \wedge is(E') \rrbracket \\
=\ & \{s, h \mid h = [] \text{ and } \llbracket E' \rrbracket s \text{ exists} \} \\
=\ & \{s, h \mid [s \mid x \mapsto \llbracket E' \rrbracket s], h \in \llbracket \text{emp} \rrbracket \} \\
=\ & \llbracket \text{emp}[E'/x] \rrbracket
\end{aligned}$$

- $\boxed{(P * Q)[E'/x] \equiv P[E'/x] * Q[E'/x]}$:

$$\begin{aligned}
=\ & \llbracket P[E'/x] * Q[E'/x] \rrbracket_\rho \\
=\ & \{s, h \mid \exists h_0, h_1.\ h_0 \sharp h_1, h_0.h_1 = h, s, h_0 \in \llbracket P[E'/x] \rrbracket_\rho \text{ and } s, h_1 \in \llbracket Q[E'/x] \rrbracket_\rho \} \\
=\ & \{s, h \mid \exists h_0, h_1.\ h_0 \sharp h_1, h_0.h_1 = h, [s \mid x \mapsto \llbracket E' \rrbracket s], h_0 \in \llbracket P \rrbracket_\rho \\
& \text{and } [s \mid x \mapsto \llbracket E' \rrbracket s], h_1 \in \llbracket Q \rrbracket_\rho \} \\
=\ & \{s, h \mid [s \mid x \mapsto \llbracket E' \rrbracket s], h \in \llbracket P * Q \rrbracket_\rho \} \\
=\ & \llbracket (P * Q)[E'/x] \rrbracket_\rho
\end{aligned}$$

- $\boxed{(P{\twoheadrightarrow}Q)[E'/x] \equiv P[E'/x]{\twoheadrightarrow}Q[E'/x]}$:

$$
\begin{aligned}
&= \quad [\![P[E'/x]{\twoheadrightarrow}Q[E'/x]]\!]_\rho \\
&= \quad \{s,h \mid \forall h'.\ \text{if } h'\sharp h \text{ and } s,h' \in [\![P[E'/x]]\!]_\rho \text{ then } s,h.h' \in [\![Q[E'/x]]\!]_\rho\} \\
&= \quad \{s,h \mid \forall h'.\ \text{if } h'\sharp h \text{ and } [s \mid x \mapsto [\![E']\!]s], h' \in [\![P]\!]_\rho \text{ then} \\
&\qquad [s \mid x \mapsto [\![E']\!]s], h.h' \in [\![Q]\!]_\rho\} \\
&= \quad \{s,h \mid [s \mid x \mapsto [\![E']\!]s], h \in [\![P{\twoheadrightarrow}Q]\!]_\rho\} \\
&= \quad [\![(P{\twoheadrightarrow}Q)[E'/x]]\!]_\rho
\end{aligned}
$$

2.6.9 sp's proofs

Case $x := E$

$$
\begin{aligned}
sp_o(\gamma(P), x := E) &= \{s',h' \mid \exists s,h.\ s,h \models P \wedge\ h' = h \wedge\ s' = [s \mid x \to [\![E]\!]s]\} \\
&= \{s',h' \mid \exists s.s, h' \models P \wedge\ s' = [s \mid x \to [\![E]\!]s]\} \\
\gamma(sp(P, x := E)) &= \{s',h' \mid s',h' \models \exists x'.\ P[x'/x] \wedge\ x = E\{x'/x\}\} \\
&= \{s',h' \mid \exists v.[s' \mid x' \to v], h' \models P[x'/x] \\
&\qquad \wedge [s' \mid x' \to v](x) = [\![E\{x'/x\}]\!][s' \mid x' \to v]\} \\
&= \{s',h' \mid \exists v.[s \mid x' \to v \mid x \to v], h' \models P \\
&\qquad \wedge s'(x) = [\![E]\!][s' \mid x' \to v \mid x \to v]\} \\
&= \{s',h' \mid \exists v.[s' \mid x' \to v \mid x \to v], h' \models P \wedge\ s'(x) = [\![E]\!][s' \mid x \to v]\} \\
&= \{s',h' \mid \exists v.[s' \mid v \mid x \to v], h' \models P \wedge\ s'(x) = [\![E]\!][s' \mid x \to v]\}
\end{aligned}
$$

The last equality is because $x' \notin Var(P)$.

We can prove the inclusion by taking $v = s(x)$ if $x \in dom(s)$ and any value otherwise.

We could also prove the inclusion in the other way by taking $s = [s' \mid x \to v]$.

So we have $sp_o(\gamma(P), x := E) = \gamma(sp(P, x := E))$.

Case $x := E.i$

$$
\begin{aligned}
sp_o(\gamma(P), x := E.i) \;=\; & \{s', h' \mid \exists s, h.s, h \models P \wedge\ h' = h \wedge\ [\![E]\!]s \in Loc \\
& \wedge\ (\exists v.\ v = \pi_i(h([\![E]\!]s)) \wedge\ s' = [s \mid x \to v])\} \\
=\; & \{s', h' \mid \exists s.s, h' \models P \wedge\ (\exists v.\ v = \pi_i(h'([\![E]\!]s)) \wedge\ s' = [s \mid x \to v])\} \\
=\; & \{s', h' \mid \exists s.s, h' \models P \wedge\ s' = [s \mid x \to \pi_i(h'([\![E]\!]s))])\} \\
\gamma(sp(P, x := E.i)) \;=\; & \{s', h' \mid s', h' \models \exists x'.\ P[x'/x] \wedge\ x = (E\{x'/x\}).i\} \\
=\; & \{s', h' \mid \exists v.[s' \mid x' \to v], h' \models P[x'/x] \\
& \wedge\ [s' \mid x' \to v](x) = \pi_i(h([\![E\{x'/x\}]\!][s' \mid x' \to v]))\} \\
=\; & \{s', h' \mid \exists v.[s \mid x' \to v \mid x \to v], h' \models P \\
& \wedge\ s'(x) = \pi_i(h([\![E]\!][s' \mid x' \to v \mid x \to v]))\} \\
=\; & \{s', h' \mid \exists v.[s' \mid x' \to v \mid x \to v], h' \models P \\
& \wedge\ s'(x) = \pi_i(h([\![E]\!][s' \mid x \to v]))\} \\
=\; & \{s', h' \mid \exists v.[s' \mid x \to v], h' \models P \wedge\ s'(x) = \pi_i(h([\![E]\!][s' \mid x \to v]))\}
\end{aligned}
$$

The last equality is because $x' \notin Var(P)$.

We can prove the inclusion by taking $v = s(x)$ if $x \in dom(s)$ and any value otherwise.

We could also prove the inclusion in the other way by taking $s = [s' \mid x \to v]$.

So we have $sp_o(\gamma(P), x := E.i)\ =\ \gamma(sp(P, x := E.i))$.

Case $E_1.i := E_2$

$$
\begin{aligned}
sp_o(\gamma(P), E_1.1 := E_2) \;=\; & \{s', h' \mid \exists h.\;\; s', h \in \gamma(P) \wedge \exists v_1, v_2.h(\llbracket E_1 \rrbracket^{s'}) = \langle v_1, v_2 \rangle \\
& \qquad \wedge h' = [h \mid \llbracket E_1 \rrbracket^{s'} \to \langle \llbracket E_2 \rrbracket^{s'}, v_2 \rangle]\} \\
\gamma(sp(P, E_1 := E_2.i)) \;=\; & \{s', h' \mid \qquad s', h' \in \gamma(\exists x_1, x_2.\; (E \mapsto E_2, x_2) * ((E_1 \mapsto x_1, x_2) \mathbin{\rightarrow\!\!\!*} P))\} \\
\;=\; & \{s', h' \mid \qquad \exists v_1, v_2.\; \exists h'_0, h'_1. h'_0 \sharp h'_1. \\
& \qquad \wedge\; h' = h'_0 \cdot h'_1 \\
& \qquad \wedge\; [s \mid x_i \mapsto v_i], h'_0 \in \gamma(E_1 \mapsto E_2, x_2) \\
& \qquad \wedge\; [s \mid x_i \mapsto v_i], h'_1 \in \gamma((E_1 \mapsto x_1, x_2) \mathbin{\rightarrow\!\!\!*} P)\} \\
\;=\; & \{s', h' \mid \qquad \exists v_1, v_2.\; \exists h'_0, h'_1. h'_0 \sharp h'_1. \\
& \qquad \wedge\; h' = h'_0 \cdot h'_1 \\
& \qquad \wedge \llbracket E_1 \rrbracket^{s'} \in Loc \\
& \qquad \wedge h'_0 = [\llbracket E_1 \rrbracket^{s'} \mapsto \langle \llbracket E_2 \rrbracket^{s'}, v_2 \rangle] \text{ (using } x_i \notin Var(E_2)) \\
& \qquad \wedge \forall h_0. \text{If } h_0 \sharp h'_1 \text{ and } h_0 = [\llbracket E_1 \rrbracket^{s'} \mapsto \langle v_1, v_2 \rangle] \\
& \qquad\quad \text{then } s', h_0 \cdot h'_1 \in \gamma(P)\} \\
\;=\; & \{s', h' \mid \qquad \exists v_1, v_2.\; \exists h'_0, h'_1. h'_0 \sharp h'_1. \\
& \qquad \wedge\; h' = h'_0 \cdot h'_1 \\
& \qquad \wedge \llbracket E_1 \rrbracket^{s'} \in Loc \\
& \qquad \wedge h'_0 = [\llbracket E_1 \rrbracket^{s'} \mapsto \langle \llbracket E_2 \rrbracket^{s'}, v_2 \rangle] \text{ (using } x_i \notin Var(E_2)) \\
& \qquad \wedge \text{If } \llbracket E_1 \rrbracket^{s'} \notin dom(h'_1) \\
& \qquad\quad \text{then } s', [\llbracket E_1 \rrbracket^{s'} \mapsto \langle v_1, v_2 \rangle] \cdot h'_1 \in \gamma(P)\} \\
\;=\; & \{s', h' \mid \qquad \exists v_1, v_2.\; \exists h'_0, h'_1. h'_0 \sharp h'_1. \\
& \qquad \wedge\; h' = h'_0 \cdot h'_1 \\
& \qquad \wedge \llbracket E_1 \rrbracket^{s'} \in Loc \\
& \qquad \wedge h'_0 = [\llbracket E_1 \rrbracket^{s'} \mapsto \langle \llbracket E_2 \rrbracket^{s'}, v_2 \rangle] \text{ (using } x_i \notin Var(E_2)) \\
& \qquad \wedge s', [\llbracket E_1 \rrbracket^{s'} \mapsto \langle v_1, v_2 \rangle] \cdot h'_1 \in \gamma(P)\}
\end{aligned}
$$

We can prove the inclusion by taking $h'_1 = h \mid_{dom(h) \backslash \llbracket E_1 \rrbracket^{s'}}$. We could also prove the inclusion in the other way by taking $h = [\llbracket E_1 \rrbracket^{s'} \mapsto \langle v_1, v_2 \rangle] \cdot h'_1$.

So we have $sp_o(\gamma(P), E_1 := E_2.i) \;=\; \gamma(sp(P, E_1 := E_2.i))$.

Case $x := \mathtt{cons}(E_1, E_2)$

Not typed yet.

Case $\mathtt{dispose}(E)$

Not typed yet.

Case $C_1; C_2$

We prove that $sp_o(\gamma(P), C_1; C_2) = \gamma(sp(P, C_1; C_2))$ by induction on the size of the command.

$$
\begin{aligned}
sp_o(\gamma(P), C_1; C_2) &= sp_o(sp_o(\gamma(P), C_1), C_2) && definition \\
&= sp_o(\gamma(sp(P, C_1)), C_2) && induction\ hypothesis \\
&= \gamma(sp((sp(P, C_1)), C_2)) && induction\ hypothesis \\
&= \gamma(sp(P, C_1; C_2)) && definition
\end{aligned}
$$

Case $if\ E\ then\ C_1\ else\ C_2$

$C = if\ E\ then\ C_1\ else\ C_2$

$$
\begin{aligned}
sp_o(\gamma(P), C) &= \{s', h' \mid \exists s, h.\ s, h \models P && \wedge\ ((\llbracket E \rrbracket s = True\ \wedge\ s', h' \in sp_o(\{s, h\}, C_1)) \\
& && \vee(\llbracket E \rrbracket s = False\ \wedge\ s', h' \in sp_o(\{s, h\}, C_2))))\} \\
&= \{s', h' \mid \exists s, h.\ ((s, h \models P && \wedge E = \texttt{true}\ \wedge\ s', h' \in sp_o(\{s, h\}, C_1)) \\
& \vee(s, h \models P && \wedge E = \texttt{false}\ \wedge\ s', h' \in sp_o(\{s, h\}, C_2)))\} \\
&= sp_o(\gamma(P \wedge E = \texttt{true}), C_1) \cup && sp_o(\gamma(P \wedge E = \texttt{false}), C_2) \\
\gamma(sp(P, C)) &= \{s', h' \mid s', h' \models && (sp(P \wedge E = \texttt{true}, C_1) \\
& && \vee sp(P \wedge E = \texttt{false}, C_2))\} \\
&= \{s', h' \mid s', h' \models && sp(P \wedge E = \texttt{true}, C_1)\} \\
& \cup \{s', h' \mid xs', h' \models && sp(P \wedge E = \texttt{false}, C_2)\} \\
&= \gamma(sp(P \wedge E = \texttt{true}, C_1)) \cup && \gamma(sp(P \wedge E = \texttt{false}, C_2))
\end{aligned}
$$

We prove by induction in the size of the command.

Case `skip`

$$
\begin{aligned}
sp_o(\gamma(P), \texttt{skip}) &= \gamma(P) \\
\gamma(sp(P, \texttt{skip})) &= \gamma(P)
\end{aligned}
$$

So we have $sp_o(\gamma(P), \texttt{skip}) = \gamma(sp(P, \texttt{skip}))$.

Case *while E do C_1*

We define $F_o = \lambda X.\ \gamma(P) \cup sp_o(X \cap \gamma(E = \mathtt{true}), C_1)$

and $F = \lambda X.\ \Gamma_{[X_v \mapsto X]}(sp(X_v \wedge E = \mathtt{true}, C_1) \vee P)$.

Lemma 2.36.
$$\forall n \geq 0.\, F_o^n(\emptyset) = F^n(\emptyset)$$

Lemma 2.36. We prove by recurrence that the $F_o^n(\emptyset) = F^n(\emptyset)$ and that $\exists Y.\ F^n(\emptyset) = \gamma(Y)$:

- Case n=0:
$$
\begin{aligned}
F_o^0(\emptyset) &= \gamma(P) \cup \emptyset \\
&= \gamma(P \vee \mathtt{false}) \\
F^0(\emptyset) &= \Gamma_{[X_v \mapsto \emptyset]}(sp(X_v \wedge E = \mathtt{true}, C_1) \vee P) \\
&= \Gamma_{[X_v \mapsto \gamma(\mathtt{false})]}(sp(X_v \wedge E = \mathtt{true}, C_1) \vee P) \\
&= \gamma(sp(\mathtt{false} \wedge E = \mathtt{true}, C_1) \vee P) \qquad \text{(by Th. 2.30)} \\
&= \gamma(\mathtt{false} \vee P)
\end{aligned}
$$

- Case n+1.: $F_o^{n+1}(\emptyset) = F_o(F_o^n(\emptyset))$

$= F_o(F^n(\emptyset))$

$= \gamma(P) \cup sp_o(F^n(\emptyset) \cap \gamma(E = \mathtt{true}), C_1)$

by induction hyp $\exists Y.\ F^n(\emptyset) = \gamma(Y)$

so $= \gamma(P) \cup sp_o(\gamma(Y) \cap \gamma(E = \mathtt{true}), C_1)$

$= \gamma(P) \cup sp_o(\gamma(Y \vee E = \mathtt{true}), C_1)$

$= \gamma(P) \cup \gamma(sp(Y \vee E = \mathtt{true}, C_1))$, by the global ind. hyp

$= \gamma(sp(Y \wedge E = \mathtt{true}, C_1) \vee P)$

$= \Gamma_{[X_v \mapsto \gamma(Y)]}(sp(X_v \wedge E = \mathtt{true}, C_1) \vee P)$, by Th. 2.30+ lemma 2.31+ X_v not free in

P

$= \Gamma_{[X_v \mapsto F^n]}(sp(X_v \wedge E = \mathtt{true}, C_1) \vee P)$

$= F^{n+1}(\emptyset)$

$\square$

Lemma 2.37.
F is upper-continuous.

Lemma 2.37. The proof come directly from lemma 2.41 and theorem 2.42 $\qquad\square$

$$
\begin{aligned}
sp_o(\gamma(P), w\ E\ d\ C_1) &= \left(\mathrm{lfp}_\emptyset^\subseteq \lambda X. \left\{ \begin{array}{l} s', h' \mid \exists s, h.\ s, h \in X \wedge \\ ((\llbracket E \rrbracket s = True\ \wedge s', h' \in sp_o(\{s,h\}, C_1)) \\ \vee(s', h' \in \gamma(P))) \end{array} \right\} \right) \\
&\quad \cap \{s', h' \mid \llbracket E \rrbracket s' = \texttt{false}\} \\
&= \left(\mathrm{lfp}_\emptyset^\subseteq \lambda X. \left\{ \begin{array}{l} s', h' \mid \exists s, h.\ s, h \in X \wedge \\ (\llbracket E \rrbracket s = True\ \wedge s', h' \in sp_o(\{s,h\}, C_1)) \end{array} \right\} \cup \gamma(P) \right) \\
&\quad \cap \gamma(E = \texttt{false}) \\
&= (\mathrm{lfp}_\emptyset^\subseteq \lambda X.\ \gamma(P) \cup\ sp_o(X \cap \gamma(E = \texttt{true}), C_1)) \\
&\quad \cap \gamma(E = \texttt{false}) \\
&= (\mathrm{lfp}_\emptyset^\subseteq F_o) \\
&\quad \cap \gamma(E = \texttt{false}) \\
(F_o\ u.c. + Tarski) &= \bigcup_{n \geq 0} F_o^n(\emptyset) \\
&\quad \cap \gamma(E = \texttt{false}) \\
(\text{Lem. } 2.36) &= \bigcup_{n \geq 0} F^n(\emptyset) \\
&\quad \cap \gamma(E = \texttt{false})
\end{aligned}
$$

$$
\begin{aligned}
\gamma(sp(P, w\ E\ d\ C_1)) &= \gamma(\mu X_v.\ (P \vee sp(X_v \wedge E = \texttt{true}, C_1)) \\
&\qquad \wedge(E = \texttt{false})) \\
&= \gamma(\mu X_v.\ (P \vee sp(X_v \wedge E = \texttt{true}, C_1))) \\
&\quad \cap \gamma(E = \texttt{false}) \\
&= \Gamma(\mu X_v.\ (P \vee sp(X_v \wedge E = \texttt{true}, C_1))) \\
&\quad \cap \gamma(E = \texttt{false}) \\
&= (\mathrm{lfp}_\emptyset^\subseteq \lambda X.\ \Gamma_{[X_v \mapsto X]}(sp(X_v \wedge E = \texttt{true}, C_1) \vee P)) \\
&\quad \cap \gamma(E = \texttt{false}) \\
&= (\mathrm{lfp}_\emptyset^\subseteq F) \\
&\quad \cap \gamma(E = \texttt{false}) \\
(\text{Lem. } 2.37 + Tarski) &= \bigcup_{n \geq 0} F^n(\emptyset) \\
&\quad \cap \gamma(E = \texttt{false})
\end{aligned}
$$

So we have that

$$sp_o(\gamma(P), while\ E\ do\ C_1) = \gamma(sp(P, while\ E\ do\ C_1)).$$

2.6.10 wlp's proofs

Most cases were already proven in other papers and are not much of interest. They are written on paper and we could type some cases if needed.

2.6.11 Upper-continuous results

Definition 2.38. $\lambda X.[\![P]\!]_\rho \in \mathcal{P}\ (S \times H) \longmapsto \mathcal{P}\ (S \times H)$ *is upper-continuous iff*

For any $\subseteq$-increasing chain x_i, $i \in \mathbb{Z}$:

$[\![P]\!]_{\rho[X_i/X]}$ *exists and:*

$$[\![P]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]} = \bigcup_{i\in\mathbb{Z}}[\![P]\!]_{\rho[X_i/X]}$$

Lemma 2.39. *If $\lambda X.[\![P]\!]_\rho$ is upper-continuous then $\lambda X.[\![P[x'/x]]\!]_\rho$ is upper-continuous.*

Lemma 2.39. By definition $[\![P[x'/x]]\!]_\rho = \{s, h \mid s[x \mapsto s(x')], h \in [\![P]\!]_\rho\}$. The lemma follows directly:

$$
\begin{aligned}
&(\lambda X.[\![P[x'/x]]\!]_\rho)(\bigcup_{i\in\mathbb{Z}} X_i)\\
=\ & [\![P[x'/x]]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\\
=\ & \{s, h \mid s[x \mapsto s(x')], h \in [\![P]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\}\\
=\ & \{s, h \mid s[x \mapsto s(x')], h \in \bigcup_{i\in\mathbb{Z}}[\![P]\!]_{\rho[X_i/X]}\} \quad \text{(by hyp.)}\\
=\ & \bigcup_{i\in\mathbb{Z}}\{s, h \mid s[x \mapsto s(x')], h \in [\![P]\!]_{\rho[X_i/X]}\}\\
=\ & \bigcup_{i\in\mathbb{Z}}[\![P[x'/x]]\!]_{\rho[X_i/X]}\\
=\ & \bigcup_{i\in\mathbb{Z}}(\lambda X.[\![P[x'/x]]\!]_\rho)(X_i)
\end{aligned}
$$
$\square$

Lemma 2.40. *If $\lambda X.F$ is upper-continuous and $\lambda X.G$ does not depend on X then $\lambda X.F \cap G$ is upper-continuous.*

Lemma 2.40.
$$
\begin{aligned}
&(\lambda X.F \cap G)(\bigcup_{i\in\mathbb{Z}} X_i)\\
=\ & F(\bigcup_{i\in\mathbb{Z}} X_i) \cap G(\bigcup_{i\in\mathbb{Z}} X_i)\\
=\ & (\bigcup_{i\in\mathbb{Z}}(F(X_i))) \cap G(\bigcup_{i\in\mathbb{Z}} X_i) \quad \text{(hyp. 1)}\\
=\ & (\bigcup_{i\in\mathbb{Z}}(F(X_i))) \cap G(X_i) \quad\quad \text{(hyp. 2)}\\
=\ & (\bigcup_{i\in\mathbb{Z}}(F(X_i) \cap G(X_i)) \quad\quad (G(X_i) \text{ does not depend on i})\\
=\ & (\bigcup_{i\in\mathbb{Z}}(\lambda X.F \cap G)(X_i)
\end{aligned}
$$
$\square$

Lemma 2.41. $\boxed{\lambda X.[\![X_v \wedge E = \texttt{true}]\!]_{[\rho|X_v \mapsto X]} \text{ is upper-continuous.}}$

Lemma 2.41.
$$\lambda X.[\![X_v \wedge E = \texttt{true}]\!]_{[\rho|X_v \mapsto X]}$$
$$= \quad \lambda X.X \cap [\![E = \texttt{true}]\!]$$
since X_v cannot occure in $E = \texttt{true}$.

The result then follows by Lemma 2.40. $\qquad\qquad\qquad\qquad$ $\square$

Theorem 2.42. $\boxed{\begin{array}{l} \forall P, C \;\; If \; \lambda X.[\![P]\!]_\rho \text{ is upper-continuous, then } \lambda X.[\![sp(P,C)]\!]_\rho \text{ is} \\ upper\text{-}continuous. \end{array}}$

Theorem 2.42. Proof by induction on the command C. Since we are working with function from sets to sets the union is always defined. Notice that we use by hypothesis $[\![P]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]} = \bigcup_{i\in\mathbb{Z}}[\![P]\!]_{\rho[X_i/X]}$ and want to prove that $[\![sp(P,C)]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]} = \bigcup_{i\in\mathbb{Z}}[\![sp(P,C)]\!]_{\rho[X_i/X]}$.

- Case C is $x := E$ then

$$\lambda X.[\![sp(P,C)]\!]_\rho(\bigcup_{i\in\mathbb{Z}} X_i)$$
$$= \quad [\![\exists x'. \; P[x'/x] \wedge x = E[x'/x]]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}$$
$$= \quad \{s,h \mid \exists v \in Val.[s \mid x \mapsto v], h \in [\![P[x'/x] \wedge x = E[x'/x]]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\}$$
$$= \quad \{s,h \mid \exists v \in Val.$$
$$[s \mid x \mapsto v], h \in [\![P[x'/x]]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]} \text{ and}$$
$$[s \mid x \mapsto v], h \in [\![x = E[x'/x]]\!]\}$$
$$= \quad \{s,h \mid \exists v \in Val.$$
$$[s \mid x \mapsto v], h \in \bigcup_{i\in\mathbb{Z}}[\![P[x'/x]]\!]_{\rho[X_i/X]} \text{ and}$$
$$[s \mid x \mapsto v], h \in [\![x = E[x'/x]]\!])\} \;\; (\text{lem. } 2.39+ \text{ hyp.})$$
$$= \quad \{s,h \mid \exists v \in Val.[s \mid x \mapsto v], h \in \bigcup_{i\in\mathbb{Z}}[\![P[x'/x] \wedge x = E[x'/x]]\!]_{\rho[X_i/X]}\}$$
$$= \quad \bigcup_{i\in\mathbb{Z}}\{s,h \mid \exists v \in Val.[s \mid x \mapsto v], h \in [\![P[x'/x] \wedge x = E[x'/x]]\!]_{\rho[X_i/X]}\}$$
$$= \quad \bigcup_{i\in\mathbb{Z}}[\![\exists x'. \; P[x'/x] \wedge x = E[x'/x]]\!]_{\rho[X_i/X]}$$
$$= \quad \bigcup_{i\in\mathbb{Z}}(\lambda X.[\![sp(P,C)]\!]_\rho(X_i))$$

- Case C is $x := E.i$ then

75

$$\lambda X.\llbracket sp(P,C)\rrbracket_\rho(\bigcup_{i\in\mathbb{Z}} X_i)$$

$$= \quad \llbracket \exists x'.\ P[x'/x] \wedge x = (E[x'/x]).i\rrbracket_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}$$

$$= \quad \llbracket \exists x'.\ P[x'/x] \wedge (\exists x_1, x_2.(E[x'/x] \hookrightarrow x_1, x_2) \wedge x = x_i)\rrbracket_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}$$

$$= \quad \{s,h \mid \exists v \in Val.$$
$$[s \mid x \mapsto v], h \in \llbracket P[x'/x] \wedge (\exists x_1, x_2.(E[x'/x] \hookrightarrow x_1, x_2) \wedge x = x_i)\rrbracket_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\}$$

$$= \quad \{s,h \mid \exists v \in Val.$$
$$[s \mid x \mapsto v], h \in \llbracket P[x'/x]\rrbracket_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]} \text{ and}$$
$$[s \mid x \mapsto v], h \in \llbracket \exists x_1, x_2.(E[x'/x] \hookrightarrow x_1, x_2) \wedge x = x_i\rrbracket)\}$$

$$= \quad \{s,h \mid \exists v \in Val.$$
$$[s \mid x \mapsto v], h \in \bigcup_{i\in\mathbb{Z}} \llbracket P[x'/x]\rrbracket_{\rho[X_i/X]} \text{ and}$$
$$[s \mid x \mapsto v], h \in \llbracket \exists x_1, x_2.(E[x'/x] \hookrightarrow x_1, x_2) \wedge x = x_i\rrbracket)\} \quad \text{(lem. 2.39 + hyp.)}$$

$$= \quad \{s,h \mid \exists v \in Val.$$
$$[s \mid x \mapsto v], h \in \bigcup_{i\in\mathbb{Z}} \llbracket P[x'/x] \wedge (\exists x_1, x_2.(E[x'/x] \hookrightarrow x_1, x_2) \wedge x = x_i)\rrbracket_{\rho[X_i/X]}\}$$

$$= \quad \bigcup_{i\in\mathbb{Z}} \{s,h \mid \exists v \in Val.$$
$$[s \mid x \mapsto v], h \in \llbracket P[x'/x] \wedge (\exists x_1, x_2.(E[x'/x] \hookrightarrow x_1, x_2) \wedge x = x_i)\rrbracket_{\rho[X_i/X]}\}$$

$$= \quad \bigcup_{i\in\mathbb{Z}} \llbracket \exists x'.\ P[x'/x] \wedge (\exists x_1, x_2.(E[x'/x] \hookrightarrow x_1, x_2) \wedge x = x_i)\rrbracket_{\rho[X_i/X]}$$

$$= \quad \bigcup_{i\in\mathbb{Z}} (\lambda X.\llbracket sp(P,C)\rrbracket_\rho(X_i))$$

- Case C is $E.1 := E'$ then

$$\lambda X.[\![sp(P,C)]\!]_\rho(\bigcup_{i\in\mathbb{Z}} X_i)$$

$$= \quad [\![\exists x_1 \exists x_2.(E \mapsto E', x_2) * ((E \mapsto x_1, x_2){-\!\!*}P)]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}$$

$$= \quad \{s,h \mid \exists v_1, v_2 \in Val.[s \mid x_i \mapsto v_i], h \in [\![E \mapsto E', x_2]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]} * ((E \mapsto x_1, x_2){-\!\!*}P))\}$$

$$= \quad \{s,h \mid \exists v_1, v_2 \in Val.\exists h_0, h_1 \; h_0 \sharp h_1, \; h = h_0.h_1 \text{ and}$$
$$[s \mid x_i \mapsto v_i], h_0 \in [\![E \mapsto E', x_2]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]} \text{ and}$$
$$[s \mid x_i \mapsto v_i], h_1 \in [\![(E \mapsto x_1, x_2){-\!\!*}P]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\}$$

$$= \quad \{s,h \mid \exists v_1, v_2 \in Val.\exists h_0, h_1. \; h_0 \sharp h_1, \; h = h_0.h_1 \text{ and}$$
$$[s \mid x_i \mapsto v_i], h_0 \in [\![E \mapsto E', x_2]\!] \text{ and}$$
$$\forall h'_0. \begin{array}{l} \text{if } h_1 \sharp h'_0 \text{ and } [s \mid x_i \mapsto v_i], h'_0 \in [\![E \mapsto x_1, x_2]\!] \\ \text{then}[s \mid x_i \mapsto v_i], h'_0.h_1 \in [\![P]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]} \end{array}\}$$

$$= \quad \{s,h \mid \exists v_1, v_2 \in Val.\exists h_0, h_1. \; h_0 \sharp h_1, \; h = h_0.h_1 \text{ and}$$
$$[s \mid x_i \mapsto v_i], h_0 \in [\![E \mapsto E', x_2]\!] \text{ and}$$
$$\forall h'_0. \begin{array}{l} \text{if } h_1 \sharp h'_0 \text{ and } [s \mid x_i \mapsto v_i], h'_0 \in [\![E \mapsto x_1, x_2]\!] \\ \text{then}[s \mid x_i \mapsto v_i], h'_0.h_1 \in \bigcup_{i\in\mathbb{Z}} [\![P]\!]_{\rho[X_i/X]} \end{array}\}$$

here we have a $\forall h'_o$ but for the if branch to be satifyied there is only one h'_0 that could work: $[[\![E]\!]s \mapsto v_1, v_2]$ so we can delete it and we also know that this is disjoint from h_1 from the previous conditions.

$$= \quad \{s,h \mid \exists v_1, v_2 \in Val.\exists h_0, h_1. \; h_0 \sharp h_1, \; h = h_0.h_1 \text{ and}$$
$$[s \mid x_i \mapsto v_i], h_0 \in [\![E \mapsto E', x_2]\!] \text{ and}$$
$$\text{if } [\![E]\!]s \text{ exists}$$
$$\text{then}[s \mid x_i \mapsto v_i], [[\![E]\!]s \mapsto v_1, v_2].h_1 \in \bigcup_{i\in\mathbb{Z}} [\![P]\!]_{\rho[X_i/X]}\}$$

$$= \quad \bigcup_{i\in\mathbb{Z}} \{s,h \mid \exists v_1, v_2 \in Val.\exists h_0, h_1. \; h_0 \sharp h_1, \; h = h_0.h_1 \text{ and}$$
$$[s \mid x_i \mapsto v_i], h_0 \in [\![E \mapsto E', x_2]\!] \text{ and}$$
$$\text{if } [\![E]\!]s \text{ exists}$$
$$\text{then}[s \mid x_i \mapsto v_i], [[\![E]\!]s \mapsto v_1, v_2].h_1 \in [\![P]\!]_{\rho[X_i/X]}\}$$

$$= \quad \bigcup_{i\in\mathbb{Z}} \{s,h \mid \exists v_1, v_2 \in Val.\exists h_0, h_1. \; h_0 \sharp h_1, \; h = h_0.h_1 \text{ and}$$
$$[s \mid x_i \mapsto v_i], h_0 \in [\![E \mapsto E', x_2]\!] \text{ and}$$
$$\forall h'_0. \begin{array}{l} \text{if } h_1 \sharp h'_0 \text{ and } [s \mid x_i \mapsto v_i], h'_0 \in [\![E \mapsto x_1, x_2]\!] \\ \text{then}[s \mid x_i \mapsto v_i], h'_0.h_1 \in [\![P]\!]_{\rho[X_i/X]} \end{array}\}$$

$$= \quad \bigcup_{i\in\mathbb{Z}} [\![\exists x_1 \exists x_2.(E \mapsto E', x_2) * ((E \mapsto x_1, x_2){-\!\!*}P)]\!]_{\rho[X_i/X]}$$

$$= \quad \bigcup_{i\in\mathbb{Z}} (\lambda X.[\![sp(P,C)]\!]_\rho(X_i))$$

- Case C is $E.2 := E'$ then

  almost the same as the previous one...

- Case C is $x := \texttt{cons}(E_1, E_2)$ then

$$
\begin{aligned}
&\lambda X.[\![sp(P,C)]\!]_\rho(\bigcup_{i\in\mathbb{Z}} X_i)\\
=\ &[\![\exists x'.(P[x'/x] * (x \mapsto E_1[x'/x], E_2[x'/x]))]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\\
=\ &\{s,h \mid \exists v'_x \in Val.[s \mid x' \mapsto v'_x], h \in [\![(P[x'/x] * (x \mapsto E_1[x'/x], E_2[x'/x]))]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\}\\
=\ &\{s,h \mid \exists v'_x \in Val.\exists h_0, h_1.\ h_0 \sharp h_1,\ h = h_0.h_1 \text{ and}\\
&[s \mid x' \mapsto v'_x], h \in [\![P[x'/x]]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]} \text{ and}\\
&[\![x \mapsto E_1[x'/x], E_2[x'/x]]\!]\}\\
=\ &\{s,h \mid \exists v'_x \in Val.\exists h_0, h_1.\ h_0 \sharp h_1,\ h = h_0.h_1 \text{ and}\\
&[s \mid x' \mapsto v'_x], h \in \bigcup_{i\in\mathbb{Z}}[\![P[x'/x]]\!]_{\rho[X_i/X]} \text{ and}\\
&[\![x \mapsto E_1[x'/x], E_2[x'/x]]\!]\} \text{ (lem. 2.39+hyp.)}\\
=\ &\bigcup_{i\in\mathbb{Z}}\{s,h \mid \exists v'_x \in Val.\exists h_0, h_1.\ h_0 \sharp h_1,\ h = h_0.h_1 \text{ and}\\
&[s \mid x' \mapsto v'_x], h \in [\![P[x'/x]]\!]_{\rho[X_i/X]} \text{ and}\\
&[\![x \mapsto E_1[x'/x], E_2[x'/x]]\!]\}\\
=\ &\bigcup_{i\in\mathbb{Z}}[\![\exists x'.(P[x'/x] * (x \mapsto E_1[x'/x], E_2[x'/x]))]\!]_{\rho[X_i/X]}\\
=\ &\bigcup_{i\in\mathbb{Z}}(\lambda X.[\![sp(P,C)]\!]_\rho(X_i))
\end{aligned}
$$

- Case C is $\texttt{dispose}(E)$ then

$$\lambda X.[\![sp(P,C)]\!]_\rho(\bigcup_{i\in\mathbb{Z}} X_i)$$

$$= \ [\![\exists x_1,x_2.\,((E\mapsto x_1,x_2)\!\rightarrow\!\!*P)]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}$$

$$= \ \{s,h\mid\exists v_1,v_2.[s\mid x_i\mapsto v_i],h\in[\![(E\mapsto x_1,x_2)\!\rightarrow\!\!*P]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\}$$

$$= \ \{s,h\mid\exists v_1,v_2.\forall h'.\ \begin{array}{l}\text{if }h\sharp h',\text{ and }[s\mid x_i\mapsto v_i],h'\in[\![E\mapsto x_1,x_2]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\\\text{then }[s\mid x_i\mapsto v_i],h.h'\in[\![P]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\end{array}\}$$

$$= \ \{s,h\mid\exists v_1,v_2.\forall h'.\ \begin{array}{l}\text{if }h\sharp h',\text{ and }[s\mid x_i\mapsto v_i],h'\in[\![E\mapsto x_1,x_2]\!]\\\text{then }[s\mid x_i\mapsto v_i],h.h'\in\bigcup_{i\in\mathbb{Z}}[\![P]\!]_{\rho[X_i/X]}\end{array}\} \qquad\text{(hyp.)}$$

$$= \ \{s,h\mid\exists v_1,v_2.\ \begin{array}{l}\text{if }[\![E]\!]s\text{ exists and }[\![E]\!]s\notin dom(h)\\\text{then }[s\mid x_i\mapsto v_i],h.[[\![E]\!]s\mapsto v_1,v_2]\in\bigcup_{i\in\mathbb{Z}}[\![P]\!]_{\rho[X_i/X]}\end{array}\}$$

(same operation as in the case of $E.i:=E'$)

$$= \ \bigcup_{i\in\mathbb{Z}}\{s,h\mid\exists v_1,v_2.\ \begin{array}{l}\text{if }[\![E]\!]s\text{ exists and }[\![E]\!]s\notin dom(h)\\\text{then }[s\mid x_i\mapsto v_i],h.[[\![E]\!]s\mapsto v_1,v_2]\in[\![P]\!]_{\rho[X_i/X]}\end{array}\}$$

$$= \ \bigcup_{i\in\mathbb{Z}}[\![\exists x_1,x_2.\,((E\mapsto x_1,x_2)\!\rightarrow\!\!*P)]\!]_{\rho[X_i/X]}$$

$$= \ \bigcup_{i\in\mathbb{Z}}(\lambda X.[\![sp(P,C)]\!]_\rho(X_i))$$

- Case C is $C_1;C_2$ then:

$$sp(P,C)=sp(sp(P,C_1),C_2)$$

$\lambda X.[\![sp(P,C_1)]\!]_\rho$ is upper-continuous by induction hypothesis on C_1 for P and $\lambda sp(P,C)$ is upper-continuous by induction hypothesis on C_2 for $sp(P,C_1)$.

- Case C is $if\ \ E\ \ then\ \ C_1\ \ else\ \ C_2$ then

$$\lambda X.[\![sp(P,C)]\!]_\rho(\bigcup_{i\in\mathbb{Z}} X_i)$$

$$= \ [\![sp(P\wedge E=\texttt{true},C_1)\vee sp(P\wedge E=\texttt{false},C_2)]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}$$

$$= \ [\![sp(P\wedge E=\texttt{true},C_1)]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}\cup[\![sp(P\wedge E=\texttt{false},C_2)]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}$$

$$= \ (\bigcup_{i\in\mathbb{Z}}[\![sp(P\wedge E=\texttt{true},C_1)]\!]_{\rho[X_i/X]})\cup(\bigcup_{i\in\mathbb{Z}}[\![sp(P\wedge E=\texttt{false},C_2)]\!]_{\rho[X_i/X]})$$
(by ind. hyp)

$$= \ \bigcup_{i\in\mathbb{Z}}(([\![sp(P\wedge E=\texttt{true},C_1)]\!]_{\rho[X_i/X]})\cup([\![sp(P\wedge E=\texttt{false},C_2)]\!]_{\rho[X_i/X]}))$$

$$= \ \bigcup_{i\in\mathbb{Z}}([\![sp(P\wedge E=\texttt{true},C_1)\vee sp(P\wedge E=\texttt{false},C_2)]\!]_{\rho[X_i/X]})$$

$$= \ \bigcup_{i\in\mathbb{Z}}([\![sp(P,C)]\!]_{\rho[X_i/X]})$$

- Case C is `skip` then

$sp(P, C) = P$ so $\lambda X.\llbracket sp(P, C) \rrbracket_\rho$ is upper-continuous by hypothesis.

- Case C is $while\ \ E\ \ do\ \ C_1$ then

$$
\begin{aligned}
&\lambda X.\llbracket sp(P, C) \rrbracket_\rho (\bigcup_{i \in \mathbb{Z}} X_i)\\
=\ &\llbracket (\mu Y_v.sp(Y_v \wedge E = \texttt{true}, C_1) \vee P) \wedge\ (E = \texttt{false}) \rrbracket_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]}\\
&\text{(def. of sp. } Y \not\in \rho[\bigcup_{i \in \mathbb{Z}} X_i/X] + Y_v \not\in P)\\
=\ &(\llbracket (\mu Y_v.sp(Y_v \wedge E = \texttt{true}, C_1) \vee P) \rrbracket_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]}) \cap \llbracket E = \texttt{false} \rrbracket\\
&\text{(since } Y_v \text{ not in } E)\\
=\ &\text{lfp}^{\subseteq}_{\emptyset} \lambda Y.\ \llbracket sp(Y_v \wedge E = \texttt{true}, C_1) \vee P \rrbracket_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]|Y_v \mapsto Y]}\\
&\cap \llbracket E = \texttt{false} \rrbracket\\
&\text{(def of } \mu)\\
=\ &\text{lfp}^{\subseteq}_{\emptyset} (\lambda Y.\ \begin{aligned}[t]&\llbracket sp(Y_v \wedge E = \texttt{true}, C_1) \rrbracket_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]|Y_v \mapsto Y]}\\ &\cup \llbracket P \rrbracket_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]|Y_v \mapsto Y]}\end{aligned})\\
&\cap \llbracket E = \texttt{false} \rrbracket\\
=\ &\text{lfp}^{\subseteq}_{\emptyset} (\lambda Y.\ \begin{aligned}[t]&\llbracket sp(Y_v \wedge E = \texttt{true}, C_1) \rrbracket_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]|Y_v \mapsto Y]}\\ &\cup \llbracket P \rrbracket_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]}\end{aligned})\\
&\cap \llbracket E = \texttt{false} \rrbracket\\
&\text{(since } Y_v \not\in P)
\end{aligned}
$$

let $G = \lambda X.(\lambda Y.\ \llbracket sp(Y_v \wedge E = \texttt{true}, C_1) \rrbracket_{\rho|Y_v \mapsto Y]} \cup \llbracket P \rrbracket_\rho)$

we have $G(\bigcup_{i \in \mathbb{Z}} X_i) = (\lambda Y.\ \llbracket sp(Y_v \wedge E = \texttt{true}, C_1) \rrbracket_{\rho|Y_v \mapsto Y][\bigcup_{i \in \mathbb{Z}} X_i/X]} \cup \llbracket P \rrbracket_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]})$

and by chose of Y we can rewrite it as:

$G(\bigcup_{i \in \mathbb{Z}} X_i) = (\lambda Y.\ \llbracket sp(Y_v \wedge E = \texttt{true}, C_1) \rrbracket_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]|Y_v \mapsto Y]} \cup \llbracket P \rrbracket_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]})$

and $G(X_i) = (\lambda Y.\ \llbracket sp(Y_v \wedge E = \texttt{true}, C_1) \rrbracket_{\rho[X_i/X]|Y_v \mapsto Y]} \cup \llbracket P \rrbracket_{\rho[X_i/X]})$

we can rewrite the former formula as:

$$
\begin{aligned}
&= \; (\mathrm{lfp}_\emptyset^\subseteq (G(\bigcup_{i\in\mathbb{Z}} X_i))) \cap [\![E = \texttt{false}]\!] \\
&= \; (\bigcup_{n\in\mathbb{Z}} (G(\bigcup_{i\in\mathbb{Z}} X_i))^n(\emptyset)) \cap [\![E = \texttt{false}]\!] \\
&\quad (G(\bigcup_{i\in\mathbb{Z}} X_i) \text{ u-c on } Y + \text{Tarski}) \\
&= \; (\bigcup_{n\in\mathbb{Z}} (\bigcup_{i\in\mathbb{Z}} G(X_i))^n(\emptyset)) \cap [\![E = \texttt{false}]\!] \\
&\quad (G \text{ u-c on } X) \\
&= \; (\bigcup_{i\in\mathbb{Z}} \bigcup_{n\in\mathbb{Z}} ((G(X_i))^n(\emptyset)) \cap [\![E = \texttt{false}]\!] \\
&= \; \bigcup_{i\in\mathbb{Z}} (\mathrm{lfp}_\emptyset^\subseteq G(X_i)) \cap [\![E = \texttt{false}]\!] \\
&\quad (G(X_i) \text{ u-c on } Y + \text{Tarski}) \\
&= \; \bigcup_{i\in\mathbb{Z}} (\mathrm{lfp}_\emptyset^\subseteq \lambda Y. \, [\![sp(Y_v \wedge E = \texttt{true}, C_1) \vee P]\!]_{[\rho[X_i/X]|Y_v\mapsto Y]}) \cap [\![E = \texttt{false}]\!] \\
&= \; \bigcup_{i\in\mathbb{Z}} ([\![\mu Y_v . sp(Y_v \wedge E = \texttt{true}, C_1) \vee P]\!]_{\rho[X_i/X]}) \cap [\![E = \texttt{false}]\!] \\
&\quad \text{def of } \mu) \\
&= \; \bigcup_{i\in\mathbb{Z}} ([\![(\mu Y_v . sp(Y_v \wedge E = \texttt{true}, C_1) \vee P) \wedge (E = \texttt{false})]\!]_{\rho[X_i/X]}) \\
&= \; \bigcup_{i\in\mathbb{Z}} ([\![sp(P,C)]\!]_{\rho[X_i/X]})
\end{aligned}
$$

Complement:

- G is upper-continuous on X:

  recall $G = \lambda X.(\lambda Y. \, [\![sp(Y_v \wedge E = \texttt{true}, C_1)]\!]_{[\rho|Y_v\mapsto Y]} \cup [\![P]\!]_\rho)$

  by hypothesis: $\lambda X.[\![P]\!]_\rho$ is upper-continuous,

  by chose of Y not in ρ we have that $[\![sp(Y_v \wedge E = \texttt{true}, C_1)]\!]_{[\rho|Y_v\mapsto Y]}$ does not depend on X so $\lambda X.[\![sp(Y_v \wedge E = \texttt{true}, C_1)]\!]_{[\rho|Y_v\mapsto Y]}$ is upper-continuous on X and so G is upper-continuous on X.

- $G(\bigcup_{i\in\mathbb{Z}} X_i)$ upper-continuous on Y:

  recall: $G(\bigcup_{i\in\mathbb{Z}} X_i) = (\lambda Y. \, [\![sp(Y_v \wedge E = \texttt{true}, C_1)]\!]_{[\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]|Y_v\mapsto Y]} \cup [\![P]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]})$

  by Lemma 2.41 we have that $\lambda Y.[\![Y_v \wedge E = \texttt{true}]\!]_{[\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]|Y_v\mapsto Y]}$ is upper-continuous

  and by ind. hyp. we have that this current theorem holds for any subprogram so it hold for C_1 and so $\lambda Y.[\![sp(Y_v \wedge E = \texttt{true}, C_1)]\!]_{[\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]|Y_v\mapsto Y]}$ is upper-continuous

  by chose of Y $[\![P]\!]_{\rho[\bigcup_{i\in\mathbb{Z}} X_i/X]}$ does not depend on Y

so $(\lambda Y. [\![sp(Y_v \wedge E = \mathtt{true}, C_1)]\!]_{[\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]|Y_v \mapsto Y]} \cup [\![P]\!]_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]})$ is upper-continuous

- $G(X_i)$ upper-continuous on Y: Same proof as for $G(\bigcup_{i \in \mathbb{Z}} X_i)$

$\square$

2.6.12 Simplification theorems

Definition 2.43. $\lambda X. [\![P]\!]_\rho \in \mathcal{P}\ (S \times H) \longmapsto \mathcal{P}\ (S \times H)$ *is* upper-continuous *iff*

For any $\subseteq$-increasing chain x_i, $i \in \mathbb{Z}$:

$[\![P]\!]_{\rho[X_i/X]}$ *exists and:*

$$[\![P]\!]_{\rho[\bigcup_{i \in \mathbb{Z}} X_i/X]} = \bigcup_{i \in \mathbb{Z}} [\![P]\!]_{\rho[X_i/X]}$$

Theorem 2.44.

> *If $\lambda X. [\![P]\!]_{[X_v \mapsto X]}$ is upper-continuous and X_v is the only $V\_variable$ which can be free in P*
>
> $$s, h \models \mu X_v\ .\ P \quad iff \quad \exists n.\ s, h \models F_P^n(\mathtt{false})$$
>
> *with $F_P^0(Q) = Q$, $F_P^{n+1}(Q) = P\{F_P^n(Q)/X_v\}$*

Theorem 2.44. If $\lambda X. [\![P]\!]_{[X_v \mapsto X]}$ is upper-continuous and X_v is the only variable which can be v-free in P then $[\![\mu X_v.P]\!]$ exists and $[\![\mu X_v.P]\!] = \mathrm{lfp}_\emptyset^\subseteq \lambda X. [\![P]\!]_{[X_v \mapsto X]}$

by Tarski constructive theorem: $[\![\mu X_v.P]\!] = \bigcup_{n \in \mathbb{Z}} (\lambda X. [\![P]\!]_{[X_v \mapsto X]})^n(\emptyset)$

so $s, h \models \mu X_v.P$ iff $\exists n.s, h \in (\lambda X. [\![P]\!]_{[X_v \mapsto X]})^n(\emptyset)$

We still need to prove that $(\lambda X. [\![P]\!]_{[X_v \mapsto X]})^n(\emptyset) = [\![F_P^n(\mathtt{false})]\!]$.

We prove it by recurrence on n:

- Case $n = 0$:
  $$(\lambda X. [\![P]\!]_{[X_v \mapsto X]})^0(\emptyset)$$
  $$= \emptyset$$
  $$= [\![\mathtt{false}]\!]$$
  $$= [\![F_P^0(\mathtt{false})]\!]$$

- Case $n + 1$

$$(\lambda X.\llbracket P \rrbracket^{n+1}_{[X_v \mapsto X]})(\emptyset)$$

$$= (\lambda X.\llbracket P \rrbracket_{[X_v \mapsto X]})((\lambda X.\llbracket P \rrbracket_{[X_v \mapsto X]})^n \emptyset)$$

$$= (\lambda X.\llbracket P \rrbracket_{[X_v \mapsto X]})(\llbracket F_P^n(\mathtt{false}) \rrbracket) \qquad \text{(ind. hyp)}$$

$$= \llbracket P \rrbracket_{[X_v \mapsto \llbracket F_P^n(\mathtt{false}) \rrbracket]}$$

$$= \llbracket P\{F_P^n(\mathtt{false})/X_v\} \rrbracket \qquad \text{(by Th. 2.30)}$$

$$= \llbracket F_P^{n+1}(\mathtt{false}) \rrbracket$$

(To use Theorem 2.30 we needed that $\llbracket F_P^n(\mathtt{false}) \rrbracket$ exists but we have it by recursion since $\llbracket \mathtt{false} \rrbracket$ exists and since $\lambda X.\llbracket P \rrbracket_{[X_v \mapsto X]}$ is upper-continuous we have $\forall X_i.\llbracket P \rrbracket_{[X_v \mapsto X_i]}$ exist.)

$\square$

Chapter 3

An abstract language for separation logic

3.1 Introduction

In Chapter 2, we extended separation logic with fixpoints, letting us express finitely separation-logic characterizations of wlp and sp for `while`-loop programs[14].

Those pre- and post-conditions allowed us to characterise a program, but not really to check errors since there is no theorem prover (not just because of fixpoints, but mostly because of quantifiers). Moreover, they created formulae that are not easy to read for a human, which can be expected since they do not make approximations. But still, separation logic seemed really suitable for a human to characterise a memory state, or sets of states. Thus, we decided to write a language, which primarily would be used as an intermediate language for translating separation logic into some other analysis domain, shape and alias ones (among other things, the language contain the usual notion of *"summary nodes"*). The language could also be used for doing directly some analysis.

Lying at the heart of the abstract language is its semantic domain of denotations. The principal caracteristic and the main point that guided the design is that the domain is a tuple, and we have semantics that is an intersection of each component and in particular the graph's semantics which is an intersection of each arrow.

In this chapter, we introduce the abstract language, giving the latter in both linear

84

(Sect. 3.3) and graphical forms. Our graphical representation resembles the graph formats found in pointer analysis and shape analysis[4,5], which are also concerned with describing properties of aliasing and heap data structures, respectively.

To keep our abstract separation-logic language as general as possible, we *parameterize* it on a numerical abstract domain, which can be instantiated with existing numerical domains, including relational ones. Within our language, we treat numerical information in the same way as memory information. We provide a semantics of our language directly in term of sets of memory (Sect. 3.4).

As part of our language, we present some functions on the language in Sect. 3.5, among with stabilization and union operators, whose precision and cost can be tuned to the specific needs of the context where the language is used. We present the translations from separation logic to the language as examples in Sect. 3.2 and more formally in Sect. 3.6. The achievements of the chapter are:

   i we define an abstract language for separation logic that is amenable to static analyis;

   ii we define operations on the language;

   iii we give a rigorous semantics and use it to prove soundness of the operations on the language;

   iiii we provide translation function for the fixpoints separation logic formulae

The domain uses a set of auxiliary variables to encode aliasing and allows to write a cheap translation of the $\wedge$ connective which does not require to check for aliasing. This formal semantics given to auxiliary variables allows to prove operations which would usually be pushed as implementation's correctness problem.

In the chapter, for a function f, we will use the notation $[f \mid x \to V]$ for the function that maps x to V and is equal to f for any element in the domain of f different from x.

We write _ when the element of the tuple is not pertinent for the example.

	formula or set states represented	abstract representation
(1)	$(x = \texttt{nil})$	$\left(\boxed{x} \longrightarrow \text{Nilt} \;,\, \text{-}, \text{-}, \text{-}, \text{-}, \text{-}, \text{-} \right)$
(2)	$\neg(x = x)$ $\{s, h \mid x \notin dom(s)\}$ see some explainations below	$\left(\boxed{x} \longrightarrow \text{Oodt} \;,\, \text{-}, \text{-}, \text{-}, \text{-}, \text{-}, \text{-} \right)$
(3)	$(x = \texttt{nil} \vee x = \texttt{true})$	$\left(\boxed{x} \longrightarrow \begin{array}{c} \text{Nilt} \\ \text{Truet} \end{array} \;,\, \text{-}, \text{-}, \text{-}, \text{-}, \text{-}, \text{-} \right)$
(4)	$(x = y)$	$\left(\begin{array}{c} \boxed{x} \\ \boxed{y} \end{array} \longrightarrow \boxed{\alpha} \longrightarrow \top \;,\, \text{-}, \text{-}, \text{-}, \text{-}, \text{-}, \text{-} \right)$
(5)	$(x = y \wedge x = \texttt{nil})$	$\left(\begin{array}{c} \boxed{x} \\ \boxed{y} \end{array} \longrightarrow \boxed{\alpha} \longrightarrow \text{Nilt} \;,\, \text{-}, \text{-}, \text{-}, \text{-}, \text{-}, \text{-} \right)$
(6)	$(\exists x.\, x = y \wedge x = \texttt{nil}) \equiv (y = \texttt{nil})$	$\left(\boxed{y} \longrightarrow \boxed{\alpha} \longrightarrow \text{Nilt} \;,\, \text{-}, \text{-}, \text{-}, \text{-}, \text{-}, \text{-} \right)$
(7)	$(x < y + 3)$	$\left(\begin{array}{cc} \boxed{x} \longrightarrow \boxed{\alpha} \longrightarrow & \text{Numt} \\ \boxed{y} \longrightarrow \boxed{\beta} \longrightarrow & \text{Numt} \end{array} \;,\, \text{-}, \text{-}, \text{-}, \text{-}, \text{-}, d \right)$ $d \in \mathcal{D}$ encodes that $\alpha < \beta + 3$

We have that $\neg(x = x) \not\equiv \texttt{false}$ because the model of the formulae uses partial functions for representing the stack. The semantics of $\texttt{false}$ is $\emptyset$, while the semantics of $\neg(x = x)$ is all the states where x is not in the domain of the stack, and there exist many of them.

Figure 3.1: *Introduction examples*

3.2 Examples: Introduction to the language, translations of formulae

In all our examples, we use $x, y, \ldots \in Var$ for program variables, and we use $\alpha, \beta, \ldots \in TVar$ for auxiliary variables that denote values of our abstract language. We use $v, v_1, v_2, \ldots$ for variables in $Var \cup TVar$.

The language we define is a set of tuples. The formal definitions of AR, PVD^+, CL_{eq} are given later in Fig. 3.3.

	formula or set states represented	abstract representation
(8)	"x is a location not allocated" $\{s,h \mid s(x) \in Loc \wedge s(x) \notin dom(h)\}$	$\left(\boxed{x} \longrightarrow Dangling\_Loc \ , \text{-}, \text{-}, \text{-}, \text{-}, \text{-}, \text{-} \right)$
(9)	emp $\{s,h \mid dom(h) = \emptyset\}$	$(\text{-}, \text{-}, \emptyset, \text{-}, \text{-}, \text{-}, \text{-})$
(10)	$(x \mapsto \texttt{true}, \texttt{nil})$ $\{s,h \mid [s(x) \to \langle True, \texttt{nil}\rangle] = h\}$	$\left(\boxed{x} \to \boxed{\alpha} \to \bullet \rightsquigarrow^1 Truet \ \rightsquigarrow_2 Nilt \ , \{\alpha\}, \{\alpha\}, \text{-}, \text{-}, \text{-}, \text{-} \right)$
(11)	$(x \hookrightarrow \texttt{true}, \texttt{nil})$ $\{s,h \mid [s(x) \to \langle True, \texttt{nil}\rangle] \subseteq h\}$	$\left(\boxed{x} \to \boxed{\alpha} \to \bullet \rightsquigarrow^1 Truet \ \rightsquigarrow_2 Nilt \ , \{\alpha\}, \texttt{full}, \text{-}, \text{-}, \text{-}, \text{-} \right)$
(12)	approx. of $(x = \texttt{true} \wedge y = \texttt{false})$ $\begin{pmatrix} x = \texttt{true} \\ \vee \\ x = \texttt{false} \end{pmatrix} \wedge \begin{pmatrix} y = \texttt{false} \\ \vee \\ y = \texttt{true} \end{pmatrix}$	$\boxed{x}, \boxed{y} \to \boxed{\boxed{\alpha}} \to Truet, Falset$
(13)	there is an finite acyclic list of $True$ starting from x $\mu X_v. \left(\begin{array}{c} (x = \texttt{nil}) \vee \exists x_2. \\ x \hookrightarrow (\texttt{true}, x_2) \ * \ X_v[x_2/x] \end{array} \right)$	$\left(\boxed{x} \to \boxed{\boxed{\alpha}} \rightsquigarrow^{*2} \circledbullet \rightsquigarrow^1 Truet, \ Nilt \ , \text{-}, \text{-}, \{\alpha\}, \text{-}, \text{-}, \text{-} \right)$
(14)	$\begin{pmatrix} x = \texttt{nil} \\ \wedge y = \texttt{true} \end{pmatrix} \vee \begin{pmatrix} x = \texttt{true} \\ \wedge y = \texttt{nil} \end{pmatrix}$	$\boxed{x} \to \alpha_1 \to Nilt, \ \alpha_2 \to Truet \ ; \ \boxed{y} \to \alpha_3 \to Nilt, \ \alpha_4 \to Truet \ ; \ \{\dagger_{eq}\}$
(15)	x and y point to the same acyclic list of $True$ but x comes first details below	$\boxed{x}, \boxed{y} \to \boxed{\boxed{\alpha}} \rightsquigarrow^{*2} \circledbullet \rightsquigarrow^1 Truet, \ Nilt \ ; \ \{\subset_{eq}\}$
(16)	$((x \mapsto \texttt{true}, \texttt{nil}) * (y \mapsto \texttt{true}, \texttt{nil}))$	$\boxed{x} \to \boxed{\alpha} \to \circledbullet \rightsquigarrow^1 Truet \rightsquigarrow_2 Nilt \ ; \ \boxed{y} \to \boxed{\beta} \to \circledbullet \rightsquigarrow^1 Truet \rightsquigarrow_2 Nilt \ ; \ \{\sharp_{eq}\}$

Details for example (15)

$$(x \neq y) \wedge \left(\mu Y_v. \left(\begin{array}{c} (y = \texttt{nil}) \vee \exists y_2. \\ y \hookrightarrow (\texttt{true}, y_2) \ * \ Y_v[y_2/y] \end{array} \right) * \mu X_v. \left(\begin{array}{c} (x = y) \vee \exists x_2. \\ x \hookrightarrow (\texttt{true}, x_2) \ * \ X_v[x_2/x] \end{array} \right) \right)$$

Figure 3.2: *Introduction examples*

Let $(ad, hu, ho, sn, sn^\infty, t, d)$ be an element of our language, which we name AR:

- $ad : VAR \overset{total}{\rightharpoonup} PVD^+$ is a function that maps variables (and auxiliary variables) to abstract denotations (represented as graphs)

- $hu \subseteq TVar$ is a set of auxiliary variables which represent an underapproximation of the locations in the heap

- $ho \subseteq TVar$ or $ho = \texttt{full}$ is a set of auxiliary variables which represent an overapproximation of the locations in the heap; it can also take the value $\texttt{full}$. We note this as $ho \in (\mathcal{P}(TVar) \uplus \texttt{full})$ having $\uplus$ for disjoint union

- $sn \subseteq TVar$ is a set of auxiliary variables which can represent a finite set of concrete values

- $sn^\infty \subseteq TVar$ is a set of auxiliary variables which can represent an infinite set of concrete values

- $t : (TVar \times TVar) \overset{total}{\rightharpoonup} CL_{eq}$ is a table which compares the values represented of two auxiliary variables.

- $d : \mathcal{D}$ contains all the numerical information, $\mathcal{D}$ being a numerical domain. If x is a numerical, then in AD it is made to point to some auxiliary which itself will point to $\{Numt\}$, and if there is more information about this numerical value, we will register it for that auxiliary variable in d.

In Fig. 3.1 and 3.2, we present examples of formulae or sets of memory states and their translations. The complete description of separation logic and its semantics are given in the previous chapter, Ch. 2.

Concrete domain Let Loc be an infinite set of heap locations, let $Val \triangleq \mathbb{Z} \uplus Bool \uplus \{\texttt{nil}\} \uplus Loc$ be the set of storable values, let $S \triangleq Var \rightharpoonup Val$ ($\rightharpoonup$ stands for partial function) be the set of temporary-variable stacks, and let $H \triangleq Loc \rightharpoonup (Val \times Val)$ be the set of heaps (partial functions that map locations to cons cells).

We define $M \triangleq S \times H$. The standard model for the logic, which is also the domain for the concretisation of our abstract language is $\mathcal{P}(M)$. Given some $s, h \in M$, the denotation of a variable x, is $s(x)$; when $s(x) \in Loc$, its dereferencing is $h(s(x))$. Since separation logic is oriented towards properties of heap structures, such two-step dereferencing dominates one's reasoning about variables, and we make it a key feature of our abstract language.

Simple abstract values One might use separation logic to assert that the value of a variable in the stack is `nil`, thus we have an abstract value $Nilt$ (see Ex. 1). As well, we translate `true` by $Truet$, `false` by $Falset$. A variable can be out of the domain of the stack which corresponds to the abstract value $Oodt$ (see Ex. 2).

The abstract language can assign to variables a set of abstract values, which can arise from a disjunction (see Ex. 3).

Auxiliary variables We use an infinite set of auxiliary variables $TVar$ to encode aliasing (see Ex. 4). The use of $TVar$ permits a cheap translation of conjunction, that is, to translate $P \wedge Q$ we first translate P then we refine the result while translating Q (see Ex. 4 and 5) and it also permits a cheap translation of quantifiers (see Ex. 5 and 6).

Auxiliary variables can be used as wanted, for example, $\boxed{x} \longrightarrow \big(\!Nilt\!\big)$ can also be represented by $\boxed{x} \longrightarrow \boxed{\alpha} \longrightarrow \big(\!Nilt\!\big)$. But for efficiency reasons, the abstract language has some constraints, for example, we forbid $\boxed{x} \longrightarrow \boxed{\alpha} \longrightarrow \big(\!\emptyset\!\big)$, which must be represented as $\boxed{x} \longrightarrow \big(\!\emptyset\!\big)$.

Numerical information To allow translation of numerical values and numerical relations, we parameterize our language on a numerical domain, $\mathcal{D}$. The graph will contain only the abstract value $Numt$, and all the numerical information are encoded in the element of the numerical domain. For example, in Ex. 7, d could be a "difference bound matrix" [19], and we would have $d(\alpha, \beta) = i$ with any $i \geq 2$.

Heap In the logic, one can say that a variable is a dangling pointer, thus we have another abstract value, $Dangling\_Loc$, to represent this information (see Ex. 8). To model pointers and the heap, the second and third component of our tuples are *(i) hu*, a set of auxiliary

variables that underapproximate the set of locations allocated in the heap, and *(ii) ho*, a similar overapproximation (*ho* could also take the value, `full`, to give no information). They are used, for example, to translate the formula `emp` which says that the heap is empty (see Ex. 9). In separation logic, one writes assertions that state the exact contents of the heap. For example, the formula, $(x \mapsto \mathtt{true}, \mathtt{nil})$, not only says that x points to a cell whose *car*-value is *True* and *cdr*-value is `nil` but also says that the cons-cell pointed to by x is the *only* cell in the heap (see Ex. 10). In the translation, $\bullet$ represents a heap location, and $\rightsquigarrow^{1}$ / $\rightsquigarrow^{2}$ represents its car/cdr. The formula $(x \hookrightarrow \mathtt{true}, \mathtt{nil})$ is like $(x \mapsto \mathtt{true}, \mathtt{nil})$ except that it allows additional cells in the heap domain, thus *ho* takes the value `full` (see Ex. 11).

Summary nodes Notice that we embed shape-graphs, $\bullet$ and $\rightsquigarrow$ being their nodes and edges. To avoid infinite graphs (or to bound the computation time), our graphs have *summary nodes*, which are nodes that represent multiple concrete values. By default, all variables represent only one value. The fourth (sn) and fifth (sn^{∞}) components are sets of auxiliary variables which are allowed to represent sets of values (possibly an infinite set for sn^{∞}). Graphically, finite summary nodes will be doubly circled/squared and infinite summary nodes will be triply circled/squared. The main difference from usual shape-graphs is that we allow summary nodes for values which are not locations (see Ex. 12). We also use summary nodes to represent a list of any size (see Ex. 13). In the formula, μ is a least-fixpoint connective and [ ] is a postponed substitution connective which is used for the recursion[14]. Informally, the formula can be written as $\mathtt{nclisttrue}(x) \triangleq (x = \mathtt{nil}) \lor \exists x_2. \, x \hookrightarrow (\mathtt{true}, x_2) * \mathtt{nclisttrue}(x_2)$.

To allow infinite lists, we would replace μ by ν in the above formula, and in our language, $\odot$ would be replaced by $\circledcirc$

Tables It is obvious that representing a union of graphs by a graph with unions (as we do, see Ex. 3) implies some approximations. It can be interesting to do this approximation but we also want the language to allow additional precision if needed. In Sect. 3.5.2, we

give an explanation of how to do a precise union. The sixth component of the tuple is a 2-dimensional table for auxiliary variables which is graphically representated by annoted arrows, $\_\_\rightarrowtail$. The annotations are sets of properties on the concrete values represented by the auxiliary variables. Details are given in Sec. 3.3. An arrow labeled with $\{\dagger_{eq}\}$ means that one "exclusive or" the other variables pointed represent no value (see Ex. 14 where if α_1 has a value, then α_2 and α_3 do not, and when α_4 has a value, then α_2 and α_3 do not). An arrow labeled with $\{\subset_{eq}\}$ means that the values represented by one variable are included in the ones represented by the other (see Ex. 15).

Separation information There are two ways to encode separation information. The first way is to use $\_\_\rightarrowtail$, in particular $\_\_\overset{\{\sharp_{eq}\}}{\rightarrowtail}$ says that two variables has no values in common in their concretisations (see Ex. 16). The second way is to use $\overset{*1}{\rightsquigarrow}$ / $\overset{*2}{\rightsquigarrow}$ instead of $\overset{1}{\rightsquigarrow}$ / $\overset{2}{\rightsquigarrow}$. In Ex. 13, this insures that the list is not cyclic: α can represent several locations and they can point to one another through their cdr-values but $*2$ says that no path through those locations is cyclic.

In an automatic translation of formulae, the use of $\_\_\overset{\{\sharp_{eq}\}}{\rightarrowtail}$ would appear when we translate $P * Q$. We would translate P and Q separately with disjoint sets of fresh auxiliary variables. Then, we will join the two abstract values in some sort of union which will respect the separation information. The two sets of allocated locations reprented by the two translations should be disjointed. So, for exemple for the variables in hu_P and hu_Q, the ones definitely representing allocated locations, they should have arrows $\_\_\overset{\{\sharp_{eq}\}}{\rightarrowtail}$ which insure that they represent disjoints sets. The use of $\overset{*1}{\rightsquigarrow}$ / $\overset{*2}{\rightsquigarrow}$ would only appear while translating fixpoints and building summary nodes.

3.2.1 Full example: tree

We present here a full example where we have the graphical and non graphical representation of the domain.

We define $ad_i \in AD$ such that $\forall x \in Var.\ ad_i(x) = \top$, and $\forall \alpha \in TVar.\ ad_i(\alpha) = \otimes$.

$t_i \in TB$ such that $\forall \alpha_1, \alpha_2 \in TVar. \, t(\alpha_1, \alpha_2) = \{\ddagger_{eq}\}$,

and finaly $ar_i = (ad_i, \emptyset, \texttt{full}, TVar, TVar, t_i, \top^{\mathcal{D}})$

To say that x points to a tree, we could use the formula

$$\mu X_v. \, (x = \texttt{nil}) \vee \exists x_1, x_2. (x \hookrightarrow x_1, x_2 \, * \, X_v[x_1/x] \, * \, X_v[x_2/x])$$

or we can in our domain define it as

$$
\left(
\begin{array}{l}
\left[ad_i | x \to \{\alpha\} | \alpha \to \left\{ \begin{array}{l} Nilt, \\ Loc(\{*1, *2\}, \alpha_1, \alpha_2) \end{array} \right\} | \alpha_1 \to \{\alpha\} | \alpha_2 \to \{\alpha\} \right], \\
\emptyset, \texttt{full}, \{\alpha, \alpha_1, \alpha_2\}, \emptyset, \\
\left[t_i \left| \begin{array}{l} (\alpha, \_), (\_, \alpha), (\alpha_1, \_), (\_, \alpha_1), (\alpha_2, \_), (\_, \alpha_2) \to \{\ddagger_{eq}, \dagger_{eq}\} \\ | (\alpha, \alpha_1), (\alpha, \alpha_2), (\alpha_2, \alpha), (\alpha_1, \alpha) \to \top_{eq} \\ | (\alpha_1, \alpha_2), (\alpha_2, \alpha_1) \to \{\ddagger_{eq}, \sharp_{eq}\} \end{array} \right. \right], \top^{\mathcal{D}}
\end{array}
\right)
$$

which can be graphicaly represented as
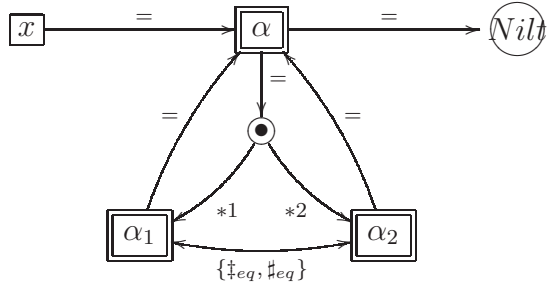
3.3   Definition of the language, AR

We now formalize the graphs from the previous section as denotations in our abstract separation-logic language.

Let Var and $TVar$ be two disjoint infinite sets of variables (typically Var will be the set of program/formula variables and $TVar$ will be auxiliary variables). We define $VAR \triangleq Var \uplus TVar$, the set of all variables. The formal definition of the syntax of the language are presented in Fig. 3.3.

$VD1$ is an abstract language for all values except locations. VD is $VD1$ plus abstract values for locations in the heap. For $Loc(A, vd1, vd2)$, A expresses some separation information, $vd1$ is the abstract car-value and $vd2$ is the abstract cdr-value. PVD^+ is either a powerset of values in VD or the undefined value, denoted $\otimes$.

$$
\begin{array}{rcl}
VD1 & ::= & Numt \mid Truet \mid Falset \mid Oodt \mid Nilt \mid Dangling\_Loc \mid TVar \\
VD & ::= & VD1 \mid Loc(\mathcal{P}(\{*1, *2\}) \times VD1 \times VD1) \\
PVD^+ & ::= & (\mathcal{P}(VD) \uplus \circledast, \sqcup, \sqcap) \\
& & \text{where } \uplus \text{ is a disjoing union and } \sqcup \text{ and } \sqcap \text{ are functions defined below} \\
AD & ::= & VAR \overset{total}{\rightarrow} PVD^+ \\
CL_{eq} & ::= & \mathcal{P}(\{\ddagger_{eq}, \dagger_{eq}, =_{eq}, \subset_{eq}, \supset_{eq}, \sharp_{eq}, \textcircled{0}_{eq}\}) \\
TB & ::= & (TVar \times TVar) \overset{total}{\rightarrow} CL_{eq} \\
AR & ::= & AD \times \mathcal{P}(TVar) \times (\mathcal{P}(TVar) \uplus \texttt{full}) \times \mathcal{P}(TVar) \times \mathcal{P}(TVar) \times TB \\
& & \times (\mathcal{D}, \llbracket \cdot \rrbracket^{\mathcal{D}} : \mathcal{D} \rightarrow \mathcal{P}(TVar \overset{total}{\rightarrow} \mathcal{P}(\mathbb{Z})))
\end{array}
$$

For the language PVD^+ we define $\circledast \sqcup S \triangleq S$, $S \sqcup \circledast \triangleq S$,
$\circledast \sqcap S \triangleq S$, $S \sqcap \circledast \triangleq S$, $\forall S_1, S_2 \neq \circledast$. $S_1 \sqcup S_2 \triangleq S_1 \cup S_2$, $S_1 \sqcap S_2 \triangleq S_1 \cap S_2$.

For the language $\mathcal{P}(TVar) \uplus \texttt{full}$ we define $\texttt{full} \cup S \triangleq \texttt{full}$, $S \cup \texttt{full} \triangleq \texttt{full}$,
$\texttt{full} \cap S \triangleq S$, $S \cap \texttt{full} \triangleq S$, $\texttt{full} \setminus S \triangleq \texttt{full}$, $\forall \alpha. \alpha \notin \texttt{full}$.

Figure 3.3: *Syntax of the language*

AD corresponds to the graphs used in the examples in Section 3.2.

TB associates pairs (α, β) of auxiliary variables to relationships defined by CL_{eq}:

- $\ddagger_{eq}$ means both α and β represent an empty set of values

- $\dagger_{eq}$ means exactly one of α and β represents an empty set of values

- $=_{eq}$ means both α and β represent the same nonempty set of values

- $\subset_{eq}$ means the nonempty set of values represented by α is strictly included in the set represented by β

- $\supset_{eq}$ means the nonempty set of values represented by β is strictly included in the set represented by α

- $\sharp_{eq}$ means α and β represent two nonempty disjoint sets of values

- $\textcircled{0}_{eq}$ means that the three sets of values represented by α, by β and by both are nonempty

93

Constraints on $\mathcal{D}$: $\exists \top^{\mathcal{D}}, \nabla^{\mathcal{D}}, \sqcup^{\mathcal{D}}, \sqcap^{\mathcal{D}}, top : (\textit{TVar} \times \mathcal{D}) \to \mathcal{D}$,
$copy : (\textit{TVar} \times \textit{TVar} \times \mathcal{D}) \to \mathcal{D}, merge : (\textit{TVar} \times \textit{TVar} \times \mathcal{D}) \to \mathcal{D}$ such that

- $\forall d \in \mathcal{D}.\alpha_1, \alpha_2 \in \textit{TVar}.$
  $[\![merge(\alpha_1, \alpha_2, d)]\!]^{\mathcal{D}} = \{g' \mid \exists g \in [\![d]\!]^{\mathcal{D}}.\ g' = [g \mid \alpha_2 \to g(\alpha_1) \cup g(\alpha_2) \mid \alpha_1 \to \mathbb{Z}]\}$

- $[\![\top^{\mathcal{D}}]\!]^{\mathcal{D}} = (\textit{TVar} \overset{total}{\to} \mathbb{Z})$

- $\forall d \in \mathcal{D}.\forall \alpha \in \textit{TVar}.\ [\![top(\alpha, d)]\!]^{\mathcal{D}} = \{g' \mid \exists g \in [\![d]\!]^{\mathcal{D}}.\ g' = [g \mid \alpha \to \mathbb{Z}]\}$

- $\forall d \in \mathcal{D}.\forall \alpha_1, \alpha_2 \in \textit{TVar}.[\![copy(\alpha_1, \alpha_2, d)]\!]^{\mathcal{D}}. = \{g' \mid \exists g \in [\![d]\!]^{\mathcal{D}}.\ g' = [g \mid \alpha_2 \to g(\alpha_1)]\}$

- $\forall d_1, d_2 \in \mathcal{D}.\ \forall g_1 \in [\![d_1]\!]^{\mathcal{D}}.\ \forall g_2 \in [\![d_2]\!]^{\mathcal{D}}.\ \exists g_{12} \in [\![d_1 \sqcup^{\mathcal{D}} d_2]\!]^{\mathcal{D}}.\ \forall \alpha.g_1(\alpha) \cup g_2(\alpha) \subseteq g_{12}(\alpha)$

- $\forall d_1, d_2 \in \mathcal{D}.\ \forall g_1 \in [\![d_1]\!]^{\mathcal{D}}.\ \forall g_2 \in [\![d_2]\!]^{\mathcal{D}}.\ \exists g_{12} \in [\![d_1 \sqcap^{\mathcal{D}} d_2]\!]^{\mathcal{D}}.\ \forall \alpha.g_1(\alpha) \cap g_2(\alpha) \subseteq g_{12}(\alpha)$

- $\forall w \in \mathbb{N} \overset{total}{\to} \mathcal{D}.\ \exists i \in \mathbb{N}.\forall i' \geq i.\ \nabla^{\mathcal{D}}(w \restriction_{[0,i] \overset{total}{\to} \mathcal{D}}) = \nabla^{\mathcal{D}}(w \restriction_{[0,i'] \overset{total}{\to} \mathcal{D}})$

- $\forall w \in \mathbb{N} \overset{total}{\to} \mathcal{D}.\ \forall i \in \mathbb{N}.\ \forall g_1 \in [\![w(i)]\!]^{\mathcal{D}}.\exists g_2 \in [\![\nabla^{\mathcal{D}} \left(w \restriction_{[0,i] \overset{total}{\to} \mathcal{D}} \right)]\!]^{\mathcal{D}}.$
  $\forall \alpha.\ g_1(\alpha) \subseteq g_2(\alpha)$

Constraints on the language : $\forall (ad, hu, ho, sn, sn^{\infty}, t, d) \in AR.$

- $\forall x \in \textit{Var}.ad(x) \neq \circledast$

- $\forall \alpha \in \textit{TVar}.ad(\alpha) \neq \emptyset$

- $\forall v \in \textit{VAR}.\alpha \in \textit{TVar}.\alpha \in ad(v) \Rightarrow ad(\alpha) \neq \circledast$

- $\forall v \in \textit{VAR}.\alpha \in \textit{TVar}.vd1 \in \textit{VD}1.\ Loc(A, \alpha, vd1) \in ad(v) \Rightarrow ad(\alpha) \neq \circledast$

- $\forall v \in \textit{VAR}.\alpha \in \textit{TVar}.vd1 \in \textit{VD}1.\ Loc(A, vd1, \alpha) \in ad(v) \Rightarrow ad(\alpha) \neq \circledast$

- $\forall \alpha \in hu.\ ad(\alpha) \neq \circledast$

- $\forall \alpha \in ho.\ ad(\alpha) \neq \circledast$

- $ho = \texttt{full} \vee hu \subseteq ho$

- $\forall \alpha \in \textit{TVar}.ad(\alpha) = \circledast \Rightarrow \forall \alpha' \in \textit{TVar}.\{\ddagger_{eq}, \dagger_{eq}\} \cap t(\alpha, \alpha') \neq \emptyset \wedge \{\ddagger_{eq}, \dagger_{eq}\} \cap t(\alpha', \alpha) \neq \emptyset$

Figure 3.4: *Constraints on the language*

Notice that if α and β are not summary nodes (so there concretisation is $\emptyset$ or a singleton), then $\subset_{eq}, \supset_{eq}, \sharp_{eq}$ and $\textcircled{0}_{eq}$ all have the same meaning : α and β both have a concrete value and their concrete values are different; we could write this as merely $\neq_{eq}$. For this reason, implementation could also work with only a sublatice of $\mathcal{P}(\{\ddagger_{eq}, \dagger_{eq}, =_{eq}, \subset_{eq}, \supset_{eq}, \sharp_{eq}, \textcircled{0}_{eq}\})$.

Comments

- We use total functions for AD and TB because it is more convenient for the proofs. So their domains are Var and $(Var \times Var)$ but the user can define some functions dom for domains so that they satisfy these constraints:

  $\forall ad \in AD.\ dom(ad) \triangleq \{x \in Var | ad(v) \neq \top\} \cup \{\alpha \in TVar | ad(\alpha) \neq \circledast\}$.

  $\forall t \in TB.\ dom(t) \triangleq \{\alpha \in TVar.\ t(\alpha, \alpha) \neq \{\ddagger_{eq}, =_{eq}\} \vee \exists \alpha' \in TVar \setminus \{\alpha\}.\ t(\alpha, \alpha') \not\subseteq \mathcal{P}(\{\ddagger_{eq}, \dagger_{eq}\}) \vee t(\alpha', \alpha) \not\subseteq \mathcal{P}(\{\ddagger_{eq}, \dagger_{eq}\})\}$

  $\forall f \in F.dom(f) \triangleq \{\alpha | f(\alpha) \neq \emptyset\}$

- By choice in our semantics, variables in $sn^\infty \setminus sn$ are not summary nodes.

We have constraints on the language that are presented in Fig. 3.4.

- The first set of constrains are constraints on the numerical domain $\mathcal{D}$. It says that the domain should have a value $\top^{\mathcal{D}}$ which would say that all the auxiliary variables can take any value. It should have a function $merge$ such that it could take two auxiliary variables and an element of the domain and it would allow the first variable to take any value while the second would take its previous value or a value that the first one could previously take. It should have a function top which would allow one variable to take any value. A function $copy$ which would allow one variable to take the same value as another. It should have $\sqcup^{\mathcal{D}}/\sqcap^{\mathcal{D}}$ to overapproximate the union/intersection of two elements. It should have a function $\nabla^{\mathcal{D}}$ which taking a sequence of values and returning a sequence of values which eventually converges and such that every element is an overapproximation of the corresponding element in the argument's sequence.

- The second set of constraints limit the number of elements of the domain whose semantics correspond to an emptyset. Most of them say that if the concrete values that an auxiliary variable can take is an emptyset, then no variable should be allowed to point to that auxiliary variable.

- In our presentation of the language, we expressed only the constraints on the numerical domain which are needed to define the functions presented. There are some other constraints, for example, the domain should provide an element which overapproximates the projection of an element, for example, $(x = 3)$ or $(x < y + 3)$. It should also allow to overapproximate a precondition for an assignment (for example, if d is in the numerical domain, we want d', which should be an overapproximation such that d corresponds to d' where we have assigned $x + 1$ to x).

3.4   Semantics of the language

The abstractions of separation-logic formulae can be efficiently implemented because we formalize them as a disjunction of eight simple semantic-interpretation functions, represented as $[\![\cdot]\!]^i$. This makes a denotation into a tuple of orthogonal elements.

First, we recall the concrete domains:

$$
\begin{aligned}
Val &\triangleq \mathbb{Z} \uplus Bool \uplus \texttt{nil} \uplus Loc & Val' &\triangleq Val \cup \{\texttt{ood}\} \\
S &\triangleq Var \rightharpoonup Val & S' &\triangleq Var \overset{total}{\rightharpoonup} Val' \\
H &\triangleq Var \rightharpoonup (Val \times Val) & F &\triangleq TVar \overset{total}{\rightharpoonup} \mathcal{P}(Val') \\
& & R &\triangleq Loc \rightharpoonup \mathcal{P}(Loc) \\
M &\triangleq S \times H & MFR &\triangleq \mathcal{P}(S' \times H \times F \times R)
\end{aligned}
$$

For simplicity, we work with total functions, so we extend Val with the "out of domain" value $\texttt{ood}$ to define Val'. S' are total stacks, that is we have a bijection between a normal stack and a total stack. We define $^- : S' \rightarrow S$ by $\bar{s'} \triangleq s' \upharpoonright_{dom(s') \cap \{x | s'(x) \neq \texttt{ood}\}}$, and $^- : S \rightarrow S'$ by $\bar{s} \triangleq [x \in dom(s) \rightarrow s(x) \mid x \notin dom(s) \rightarrow \texttt{ood}]$.

The domain F, should be seen as a stack for auxiliary variables. So the abstract language

embeds information about the normal variables which is represented in the concrete domain with the stacks in $S' \triangleq Var \stackrel{total}{\rightarrow} Val'$, the abstract language embeds information about auxiliary variable which is represented in the concrete domain with values in $F \triangleq TVar \stackrel{total}{\rightarrow} \mathcal{P}(Val')$. We have $TVar \stackrel{total}{\rightarrow} \mathcal{P}(Val')$ instead of $TVar \stackrel{total}{\rightarrow} Val'$ because we want to have summary nodes which will represent several concrete values.

To lighten the notation, we define a union of stacks in S' and stacks for auxiliary variables in $F : \cdot^+ \cdot : (S' \times F) \rightarrow (VAR \stackrel{total}{\rightarrow} \mathcal{P}(Val'))$ such that if $x \in Var$ then $s^+f(x) \triangleq \{s(x)\}$ and if $\alpha \in TVar$ then $s^+f(\alpha) \triangleq f(\alpha)$.

For $(s, h, f, r) \in MFR$, the s corresponds to a (total) stack, h is the heap, f is a stack for the variables in $TVar$, where a variable can map to a set of values (cf. a summary node), and r maps locations to their reachable set of locations, thus encoding separation information.

We define $[\![\cdot]\!]$, the semantics of elements of our language, in terms of $\mathcal{P}(M)$:

$$[\![\cdot]\!] \quad \in \quad \boldsymbol{AR} \rightarrow \boldsymbol{\mathcal{P}(M)}$$
$$[\![ar]\!] \quad \triangleq \quad \{\bar{s}, h \mid s, h, f, r \in [\![ar]\!]'\}$$

It uses an intermediate semantics, $[\![\cdot]\!]' : AR \rightarrow MFR$, which is an intersection of the semantics of every component of the tuple, ar. (The need for $[\![\cdot]\!]'$ — instead of having directly $[\![\cdot]\!]$ as a conjunction — is explained in Sect. 3.8.14):

We would like to apologize for using superscripts to differenciate the different semantics depending on their domains ($[\![\cdot]\!]^1$, $[\![\cdot]\!]^2$, ...). The numbers themselves have no special meaning and the ordering is just historical.

$$[\![\cdot]\!]' \quad \in \quad \boldsymbol{AR} \rightarrow \boldsymbol{MFR}$$
$$[\![(ad, hu, ho, sn, sn^\infty, t, d)]\!]' \quad \triangleq \quad [\![ad]\!]^4 \cap [\![hu]\!]^1 \cap [\![ho]\!]^{1'} \cap [\![sn]\!]^2 \cap [\![sn^\infty]\!]^{2'}$$
$$\cap [\![t]\!]^3 \cap [\![d]\!]^7 \cap sem*$$

The semantics of the graph ad is also a conjunction for all assignments:

$$[\![\cdot]\!]^4 \quad \in \quad \boldsymbol{AD} \rightarrow \boldsymbol{MFR}$$
$$[\![ad]\!]^4 \quad \triangleq \quad \bigcap_{v \in VAR} [\![v, ad(v)]\!]^5$$

An assignment to a set is a disjunction of assignments:

$$\llbracket \cdot \rrbracket^5 \in (VAR \times PVD^+) \to MFR$$

$$\llbracket v, \top \rrbracket^5 \triangleq MFR$$

$$\llbracket v, \circledast \rrbracket^5 \triangleq \{s, h, f, r \mid s^+f(v) = \emptyset\}$$

$$\llbracket v, S \rrbracket^5 \triangleq \{s, h, f, r \mid s^+f(v) \subseteq \bigcup_{vd \in S} \llbracket vd, (h, f, r) \rrbracket^8\}, \text{ if } S \notin \{\top, \circledast\}$$

The rule for $\top$ gives no information, for $\circledast$ it is used for "fresh" variables. Thus they should represent an empty set of concrete values. For S, the rule says that if a variable points to a set of abstract values, then its concrete values should be included in the union of all the concrete values represented by each abstract value.

We use an auxiliary function, $\llbracket \cdot \rrbracket^8$, which gives the sets of concrete values in Val' that correspond to an abstract value in VD for a particular element of $H \times F \times R$:

$$\llbracket \cdot \rrbracket^8 \in (VD \times (H \times F \times R)) \to \mathcal{P}(Val')$$

$$\llbracket Nilt, \_ \rrbracket^8 \triangleq \{\texttt{nil}\} \qquad\qquad \llbracket Truet, \_ \rrbracket^8 \triangleq \{True\}$$

$$\llbracket Falset, \_ \rrbracket^8 \triangleq \{False\} \qquad\qquad \llbracket Oodt, \_ \rrbracket^8 \triangleq \{\texttt{ood}\}$$

$$\llbracket Numt, \_ \rrbracket^8 \triangleq \mathbb{Z} \qquad \llbracket Dangling\_Loc, (h, \_, \_) \rrbracket^8 \triangleq Loc \setminus dom(h)$$

$$\llbracket v, (\_, f, \_) \rrbracket^8 \triangleq f(v) \text{ when } v \in TVar$$

The semantics of $*1$ and $*2$ are defined by these two definitions:

$$\llbracket Loc(A, vd1, vd2), (h, f, r) \rrbracket^8$$

$$\triangleq \left\{ l \in dom(h) \left| \begin{array}{l} \bullet \quad \Pi_1(h(l)) \in \llbracket vd1, (h, f, r) \rrbracket^8 \\ \bullet \quad \Pi_2(h(l)) \in \llbracket vd2, (h, f, r) \rrbracket^8 \\ \bullet \quad *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r(l) \\ \bullet \quad *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r(l) \end{array} \right. \right\}$$

$$sem* \in MFR$$

$$sem* \triangleq \left\{ s, h, f, r \left| \forall l \in dom(r). \begin{array}{l} \bullet \quad l \notin r(l) \\ \bullet \quad \forall l' \in r(l) \cap dom(r). r(l') \subseteq r(l) \end{array} \right. \right\}$$

$sem*$ says that the concrete function r should be a reachability function, so all the reachable from a reachable are reachable. And first, that there are no cycles, so no one should be reachable from itself.

The semantics of the lower bound of heap's domain is given by $\llbracket \cdot \rrbracket^1$:

$$\llbracket \cdot \rrbracket^1 \in \mathcal{P}(TVar) \to MFR$$

$$\llbracket hu \rrbracket^1 \triangleq \bigcap_{\alpha \in hu} \{s, h, f, r \mid f(\alpha) \cap dom(h) \neq \emptyset\}$$

and the semantics of the upper bound of a heap's domain is given by $\llbracket \cdot \rrbracket^{1'}$:

$$\llbracket \cdot \rrbracket^{1'} \ \in \ (\mathcal{P}(\mathit{TVar}) \uplus \mathtt{full}) \rightarrow \boldsymbol{MFR}$$

$$\llbracket \mathtt{full} \rrbracket^{1'} \ \triangleq \ MFR$$

$$\llbracket ho \rrbracket^{1'} \ \triangleq \ \{s, h, f, r \mid dom(h) \subseteq \bigcup_{\alpha \in ho} f(\alpha)\}$$

The semantics of summary nodes are given by $\llbracket \cdot \rrbracket^{2}$ and $\llbracket \cdot \rrbracket^{2'}$

$$\llbracket \cdot \rrbracket^{2} \ \in \ \mathcal{P}(\mathit{TVar}) \rightarrow \boldsymbol{MFR}$$

$$\llbracket sn \rrbracket^{2} \ \triangleq \ \{s, h, f, r \mid \forall \alpha \in \mathit{TVar} \setminus sn.|f(\alpha)| \leq 1\}$$

$$\llbracket \cdot \rrbracket^{2'} \ \in \ \mathcal{P}(\mathit{TVar}) \rightarrow \boldsymbol{MFR}$$

$$\llbracket sn^{\infty} \rrbracket^{2'} \ \triangleq \ \{s, h, f, r \mid \forall \alpha \in \mathit{TVar} \setminus sn^{\infty}.\ f(\alpha) \text{ is finite}\}$$

The semantics of the table is a conjunction of all semantics of its assignments:

$$\llbracket \cdot \rrbracket^{3} \ \in \ \boldsymbol{TB} \rightarrow \boldsymbol{MFR}$$

$$\llbracket t \rrbracket^{3} \ \triangleq \ \bigcap_{(\alpha_1, \alpha_2) \in \mathit{TVar} \times \mathit{TVar}} \{s, h, f, r \mid \llbracket t(\alpha_1, \alpha_2), f(\alpha_1), f(\alpha_2) \rrbracket^{6'}\}$$

An assignment of the table is a disjunction of assignments:

$$\llbracket \cdot \rrbracket^{6'} \ \in \ (\boldsymbol{CL_{eq}} \times \boldsymbol{\mathcal{P}(Val')} \times \boldsymbol{\mathcal{P}(Val')}) \rightarrow \boldsymbol{Bool}$$

$$\llbracket S, A, B \rrbracket^{6'} \ \triangleq \ \bigvee_{l \in S} \llbracket l, A, B \rrbracket^{6}$$

The table encodes aliasing, non-aliasing information, and mutual exclusion. (For example, $\dagger_{eq}$ insures that two variables cannot live at the same time, and $\sharp_{eq}$ insures that two variables are not aliased.)

$$\llbracket \cdot \rrbracket^{6} \ \in \ (\top_{\boldsymbol{eq}} \times \boldsymbol{\mathcal{P}(Val')} \times \boldsymbol{\mathcal{P}(Val')}) \rightarrow \boldsymbol{Bool}$$

$$\llbracket \ddagger_{eq}, A, B \rrbracket^{6} \ \triangleq \ A \cup B = \emptyset \qquad \llbracket \dagger_{eq}, A, B \rrbracket^{6} \ \triangleq \ A = \emptyset \text{ xor } B = \emptyset$$

$$\llbracket =_{eq}, A, B \rrbracket^{6} \ \triangleq \ A = B \wedge A \neq \emptyset \quad \llbracket \subset_{eq}, A, B \rrbracket^{6} \ \triangleq \ A \subset B \wedge A \neq \emptyset$$

$$\llbracket \supset_{eq}, A, B \rrbracket^{6} \ \triangleq \ A \supset B \wedge B \neq \emptyset \quad \llbracket \sharp_{eq}, A, B \rrbracket^{6} \ \triangleq \ A \cap B = \emptyset \wedge A \neq \emptyset \wedge B \neq \emptyset$$

$$\llbracket \bigcirc\!\!\!\!\bigcirc_{eq}, A, B \rrbracket^{6} \ \triangleq \ A \cap B \neq \emptyset \wedge A \setminus B \neq \emptyset \wedge B \setminus A \neq \emptyset$$

The semantics of the numerical part is given by $\llbracket \cdot \rrbracket^{7}$:

$$\llbracket \cdot \rrbracket^{7} \ \in \ \boldsymbol{\mathcal{D}} \rightarrow \boldsymbol{MFR}$$

$$\llbracket d \rrbracket^{7} \ \triangleq \ \bigcup_{g \in \llbracket d \rrbracket^{\mathcal{D}}} \bigcap_{\alpha \in \mathit{TVar}} \{s, h, f, r \mid f(\alpha) \cap \mathbb{Z} \subseteq g(\alpha)\}$$

3.5 Operations on the language

We now present five key operations and describe how they are computed within the abstract language. All the operations have been proved sound. We first present an *extension* function to add an auxiliary variable to a graph, which is used to tune the precision of the *union* function presented second. Then we present a *merging* function which is used to merge two

99

nodes to reduce the size of the graph, which is used for the stabilization function presented just after. Then we present a function, *ast*, which is used to translate the connective $*$.

Below, P is a formula in the separation logic with fixpoints defined in Chapter 2, and $T(P) \in AR$ is the translation of the formula.

For all the transformations T of elements of the language, we provide theorems of this form: $\forall s, h, f, r \in MFR.\exists g.\ s, h, f, r \in \llbracket ar \rrbracket' \Rightarrow s, h, g(f), r \in \llbracket T(ar) \rrbracket'$. The "$g$" being there because we wish to allow changes for auxiliary variables (like for the function, *merge*), so the values taken by the auxiliary variables do not have to be characterised by f in both sides of the implication, but can be transformed by a function g on the right side of the implication.

To explain, we want to translate formulae in *separation logic with fixpoints* into the language AR. We want $\llbracket P \rrbracket \subseteq \llbracket T(P) \rrbracket$. For efficiency on the computation of conjunction, we will define another translation, $T' : AR \times BI^{\mu\nu} \to AR$, from an element of the language and a formula to an element of the language, so that we have $T'(ar, P_1 \wedge P_2) = T'(T'(ar, P_1), P_2)$.

We do the proofs by induction on the syntax of the formula. We do the proofs at the level of $\llbracket \cdot \rrbracket'$ and need to keep the properties at this level and not $\llbracket \cdot \rrbracket$ because $\llbracket ar_1 \rrbracket \subseteq \llbracket ar_2 \rrbracket \not\Rightarrow \llbracket ar_1 \rrbracket' \subseteq \llbracket ar_2 \rrbracket'$. Then at the end we define $T(P) = T'(ar_i, P)$. Since we have $\forall s, h, f, r \in MFR.\ \exists g.\ s, h, f, r \in ar_i \wedge \bar{s}, h \in \llbracket P \rrbracket \Rightarrow s, h, g(f), r \in \llbracket T(P) \rrbracket'$ and $\llbracket ar_i \rrbracket' = MFR$, we get $\llbracket P \rrbracket \subseteq \llbracket T(P) \rrbracket$.

Recall that $\llbracket \cdot \rrbracket$ is the semantic on the state domain $S \times H$ like for formulae, in this semantics, the auxiliary variables can be renamed by alpha-renaming. $\llbracket \cdot \rrbracket'$ is on $S' \times H \times F \times R$ and the auxiliary variables can not be renamed.

3.5.1 Extension

Description

We present two extension functions, which add a "fresh variable" (a variable whose value is $\otimes$) as an intermediary between a variable and its assignment. Both functions have the same

properties. The first one is cheaper but the second one contributes better to the precision of the union than the first.

The first version of extension, named $extend(v, \alpha, \_)$, (which is cheaper but induce less precision) will replace

by
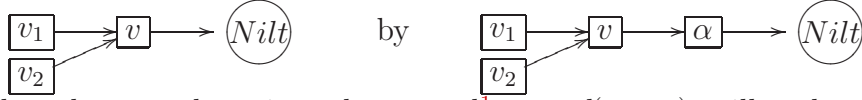
but the second version, also named[1] $extend(v, \alpha, \_)$, will replace it by
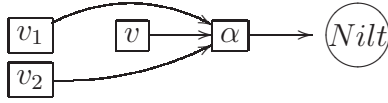
Properties

Both functions will satisfy the property

Proposition 3.1. $\forall v \in VAR.\alpha \in TVar.[ar \mid \alpha \rightarrow \circledast] \in AR.(s, h, f, r) \in MFR.$

$s, h, f, r \in [\![[ar \mid \alpha \rightarrow \circledast]]\!]' \Leftrightarrow s, h, [f \mid \alpha \rightarrow s^+ f(v)] \in [\![extend(v, \alpha, [ar \mid \alpha \rightarrow \circledast])]\!]'$

Corollary 3.2. $\forall v \in VAR.\alpha \in TVar.ar \in AR.$ $[\![ar]\!] = [\![extend(v, \alpha, ar)]\!]$

The proof for the cheap extension function is available in Sect. 3.8.2 and the proof for the more precise extension is in Sect. 3.8.3.

Definitions

The cheap extension is defined as follows:

---

[1]we use the same name since they satisfy the same proposition and the choice of one or the other will be left as an implementation heuristic

$extend\colon \boldsymbol{VAR} \rightarrow \mathit{TVar} \rightarrow \boldsymbol{AR} \rightarrow \boldsymbol{AR}$

$extend\,(v, \alpha, ([ad \mid \alpha \rightarrow \otimes], hu, ho, sn, sn^{\infty}, t, d)) \triangleq$

$$
\left(
\begin{array}{l}
[ad \mid v \rightarrow \{\alpha\}, \alpha \rightarrow ad(v)] \\[4pt]
hu, \\[4pt]
ho \setminus \{\alpha\}, \\[4pt]
\text{if } v \in sn \text{ then } sn \cup \{\alpha\} \text{ else } sn \setminus \{\alpha\}, \\[4pt]
\text{if } v \in sn^{\infty} \text{ then } sn^{\infty} \cup \{\alpha\} \text{ else } sn^{\infty} \setminus \{\alpha\}, \\[4pt]
\quad \text{if } v \in \mathit{TVar} \\[4pt]
\quad \text{then } \left[
\begin{array}{l}
t \\
\mid (\alpha, \alpha') \rightarrow t(v, \alpha') \\
\mid (\alpha', \alpha) \rightarrow t(\alpha', v)
\end{array}
\right] \\[20pt]
\quad \text{else } \left[
\begin{array}{l}
t \\
\mid (\alpha, \alpha') \rightarrow \text{if } ad(\alpha') = \otimes \ \text{ then } \{\ddagger_{eq}, \dagger_{eq}\} \text{ else } CL_{eq} \\
\mid (\alpha', \alpha) \rightarrow \text{if } ad(\alpha') = \otimes \ \text{ then } \{\ddagger_{eq}, \dagger_{eq}\} \text{ else } CL_{eq} \\
\mid (\alpha, \alpha) \rightarrow \{\ddagger_{eq}, =_{eq}\}
\end{array}
\right], \\[28pt]
\text{if } v \in \mathit{Var} \text{ then } top(\alpha, d) \text{ else } copy(v, \alpha, d)
\end{array}
\right)
$$

The function needs that α already points to $\otimes$. It makes v point to α in the graph and α points to what v was previously pointing to. For the rest of the components, it just updates them knowing that α is equal to v.

The better extension is defined as follows:

$extend$**$: VAR \rightarrow TVar \rightarrow AR \rightarrow AR$**

$extend\,(v, \alpha, ([ad \mid \alpha \rightarrow \circledast], hu, ho, sn, sn^\infty, t)) \triangleq$

$$
\left(
\begin{array}{l}
replace(v, \alpha) \circ [ad \mid v \rightarrow \{\alpha\}, \alpha \rightarrow ad(v)] \\
\text{if } v \in hu \text{ then } hu \setminus \{v\} \cup \{\alpha\} \text{ else } hu, \\
\text{if } v \in ho \text{ then } ho \setminus \{v\} \cup \{\alpha\} \text{ else } ho \setminus \{\alpha\}, \\
\text{if } v \in sn \text{ then } sn \cup \{\alpha\} \text{ else } sn \setminus \{\alpha\}, \\
\text{if } v \in sn^\infty \text{ then } sn^\infty \cup \{\alpha\} \text{ else } sn^\infty \setminus \{\alpha\}, \\
\text{ if } v \in TVar \\
\quad \text{then } \left[\begin{array}{l} t \\ \mid (\alpha, \alpha') \rightarrow t(v, \alpha') \\ \mid (\alpha', \alpha) \rightarrow t(\alpha', v) \end{array}\right] \\
\quad \text{else } \left[\begin{array}{l} t \\ \mid (\alpha, \alpha') \rightarrow \text{if } ad(\alpha') = \circledast \ \text{then } \{\ddagger_{eq}, \dagger_{eq}\} \text{ else } CL_{eq} \\ \mid (\alpha', \alpha) \rightarrow \text{if } ad(\alpha') = \circledast \ \text{then } \{\ddagger_{eq}, \dagger_{eq}\} \text{ else } CL_{eq} \\ \mid (\alpha, \alpha) \rightarrow \{\ddagger_{eq}, =_{eq}\} \end{array}\right] , \\
\text{if } v \in Var \text{ then } top(\alpha, d) \text{ else } top(v, copy(v, \alpha, d))
\end{array}
\right)
$$

This more precise version first does the same thing as other one and then replaces in the graph the arrows pointing to v by arrows pointing to α.

For the proofs, we defined the extension in the concrete domain as

$$extend \quad : \quad \boldsymbol{VAR} \rightarrow TVar \rightarrow \boldsymbol{S'} \rightarrow \boldsymbol{F} \rightarrow \boldsymbol{F}$$

$$extend(v, \alpha, s, f) \quad \triangleq \quad [f \mid \alpha \rightarrow s^+ f(v)]$$

3.5.2 Union

The basic union is almost a simple union of all nodes and edges except that if a variable in *Var* has no outgoing edges (that is, the variable is assigned to $\top$), the union graph does not have an edge from this variable.

Definition

$$union \quad : \quad (\boldsymbol{AR} \times \boldsymbol{AR}) \to \boldsymbol{AR}$$

$$basic\_union \left(\left(\begin{array}{c} ad_1, \\ hu_1, \\ ho_1, \\ sn_1, \\ sn_1^\infty, \\ t_1, \\ d_1 \end{array} \right) , \left(\begin{array}{c} ad_2, \\ hu_2, \\ ho_2, \\ sn_2, \\ sn_2^\infty, \\ t_2, \\ d_2 \end{array} \right) \right) \triangleq \left(\begin{array}{c} ad_1 \dot{\sqcup}\, ad_2, \\ hu_1 \cap hu_2, \\ ho_1 \cup ho_2, \\ sn_1 \cup sn_2, \\ sn_1^\infty \cup sn_2^\infty, \\ t_1 \dot{\cup}\, t_2, \\ d_1 \sqcup^{\mathcal{D}} d_2 \end{array} \right)$$

$\dot{\sqcup}/\dot{\cup}$ being the pointwise applications of $\sqcup/\cup$, respectively

Properties

Proposition 3.3. $\forall ar_1, ar_2 \in AR.\ [\![ar_1]\!]' \cup [\![ar_2]\!]' \subseteq [\![basic\_union(ar_1, ar_2)]\!]'$

Corollary 3.4. $\forall ar_1, ar_2 \in AR.\ [\![ar_1]\!] \cup [\![ar_2]\!] \subseteq [\![basic\_union(ar_1, ar_2)]\!]$

Proofs are found in Sect. 3.8.4 (but there, *basic_union* is just called *union*).

More precise union

The basic union defined above is not precise. For example, it translates $(x = y \wedge y = \texttt{nil}) \vee (y = \texttt{true})$ into
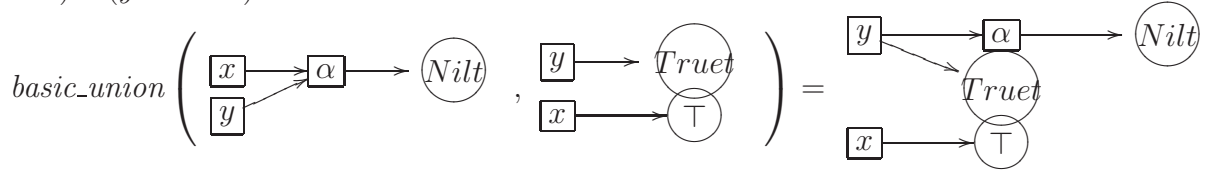
which is merely $(y = \texttt{nil}) \vee (y = \texttt{true})$.

So we will combine the basic union function with *extend*. For the same example, we obtain

$$union\left(\;\boxed{x}\rightarrow\boxed{\alpha}\rightarrow\bigcirc\!\!Nilt \quad,\quad \boxed{y}\rightarrow Truet \atop \boxed{x}\rightarrow\top \;\right)$$

$$=\;basic\_union\left(\;\boxed{x}\rightarrow\boxed{\alpha}\rightarrow\bigcirc\!\!Nilt \quad,\quad \boxed{y}\rightarrow Truet \atop \boxed{x}\rightarrow\boxed{\bar{\beta}}\rightarrow\top \;\right)$$

$$=\qquad \boxed{x}\rightarrow\boxed{\beta}\rightarrow\top \atop \{\dagger_{eq}\} \atop \boxed{\alpha}\rightarrow\bigcirc\!\!Nilt \atop \boxed{y}\rightarrow Truet$$
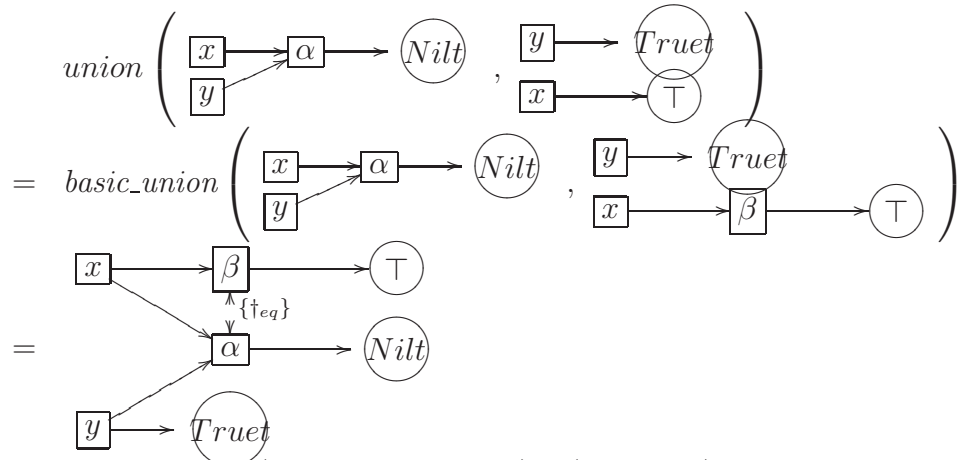
This is exactly $(x = \texttt{nil} \wedge y = \texttt{nil}) \vee (y = \texttt{true})$

In a non graphical presentation, this example is written

$$
union \left(
\begin{array}{l}
\left(
\begin{array}{l}
[ar_i|x \to \{\alpha\}|y \to \{\alpha\}|\alpha \to \{Nilt\}], \\
\emptyset, TVar, \emptyset, \emptyset, \\
\left[t_i \;\middle|\;
\begin{array}{c|c|c}
 & \alpha & \beta \\ \hline
\alpha & \{=_{eq}\} & \{\dagger_{eq}\} \\ \hline
\beta & \{\dagger_{eq}\} & \{\ddagger_{eq}\}
\end{array}
\right], \top^{\mathcal{D}}
\end{array}
\right), \\[2em]
\left(
\begin{array}{l}
[ar_i|y \to \{Truet\}], \\
\emptyset, TVar, \emptyset, \emptyset, \\
\left[t_i \;\middle|\;
\begin{array}{c|c|c}
 & \alpha & \beta \\ \hline
\alpha & \{\ddagger_{eq}\} & \{\ddagger_{eq}\} \\ \hline
\beta & \{\ddagger_{eq}\} & \{\ddagger_{eq}\}
\end{array}
\right], \top^{\mathcal{D}}
\end{array}
\right)
\end{array}
\right)
$$

$$
= basic\_union \left(
\begin{array}{l}
\left(
\begin{array}{l}
[ar_i|x \to \{\alpha\}|y \to \{\alpha\}|\alpha \to \{Nilt\}], \\
\emptyset, TVar, \emptyset, \emptyset, \\
\left[t_i \;\middle|\;
\begin{array}{c|c|c}
 & \alpha & \beta \\ \hline
\alpha & \{=_{eq}\} & \{\dagger_{eq}\} \\ \hline
\beta & \{\dagger_{eq}\} & \{\ddagger_{eq}\}
\end{array}
\right], \top^{\mathcal{D}}
\end{array}
\right), \\[2em]
\left(
\begin{array}{l}
[ar_i|y \to \{Truet\}|x \to \{\beta\}|\beta \to \top], \\
\emptyset, TVar, \emptyset, \emptyset, \\
\left[t_i \;\middle|\;
\begin{array}{c|c|c}
 & \alpha & \beta \\ \hline
\alpha & \{\ddagger_{eq}\} & \{\dagger_{eq}\} \\ \hline
\beta & \{\dagger_{eq}\} & \{\ddagger_{eq}, =_{eq}\}
\end{array}
\right], \top^{\mathcal{D}}
\end{array}
\right)
\end{array}
\right)
$$

$$
= \left(
\begin{array}{l}
[ar_i|x \to \{\alpha,\beta\}|y \to \{\alpha, Truet\}|\alpha \to \{Nilt\}|\beta \to \top], \\
\emptyset, TVar, \emptyset, \emptyset, \\
\left[t_i \;\middle|\;
\begin{array}{c|c|c}
 & \alpha & \beta \\ \hline
\alpha & \{\ddagger_{eq}, =_{eq}\} & \{\dagger_{eq}\} \\ \hline
\beta & \{\dagger_{eq}\} & \{\ddagger_{eq}, =_{eq}\}
\end{array}
\right], \top^{\mathcal{D}}
\end{array}
\right)
$$

So one can tune the precision of the union by combining it with *extend*, but it is more expensive. Using it everywhere would be comparable to renaming all variables before union. We suggest applying *extend* only when we must union $\top$ with $S \neq \top$ such that $\exists \alpha \in TVar. \alpha \in S$.

We do not suggest to first do several *extends*, then do the *basic_union* _ the *extend* could be done on the fly _ we would just have to be careful while applying *extend* that the *fresh* variables are fresh for both arguments of the union.

3.5.3 Merging nodes

Description

We define polymorphic functions for merging two nodes. The first node's information gets included in the second one and the first node is removed. This function will be used to reduce the number of auxiliary variables which are used and will create summary nodes. In particular, we will use this function for stabilization which is crucial for translating fixpoints.

For example, after merging α_1 into α_2 in

we get

 .
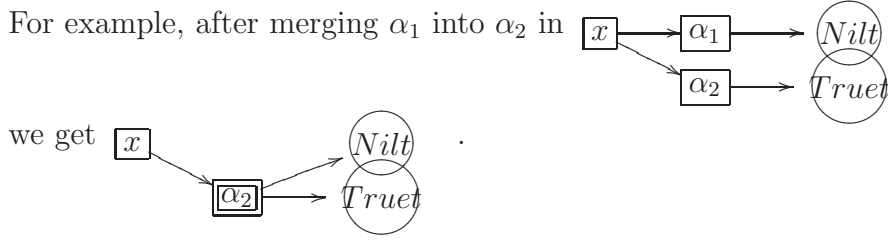
Definition

The merge operation is defined as follows:

$$
\begin{aligned}
&merge\colon TVar \to TVar \to \mathbf{AR} \to \mathbf{AR}\\
&merge\,(\alpha_1, \alpha_2, (ad, hu, ho, sn, sn^\infty, t, d)) \triangleq\\
&\left(
\begin{array}{l}
replace(\alpha_1, \alpha_2) \circ [ad \mid \alpha_2 \to ad(\alpha_1) \sqcup ad(\alpha_2) \mid \alpha_1 \to \otimes],\\
\text{if } \{\alpha_1, \alpha_2\} \not\subseteq hu \text{ then } hu \setminus \{\alpha_1, \alpha_2\} \text{ else } hu \setminus \{\alpha_1\},\\
\text{if } \{\alpha_1, \alpha_2\} \cap ho = \emptyset \text{ then } ho \text{ else } ho \setminus \{\alpha_1\} \cup \{\alpha_2\},\\
sn \setminus \{\alpha_1\} \cup \{\alpha_2\},\\
\text{if } \{\alpha_1, \alpha_2\} \cap sn^\infty = \emptyset \text{ then } sn^\infty \text{ else } sn^\infty \setminus \{\alpha_1\} \cup \{\alpha_2\},\\
t \left[
\begin{array}{l}
\mid (\alpha_2, v) \to UR_{eq}(t(\alpha_1, v), t(\alpha_2, v))\\
\mid (v, \alpha_2) \to UL_{eq}(t(v, \alpha_1), t(v, \alpha_2))\\
\mid (\alpha_2, \alpha_2) \to \{\ddagger_{eq}, =_{eq}\}\\
\mid (\_, \alpha_1) \to \{\ddagger_{eq}, \dagger_{eq}\}\\
\mid (\alpha_1, \_) \to \{\ddagger_{eq}, \dagger_{eq}\}\\
\mid (\alpha_1, \alpha_1) \to \{\ddagger_{eq}\}
\end{array}
\right],\\
merge(\alpha_1, \alpha_2, d)
\end{array}
\right)
\end{aligned}
$$

$$merge \quad : \quad ((\bigcup_{n\in\mathbb{Z}}[0,n] \overset{total}{\to} (TVar \times TVar)) \times \boldsymbol{AR}) \to \boldsymbol{AR}$$

$$merge([0 \to (\alpha_1, \alpha_2)], ar) \quad \triangleq \quad merge(\alpha_1, \alpha_2, ar)$$

$$merge([w|(n+1) \to (\alpha_1, \alpha_2)] \restriction_{[0,n+1]}, ar) \quad \triangleq \quad merge(w \restriction_{[0,n]}, merge(\alpha_1, \alpha_2, ar))$$

$$merge \quad : \quad TVar \to TVar \to \boldsymbol{F} \to \boldsymbol{F}$$

$$merge(\alpha_1, \alpha_2, f) \quad \triangleq \quad [f \mid \alpha_1 \to \emptyset \mid \alpha_2 \to f(\alpha_1) \cup f(\alpha_2)]$$

$$merge \quad : \quad ((\bigcup_{n\in\mathbb{Z}}[0,n] \overset{total}{\to} (TVar \times TVar)) \times \boldsymbol{F}) \to \boldsymbol{F}$$

$$merge([0 \to (\alpha_1, \alpha_2)], f) \quad \triangleq \quad merge(\alpha_1, \alpha_2, f)$$

$$merge([w|(n+1) \to (\alpha_1, \alpha_2)] \restriction_{[0,n+1]}, ar) \quad \triangleq \quad merge(w \restriction_{[0,n]}, merge(\alpha_1, \alpha_2, f))$$

We define UR_{eq} and UL_{eq} in Subsection .

Properties

Proposition 3.5. $\forall \alpha_1, \alpha_2 \in TVar.\forall ar \in AR.\ (s, h, f, r \in \llbracket ar \rrbracket' \Rightarrow s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, ar) \rrbracket')$

Corollary 3.6. $\forall w \in (\bigcup_{n\in\mathbb{Z}}[0,n] \overset{total}{\to} (TVar \times TVar)).\ \forall ar \in AR.\ (s, h, f, r \in \llbracket ar \rrbracket' \Rightarrow s, h, merge(w, f), r \in \llbracket merge(w, ar) \rrbracket')$

Corollary 3.7. $\forall \alpha_1, \alpha_2 \in TVar.ar \in AR.\ \llbracket ar \rrbracket \subseteq \llbracket merge(\alpha_1, \alpha_2, ar) \rrbracket$

The *used* variables are defined by

$$used: \boldsymbol{AR} \to \mathcal{P}(TVar)$$
$$used((ad, hu, ho, sn, sn^\infty, t, d)) \triangleq$$
$$\begin{pmatrix} & \{\alpha \in TVar \mid ad(\alpha) \neq \circledast\} \\ \cup & \{\alpha \in TVar \mid \exists v \in VAR.\alpha \in ad(v)\} \\ \cup & \{\alpha \in TVar \mid \exists v \in VAR.vd \in VD1.\ Loc(A, \alpha, vd) \in ad(v)\} \\ \cup & \{\alpha \in TVar \mid \exists v \in VAR.vd \in VD1.\ Loc(A, vd, \alpha) \in ad(v)\} \\ \cup & hu \cup \{\alpha \mid \alpha \in ho\} \cup sn \cup sn^\infty \\ \cup & \{\alpha \mid t(\alpha, \alpha) \neq \{\ddagger_{eq}\}\} \\ \cup & \{\alpha \mid \exists \alpha'.t(\alpha, \alpha') \neq \{\ddagger_{eq}, \dagger_{eq}\}\} \\ \cup & \{\alpha \mid \exists \alpha'.t(\alpha', \alpha) \neq \{\ddagger_{eq}, \dagger_{eq}\}\} \\ \cup & \{\alpha \mid top(\alpha, d) \neq d\} \end{pmatrix}$$

Proposition 3.8. $\forall \alpha_1, \alpha_2 \in \mathit{TVar}.\forall ar \in AR.\ \mathit{used}(\mathit{merge}(\alpha_1, \alpha_2, ar)) = \mathit{used}\,(ar)\ \backslash\ \{\alpha_1\}$

Corollary 3.9. $\forall w \in (\bigcup_{n \in \mathbb{N}} [0, n] \overset{total}{\to} (\mathit{TVar} \times \mathit{TVar})).\ \forall ar \in AR.\ \mathit{used}(\mathit{merge}(w, ar)) = \mathit{used}\,(ar)\ \backslash\ \mathit{fst}(\mathit{range}(w))$

Proposition 3.10. $\forall A \in \mathcal{P}(\mathit{TVar}).\forall d' \in \mathcal{D}.\ A \textit{ is finite} \Rightarrow \{(ad, hu, ho, sn, sn^\infty, t, d) \in AR \mid d = d' \wedge \mathit{used}(AR) \subseteq A\}$ *is finite*

Proofs are found in Sect. 3.8.5.

Functions on CL_{eq}

Here, we define two functions UR_{eq} and UL_{eq} which are used to update the tabular component while merging two nodes.

They are defined such that, if we have 3 sets of values A, B and C

- If we know that C and A are in the relationship $l1 \in \top_{eq}$ and C and B are in the relationship $l2 \in \top_{eq}$ then C and $A \cup B$ will be in one of the relationship in $ur_{eq}(l1, l2)$.

- If we know that A and C are in the relationship $l1 \in \top_{eq}$ and B and C are in the relationship $l2 \in \top_{eq}$ then $A \cup B$ and C will be in one of the relationship in $ul_{eq}(l1, l2)$.

We define UR_{eq} as follows

$UR_{eq}: (CL_{eq} \times CL_{eq}) \rightarrow CL_{eq}$

$$UR_{eq}(S^1_{eq}, S^2_{eq}) \triangleq \bigcup_{l_1, l_2 \in S^1_{eq} \times S^2_{eq}} ur_{eq}(l_1, l_2)$$

$ur_{eq}: (\top_{eq} \times \top_{eq}) \rightarrow CL_{eq} \uplus \emptyset$

$ur_{eq}(l_1, l_2) \triangleq$

l_1 \ l_2	$\ddagger_{eq}$	$\dagger_{eq}$	$=_{eq}$	$\supset_{eq}$	$\subset_{eq}$	$\sharp_{eq}$	$\textcircled{O}_{eq}$
$\ddagger_{eq}$	$\{\ddagger_{eq}\}$	$\{\dagger_{eq}\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\dagger_{eq}$	$\{\dagger_{eq}\}$	$\{\dagger_{eq}\}$	$\{=_{eq}\}$	$\{\supset_{eq}\}$	$\{\subset_{eq}\}$	$\{\sharp_{eq}\}$	$\{\textcircled{O}_{eq}\}$
$=_{eq}$	$\emptyset$	$\{=_{eq}\}$	$\{=_{eq}\}$	$\{=_{eq}\}$	$\{\subset_{eq}\}$	$\{\subset_{eq}\}$	$\{\subset_{eq}\}$
$\supset_{eq}$	$\emptyset$	$\{\supset_{eq}\}$	$\{=_{eq}\}$	$\{\supset_{eq}, =_{eq}\}$	$\{\subset_{eq}\}$	$\{\textcircled{O}_{eq}\}$	$\{\subset_{eq}, \textcircled{O}_{eq}\}$
$\subset_{eq}$	$\emptyset$	$\{\subset_{eq}\}$	$\{\subset_{eq}\}$	$\{\subset_{eq}\}$	$\{\subset_{eq}\}$	$\{\subset_{eq}\}$	$\{\subset_{eq}\}$
$\sharp_{eq}$	$\emptyset$	$\{\sharp_{eq}\}$	$\{\subset_{eq}\}$	$\{\textcircled{O}_{eq}\}$	$\{\subset_{eq}\}$	$\{\sharp_{eq}\}$	$\{\textcircled{O}_{eq}\}$
$\textcircled{O}_{eq}$	$\emptyset$	$\{\textcircled{O}_{eq}\}$	$\{\subset_{eq}\}$	$\{\subset_{eq}, \textcircled{O}_{eq}\}$	$\{\subset_{eq}\}$	$\{\textcircled{O}_{eq}\}$	$\{\subset_{eq}, \textcircled{O}_{eq}\}$

We have (proof in Sect. 3.8.6)

Proposition 3.11. $\forall S^1_{eq}, S^2_{eq} \in CL_{eq}. \ \forall S, S^1, S^2 \in \mathcal{P}(Val'). \ [\![S^1_{eq}, S, S^1]\!]^{6'} \wedge [\![S^2_{eq}, S, S^2]\!]^{6'} \Rightarrow [\![UR_{eq}(S^1_{eq}, S^2_{eq}), S, S^1 \cup S^2]\!]^{6'}$

We define UL_{eq} as follows

$UL_{eq}: (CL_{eq} \times CL_{eq}) \rightarrow CL_{eq}$

$$UL_{eq}(S^1_{eq}, S^2_{eq}) \triangleq \bigcup_{l_1, l_2 \in S^1_{eq} \times S^2_{eq}} ul_{eq}(l_1, l_2)$$

$ul_{eq}: (\top_{eq} \times \top_{eq}) \rightarrow CL_{eq} \uplus \emptyset$

$ul_{eq}(l_1, l_2) \triangleq$

l_1 \ l_2	$\ddagger_{eq}$	$\dagger_{eq}$	$=_{eq}$	$\subset_{eq}$	$\supset_{eq}$	$\sharp_{eq}$	$\textcircled{O}_{eq}$
$\ddagger_{eq}$	$\{\ddagger_{eq}\}$	$\{\dagger_{eq}\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\dagger_{eq}$	$\{\dagger_{eq}\}$	$\{\dagger_{eq}\}$	$\{=_{eq}\}$	$\{\subset_{eq}\}$	$\{\supset_{eq}\}$	$\{\sharp_{eq}\}$	$\{\textcircled{O}_{eq}\}$
$=_{eq}$	$\emptyset$	$\{=_{eq}\}$	$\{=_{eq}\}$	$\{=_{eq}\}$	$\{\supset_{eq}\}$	$\{\supset_{eq}\}$	$\{\supset_{eq}\}$
$\subset_{eq}$	$\emptyset$	$\{\subset_{eq}\}$	$\{=_{eq}\}$	$\{\subset_{eq}, =_{eq}\}$	$\{\supset_{eq}\}$	$\{\textcircled{O}_{eq}\}$	$\{\supset_{eq}, \textcircled{O}_{eq}\}$
$\supset_{eq}$	$\emptyset$	$\{\supset_{eq}\}$	$\{\supset_{eq}\}$	$\{\supset_{eq}\}$	$\{\supset_{eq}\}$	$\{\supset_{eq}\}$	$\{\supset_{eq}\}$
$\sharp_{eq}$	$\emptyset$	$\{\sharp_{eq}\}$	$\{\supset_{eq}\}$	$\{\textcircled{O}_{eq}\}$	$\{\supset_{eq}\}$	$\{\sharp_{eq}\}$	$\{\textcircled{O}_{eq}\}$
$\textcircled{O}_{eq}$	$\emptyset$	$\{\textcircled{O}_{eq}\}$	$\{\supset_{eq}\}$	$\{\supset_{eq}, \textcircled{O}_{eq}\}$	$\{\supset_{eq}\}$	$\{\textcircled{O}_{eq}\}$	$\{\supset_{eq}, \textcircled{O}_{eq}\}$

We have (proof in Sect. 3.8.6)

Proposition 3.12. $\forall S_{eq}^1, S_{eq}^2 \in CL_{eq}. \ \forall S, S^1, S^2 \in \mathcal{P}(Val'). \ [\![S_{eq}^1, S^1, S]\!]^{6'} \wedge [\![S_{eq}^2, S^2, S]\!]^{6'} \Rightarrow [\![UL_{eq}(S_{eq}^1, S_{eq}^2), S^1 \cup S^2, S]\!]^{6'}$

3.5.4 Stabilization

Description

This stabilization function does not insure convergence to a single value but that the stabilization will take a finite number of values.

The stabilization operator combines the stabilization on the numerical domain and a strategy to bound the number of used variables. First, the stabilization operator use the function *merge* to merge nodes to keep the number of auxiliary varibles used bounded, then the only possibility for divergence is from the numerical domain, so it applies the numerical stabilization for the numerical component.

Typically, a numerical widening is a numerical stabilization, so when computing a least-fixpoint, we will apply the stabilization , with the numerical widening as numerical stabilization, to a chain which we already know is increasing thus we will insure convergence and overapproximation.

A numerical narrowing is not in general a numerical stabilization because we define stabilization to be an overapproximation of the current element of the chain, while the narrowing is an overapproximation of the intersection of all the previous elements, but in case of a decreasing chain, the numerical narrowing behaves like a numerical stabilization. So when computing a greatest-fixpoint, we will apply the stabilization with the numerical narrowing as numerical stabilization, to a chain which we already know is decreasing thus we will insure convergence and overapproximation.

Definition

Remember that we have as a condition that the numerical domain $\mathcal{D}$ should have function $\nabla^{\mathcal{D}} : (\bigcup\limits_{n\in\mathbb{N}} [0,n] \overset{total}{\to} \mathcal{D}) \to \mathcal{D}$, such that

1. $\forall w \in \mathbb{N} \overset{total}{\to} \mathcal{D}. \exists i \in \mathbb{N}.\forall i' \geq i. \nabla^{\mathcal{D}}(w \upharpoonright_{[0,i']^{total}\mathcal{D}}) = \nabla^{\mathcal{D}}(w \upharpoonright_{[0,i]^{total}\mathcal{D}})$

2. $\forall w \in \mathbb{N} \overset{total}{\to} \mathcal{D}. \forall i \in \mathbb{N}. \forall g_1 \in [\![w(i)]\!]^{\mathcal{D}}.\exists g_2 \in [\![\nabla^{\mathcal{D}}\left(w \upharpoonright_{[0,i]^{total}\mathcal{D}}\right)]\!]^{\mathcal{D}}.$
   $\forall \alpha \in TVar. g_1(\alpha) \subseteq g_2(\alpha)$

The conditions 1 is for convergence and the condition 2 is for overapproximation.

To compare we can recall the usual condition of a numerical widening $\nabla^{\sharp}$:

Definition 3.13. $\nabla^{\sharp}$ *is a widening iff*
1. $\forall w \in \mathbb{N} \overset{total}{\to} \mathcal{D}. \exists i \in \mathbb{N}.\forall i' \geq i. Y_i = Y_{i'} \ with \ Y_0 \triangleq w(0) \ and \ Y_{i+1} \triangleq Y_i \ \nabla^{\sharp} \ w(i+1)$
2. $\forall d_1, d_2 \in \mathcal{D}. \forall g_1 \in [\![d_1]\!]^{\mathcal{D}}. \forall g_2 \in [\![d_2]\!]^{\mathcal{D}}. \forall g_3 \in [\![d_1\nabla^{\sharp}d_2]\!]^{\mathcal{D}}. \forall \alpha \in TVar.$
   $g_1(\alpha), g_2(\alpha) \subseteq g_3(\alpha)$

It is very simple to see that if we take $\nabla^{\mathcal{D}}(w \upharpoonright_{[0,i]^{total}\mathcal{D}}) \triangleq Y_i$ then $\nabla^{\mathcal{D}}$ satify the conditions.

In the case of narrowing

Definition 3.14. $\triangle^{\sharp}$ *is a narrowing iff*
1. $\forall w \in \mathbb{N} \overset{total}{\to} \mathcal{D}. \exists i \in \mathbb{N}.\forall i' \geq i. Y_i = Y_{i'} \ with \ Y_0 \triangleq w(0) \ and \ Y_{i+1} \triangleq Y_i \ \nabla^{\sharp} \ w(i+1)$
2. $\forall d_1, d_2 \in \mathcal{D}. \forall g_1 \in [\![d_1]\!]^{\mathcal{D}}. \forall g_2 \in [\![d_2]\!]^{\mathcal{D}}. \forall g_3 \in [\![d_1\nabla^{\sharp}d_2]\!]^{\mathcal{D}}. \forall \alpha \in TVar.$
   $g_1(\alpha) \cap g_2(\alpha) \subseteq g_3(\alpha) \subseteq g_1(\alpha)$

Here, if we take $\nabla^{\mathcal{D}}(w \upharpoonright_{[0,i]^{total}\mathcal{D}}) \triangleq Y_i$ then $\nabla^{\mathcal{D}}$ satisfies the condition 1 of convergence, but it satisfies the condition 2 of overapproximation only when the chain w is decreasing.

We define $give\_d : AR \to \mathcal{D}, (ad, hu, ho, sn, sn^{\infty}, t, d) \mapsto d$ and $give\_d : (\mathbb{N} \overset{total}{\to} AR) \to (\mathbb{N} \overset{total}{\to} \mathcal{D}), w \mapsto (i \mapsto give\_d(w(i)))$ which are projection functions $set\_d : (AR \times \mathcal{D}) \to AR, ((ad, hu, ho, sn, sn^{\infty}, t, d), d') \mapsto (ad, hu, ho, sn, sn^{\infty}, t, d')$ which is a function to update the numerical information of an element in AR.

We can extend $\nabla^{\mathcal{D}}$ to be applied with elements of AR:

$$\nabla^{\mathcal{D}}_{AR} : (\bigcup\limits_{n\in\mathbb{N}}[0,n] \overset{total}{\to} AR) \to AR$$

112

such that

$$\forall w \in \mathbb{N} \overset{total}{\to} AR. \ \forall i \in \mathbb{N}. \nabla^{\mathcal{D}}_{AR}(w \upharpoonright_{[0,i]^{total}_{\to}\mathcal{D}}) = set\_d(w(i), \nabla^{\mathcal{D}}(give\_d(w \upharpoonright_{[0,i]^{total}_{\to}\mathcal{D}})))$$

We can easily see that we have:

1 $\forall w \in \mathbb{N} \overset{total}{\to} AR. \ \exists i \in \mathbb{N}. \forall i' \geq i. \ give\_d(\nabla^{\mathcal{D}}_{AR}(w \upharpoonright_{[0,i']^{total}_{\to}\mathcal{D}})) = give\_d(\nabla^{\mathcal{D}}_{AR}(w \upharpoonright_{[0,i]^{total}_{\to}\mathcal{D}}))$

2 $\forall w \in \mathbb{N} \overset{total}{\to} AR. \ \forall i \in \mathbb{N}. \ \forall g_1 \in [\![give\_d(w(i))]\!]^{\mathcal{D}}. \exists g_2 \in [\![give\_d(\nabla^{\mathcal{D}}_{AR}\left(w \upharpoonright_{[0,i]^{total}_{\to}AR}\right))]\!]^{\mathcal{D}}.$
   $\forall \alpha \in TVar. \ g_1(\alpha) \subseteq g_2(\alpha)$

Suppose that we have a strategy for merging:

$$\nabla^{merge} : (\bigcup_{n \in \mathbb{N}} [0,n] \overset{total}{\to} AR) \to (\bigcup_{n \in \mathbb{N}} [0,n] \overset{total}{\to} (TVar \times TVar))$$

such that:

$$\forall w \in \mathbb{N} \overset{total}{\to} AR. \ \exists A \in \mathcal{P}(TVar). \ (A \text{ is finite}) \wedge \exists i \in \mathbb{N}. \forall i' \geq i.$$
$$(used(merge(\nabla^{merge}(w \upharpoonright_{[0,i']^{total}_{\to}AR}), w(i'))) \subseteq A) \tag{3.1}$$

This stategy could for example consist on merging variables which have been used while analysing the same program point or the same part of a formula (typically the variable built for translating $\exists$).

Definition 3.15. $\nabla^{AR} : (\bigcup_{n \in \mathbb{N}} [0,n] \overset{total}{\to} AR) \to AR)$ *such that*

$$\forall w \in \mathbb{N} \overset{total}{\to} AR. \ \forall i \in \mathbb{N}.$$
$$\nabla^{AR}(w \upharpoonright_{[0,i]^{total}_{\to}AR}) \triangleq \nabla^{\mathcal{D}}_{AR}\left(\left[w \upharpoonright_{[0,i]^{total}_{\to}AR} \mid i \to merge(\nabla^{merge}(w \upharpoonright_{[0,i]^{total}_{\to}AR}), w(i)) \right] \right) \tag{3.2}$$

The definition is first merge the variable that have to be merged, then use the numerical stabilization .

Properties

Proposition 3.16. $\quad \forall w \in \mathbb{N} \overset{total}{\to} AR. \ \exists A \in \mathcal{P}(AR). \ (A \text{ is finite}) \wedge \exists i \in \mathbb{N}. \forall i' \geq i.$
$\nabla^{AR}(w \upharpoonright_{[0,i']^{total}_{\to}AR}) \in A$

This proposition says that values taken by the stabilization is a finite set (A).

Proposition 3.17. $\forall w \in \mathbb{N} \overset{total}{\rightharpoonup} AR.\ \forall i \in \mathbb{N}.\ \forall s, h, f, r.\exists g.\ s, h, f, r \in [\![w(i)]\!]' \Rightarrow s, h, g(f), r \in$
$[\![\nabla^{AR}(w \restriction_{[0,i]^{total}\rightharpoonup AR})]\!]'$

This proposition says that it is an overapproximation.

The proofs are found in Sect. 3.8.7.

3.5.5   ast

Here we present a function $ast : (AR \times AR) \to (AR \uplus \Omega)$ which we user for translation of the connective $*$. (Ω is an error result, we could have avoid this add by replacing it by any element of AR whose semantics is $\emptyset$ but it makes the analysis clearer if we add it.)

We present ast's definition via an informal 3 step algorithm.

We want ast to have the property :

Proposition 3.18. $\forall ar_0, ar_1, s, h, f_0, f_1, r_0, r_1$

$$\left(\begin{array}{l} \exists h_0, h_1.\ h_0 \sharp h_1 \wedge h = h_0 \cdot h_1 \wedge \\ s, h_0, f_0, r_0 \in [\![ar_0]\!]' \wedge \\ s, h_1, f_1, r_1 \in [\![ar_1]\!]' \wedge \\ used(ar_0) \cap used(ar_1) = \emptyset \end{array} \right) \Rightarrow s, h, f_0 \dot\cup f_1, r_0 \dot\cup r_1 \in [\![ast(ar_0, ar_1)]\!]'$$

Recall that $[\![\cdot]\!]' : AR \to MFR$ is defined in Sect. 3.4.

To help us prove the Proposition 3.18, we impose the three following restrictions:.

Restriction 1 First, we extend the usual definition of $\in$ on sets such that $\forall \alpha \in TVar.\ \alpha \notin \top$.

Next, we limit ast to be applied to elements of the domains

$$\begin{array}{lcl} VD1 & ::= & Numt \mid Truet \mid Falset \mid Oodt \mid Nilt \mid TVar \\ VD & ::= & VD1 \mid \textbf{\textcolor{red}{Loct}} \mid \textbf{\textcolor{red}{Dangling\_Loc}} \mid Loc(\mathcal{P}(\{*1, *2\}) \times VD1 \times VD1) \end{array}$$

where we use the definition, $[\![Loct, (h, \_, \_)]\!]^8 \triangleq dom(h)$, $Loct$ is introduced here to mean $Loc(\_, \top, \top)$.

This simplifies the proofs for technical reasons. The idea is that with those restrictions (which we could get from the previous definition by using some extra auxiliary variables),

the car or cdr of a *Loc* are either a variable, so we have some constrains on them, or they are an abstract value which does not represent locations and it makes things simpler for doing the *ast*.

We change the semantics of *sem∗* to read as follows:

$$sem* \triangleq \left\{ s, h, f, r \;\middle|\; \forall l \in dom(r). \begin{bmatrix} \bullet & \boldsymbol{dom(r) \cup codom(r) \subseteq dom(h)} \\ \bullet & l \notin r(l) \\ \bullet & \forall l' \in r(l) \cap dom(r).r(l') \subseteq r(l) \end{bmatrix} \right\}$$

Restriction 2 We suppose that for all non-auxiliary variables that appear in the shape graph that either (i) they are pointing to $\top$ on both sides, or (ii) they are pointing to a singleton of an auxiliary variable on both sides. (If not, we insure this using the function *extend*, being careful to use disjoint fresh variables on both sides.)

Restriction 3 Prop. 3.18, you can notice the restriction, $used(ar_0) \cap used(ar_1) = \emptyset$. The restriction is there because we restrict the application of *ast* to the cases where the auxiliary graphs are disjoint. This condition is satisfied when doing the translation of formulae, and if the function ast is needed for another use, this restriction could be satisfied by variable renaming. This restriction lets us do the first step of the algorithm:

For simplicity of presentation, we assume we apply the function *ast* on two arguments $ar_0 = (ad_0, hu_0, ho_0, sn_0, sn_0^\infty, t_0, d_0)$ and $ar_1 = (ad_1, hu_1, ho_1, sn_1, sn_1^\infty, t_1, d_1)$ and that its result is $ar_{01} = (ad_{01}, hu_{01}, ho_{01}, sn_{01}, sn_{01}^\infty, t_{01}, d_{01})$.

The first step is to compute the result part for the other components than the graphs, and the two next steps are only dealing with the graph part.

AST ALGORITHM, STEP 1: Action on the non-graph components:

Definition 3.19.
- $sn_{01} \triangleq sn_0 \cup sn_1$

- $sn_{01}^\infty \triangleq sn_0^\infty \cup sn_1^\infty$

- $hu_{01} \triangleq hu_0 \cup hu_1$ *(here, we need not check if $hu_0 \cap hu_1 = \emptyset$ because we know this holds by the restriction on used variables)*

- $ho_{01} \triangleq ho_0 \cup ho_1$

- $\forall \alpha, \beta.\ t_0(\alpha, \alpha) = t_0(\beta, \beta) = \{\ddagger_{eq}\} \Rightarrow t_{01}(\alpha, \beta) \triangleq t_1(\alpha, \beta)$

- $\forall \alpha, \beta.\ t_1(\alpha, \alpha) = t_1(\beta, \beta) = \{\ddagger_{eq}\} \wedge (t_0(\alpha, \alpha) \neq \{\ddagger_{eq}\} \vee t_0(\beta, \beta) \neq \{\ddagger_{eq}\}) \Rightarrow t_{01}(\alpha, \beta) \triangleq$ $t_0(\alpha, \beta)$

- $\forall \alpha, \beta.\ t_0(\alpha, \alpha) \neq \{\ddagger_{eq}\} \wedge t_1(\alpha, \alpha) \neq \{\ddagger_{eq}\} \Rightarrow$ *this case is impossible because by auxiliary* *variable constraints, we have* $\forall \alpha.t_0(\alpha, \alpha) = \{\ddagger_{eq}\} \vee t_1(\alpha, \alpha) = \{\ddagger_{eq}\}$

The properties and proofs are found at Sect. 3.8.9

AST ALGORITHM, STEP 2: Joining the two graphs of auxiliary variables :
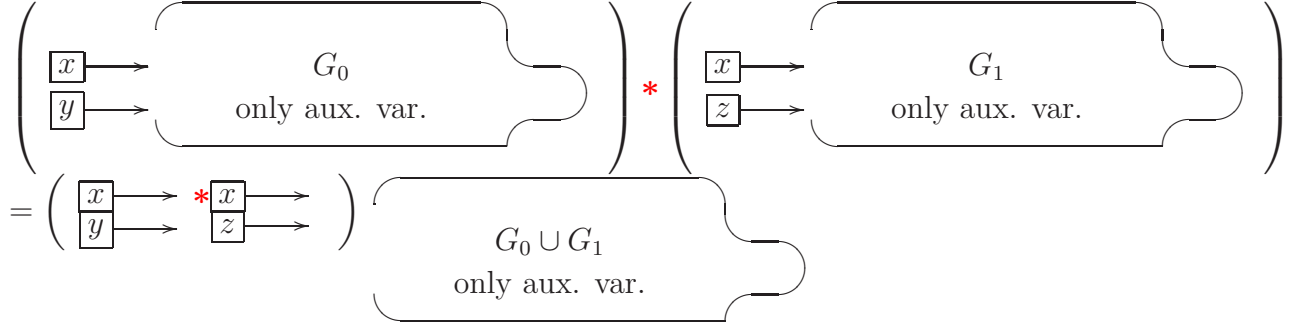
AST ALGORITHM, STEP 3: For all variables $x \in Var$, such that x points to $\{\alpha\}$ in one graph and $\{\beta\}$ in the other graph

- make x point to $\{\alpha\}$ in the result (this is an arbitrary choice, we could have taken $\{\beta\}$ instead)

- apply the procedure **do_2_var** for α, β

Procedure do_2_var

1. If one of the variables points to $\emptyset$ or $\otimes$, then it is already an error value and we return Ω. This comes directly because if we have an non-auxiliary variable, it always has a value, (because if it is not in the stack, it is assigned to $\{Oodt\}$), and if it is an auxiliary variable, it comes because we have reached it through a path starting from a non-auxiliary variable.

116

2. ElseIf on one graph, a variable points to a non-singleton set, then use the procedure **mk_sets_of_graphs** to make it a singleton set and then go the the following case:

Procedure mk_sets_of_graphs

(a) **Case non-summary node:**

$$\left(v \to A,\, B \right) \;=\; \left(v \to A \right) \;\bigcup\; \left(v \to B \right)$$
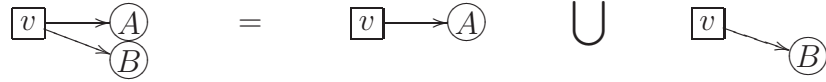
This is correct because, if v in not a summary node, $|s^+f(v)| \le 1$, thus,

$$[\![v, ar(v)]\!]^5 \wedge P$$

$$\triangleq \left\{ s, h, f, r \mid s^+f(v) \subseteq \bigcup_{vd \in ar(v)} [\![vd, (h, f, r)]\!]^8 \right\} \wedge P$$

$$= \bigcup_{vd \in ar(v)} \left(\{ s, h, f, r \mid s^+f(v) \subseteq [\![vd, (h, f, r)]\!]^8 \} \wedge P \right)$$

$$\triangleq \bigcup_{vd \in ar(v)} \left([\![v, \{vd\}]\!]^5 \wedge P \right)$$

(b) **Case of summary nodes:** Obviously, we cannot do the same as for non summary nodes because

$$\left(x \to \boxed{\alpha} \to A,\, B \right) \neq \left(x \to \boxed{\alpha} \to A \right) \cup \left(x \to \boxed{\alpha} \to B \right)$$

but we can do

$$\left(x \to \boxed{\alpha} \to A,\, B \right) \subseteq \left(\boxed{\alpha} \to A,\, B \; ; \; x \to A \right) \cup \left(\boxed{\alpha} \to A,\, B \; ; \; x \to B \right)$$
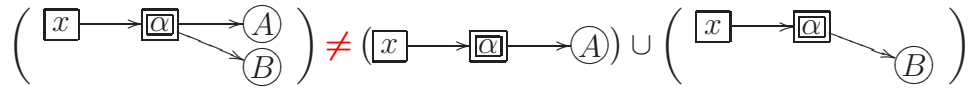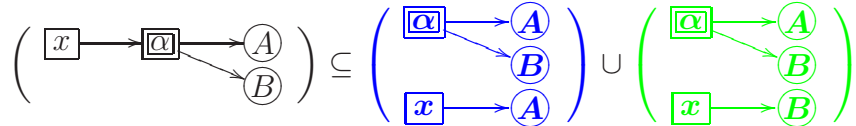
Because

$$s(x) \in f(\alpha) \subseteq (A \cup B) \Rightarrow$$
$$(s(x) \in A \wedge f(\alpha) \subseteq (A \cup B)) \vee (s(x) \in B \wedge f(\alpha) \subseteq (A \cup B))$$

(c) **Case of summary nodes in locations:** This case, in fact, never occurs for the function ast, but since we will use almost the same algorithm for the intersection operation, where this case might occur, we add it here:

$$\left(x \to \bullet \overset{1}{\leadsto} \boxed{\alpha} \to A ; \overset{2}{\leadsto} Nil , B \right) \subseteq \left(\boxed{\alpha} \to A, B ; x \to \bullet \overset{1}{\leadsto} A \overset{2}{\leadsto} Nil \right) \cup \left(\boxed{\alpha} \to A, B ; x \to \bullet \overset{1}{\leadsto} B \overset{2}{\leadsto} Nil \right)$$

117
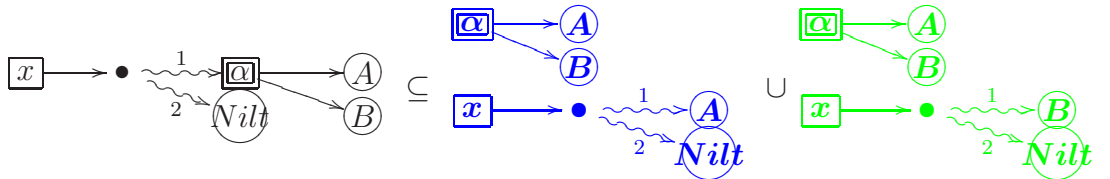
because
$$\Pi_1(h(s(x))) \in f(\alpha) \subseteq (A \cup B) \Rightarrow$$
$$\begin{pmatrix} & (\mathbf{\Pi_1(h(s(x))) \in A \wedge f(\alpha) \subseteq (A \cup B))} \\ \vee & (\mathbf{\Pi_1(h(s(x))) \in B \wedge f(\alpha) \subseteq (A \cup B))} \end{pmatrix}$$

We have extended the union on AD to $AD \uplus \Omega$ such that $\forall ad \in AD.\ \Omega \cup ar \triangleq ar \triangleq ar \cup \Omega$ and $\Omega \cup \Omega \triangleq \Omega$.

3. ElseIf both sides point to a singleton without variables or $\top$, then apply the function *basic_ast* (presented in the followin Sect. 3.5.5) on the two values. If it returns Ω, then return Ω, else

   - if the two variables are the same, then the temporary result will make this variable point to the result $(\boxed{\alpha} \longrightarrow \!\!\bigcirc\!\!A * \boxed{\alpha} \longrightarrow \!\!\bigcirc\!\!B) = \boxed{\alpha} \longrightarrow \boxed{\boldsymbol{basic\_ast}(A, B)}$

     (This case also does not appear for the case of *ast*.)

   - otherwise, the temporary result will make the two variables points to a fresh one which will points to the result

     $(\boxed{\alpha} \longrightarrow \!\!\bigcirc\!\!A * \boxed{\beta} \longrightarrow \!\!\bigcirc\!\!B) = \boxed{\alpha} \longrightarrow \boxed{\delta} \longrightarrow \boxed{\boldsymbol{basic\_ast}(A, B)}$ with $\boxed{\beta}$

4. ElseIf on one side, we point to a non-summary-node variable for which we never called the procedure **mk_sets_of_graphs**, then we call it for that pointed variable instead.

   This lead to the schema:
   $(\boxed{\alpha_1} \longrightarrow \boxed{\alpha_2} \longrightarrow \!\!\bigcirc\!\!A) * (\boxed{\beta} \longrightarrow \!\!\bigcirc\!\!B)$
   $= \boxed{\alpha_1} \longrightarrow \boxed{\alpha_2} \quad (\boxed{\alpha_2} \longrightarrow \!\!\bigcirc\!\!A * \boxed{\beta} \longrightarrow \!\!\bigcirc\!\!B)$
   $= \boxed{\alpha_1} \longrightarrow \boxed{\alpha_2} \longrightarrow \boxed{\delta} \longrightarrow \boxed{\boldsymbol{basic\_ast}(A, B)}$ with $\boxed{\beta}$

5. ElseIf on one side, we point to a non-summary-node variable for which we already called the procedure **mk_sets_of_graphs**, then we change this arrow for an arrow pointing to $\top$ and call recursively the procedure for the same variables but with the updated graph.

This leads to the schema:

$$
\begin{aligned}
&(\boxed{\alpha} \longrightarrow \boxed{\alpha_0} \longrightarrow \cdots \longrightarrow \boxed{\alpha_n}) * (\boxed{\beta} \longrightarrow \bigcirc{B})) \\
=\ &(\boxed{\alpha} \longrightarrow \boxed{\alpha_0} \longrightarrow \cdots \longrightarrow \boxed{\alpha_n} \longrightarrow \bigcirc{\top})) * (\boxed{\beta} \longrightarrow \bigcirc{B})) \\
=\ &(\boxed{\alpha} \longrightarrow \boxed{\alpha_0} \longrightarrow \cdots \longrightarrow \boxed{\alpha_n} \longrightarrow \boxed{\delta} \longrightarrow \boxed{\boldsymbol{basic\_ast}(\top, B)})
\end{aligned}
$$
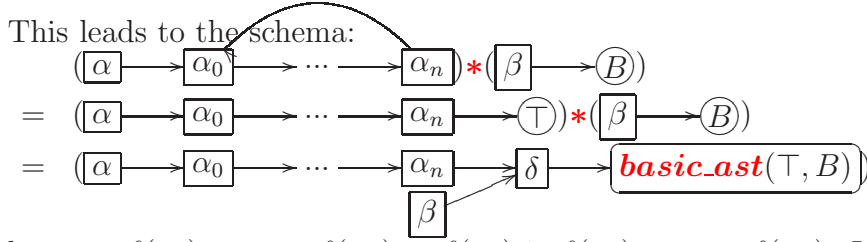
because $f(\alpha_0) \subseteq \ldots \subseteq f(\alpha_n) \subseteq f(\alpha_0)$ is $f(\alpha_0) = \ldots = f(\alpha_n)$. It is safe since we work with overapproximations and we can always assign a variable to $\top$ inducing some more approximation. Here we do not lose information because the variable α is reached from a non-auxiliary variable, thus $|f(\alpha_0)| = 1$ and we already have $|f(\alpha_n)| \leq 1$.

6. ElseIf on one side, we point to a summary-node variable then we use the Case 2b of the procedure **mk_sets_of_graphs** and call recursively the current procedure.

Formally, we prove that the "jumping of variables" is correct by using the functions *reach* and *reach*2 presented in the following Subsection 3.5.5.

Comments: The algorithm just presented is a sketch of the real algorithm:

- In reality, we work not only with the current version of the graph of auxiliary variables, but we also need to keep the two old graphs (ad_0 and ad_1) and for each variable $x \in Var$, we consult the old values, and at the end, update the result of applying the function *basic_ast*, intersect the result with what is the current value, using a function *basic_equal* presented later.

- We see later that $basic\_ast(\top, \{Dangling\_Loc\}) \triangleq \{Dangling\_Loc, Loct\}$, which means that if we know that a variable maps to a dangling location on one side and can be any value at all on the other side, then we only know that it is a location. So, when we do the union of the two auxiliary graphs, we must add $Loct$ to all sets containing $Dangling\_Loc$. This leads to imprecision if we work only with the new auxiliary graph, so we return to the old graphs to get information needed to maintain better precision.

119

basic_ast

Description   In the following table, we present the function *basic_ast*, which corresponds to *ast* at the level of PVD^+.

For simple elements (A or B), *ast* behaves as an intersection. For *Dangling*, if a location is dangling in two disjoint heaps, it is still dangling in the union of those heaps. If we have a location dangling in one heap and allocated in the other, then it is allocated in the union of them. One might worry that the allocated location in the union does not have the same properties in the union than it has in only a part of it. This problem does not occur because in $Loc(\_, s1, s2)$ we forbid $s1$ and $s2$ to be *Loc* or dangling, so it is either a simple type (whose semantics is not affected by the union of heaps) or an auxiliary variable (which is not affected, since we have the restriction on disjoint sets of used auxiliary variables). Lastly, we cannot have an allocated location in two disjoint heaps, so *basic_ast* must return Ω for any combination of allocated locations.

Definition: $basic\_ast \; : \; (PVD^+ \times PVD^+) \rightharpoonup (PVD^+ \uplus \Omega)$

We define the partial function *basic_ast* as in the following table:

Definition 3.20. *We write Dgt for Dangling_Loc.*

Let $A, B \in \{Nilt, Truet, Falset, Oodt, Numt\}$ with $A \neq B$,

basic_ast	$\{A\}$	$\{Dgt\}$	$\{Loct\}$	$\{Dgt, Loct\}$	$\{Loc(...)\}$	$\top$
$\{A\}$	$\{A\}$	Ω	Ω	Ω	Ω	$\{A\}$
$\{B\}$	Ω	Ω	Ω	Ω	Ω	$\{B\}$
$\{Dgt\}$	Ω	$\{Dgt\}$	$\{Loct\}$	$\{Dgt, Loct\}$	$\{Loc(...)\}$	$\{Dgt, Loct\}$
$\{Loct\}$	Ω	$\{Loct\}$	Ω	$\{Loct\}$	Ω	$\{Loct\}$
$\{Dgt, Loct\}$	Ω	$\{Dgt, Loct\}$	$\{Loct\}$	$\{Dgt, Loct\}$	$\{Loc(...)\}$	$\{Dgt, Loct\}$
$\{Loc(...)\}$	Ω	$\{Loc(...)\}$	Ω	$\{Loc(...)\}$	Ω	$\{Loc(...)\}$
$\top$	$\{A\}$	$\{Dgt, Loct\}$	$\{Loct\}$	$\{Dgt, Loct\}$	$\{Loc(...)\}$	$\top$

and we have $basic\_ast(\Omega, \_) = basic\_ast(\_, \Omega) = basic\_ast(\emptyset, \_) = basic\_ast(\_, \emptyset) = \Omega$.

Properties

Proposition 3.21.

$$\bigcup_{vd\in vd_0} [\![vd,(s,h_0,f_0,r_0)]\!]^8 \cap \bigcup_{vd\in vd_1} [\![vd,(s,h_1,f_1,r_1)]\!]^8 \subseteq \bigcup_{vd\in basic\_ast(vd_0,vd_1)} [\![vd,(s,h,f,r)]\!]^8$$

if we define $\bigcup_{vd\in\Omega} [\![vd,(s,h,f,r)]\!]^8 \triangleq \emptyset$ *and* $\bigcup_{vd\in\top} [\![vd,(s,h,f,r)]\!]^8 \triangleq MFR$

Lemma 3.22.   • $[\![Nilt,(s_0,h_0,f_0,r_0)]\!]^8 = [\![Nilt,(s,h,f,r)]\!]^8$

- $[\![Truet,(s_0,h_0,f_0,r_0)]\!]^8 = [\![Truet,(s,h,f,r)]\!]^8$

- $[\![Falset,(s_0,h_0,f_0,r_0)]\!]^8 = [\![Falset,(s,h,f,r)]\!]^8$

- $[\![Oodt,(s_0,h_0,f_0,r_0)]\!]^8 = [\![Oodt,(s,h,f,r)]\!]^8$

- $[\![Numt,(s_0,h_0,f_0,r_0)]\!]^8 = [\![Numt,(s,h,f,r)]\!]^8$

- $[\![Dgt,(s_0,h_0,f_0,r_0)]\!]^8 = Loc \setminus dom(h_0) \supseteq Loc \setminus dom(h) = [\![Dgt,(s,h,f,r)]\!]^8$

- $[\![\alpha,(s_0,h_0,f_0,r_0)]\!]^8 = f_0(\alpha) \subseteq f(\alpha) = [\![\alpha,(s,h,f,r)]\!]^8$

The proofs are found in Sect. 3.8.8.

Reaching functions

Definition 3.23. *For a graph* $G \in \mathcal{P}(TVar \times PDV^+)$, $\alpha \in TVar$, *we define recursively* $reach(G,\alpha) \triangleq reachrec(G,\{\alpha\},\alpha)$, *with*

conditions	$reachrec(G,V,\alpha)$
$\alpha \notin dom(G)$	$(\alpha,\top)$
$(\alpha,\{vd\}) \in G, vd \notin TVar$	$(\alpha,\{vd\})$
$(\alpha,\{\beta\}) \in G, \beta \in V, (\beta,\{\beta'\}) \in G$	$(\beta,(\beta',\alpha))$
$(\alpha,\{\beta\}) \in G, \beta \notin V \cup sn_{01}, \beta \in TVar$	$reachrec(G,V\cup\{\beta\},\beta)$
$(\alpha,\{\beta\}) \in G, \beta \in sn_{01}$	$(\alpha,\{\beta\})$

121

This definition, from a graph G and an auxiliary variable α, gives either the last edge reaches a summary node; or in the case of a cycle, it gives the two edges before and after the point where we "enter" the cycle; or else, it give the last edge ending to a leaf.

Proposition 3.24. $\forall G \in \mathcal{P}(TVar \times PVD^+). \ \forall s, h, f, r. \ (\forall (\alpha, S) \in G. \ s, h, f, r \in [\![\alpha, S]\!]^5) \Rightarrow$ $\forall \alpha \in TVar. \ \forall (\alpha', S) = reach(\alpha).$

$$
\left(\begin{array}{l} \left(f(\alpha) \subseteq f(\alpha') \wedge \left((\alpha', S) \in G \vee \exists \beta_1, \beta_2. \left(\begin{array}{l} S = (\beta_1, \beta_2) \wedge \\ (\alpha', \{\beta_1\}) \in G \wedge \\ (\beta_2, \{\alpha'\}) \in G \wedge \\ f(\alpha') = f(\beta_1) = f(\beta_2) \end{array} \right) \right) \right) \\ \wedge (\alpha \notin sn_{01} \Rightarrow \alpha' \notin sn_{01}) \end{array} \right)
$$

Definition 3.25. *For a graph $g \in (Var \times PDV^+)$ and a graph $G \in \mathcal{P}(TVar \times PDV^+)$, $x \in Var$, we define the partial function $reach2(g, G, x)$ such that*

- $\forall vd. \ (x, vd) \notin g$ *then* $reach2(g, G, x) \triangleq (x, \top)$

- *if* $(x, vd) \in g$ *and* $\forall \alpha. \ \alpha \notin vd$ *then* $reach2(g, G, x) \triangleq (x, vd)$

- *if* $(x, vd) \in g$ *and* $\exists \alpha \in vd \cap sn_{01}$ *then* $reach2(g, G, x) \triangleq (x, vd)$

- *if* $(x, \{\alpha\}) \in g$ *and* $\alpha \notin sn_{01}$ *then* $reach2(\alpha, G) \triangleq reach(\alpha, G)$

Proposition 3.26. $\forall g \in \mathcal{P}(Var \times PVD^+). \ G \in \mathcal{P}(TVar \times PVD^+). \ \forall s, h, f, r. \ (\forall (v, S) \in$ $g \uplus G. \ s, h, f, r \in [\![v, S]\!]^5) \Rightarrow \forall (v, S) \in reach2(g, G, x).$

$$
\left(s(x) = s^+ f(v) \wedge \left(S = \top \vee (v, S) \in g \uplus G \vee \exists \beta_1, \beta_2. \left(\begin{array}{l} S = (\beta_1, \beta_2) \wedge \\ (v, \{\beta_1\}) \in G \wedge \\ (\beta_2, \{v\}) \in G \wedge \\ s(x) = f(\delta_1) = f(\delta_2) \end{array} \right) \right) \right)
$$

The proof are at Sect. 3.8.11.

3.5.6 Intersection

The algorithm for the intersection procedure is similar to the graph part of the *ast* procedure presented in the previous subsection, except that instead of using the function *basic_ast*,

it uses the function *basic_equal*, defined below. (Recall that the semantics of $*$ is like the semantics of $\wedge$ plus some extra constraints about heap domains.)

Also, the intersection procedure does not require that the two elements to intersect have disjoint sets of used auxiliary variables. This restriction was used for technical simplicity in the soundness proof but also because otherwise to catch the ast information on auxiliary variables not reachable from non-auxiliary variables would have been very costly. Here, we do not have those constraints on the heap domain and we can just do a big intersection of all the information (the semantics of the graphs being already a big intersection on the semantics of each arrow within the graph). The procedure **do_2_var**, used in the definition of *ast* and reused here, will be in fact a procedure, "work on two arrows", that will be used at the beginning of the arrows starting from the same variable; we will also keep a set, *eq*, of equalities that need to be treated. We will also have rules that make things more efficient than for the definition of *ast*: if we have in both sides the same variable pointing to the same value, then we directly have that this variable points to that value.

Proposition 3.27. $\forall ar_0, ar_1, s, h, f, r.\ \exists g.$

$$
\left(
\begin{array}{l}
s, h, f, r \in [\![ar_0]\!]' \wedge \\
s, h, f, r \in [\![ar_1]\!]'
\end{array}
\right) \Rightarrow s, h, g(f), r \in [\![inter(ar_0, ar_1)]\!]'
$$

We need f to be the same for ar_0 and ar_1 because we do not restrict the auxiliary variables used. But if we need a theorem on *inter* for different f, Prop. 3.27 implies

Corollary 3.28. $\forall ar_0, ar_1, s, h, f_0, f_1, r_0, r_1.\ \exists g.$

$$
\left(
\begin{array}{l}
s, h, f_0, r_0 \in [\![ar_0]\!]' \wedge \\
s, h, f_1, r_1 \in [\![ar_1]\!]' \wedge \\
used(ar_0) \cap use(ar_1) = \emptyset \wedge \\
dom(f_0) \cap dom(f_1) = \emptyset
\end{array}
\right) \Rightarrow s, h, g(f_0 \dot{\cup} f_1), r_0 \dot{\cup} r_1 \in [\![inter(ar_0, ar_1)]\!]'
$$

Intersection for the non-graph components

1. $[\![ar_\top \sqcap ar_2]\!]' \triangleq [\![ar_2]\!]' = [\![ar_\top]\!]' \cap [\![ar_2]\!]'$

2. $\llbracket ar_1 \sqcap ar_\top \rrbracket' \triangleq \llbracket ar_1 \rrbracket' = \llbracket ar_1 \rrbracket' \cap \llbracket ar_\top \rrbracket'$

3.
$$
\begin{aligned}
& \llbracket hu_1 \sqcap hu_2 \rrbracket^1 \\
\triangleq\ & \llbracket hu_1 \cup hu_2 \rrbracket^1 \\
=\ & \bigcap_{\alpha \in hu_1 \cup hu_2} \{s, h, f, r \mid f(\alpha) \cap dom(h) \neq \emptyset\} \\
=\ & \bigcap_{\alpha \in hu_1} \{s, h, f, r \mid f(\alpha) \cap dom(h) \neq \emptyset\} \cap \bigcap_{\alpha \in hu_2} \{s, h, f, r \mid f(\alpha) \cap dom(h) \neq \emptyset\} \\
=\ & \llbracket hu_1 \rrbracket^1 \cap \llbracket hu_2 \rrbracket^1
\end{aligned}
$$

4.
$$
\begin{aligned}
& \llbracket ho_1 \sqcap \texttt{full} \rrbracket^{1'} \\
\triangleq\ & \llbracket ho_1 \rrbracket^{1'} \\
=\ & \llbracket ho_1 \rrbracket^{1'} \cap \llbracket \texttt{full} \rrbracket^{1'}
\end{aligned}
$$

5.
$$
\begin{aligned}
& \llbracket \texttt{full} \sqcap ho_2 \rrbracket^{1'} \\
\triangleq\ & \llbracket ho_2 \rrbracket^{1'} \\
=\ & \llbracket \texttt{full} \rrbracket^{1'} \cap \llbracket ho_2 \rrbracket^{1'}
\end{aligned}
$$

6.
$$
\begin{aligned}
& \llbracket ho_1 \sqcap ho_2 \rrbracket^{1'} \\
\triangleq\ & \llbracket ho_2 \rrbracket^{1'} \\
=\ & \{s, h, f, r \mid dom(h) \subseteq \bigcup_{\alpha \in ho_2} f(\alpha)\} \\
\supseteq\ & \{s, h, f, r \mid dom(h) \subseteq (\bigcup_{\alpha \in ho_1} f(\alpha)) \cap (\bigcup_{\alpha \in ho_2} f(\alpha))\} \\
=\ & \{s, h, f, r \mid dom(h) \subseteq \bigcup_{\alpha \in ho_1} f(\alpha)\} \cap \{s, h, f, r \mid dom(h) \subseteq \bigcup_{\alpha \in ho_2} f(\alpha)\} \\
=\ & \llbracket ho_1 \rrbracket^{1'} \cap \llbracket ho_2 \rrbracket^{1'}
\end{aligned}
$$

Notice that this approximation is a choice; we could choose also $ho_1 \sqcap ho_2 \triangleq ho_1$.

7.
$$
\begin{aligned}
& \llbracket sn_1 \sqcap sn_2 \rrbracket^2 \\
\triangleq\ & \llbracket sn_1 \cap sn_2 \rrbracket^2 \\
=\ & \{s, h, f, r \mid \forall \alpha \in TVar \setminus (sn_1 \cap sn_2).|f(\alpha)| \leq 1\} \\
=\ & \{s, h, f, r \mid \forall \alpha \in TVar \setminus sn_1.|f(\alpha)| \leq 1\} \\
& \cap \{s, h, f, r \mid \forall \alpha \in TVar \setminus sn_2.|f(\alpha)| \leq 1\} \\
=\ & \llbracket sn_1 \rrbracket^2 \cap \llbracket sn_2 \rrbracket^2
\end{aligned}
$$

8.
$$
\begin{aligned}
& \llbracket sn_1^\infty \sqcap sn_2^\infty \rrbracket^{2'} \\
\triangleq\ & \llbracket sn_1^\infty \cap sn_2^\infty \rrbracket^{2'} \\
=\ & \{s, h, f, r \mid \forall \alpha \in TVar \setminus (sn_1^\infty \cap sn_2^\infty).\ f(\alpha) \text{ is finite}\} \\
=\ & \{s, h, f, r \mid \forall \alpha \in TVar \setminus sn_1^\infty.\ f(\alpha) \text{ is finite}\} \\
& \cap \{s, h, f, r \mid \forall \alpha \in TVar \setminus sn_2^\infty.\ f(\alpha) \text{ is finite}\} \\
=\ & \llbracket sn_1^\infty \rrbracket^{2'} \cap \llbracket sn_2^\infty \rrbracket^{2'}
\end{aligned}
$$

124

9.
$$
\begin{aligned}
& [\![t_1 \sqcap t_2]\!]^3 \\
\triangleq\ & [\![t_1 \dot{\sqcap} t_2]\!]^3 \\
=\ & \bigcap_{(\alpha_1,\alpha_2)\in TVar \times TVar} \left\{ s,h,f,r \mid [\![t_1(\alpha_1,\alpha_2) \cap t_2(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \right\} \\
\supseteq\ & \bigcap_{(\alpha_1,\alpha_2)\in TVar \times TVar} \left\{ s,h,f,r \ \middle|\ \begin{array}{l} [\![t_1(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \\ \wedge [\![t_2(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \end{array} \right\} \\
=\ & \bigcap_{(\alpha_1,\alpha_2)\in TVar \times TVar} \left(\begin{array}{l} \left\{ s,h,f,r \mid [\![t_1(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \right\} \\ \cap \left\{ s,h,f,r \mid [\![t_2(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \right\} \end{array} \right) \\
=\ & \bigcap_{(\alpha_1,\alpha_2)\in TVar \times TVar} \left\{ s,h,f,r \mid [\![t_1(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \right\} \\
& \cap \bigcap_{(\alpha_1,\alpha_2)\in TVar \times TVar} \left\{ s,h,f,r \mid [\![t_2(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \right\} \\
=\ & [\![t_1]\!]^3 \cap [\![t_2]\!]^3
\end{aligned}
$$

Definition of $basic\_equal$: $(PVD^+ \times PVD^+ \times \mathcal{P}(PVD^+ \times PVD^+)) \rightharpoonup ((PVD^+ \times \mathcal{P}(TVar \times TVar)) \uplus \Omega)$

We define the partial function $basic\_equal$ as in the following table:

Definition 3.29. *We write Dgt for $Dangling\_Loc$;*

let $A, B \in \{Nilt, Truet, Falset, Oodt, Numt\}$ with $A \neq B$.

$basic\_equal$	$\{A\}$	$\{Dgt\}$	$\{Loct\}$	$\{Dgt, Loct\}$	$\{Loc(A_1, l_1^1, l_1^2)\}$
$\{A\}$	$\{A\}$	Ω	Ω	Ω	Ω
$\{B\}$	Ω	Ω	Ω	Ω	Ω
$\{Dgt\}$	Ω	$\{Dgt\}$	Ω	$\{Dgt\}$	Ω
$\{Loct\}$	Ω	Ω	$\{Loct\}$	$\{Loct\}$	$\{Loc(A_1, l_1^1, l_1^2)\}$
$\{Dgt, Loct\}$	Ω	$\{Dgt\}$	$\{Loct\}$	$\{Dgt, Loct\}$	$\{Loc(A_1, l_1^1, l_1^2)\}$
$\{Loc(A_2, l_2^1, l_2^2)\}$	Ω	Ω	$\{Loc(A_2, l_2^1, l_2^2)\}$	$\{Loc(A_2, l_2^1, l_2^2)\}$	*Case_loc_loc*

In the table, we did not give the second term of the result (the eq part) because it is the same

as the third component of the input entry, except for *Case_loc_loc* *which definition is*

$$basic\_equal(\{Loc(A_1, l_1^1, l_1^2)\}, \{Loc(A_2, l_2^1, l_2^2)\}, eq) \triangleq$$

$$
\begin{cases}
(\{Loc(A_1, l_1^1, l_1^2)\}, eq) & if \quad (A_1, l_1^1, l_1^2) = (A_2, l_2^1, l_2^2) \\
(Loc(A_1 \cup A_2, l_{12}^1, l_{12}^2), eq') \text{ or } \Omega & else \\
where\ l_{12}^1, l_{12}^2\ and\ eq'\ are\ computed\ by & \\
\quad \begin{bmatrix} \bullet \ eq' = eq \\ \bullet \ if\ l_1^1 = l_2^1\ then\ let\ l_{12}^1 = l_1^1 \\ \quad elseif\ l_1^1 \in TVar\ then\ l_{12}^1 = l_1^1\ and\ eq' = eq' \cup \{(l_1^1, l_2^1)\} \\ \quad elseif\ l_2^1 \in TVar\ then\ l_{12}^1 = l_2^1\ and\ eq' = eq' \cup \{(l_1^1, l_2^1)\} \\ \quad else\ \Omega \\ \bullet \ if\ l_1^2 = l_2^2\ then\ let\ l_{12}^2 = l_1^2 \\ \quad elseif\ l_1^2 \in TVar\ then\ l_{12}^2 = l_1^2\ and\ eq' = eq' \cup \{(l_1^2, l_2^2)\} \\ \quad elseif\ l_2^2 \in TVar\ then\ l_{12}^2 = l_2^2\ and\ eq' = eq' \cup \{(l_1^2, l_2^2)\} \\ \quad else\ \Omega \end{bmatrix} &
\end{cases}
$$

We also have

- $basic\_equal(\Omega, \_) = basic\_equal(\_, \Omega) = basic\_equal(\emptyset, \_) = basic\_equal(\_, \emptyset) = \Omega$

- $basic\_equal(\top, S) = basic\_equal(S, \top) = S$

- *(as shown in the table)* $basic\_equal(S, S) = S$

Proposition 3.30. *We define* $\bigcup_{vd \in \Omega} [\![vd, (s, h, f, r)]\!]^8 \triangleq \emptyset$ *and* $\bigcup_{vd \in \top} [\![vd, (s, h, f, r)]\!]^8 \triangleq MFR$
If

- $\forall (vd_0, vd_1) \in eq.\ vd_0 \in TVar \Rightarrow f(vd_0) \in [\![vd_1, (s, h, f, r)]\!]^8$

- $\forall (vd_0, vd_1) \in eq.\ vd_1 \in TVar \Rightarrow f(vd_1) \in [\![vd_0, (s, h, f, r)]\!]^8$

- *if* vd_0 *or* $vd_1 = \{Loc(A, l^1, l^2)\}$ *then* $l^1, l^2 \notin sn$

- $basic\_equal(vd_0, vd_1, eq) = (vd_{01}, eq')$

then

- $\bigcup_{vd \in vd_0} [\![vd, (s, h, f, r)]\!]^8 \cap \bigcup_{vd \in vd_1} [\![vd, (s, h, f, r)]\!]^8 \subseteq \bigcup_{vd \in vd_{01}} [\![vd, (s, h, f, r)]\!]^8$

126

- $\forall (vd_0, vd_1) \in eq'.\ vd_0 \in TVar \Rightarrow f(vd_0) \in [\![vd_1, (s, h, f, r)]\!]^8$

- $\forall (vd_0, vd_1) \in eq'.\ vd_1 \in TVar \Rightarrow f(vd_1) \in [\![vd_0, (s, h, f, r)]\!]^8$

For the proof, see Sect. 3.8.10.

When we use this function, the Case_loc_loc of intersection of two $Loc(..)$ increases the number of equalities to treat (in the argument eq), but since we will update the values, the number of $Loc(...)$ to be treated will decrease and the procedure terminate.

3.6 Translation of formulae

In this section, we present the translation of the connectives of the logic. They are theoretical translations and do not necessarily correspond exactly to what is implemented because we can improve the precision of the translation by splitting subcases while it makes the proofs more complicated.

3.6.1 Properties of the translation

Let $T' \in BI^{\mu\nu} \to AR$ be the translation function from formulae of the logic to elements of our language, and $T \in (AR \times BI^{\mu\nu}) \to AR$ be an auxiliary translation function which takes an element of the language and refines it with the translation of a formula.

We define $T'(P) \triangleq T(ar_\top, P)$ where $ar_\top$ is the top element of our language.

We prove that, $\boxed{\forall P \in BI^{\mu\nu}.[\![P]\!] \subseteq [\![T'(P)]\!]}$ which is $\forall P \in BI^{\mu\nu}.[\![P]\!] \subseteq [\![T(ar_\top, P)]\!]$

To prove this as a consequence, we proved :

Theorem 3.31. $\quad \forall P \in BI^{\mu\nu}.\forall ar \in AR.\forall (s, h, f, r) \in MFR.\exists g : F \to F.$
$\quad s, h, f, r \in [\![ar]\!]' \wedge \bar{s}, h \in [\![P]\!] \Rightarrow s, h, g(f), r \in [\![T(ar, P)]\!]'$

The function g is here to allow us to do auxiliary variable renaming, to allow use of function extension, or to build summary nodes.

This directly implies that (because $\forall s, h, f, r.\ s, h, f, r \in [\![ar_\top]\!]'$)
$\forall P \in BI^{\mu\nu}.\forall ar \in AR.\forall (s, h, f, r) \in MFR.\exists g : F \to F.$
$\bar{s}, h \in [\![P]\!] \Rightarrow s, h, g(f), r \in [\![T(ar_\top, P)]\!]'$
which itself implies that (because $\forall s, h, f, r, ar.\ s, h, f, r \in [\![ar]\!]' \Rightarrow \bar{s}, h \in [\![ar]\!]$)

$$\forall P \in BI^{\mu\nu}.\forall ar \in AR.\forall \bar{s}, h \in S \times H.\ \bar{s}, h \in [\![P]\!] \Rightarrow \bar{s}, h \in [\![T(ar_\top, P)]\!]$$

which is what we want.

Translation of *ast*

Definition 3.32. $T(ar, F * G) \triangleq inter(ar, ast(T(ar\top, F), T(ar\top, G)))$

Th. 3.31 for Def. 3.32. We want to prove that: $\forall s, h, f, r.\ \exists g.$

$$\left(\begin{array}{l} \exists h_0, h_1.\ h_0 \sharp h_1.,\ h = h_0 \cdot h_1. \\ s, h, f, r \in [\![ar]\!]' \wedge \\ \bar{s}, h_0 \in [\![P]\!] \wedge \\ \bar{s}, h_1 \in [\![Q]\!] \end{array} \right) \Rightarrow s, h, g(f), r \in [\![T(ar, P * Q)]\!]'$$

By recursion we have :

$$\forall s, h_0, f, r.\ \exists g_0.\bar{s}, h_0 \in [\![P]\!] \Rightarrow s, h_0, g_0(f), r \in [\![T(ar_\top, P)]\!]'$$

and

$$\forall s, h_1, f, r.\ \exists g_1.\bar{s}, h_1 \in [\![Q]\!] \Rightarrow s, h_1, g_1(f), r \in [\![T(ar_\top, Q)]\!]'$$

which gives us

$$\forall s, h, f, r.\ \exists g_0, g_1.$$
$$\left(\begin{array}{l} s, h, f, r \in [\![ar]\!]' \wedge \\ \bar{s}, h_0 \in [\![P]\!] \wedge \\ \bar{s}, h_1 \in [\![Q]\!] \end{array} \right) \Rightarrow \left(\begin{array}{l} s, h, f, r \in [\![ar]\!]' \wedge \\ s, h_0, g_0(f), r \in [\![T(ar_\top, P)]\!]' \wedge \\ s, h_1, g_1(f), r \in [\![T(ar_\top, Q)]\!]' \end{array} \right)$$

with Prop. 3.18 on the function *ast* we have

$$\forall s, h, f, r.\ \exists g_0, g_1.$$
$$\left(\begin{array}{l} s, h, f, r \in [\![ar]\!]' \wedge \\ \bar{s}, h_0 \in [\![P]\!] \wedge \\ \bar{s}, h_1 \in [\![Q]\!] \end{array} \right) \Rightarrow \left(\begin{array}{l} s, h, f, r \in [\![ar]\!]' \wedge \\ s, h, g_0(f) \dot{\cup} g_1(f), r_0 \dot{\cup} r_1 \in [\![ast(T(ar_\top, P), T(ar_\top, Q))]\!] \end{array} \right)$$

Assuming that we translated P and Q such that they have no common auxiliary variables with ar and between them. From Prop. 3.27 on the function *inter* we have

$$\forall s, h, f, r.\ \exists g.$$
$$\left(\begin{array}{l} \exists h_0, h_1.\ h_0 \sharp h_1.,\ h = h_0 \cdot h_1. \\ s, h, f, r \in [\![ar]\!]' \wedge \\ \bar{s}, h_0 \in [\![P]\!] \wedge \\ \bar{s}, h_1 \in [\![Q]\!] \end{array} \right) \Rightarrow s, h, g(f), r \in [\![inter(ar, ast(T(ar_\top, P), T(ar_\top, Q)))]\!]'$$

having $g(f) = f \dot{\cup} g_0(f) \dot{\cup} g_1(f)$

$\square$

Translation of $\twoheadrightarrow\!\!*$

We cannot write a general precise translation of $\twoheadrightarrow\!\!*$ because

$\llbracket P\twoheadrightarrow\!\!* Q\rrbracket = \{s,h \mid \forall h'. \text{ if } h\sharp h' \text{ and } s,h' \in \llbracket P\rrbracket \text{ then } s,h\cdot h' \in \llbracket Q\rrbracket\}$

so if $P \equiv \texttt{false}$, then $P\twoheadrightarrow\!\!* Q \equiv \texttt{true}$

So, since we do overapproximation, we cannot insure that the formula P is not $\texttt{false}$, then the safe translation of $P\twoheadrightarrow\!\!* Q$ is $ar_\top$.

In fact, we in some cases translate into a sure $\texttt{false}$ or a sure $\texttt{true}$.

Translation of $[\,/\,]$

Definition 3.33. *When $x \notin Var(E)$: $T(ar, P[E/x]) = T(ar, \exists x.\ x = E \wedge P)$*

Th. 3.31 for Def. 3.33. When $x \notin Var(E)$
$$\begin{aligned}
\llbracket P[E/x]\rrbracket &= \{s,h \mid [s \mid x \rightarrow \llbracket E\rrbracket^s], h \in \llbracket P\rrbracket\} \\
&= \{s,h \mid [s \mid x \rightarrow \llbracket E\rrbracket^s], h \in \llbracket x = E \wedge P\rrbracket\} \\
&= \{s',h \mid \exists s.s' - x = s - x \wedge s,h \in \llbracket x = E \wedge P\rrbracket\} \qquad\square \\
&= \{[s \mid x \rightarrow v],h \mid s,h \in \llbracket y = x \wedge P\rrbracket\} \cup \{s - x, h \mid s,h \in \llbracket x = E \wedge P\rrbracket\}\} \\
&= \llbracket \exists x.\ x = E \wedge P\rrbracket
\end{aligned}$$

Definition 3.34. $T(ar, P[x/x]) = T(ar, x = x \wedge P)$

Th. 3.31 for Def. 3.34.
$$\begin{aligned}
\llbracket P[x/x]\rrbracket &= \{s,h \mid [s \mid x \rightarrow s(x)], h \in \llbracket P\rrbracket\} \\
&= \{s,h \mid [s \mid x \rightarrow s(x)], h \in \llbracket x = x \wedge P\rrbracket\} \qquad\square \\
&= \llbracket x = x \wedge P\rrbracket
\end{aligned}$$

As presented in Sect. 2, we can use some equivalences:

- If P does not contain any v-variable or fixpoint or postponed substitution, then
  $P[E/x] \equiv P\{E/x\} \wedge is(E)$.

- If P is v-closed and if $x_1 \notin Var(E)$ and $x_1 \neq x_2$, then:
  $(\exists x_1.P)[E/x_2] \equiv \exists x_1.(P[E/x_2])$.

- $(\exists x.P)[E/x] \equiv (\exists x.P) \wedge is(E)$.

- $(A \lor C)[E/x] \equiv (A[E/x]) \lor (C[E/x])$.

- If $y \notin Var(P)$, then

  $(\mu X_v.P)[y/x] \equiv (\mu X_v.P\{[y/x]\}) \land is(y)$

  $(\nu X_v.P)[y/x] \equiv (\nu X_v.P\{[y/x]\}) \land is(y)$.

For other cases, we cannot directly define the translation, the precision will depend on the precision of the numerical domain and in particular, on the precision of the reverse function. For example, we cannot give a translation of $P[x+1/x]$ for arbitrary P if the previous cases cannot be applied: we translate P, and we see that x has to be a number in the translation, but the precision depends on the capability of the numerical domain to reverse the operation $x + 1$.

3.6.2 Translation of $\land$

Definition 3.35. $T(ar, P_1 \land P_2) = T(T(ar, P_1), P_2)$

Th. 3.31 for Def. 3.35. By induction we have, $\forall s, h, f, r.\ \exists g.$

$$s, h, f, r \in [\![ar]\!]' \land \bar{s}, h \in [\![P_1]\!] \Rightarrow s, h, g(f), r \in [\![T(ar, P_1)]\!]'$$

$$s, h, g(f), r \in [\![T(ar, P_1)]\!]' \land \bar{s}, h \in [\![P_2]\!] \Rightarrow s, h, g(g(f)), r \in [\![T(T(ar, P_1), P_2)]\!]'$$

using those two results, we get that

$$s, h, f, r \in [\![ar]\!]' \land \bar{s}, h \in [\![P_1]\!] \land s, h \in [\![P_2]\!] \Rightarrow s, h, g(g(f)), r \in [\![T(T(ar, P_1), P_2)]\!]'$$

which is as expected: $\forall s, h, f, r\ .\exists g'.\ (g' = g \circ g)$

$$s, h, f, r \in [\![ar]\!]' \land \bar{s}, h \in [\![P_1 \land P_2]\!] \Rightarrow s, h, g'(f), r \in [\![T(T(ar, P_1), P_2)]\!]'$$

$\square$

3.6.3 Translation of $\vee$

Definition 3.36. $T(ar, P_1 \vee P_2) = union(T(ar, P_1), T(ar, P_2))$

Th. 3.31 for Def. 3.36. By recurrence we have $\forall s, h, f, r.\exists g.$,

$$s, h, f, r \in [\![ar]\!]' \wedge \bar{s}, h \in [\![P_1]\!] \Rightarrow s, h, g(f), r \in [\![T(ar, P_1)]\!]'$$

and

$$s, h, f, r \in [\![ar]\!]' \wedge \bar{s}, h \in [\![P_2]\!] \Rightarrow s, h, g(f), r \in [\![T(ar, P_2)]\!]'$$

so we get

$$s, h, f, r \in [\![ar]\!]' \wedge (\bar{s}, h \in [\![P_1]\!] \vee \bar{s}, h \in [\![P_2]\!]) \Rightarrow$$
$$(s, h, g(f), r \in [\![T(ar, P_1)]\!]' \vee s, h, g(f), r \in [\![T(ar, P_2)]\!]')$$

which is

$$s, h, f, r \in [\![ar]\!]' \wedge \bar{s}, h \in [\![P_1 \vee P_2]\!] \Rightarrow s, h, g(f), r \in ([\![T(ar, P_1)]\!]' \cup [\![T(ar, P_2)]\!]')$$

and by Prop. 3.3

$$s, h, f, r \in [\![ar]\!]' \wedge \bar{s}, h \in [\![P_1 \vee P_2]\!] \Rightarrow s, h, g(f), r \in ([\![union(T(ar, P_1), T(ar, P_2))]\!]')$$

$\square$

3.6.4 Translation of $\exists$

Definition 3.37. $T(ar, \exists x.\ P) = class(x, T(class(x, ar), x = x \wedge P))$

Define

(* $class(x, D)$ is removing all information about $x \in Var$ in D *)

$$
\begin{aligned}
class \quad &: \quad \boldsymbol{Var \rightarrow AR \rightarrow AR} \\
class\,(x, (ad, hu, ho, sn, sn^{\infty}, t, d)) \quad &\triangleq \quad ([ad \mid x \rightarrow \top], hu, ho, sn, sn^{\infty}, t, d) \\
&\qquad \text{when } ad(x) \neq \emptyset \\
class\,(x, ([ad \mid x \rightarrow \emptyset], hu, ho, sn, sn^{\infty}, t, d)) \quad &\triangleq \quad ([ad \mid x \rightarrow \emptyset], hu, ho, sn, sn^{\infty}, t, d) \\
class \quad &: \quad \boldsymbol{(Var \times MFR) \rightarrow \mathcal{P}(MFR)} \\
class(x, (s, h, f, r)) \quad &\triangleq \quad \{s', h, f, r \mid [s' \mid x \rightarrow \mathsf{ood}] = [s \mid x \rightarrow \mathsf{ood}]\} \\[2mm]
class \quad &: \quad \boldsymbol{(Var \times \mathcal{P}(MFR) \rightarrow \mathcal{P}(MFR)} \\
class(x, S) \quad &\triangleq \quad \bigcup_{mfr \in S} class(x, mfr) \\[2mm]
noclass \quad &: \quad \boldsymbol{\mathcal{P}(MFR) \rightarrow \mathcal{P}(Var)} \\
noclass(S) \quad &\triangleq \quad \{x \in Var \mid class(x, S) \nsubseteq S\}
\end{aligned}
$$

.

The proof of the translation is in Sect. 3.8.13. The properties and proofs about the function *class* are in Sect. 3.8.12.

3.6.5 Translation of $E_1 = E_2$

Most of the cases for translating equality have been presented in Sect. 3.1. The translation will use the intersection function, so for example, to translate $x = y$, we will first check if x or y points the singletons of an auxiliary variable or insure it using the *extension* function (this is always the case in our implementation), then make them both point to one of the auxiliary variable and apply the intersection version of procedure **do_2_var** for those two auxiliary variables.

3.6.6 Translation of $x \mapsto E_1, E_2$

The situation is similar as for equality, like $x = Loc(E_1, E_2)$. We use auxiliary variables for translating the information about E_1 and E_2.

3.6.7 Translation of μ and ν

ν **case:** Since we have $\nu X_v.\, P \equiv P\{\nu X_v.\, P/X_v\}$, by the unfolding theorem Th. 2.29, and we do an overapproximation, we can safely translate $\nu X_v.P$ by translating $T(ar_\top, P)$ with $ar_\top$ as first approximation for the translation of X_v and then increase the precision of the translation by translating it again $T(ar_\top, P)$ with the current result as translation for X_v. We will stop the proceedure using a narrowing operator, which is the stabilization operator using the numerical narrowing as numerical stabilization. Then, we translate a last time $T(ar, P)$ with the last result for translating X_v.

In fact, we will have two translations in the implementation domain _ two $ar_\top$ _ one is what we intend as the $\top$ in AR knowing that we might have done some approximation. The second corresponds to the translation of a formula we know being `true`. The semantics are the same, except that in one case we indicate that we did not do approximation. Their difference will be seen while translating $\neg$: if we have $\neg P$ with the translation of P being the normal $ar_\top$, $\neg ar_\top$ translates also to this $ar_\top$, because we overapproximate and we cannot know whether the real result is not $\bot$. If we have the second meaning `true`, the translation of the negation will be a value of AR meaning the formula is false.

So, if we know that the formula $\nu X_v.\, P$ has a semantics which correspond to the infinite intersection (example, for formulae coming from our functions wlp or sp), then we start with the $ar_\top$ meaning `true`, and when encounting $\neg X_v$, it will give us the value meaning `false`. But if we don't know we have that semantics, we will keep safe by starting the translation with the normal $ar_\top$, which makes the translation of $\nu X_v.\, (\neg X_v)$ terminates to $ar_\top$.

μ **case:** It is similar to the ν case, except that if we know that the formula corresponds to a big union, we start with the value of AR that represents the `false` formula, and we use the stabilization operator with the numerical widening as numerical stabilization. Otherwise, if we don't know, we can always be safe and imprecise by translating μ as a ν.

3.7 Conclusion

In this chapter, we presented a new intermediate language for separation logic whose denotations resemble shape graphs. The improvement is that numerics and locations are treated the same way, thus we can have numerical summary nodes. This language is designed for the abstraction of separation logic with fixpoints. To keep the language as general as possible, it is parameterized by a numerical abstract domain which can be instantiated as needed by existing ones including relational ones.

The originality in the design is a semantics stated in terms of sets of memory which is the usual concrete model for separation logic as well. This is suitable for proving the correctness of functions on the language. In particular, the semantics of a graph is the disjunction of the semantics of its edges. Elements of our language being tuples, the semantics is the disjunction of each elements of the tuple, thus one can drop some elements of the tuples, losing precision but not correctness.

We provide an stabilization (for widening and narrowing) and a union (along with their correctness proofs) where precision/cost can be tuned to the specific needs of the context where the language is used.

We have designed and proved the translation of the formulae into the language.

The language was designed with the goal of building a translation toward/from existing shape-graphs, so we believe that the language along with its semantics will prove useful for both separation-logic and also heap-shape analysis. Please note that a lonely outgoing edge can be seen as a "must" arrow (valued 1 in a three-valued logic), and several outgoing edges from a variable can be seen as a "may" arrow (valued 1/2, but it is a bit more precise because we know that one of the arrow must exist), and an edge to $\emptyset$ can be seen as a "must not" arrow (valued 0).

3.8 Appendix

3.8.1 Replace

We define polymorphic functions :

$$(* \; replace(v_1, v_2, D) \quad is \quad \text{replacing } v_1 \text{ by } v_2 \text{ in } D \; *)$$

$$
\begin{aligned}
replace \;\; &: \quad \boldsymbol{VAR \to VAR \to VAR \to VAR} \\
replace(v_1, \_, v_3) \;\; &\triangleq \quad v_3 \;\; \text{when } v_3 \neq v_1 \\
replace(v_1, v_2, v_1) \;\; &\triangleq \quad v_2 \\[1em]
replace \;\; &: \quad \boldsymbol{VAR \to VAR \to VD \to VD} \\
replace(\_, \_, Nilt) \;\; &\triangleq \quad Nilt \\
replace(\_, \_, Truet) \;\; &\triangleq \quad Truet \\
replace(\_, \_, Falset) \;\; &\triangleq \quad Falset \\
replace(\_, \_, Oodt) \;\; &\triangleq \quad Oodt \\
replace(\_, \_, Numt) \;\; &\triangleq \quad Numt \\
replace(\_, \_, Dangling\_Loc) \;\; &\triangleq \quad Dangling\_Loc \\
replace(v_1, v_2, v_3) \;\; &\triangleq \quad replace(v_1, v_2, v_3) \\
replace(v_1, v_2, Loc(A, vd1, vd2)) \;\; &\triangleq \quad Loc(A, replace(v_1, v_2, vd1), replace(v_1, v_2, vd2)) \\[1em]
replace \;\; &: \quad \boldsymbol{VAR \to VAR \to PVD^+ \to PVD^+} \\
replace(v_1, v_2, \circledast) \;\; &\triangleq \quad \circledast \\
replace(v_1, v_2, \top) \;\; &\triangleq \quad \top \\
replace(v_1, v_2, S) \;\; &\triangleq \quad \bigcup_{vd \in S} \{replace(v_1, v_2, vd)\} \;\; \text{when } S \neq \top, \circledast
\end{aligned}
$$

3.8.2 Cheap extension proofs

Corollary 3.2. $\forall v \in VAR . \alpha \in TVar . ar \in AR. \; [\![ar]\!] = [\![extend(v, \alpha, ar)]\!]$

Cor. 3.2. $s, h \in [\![ar]\!] \Leftrightarrow (\text{by def.}) \exists f, r. \; \bar{s}, h, f, r \in [\![ar]\!]' \Leftrightarrow (\text{by Prop. 3.1}) \exists f, r. \bar{s}, h, [f \mid \alpha \to s^+ f(v)], r \in [\![extend(v, \alpha, ar)]\!]' \Leftrightarrow s, h \in [\![extend(v, \alpha, ar)]\!] \; \square$ ∎

Proposition 3.1. $\forall v \in VAR . \alpha \in TVar . [ar \mid \alpha \to \circledast] \in AR . (s, h, f, r) \in MFR.$

$s, h, f, r \in [\![[ar \mid \alpha \to \circledast]]\!]' \Leftrightarrow s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![extend(v, \alpha, [ar \mid \alpha \to \circledast])]\!]'$

Prop. 3.1. • It's direct to show that the constains on the domain remains.

• $$\begin{aligned}
& s,h,f,r \in [\![([ad \mid \alpha \to\circledast], hu, ho, sn, sn^\infty, t)]\!]' \\
\Leftrightarrow\ & s,h,f,r \in [\![[ad \mid \alpha \to\circledast]]\!]^4 \cap [\![hu]\!]^1 \cap [\![ho]\!]^{1'} \cap [\![sn]\!]^2 \cap [\![sn^\infty]\!]^{2'} \cap [\![t]\!]^3 \cap sem* \\
\Leftrightarrow\ & s,h,[f \mid \alpha \to s^+f(v)],r \in [\![[ad \mid v \to \{\alpha\} \mid \alpha \to ad(v)]]\!]^4 \cap [\![hu]\!]^1 \cap [\![ho \setminus \{\alpha\}]\!]^{1'} \\
& \cap [\![\text{if } v \in sn \text{ then } sn \cup \{\alpha\} \text{ else } sn \setminus \{\alpha\}]\!]^2 \\
& \cap [\![\text{if } v \in sn^\infty \text{ then } sn^\infty \cup \{\alpha\} \text{ else } sn^\infty \setminus \{\alpha\}]\!]^{2'} \\
& \cap [\![[t \mid (\alpha,\alpha') \to t(v,\alpha') \mid (\alpha',\alpha) \to t(\alpha',\alpha)]]\!]^3 \cap sem* \\
\Leftrightarrow\ & s,h,[f \mid \alpha \to s^+f(v)],r \in [\![extend(v, \alpha, ([ad \mid \alpha \to\circledast], hu, ho, sn, sn^\infty, t))
\end{aligned}$$

• $$\begin{aligned}
& s,h,[f \mid \alpha \to s^+f(v)],r \in [\![[ad \mid v \to \{\alpha\} \mid \alpha \to ad(v)]]\!]^4 \\
\Leftrightarrow\ & s,h,[f \mid \alpha \to s^+f(v)],r \in \bigcap_{v' \in VAR\setminus\{v,\alpha\}} [\![v', ad(v')]\!]^5 \cap [\![v, \{\alpha\}]\!]^5 \cap [\![\alpha, ad(v)]\!]^5 \\
\Leftrightarrow\ & \left[\begin{aligned}
& \bullet \quad s,h,[f \mid \alpha \to s^+f(v)],r \in \bigcap_{v' \in VAR\setminus\{v,\alpha\}} [\![v', ad(v')]\!]^5 \\
& \bullet \quad s^+[f \mid \alpha \to s^+f(v)](v) \subseteq [f \mid \alpha \to s^+f(v)](\alpha) \\
& \bullet \quad s,h,[f \mid \alpha \to s^+f(v)],r \in [\![\alpha, ad(v)]\!]^5
\end{aligned}\right. \\
\Leftrightarrow\ & \left[\begin{aligned}
& \bullet \quad s,h,[f \mid \alpha \to s^+f(v)],r \in \bigcap_{v' \in VAR\setminus\{v,\alpha\}} [\![v', ad(v')]\!]^5 \\
& \bullet \quad s,h,[f \mid \alpha \to s^+f(v)],r \in [\![\alpha, ad(v)]\!]^5
\end{aligned}\right. \\
& \text{by Prop. 3.38 and 3.39} \\
& \text{and } ad(\alpha) = \circledast \Rightarrow \text{ contrains, we get:} \\
\Leftrightarrow\ & \left[\begin{aligned}
& \bullet \quad s,h,[f \mid \alpha \to \emptyset],r \in \bigcap_{v' \in VAR\setminus\{v,\alpha\}} [\![v', ad(v')]\!]^5 \\
& \bullet \quad f(\alpha) = \emptyset \\
& \bullet \quad s,h,[f \mid \alpha \to \emptyset],r \in [\![v, ad(v)]\!]^5
\end{aligned}\right. \\
\Leftrightarrow\ & \left[\begin{aligned}
& \bullet \quad s,h,f,r \in \bigcap_{v' \in VAR\setminus\{v,\alpha\}} [\![v', ad(v')]\!]^5 \\
& \bullet \quad f(\alpha) = \emptyset \\
& \bullet \quad s,h,f,r \in [\![v, ad(v)]\!]^5
\end{aligned}\right. \\
\Leftrightarrow\ & s,h,f,r \in \bigcap_{v' \in VAR\setminus\{v,\alpha\}} [\![v', ad(v')]\!]^5 \cap [\![\alpha, \circledast]\!]^5 \cap [\![v, ad(v)]\!]^5 \\
\Leftrightarrow\ & s,h,f,r \in [\![[ad \mid \alpha \to\circledast]]\!]^4
\end{aligned}$$

• $\alpha \notin hu$ by the constraints on the domain
$$\begin{aligned}
& s,h,f,r \in [\![hu]\!]^1 \\
\Leftrightarrow\ & \forall \alpha' \in hu.f(\alpha') \cap dom(h) \neq \emptyset \\
\Leftrightarrow\ & \forall \alpha' \in hu.[f \mid \alpha \to s^+f(v)](\alpha') \cap dom(h) \neq \emptyset \\
\Leftrightarrow\ & s,h,[f \mid \alpha \to s^+f(v)],r \in [\![hu]\!]^1
\end{aligned}$$

• $\alpha \notin hu$ by the constraints on the domain, when $v \in hu$

136

$$
\left[
\begin{array}{ll}
& s, h, f, r \in [\![hu]\!]^1 \\
\Leftrightarrow & \forall \alpha' \in hu.f(\alpha') \cap dom(h) \neq \emptyset \\
\Leftrightarrow & (\forall \alpha' \in hu \setminus \{v\}.f(\alpha') \cap dom(h) \neq \emptyset) \wedge (f(v) \cap dom(h) \neq \emptyset) \\
\Leftrightarrow & \forall \alpha' \in hu \setminus \{v\} \cup \{\alpha\}.[f \mid \alpha \rightarrow f(v)](\alpha') \cap dom(h) \neq \emptyset \\
\Leftrightarrow & s, h, [f \mid \alpha \rightarrow s^+ f(v)], r \in [\![hu \setminus \{v\} \cup \{\alpha\}]\!]^1
\end{array}
\right.
$$

- $\left[
\begin{array}{ll}
& s, h, f, r \in [\![\texttt{full}]\!]^{1'} \\
\Leftrightarrow & True \\
\Leftrightarrow & s, h, [f \mid \alpha \rightarrow s^+ f(v)], r \in [\![\texttt{full} \setminus \{\alpha\}]\!]^{1'}
\end{array}
\right.$

- we know $f(\alpha) = \emptyset$, $v \in ho$

$$
\left[
\begin{array}{ll}
& s, h, f, r \in [\![ho]\!]^{1'} \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho} f(\alpha') \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{v\}} f(\alpha') \cup f(v) \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{v\}} [f \mid \alpha \rightarrow f(v)](\alpha') \cup [f \mid \alpha \rightarrow f(v)](\alpha) \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{v\} \cup \{\alpha\}} [f \mid \alpha \rightarrow f(v)](\alpha') \\
\Leftrightarrow & s, h, [f \mid \alpha \rightarrow s^+ f(v)], r \in [\![ho \setminus \{v\} \cup \{\alpha\}]\!]^{1'} \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho} f(\alpha') \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} f(\alpha') \text{ since } f(\alpha) = \emptyset \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} f(\alpha') \cup f(v) \text{ since } v \in ho \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} [f \mid \alpha \rightarrow f(v)](\alpha') \cup [f \mid \alpha \rightarrow f(v)](\alpha) \\
\Leftrightarrow & s, h, [f \mid \alpha \rightarrow s^+ f(v)], r \in [\![ho]\!]^{1'} \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} f(\alpha') \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} [f \mid \alpha \rightarrow f(v)](\alpha') \\
\Leftrightarrow & s, h, [f \mid \alpha \rightarrow s^+ f(v)], r \in [\![ho \setminus \{\alpha\}]\!]^{1'} \\
\Leftrightarrow & s, h, [f \mid \alpha \rightarrow s^+ f(v)], r \in [\![ho \cup \{\alpha\}]\!]^{1'}
\end{array}
\right.
$$

- we know that $f(\alpha) = \emptyset$, when $v \notin ho$

$$
\left[
\begin{array}{ll}
& s, h, f, r \in [\![ho]\!]^{1'} \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho} f(\alpha') \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} f(\alpha') \text{ since } f(\alpha) = \emptyset \\
\Leftrightarrow & dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} [f \mid \alpha \rightarrow s^+(v)](\alpha') \\
\Leftrightarrow & s, h, [f \mid \alpha \rightarrow s^+ f(v)], r \in [\![ho \setminus \{\alpha\}]\!]^{1'}
\end{array}
\right.
$$

137

- we know $f(\alpha) = \emptyset$, when $v \in sn$

$$
\begin{aligned}
& \quad\; s, h, f, r \in [\![sn]\!]^2 \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus sn.|f(\alpha')| \le 1 \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus (sn \cup \{\alpha\}).|f(\alpha')| \le 1 \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus (sn \cup \{\alpha\}).|[f \mid \alpha \to s^+ f(v)](\alpha')| \le 1 \\
\Leftrightarrow\;& s, h, [f \mid \alpha \to s^+ f(v)], r [\![sn \cup \{\alpha\}]\!]^2
\end{aligned}
$$

- we know $f(\alpha) = \emptyset$, when $v \notin sn$

$$
\begin{aligned}
& \quad\; s, h, f, r \in [\![sn]\!]^2 \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus sn.|f(\alpha')| \le 1 \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus (sn \setminus \{v\}).|f(\alpha')| \le 1 \\
\Leftrightarrow\;& (\forall \alpha' \in TVar \setminus sn.|f(\alpha')| \le 1) \wedge (|f(v)| \le 1) \\
\Leftrightarrow\;& (\forall \alpha' \in TVar \setminus (sn \cup \{\alpha\}).|f(\alpha')| \le 1) \wedge (|f(v)| \le 1) \\
\Leftrightarrow\;& (\forall \alpha' \in TVar \setminus (sn \cup \{\alpha\}).|[f \mid \alpha \to s^+ f(v)](\alpha')| \le 1) \\
& \quad \wedge (|[f \mid \alpha \to s^+ f(v)](\alpha)| \le 1) \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus (sn \setminus \{\alpha\}).|[f \mid \alpha \to s^+ f(v)](\alpha')| \le 1 \\
\Leftrightarrow\;& s, h, [f \mid \alpha \to s^+ f(v)], r [\![sn \setminus \{\alpha\}]\!]^2
\end{aligned}
$$

- we know $f(\alpha) = \emptyset$, when $v \in sn^\infty$

$$
\begin{aligned}
& \quad\; s, h, f, r \in [\![sn^\infty]\!]^{2'} \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus sn^\infty.f(\alpha') \text{ is finite} \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus (sn^\infty \cup \{\alpha\}).f(\alpha') \text{ is finite} \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus (sn^\infty \cup \{\alpha\}).[f \mid \alpha \to s^+ f(v)](\alpha') \text{ is finite} \\
\Leftrightarrow\;& s, h, [f \mid \alpha \to s^+ f(v)], r [\![sn^\infty \cup \{\alpha\}]\!]^{2'}
\end{aligned}
$$

- we know $f(\alpha) = \emptyset$, when $v \notin sn^\infty$

$$
\begin{aligned}
& \quad\; s, h, f, r \in [\![sn^\infty]\!]^{2'} \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus sn^\infty.f(\alpha') \text{ is finite} \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus (sn^\infty \setminus \{v\}).f(\alpha') \text{ is finite} \\
\Leftrightarrow\;& (\forall \alpha' \in TVar \setminus sn^\infty.f(\alpha') \text{ is finite}) \wedge (f(v) \text{ is finite}) \\
\Leftrightarrow\;& (\forall \alpha' \in TVar \setminus (sn^\infty \cup \{\alpha\}).f(\alpha') \text{ is finite}) \wedge (f(v) \text{ is finite}) \\
\Leftrightarrow\;& (\forall \alpha' \in TVar \setminus (sn^\infty \cup \{\alpha\}).[f \mid \alpha \to s^+ f(v)](\alpha') \text{ is finite}) \\
& \quad \wedge ([f \mid \alpha \to s^+ f(v)](\alpha) \text{ is finite}) \\
\Leftrightarrow\;& \forall \alpha' \in TVar \setminus (sn^\infty \setminus \{\alpha\}).[f \mid \alpha \to s^+ f(v)](\alpha') \text{ is finite} \\
\Leftrightarrow\;& s, h, [f \mid \alpha \to s^+ f(v)], r [\![sn^\infty \setminus \{\alpha\}]\!]^{2'}
\end{aligned}
$$

- when $v \in TVar$

$$
\begin{aligned}
&\quad s, h, f, r \in [\![t]\!]^3 \\
\Leftrightarrow &\quad \forall \alpha_1, \alpha_2 \in \mathit{TVar}.[\![t(\alpha_1, \alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\Leftrightarrow &\quad \forall \alpha_1, \alpha_2 \in \mathit{TVar}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'}
\end{aligned}
$$

$$
\Leftrightarrow
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{bmatrix}
$$

$$
\Leftrightarrow
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{bmatrix}
$$

$$
\Leftrightarrow
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{bmatrix}
$$

$$
\Leftrightarrow
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{bmatrix}
$$

$$
\Leftrightarrow
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, \emptyset, f(\alpha')]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), \emptyset]\!]^{6'}
\end{bmatrix}
$$

$$
\Leftrightarrow
\begin{bmatrix}
\text{when } v = \alpha &
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, \emptyset, f(\alpha')]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), \emptyset]\!]^{6'}
\end{bmatrix} \\[2ex]
\text{when } v \neq \alpha &
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, \emptyset, f(\alpha')]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), \emptyset]\!]^{6'}
\end{bmatrix}
\end{bmatrix}
$$

by domain contrains, we get:

$$
\Leftrightarrow
\begin{bmatrix}
\text{when } v = \alpha &
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, \emptyset, f(\alpha')]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), \emptyset]\!]^{6'}
\end{bmatrix} \\[1ex]
\text{when } v \neq \alpha &
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'}
\end{bmatrix}
\end{bmatrix}
$$

$$
\Leftrightarrow
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(v, \alpha').[\![l, f(v), f(\alpha')]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in t(\alpha', v).[\![l, f(\alpha'), f(v)]\!]^{6'}
\end{bmatrix}
$$

$$
\Leftrightarrow
\begin{bmatrix}
\bullet \;\; \forall \alpha_1, \alpha_2 \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in [t \mid (\alpha, \alpha') \to t(v, \alpha') \mid (\alpha', \alpha) \to t(\alpha', v)](\alpha_1, \alpha_2). \\
\quad [\, [f \mid \alpha \to s^+ f(v)](\alpha_2)]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in [t \mid (\alpha, \alpha') \to t(v, \alpha') \mid (\alpha', \alpha) \to t(\alpha', v)](\alpha, \alpha'). \\
\quad [\, [f \mid \alpha \to s^+ f(v)](\alpha')]\!]^{6'} \\
\bullet \;\; \forall \alpha' \in \mathit{TVar} \setminus \{\alpha\}.\exists l \in [t \mid (\alpha, \alpha') \to t(v, \alpha') \mid (\alpha', \alpha) \to t(\alpha', v)](\alpha', \alpha). \\
\quad [\, [f \mid \alpha \to s^+ f(v)](\alpha)]\!]^{6'}
\end{bmatrix}
$$

$$
\begin{aligned}
\Leftrightarrow &\quad \forall \alpha_1, \alpha_2 \in \mathit{TVar}.\exists l \in [t \mid (\alpha, \alpha') \to t(v, \alpha') \mid (\alpha', \alpha) \to t(\alpha', v)](\alpha_1, \alpha_2). \\
&\quad [\, [f \mid \alpha \to s^+ f(v)](\alpha_2)]\!]^{6'} \\
\Leftrightarrow &\quad s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![[t \mid (\alpha, \alpha') \to t(v, \alpha') \mid (\alpha', \alpha) \to t(\alpha', v)]]\!]^3
\end{aligned}
$$

- when $v \notin \mathit{TVar}$, by domain constraints we have $\forall \alpha'.\{\ddagger_{eq}, \dagger_{eq}\} \cap t(\alpha, \alpha') \neq \emptyset \wedge \{\ddagger_{eq}, \dagger_{eq}\} \cap t(\alpha', \alpha) \neq \emptyset$

$$
\begin{aligned}
& s, h, f, r \in [\![t]\!]^3 \\
\Leftrightarrow\ & \forall \alpha_1, \alpha_2 \in TVar.[\![t(\alpha_1, \alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\Leftrightarrow\ & \forall \alpha_1, \alpha_2 \in TVar.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\Leftrightarrow\ & \left[
\begin{aligned}
&\bullet\ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \left[
\begin{aligned}
&\bullet\ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \left[
\begin{aligned}
&\bullet\ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \left[
\begin{aligned}
&\bullet\ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \left[
\begin{aligned}
&\bullet\ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha, \alpha').[\![l, \emptyset, f(\alpha')]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha', \alpha).[\![l, f(\alpha'), \emptyset]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\Leftrightarrow\ & \left[
\begin{aligned}
&\bullet\ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1, \alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\ \exists l \in \{\ddagger_{eq}, =_{eq}\}.[\![l, s(v), s(v)]\!]^{6'} \\
&\bullet\ \exists l \in \{\ddagger_{eq}, \dagger_{eq}\}.[\![l, f(v), \emptyset]\!]^{6'} \\
&\bullet\ \exists l \in \{\ddagger_{eq}, \dagger_{eq}\}.[\![l, \emptyset, s(v)]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar.\exists l \in CL_{eq}.[\![l, f(\alpha'), s(v)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \left[
\begin{aligned}
&\bullet\ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}. \\
&\quad \exists l \in t(\alpha_1, \alpha_2).[\, [f \mid \alpha \to s^+f(v)](\alpha_2)]\!]^{6'} \\
&\bullet\ \exists l \in \{\ddagger_{eq}, =_{eq}\}.[\, [f \mid \alpha \to s^+f(v)](\alpha)]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.ad(\alpha') =\otimes\Rightarrow \exists l \in \{\ddagger_{eq}, \dagger_{eq}\}. \\
&\quad [\, [f \mid \alpha \to s^+f(v)](\alpha')]\!]^{6'} \\
&\bullet\ \exists l \in \{\ddagger_{eq}, \dagger_{eq}\}.[\, [f \mid \alpha \to s^+f(v)](\alpha)]\!]^{6'} \\
&\bullet\ \forall \alpha' \in TVar \setminus \{\alpha\}.ad(\alpha') \neq\otimes\Rightarrow \exists l \in CL_{eq}. \\
&\quad [\, [f \mid \alpha \to s^+f(v)](\alpha)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & s, h, [f \mid \alpha \to s^+f(v)], r \in
\left[\!\!\left[
\begin{aligned}
&t \\
&\mid (\alpha, \alpha') \to (ad(\alpha') =\otimes?\{\ddagger_{eq}, \dagger_{eq}\} : CL_{eq}) \\
&\mid (\alpha', \alpha) \to (ad(\alpha') =\otimes?\{\ddagger_{eq}, \dagger_{eq}\} : CL_{eq}) \\
&\mid (\alpha, \alpha) \to \{\ddagger_{eq}, =_{eq}\}
\end{aligned}
\right]\!\!\right]^3
\end{aligned}
$$

141

- we know that $f(\alpha) = \emptyset$, when $v \in Var$

$$
\begin{aligned}
& s, h, f, r \in [\![d]\!]^7 \\
\Leftrightarrow\ & \exists g \in [\![d]\!]^\mathcal{P}.\ \forall \alpha' \in TVar.\ f(\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\
\Leftrightarrow\ & \exists g \in [\![d]\!]^\mathcal{P}.\ \left[\begin{array}{l} \bullet \quad \forall \alpha' \neq \alpha.\ f(\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\ \bullet \quad f(\alpha) \cap \mathbb{Z} \subseteq g(\alpha) \end{array}\right. \\
\Leftrightarrow\ & \exists g \in [\![d]\!]^\mathcal{P}.\ \forall \alpha' \neq \alpha.\ [f \mid \alpha \rightarrow s(v)](\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\
\Leftrightarrow\ & \exists g \in [\![d]\!]^\mathcal{P}.\ \left[\begin{array}{l} \bullet \quad \forall \alpha' \neq \alpha.\ [f \mid \alpha \rightarrow s(v)](\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\ \bullet \quad s(v) \cap \mathbb{Z} \subseteq \mathbb{Z} \end{array}\right. \\
\Leftrightarrow\ & \exists g' \in [\![top(\alpha, d)]\!]^\mathcal{P}.\ \left[\begin{array}{l} \bullet \quad \forall \alpha' \neq \alpha.\ [f \mid \alpha \rightarrow s(v)](\alpha') \cap \mathbb{Z} \subseteq g'(\alpha') \\ \bullet \quad s(v) \cap \mathbb{Z} \subseteq g'(\alpha) \end{array}\right. \\
\Leftrightarrow\ & \exists g' \in [\![top(\alpha, d)]\!]^\mathcal{P}.\ \forall \alpha' \in TVar.\ [f \mid \alpha \rightarrow s(v)](\alpha') \cap \mathbb{Z} \subseteq g'(\alpha') \\
\Leftrightarrow\ & s, h, [f \mid \alpha \rightarrow s(v)], r \in [\![top(\alpha, d)]\!]^d
\end{aligned}
$$

- we know that $f(\alpha) = \emptyset$, when $v \in TVar$

$$
\begin{aligned}
& s, h, f, r \in [\![d]\!]^7 \\
\Leftrightarrow\ & \exists g \in [\![d]\!]^\mathcal{P}.\ \forall \alpha' \in TVar.\ f(\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\
\Leftrightarrow\ & \exists g \in [\![d]\!]^\mathcal{P}.\ \left[\begin{array}{l} \bullet \quad \forall \alpha' \neq \alpha.\ f(\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\ \bullet \quad f(\alpha) \cap \mathbb{Z} \subseteq g(\alpha) \end{array}\right. \\
\Leftrightarrow\ & \exists g \in [\![d]\!]^\mathcal{P}.\ \forall \alpha' \neq \alpha.\ [f \mid \alpha \rightarrow f(v)](\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\
\Leftrightarrow\ & \exists g \in [\![d]\!]^\mathcal{P}.\ \left[\begin{array}{l} \bullet \quad \forall \alpha' \neq \alpha.\ [f \mid \alpha \rightarrow f(v)](\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\ \bullet \quad f(v) \cap \mathbb{Z} \subseteq g(v) \end{array}\right. \\
\Leftrightarrow\ & \exists g' \in [\![copy(v, \alpha, d)]\!]^\mathcal{P}.\ \left[\begin{array}{l} \bullet \quad \forall \alpha' \neq \alpha.\ [f \mid \alpha \rightarrow f(v)](\alpha') \cap \mathbb{Z} \subseteq g'(\alpha') \\ \bullet \quad f(v) \cap \mathbb{Z} \subseteq g'(\alpha) \end{array}\right. \\
\Leftrightarrow\ & \exists g' \in [\![copy(v, \alpha, d)]\!]^\mathcal{P}.\ \forall \alpha' \in TVar.\ [f \mid \alpha \rightarrow f(v)](\alpha') \cap \mathbb{Z} \subseteq g'(\alpha') \\
\Leftrightarrow\ & s, h, [f \mid \alpha \rightarrow f(v)], r \in [\![copy(v, \alpha, d)]\!]^d
\end{aligned}
$$

$\square$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\blacksquare$

Proposition 3.38. $\forall \alpha \in TVar. v' \in VAR \setminus \{\alpha\}. pvd \in PVD^+. S \in \mathcal{P}(Val').(s, h, f, r) \in MFR$ when $\forall vd \in S. \alpha \text{``} \not\in'' vd$

$$s, h, [f \mid \alpha \rightarrow \emptyset], r \in [\![v', pvd]\!]^5 \Leftrightarrow s, h, [f \mid \alpha \rightarrow S], r \in [\![v', pvd]\!]^5$$

Prop. *3.38.* $\quad \bullet \left[\begin{array}{l} \quad s, h, [f \mid \alpha \rightarrow S], r \in [\![v', \top]\!]^5 \\ \Leftrightarrow\ s, h, [f \mid \alpha \rightarrow S], r \in MFR \\ \Leftrightarrow\ True \end{array}\right.$

$$
\bullet \quad \left[\begin{array}{ll}
& s, h, [f \mid \alpha \to \emptyset], r \in [\![v', \circledast]\!]^5 \\
\Leftrightarrow & (s^+[f \mid \alpha \to \emptyset](v') = \emptyset) \\
\Leftrightarrow & (s^+f(v') = \emptyset) \\
\Leftrightarrow & (s^+[f \mid \alpha \to S](v') = \emptyset) \\
\Leftrightarrow & s, h, [f \mid \alpha \to S], r \in [\![v', \circledast]\!]^5
\end{array}\right.
$$

- when $pvd \neq \top, \circledast$

$$
\begin{array}{rl}
& s, h, [f \mid \alpha \to \emptyset], r \in [\![v', pvd]\!]^5 \\
\Leftrightarrow & s^+[f \mid \alpha \to \emptyset](v') \subseteq \bigcup_{vd \in pvd} [\![vd, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 \\
\Leftrightarrow & s^+f(v') \subseteq \bigcup_{vd \in pvd} [\![vd, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 \\
\Leftrightarrow \text{(Prop. 3.40)} & s^+f(v') \subseteq \bigcup_{vd \in pvd} [\![vd, (h, [f \mid \alpha \to S], r)]\!]^8 \\
\Leftrightarrow & s^+[f \mid \alpha \to S](v') \subseteq \bigcup_{vd \in pvd} [\![vd, (h, [f \mid \alpha \to S], r)]\!]^8 \\
\Leftrightarrow & s, h, [f \mid \alpha \to S], r \in [\![v', pvd]\!]^5
\end{array}
$$

□   ∎

Proposition 3.39. $\forall \alpha \in TVar.v \in VAR.S \in PVD^+.(s, h, f, r) \in MFR$ *when* $\forall vd \in$

$S.\alpha \text{``} \notin'' vd$

$s, h, [f \mid \alpha \to \emptyset], r \in [\![v, S]\!]^5 \Leftrightarrow s, h, [f \mid \alpha \to s^+(v)], r \in [\![\alpha, S]\!]^5$

$$
Prop.\ 3.39. \quad \bullet \quad \left[\begin{array}{ll}
& s, h, [f \mid \alpha \to s^+f(v)], r \in [\![\alpha, \top]\!]^5 \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+f(v)], r \in MFR \\
\Leftrightarrow & True
\end{array}\right.
$$

$$
\bullet \quad \left[\begin{array}{ll}
& s, h, [f \mid \alpha \to \emptyset], r \in [\![v, \circledast]\!]^5 \\
\Leftrightarrow & (s^+[f \mid \alpha \to \emptyset](v) = \emptyset) \\
\Leftrightarrow & (s^+f(v) = \emptyset) \\
\Leftrightarrow & (s^+[f \mid \alpha \to s^+f(v)](\alpha) = \emptyset) \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+f(v)], r \in [\![\alpha, \circledast]\!]^5
\end{array}\right.
$$

- when $S \neq \top, \circledast$

$$
\begin{array}{rl}
& s, h, [f \mid \alpha \to \emptyset], r \in [\![v, S]\!]^5 \\
\Leftrightarrow & s^+[f \mid \alpha \to \emptyset](v) \subseteq \bigcup_{vd \in S} .[\![vd, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 \\
\Leftrightarrow & s^+f(v) \subseteq \bigcup_{vd \in S} [\![vd, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 \\
\Leftrightarrow \text{(by Prop. 3.40)} & s^+f(v) \subseteq \bigcup_{vd \in S} [\![vd, (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8 \\
\Leftrightarrow & s^+[f \mid \alpha \to s^+f(v)](\alpha) \subseteq \bigcup_{vd \in S} [\![vd, (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8 \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+f(v)], r \in [\![\alpha, S]\!]^5
\end{array}
$$

□ □

Proposition 3.40. $\forall \alpha \in TVar. l \in Val', vd \in VD, (h, f, r) \in (H \times F \times R). S \in \mathcal{P}(Val')$

when $\alpha\,``\notin''\,vd$

$$[\![vd, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 \Leftrightarrow [\![vd, (h, [f \mid \alpha \to S], r)]\!]^8$$

Prop. 3.40. We proceed by cases on vd

- $[\![Nilt, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = \{\texttt{nil}\}$
  $= [\![Nilt, (h, [f \mid \alpha \to S], r)]\!]^8$

- $[\![Truet, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = \{True\} = [\![Truet, (h, [f \mid \alpha \to S], r)]\!]^8$

- $[\![Falset, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = \{False\} = [\![Falset, (h, [f \mid \alpha \to S], r)]\!]^8$

- $[\![Oodt, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = \{\texttt{ood}\} = [\![Oodt, (h, [f \mid \alpha \to S], r)]\!]^8$

- $[\![Numt, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = \mathbb{Z} = [\![Numt, (h, [f \mid \alpha \to S], r)]\!]^8$

- $[\![Dangling\_Loc, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = Loc \setminus dom(h) = [\![Dangling\_Loc, (h, [f \mid \alpha \to S], r)]\!]^8$

- when $\alpha' \neq \alpha$
  $[\![\alpha', (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = [f \mid \alpha \to \emptyset](\alpha') = f(\alpha') = [f \mid \alpha \to S](\alpha') == [\![\alpha', (h, [f \mid \alpha \to S], r)]\!]^8$

- $\begin{aligned}
  &[\![Loc(A, vd1, vd2), (h, [f \mid \alpha \to \emptyset], r)]\!]^8 \\
  &= \left\{ l \in dom(h) \;\middle|\; \begin{array}{l} \bullet \;\; \Pi_1(h(l)) \in [\![vd1, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 \\ \bullet \;\; \Pi_2(h(l)) \in [\![vd2, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 \\ \bullet \;\; *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r(l) \\ \bullet \;\; *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r(l) \end{array} \right\} \\
  &\quad \text{by previous cases, we get:} \\
  &= \left\{ l \in dom(h) \;\middle|\; \begin{array}{l} \bullet \;\; \Pi_1(h(l)) \in [\![vd1, (h, [f \mid \alpha \to S], r)]\!]^8 \\ \bullet \;\; \Pi_2(h(l)) \in [\![vd2, (h, [f \mid \alpha \to S], r)]\!]^8 \\ \bullet \;\; *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r(l) \\ \bullet \;\; *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r(l) \end{array} \right\} \\
  &= [\![Loc(A, vd1, vd2), (h, [f \mid \alpha \to S], r)]\!]^8
  \end{aligned}$

144

□ □

3.8.3 Extension proofs

Corollary 3.2. $\forall v \in VAR.\alpha \in TVar.ar \in AR. [\![ar]\!] = [\![extend(v, \alpha, ar)]\!]$

Cor. 3.2. $s, h \in [\![ar]\!] \Leftrightarrow$ (by def.)$\exists f, r. \ \bar{s}, h, f, r \in [\![ar]\!]' \Leftrightarrow$ (by Prop. 3.1)$\exists f, r.\bar{s}, h, [f \mid \alpha \to s^+ f(v)], r \in [\![extend(v, \alpha, ar)]\!]' \Leftrightarrow s, h \in [\![extend(v, \alpha, ar)]\!]$ □

Proposition 3.1. $\forall v \in VAR.\alpha \in TVar.[ar \mid \alpha \to \circledast] \in AR.(s, h, f, r) \in MFR.$

$s, h, f, r \in [\![[ar \mid \alpha \to \circledast]]\!]' \Leftrightarrow s, h, [f \mid \alpha \to s^+ f(v)] \in [\![extend(v, \alpha, [ar \mid \alpha \to \circledast])]\!]'$

Prop. 3.1.   • It's direct to show that the constains on the domain remains.

•
$$
\begin{array}{rl}
& s, h, f, r \in [\![([ad \mid \alpha \to \circledast], hu, ho, sn, sn^\infty, t)]\!]' \\
\Leftrightarrow & s, h, f, r \in [\![[ad \mid \alpha \to \circledast]]\!]^4 \cap [\![hu]\!]^1 \cap [\![ho]\!]^{1'} \cap [\![sn]\!]^2 \cap [\![sn^\infty]\!]^{2'} \cap [\![t]\!]^3 \cap sem* \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![[ad \mid v \to \{\alpha\} \mid \alpha \to ad(v)]]\!]^4 \cap [\![hu]\!]^1 \cap [\![ho \setminus \{\alpha\}]\!]^{1'} \\
& \cap [\![\text{if } v \in sn \text{ then } sn \cup \{\alpha\} \text{ else } sn \setminus \{\alpha\}]\!]^2 \\
& \cap [\![\text{if } v \in sn^\infty \text{ then } sn^\infty \cup \{\alpha\} \text{ else } sn^\infty \setminus \{\alpha\}]\!]^{2'} \\
& \cap [\![[t \mid (\alpha, \alpha') \to t(v, \alpha') \mid (\alpha', \alpha) \to t(\alpha', \alpha)]]\!]^3 \cap sem* \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![extend(v, \alpha, ([ad \mid \alpha \to \circledast], hu, ho, sn, sn^\infty, t))]\!]
\end{array}
$$

- $$\begin{aligned}
&\quad s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![replace(v, \alpha) \circ [ad \mid v \to \{\alpha\} \mid \alpha \to ad(v)]]\!]^4 \\
&\Leftrightarrow s, h, [f \mid \alpha \to s^+ f(v)], r \in \bigcap_{v' \in VAR \setminus \{v, \alpha\}} [\![v', replace(v, \alpha, ad(v'))]\!]^5 \cap [\![v, \{\alpha\}]\!]^5 \\
&\quad \cap [\![\alpha, replace(v, \alpha, ad(v))]\!]^5 \\
&\Leftrightarrow \left[\begin{array}{l}
\bullet \quad s, h, [f \mid \alpha \to s^+ f(v)], r \in \bigcap_{v' \in VAR \setminus \{v, \alpha\}} [\![v', replace(v, \alpha, ad(v'))]\!]^5 \\
\bullet \quad s^+ [f \mid \alpha \to s^+ f(v)](v) \subseteq [f \mid \alpha \to s^+ f(v)](\alpha) \\
\bullet \quad s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![\alpha, replace(v, \alpha, ad(v))]\!]^5
\end{array}\right. \\
&\Leftrightarrow \left[\begin{array}{l}
\bullet \quad s, h, [f \mid \alpha \to s^+ f(v)], r \in \bigcap_{v' \in VAR \setminus \{v, \alpha\}} [\![v', replace(v, \alpha, ad(v'))]\!]^5 \\
\bullet \quad s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![\alpha, replace(v, \alpha, ad(v))]\!]^5
\end{array}\right. \\
&\quad \text{by Prop. } 3.41 \text{ and } 3.42
\end{aligned}$$

  and $ad(\alpha) = \otimes \Rightarrow$ contrains, we get:
  $$\begin{aligned}
&\Leftrightarrow \left[\begin{array}{l}
\bullet \quad s, h, [f \mid \alpha \to \emptyset], r \in \bigcap_{v' \in VAR \setminus \{v, \alpha\}} [\![v', ad(v')]\!]^5 \\
\bullet \quad f(\alpha) = \emptyset \\
\bullet \quad s, h, [f \mid \alpha \to \emptyset], r \in [\![v, ad(v)]\!]^5
\end{array}\right. \\
&\Leftrightarrow \left[\begin{array}{l}
\bullet \quad s, h, f, r \in \bigcap_{v' \in VAR \setminus \{v, \alpha\}} [\![v', ad(v')]\!]^5 \\
\bullet \quad f(\alpha) = \emptyset \\
\bullet \quad s, h, f, r \in [\![v, ad(v)]\!]^5
\end{array}\right. \\
&\Leftrightarrow s, h, f, r \in \bigcap_{v' \in VAR \setminus \{v, \alpha\}} [\![v', ad(v')]\!]^5 \cap [\![\alpha, \otimes]\!]^5 \cap [\![v, ad(v)]\!]^5 \\
&\Leftrightarrow s, h, f, r \in [\![[ad \mid \alpha \to \otimes]]\!]^4
\end{aligned}$$

- $\alpha \notin hu$ by the constraints on the domain
  $$\begin{aligned}
&\quad s, h, f, r \in [\![hu]\!]^1 \\
&\Leftrightarrow \forall \alpha' \in hu. f(\alpha') \cap dom(h) \neq \emptyset \\
&\Leftrightarrow \forall \alpha' \in hu. [f \mid \alpha \to s^+ f(v)](\alpha') \cap dom(h) \neq \emptyset \\
&\Leftrightarrow s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![hu]\!]^1
\end{aligned}$$

- $\alpha \notin hu$ by the constraints on the domain, when $v \in hu$
  $$\begin{aligned}
&\quad s, h, f, r \in [\![hu]\!]^1 \\
&\Leftrightarrow \forall \alpha' \in hu. f(\alpha') \cap dom(h) \neq \emptyset \\
&\Leftrightarrow (\forall \alpha' \in hu \setminus \{v\}. f(\alpha') \cap dom(h) \neq \emptyset) \wedge (f(v) \cap dom(h) \neq \emptyset) \\
&\Leftrightarrow \forall \alpha' \in hu \setminus \{v\} \cup \{\alpha\}. [f \mid \alpha \to f(v)](\alpha') \cap dom(h) \neq \emptyset \\
&\Leftrightarrow s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![hu \setminus \{v\} \cup \{\alpha\}]\!]^1
\end{aligned}$$

- $$\begin{aligned}
&\quad s, h, f, r \in [\![\texttt{full}]\!]^{1'} \\
&\Leftrightarrow True \\
&\Leftrightarrow s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![\texttt{full} \setminus \{\alpha\}]\!]^{1'}
\end{aligned}$$

146

- $$\left[\begin{array}{l} \quad s, h, f, r \in \llbracket \texttt{full} \rrbracket^{1'} \\ \Leftrightarrow \quad True \\ \Leftrightarrow \quad s, h, [f \mid \alpha \to s^+ f(v)], r \in \llbracket \texttt{full} \setminus \{v\} \cup \{\alpha\} \rrbracket^{1'} \end{array}\right.$$

- we know $f(\alpha) = \emptyset$, $v \in ho$

$$\left[\begin{array}{l} \quad s, h, f, r \in \llbracket ho \rrbracket^{1'} \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho} f(\alpha') \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{v\}} f(\alpha') \cup f(v) \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{v\}} [f \mid \alpha \to f(v)](\alpha') \cup [f \mid \alpha \to f(v)](\alpha) \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{v\} \cup \{\alpha\}} [f \mid \alpha \to f(v)](\alpha') \\ \Leftrightarrow \quad s, h, [f \mid \alpha \to s^+ f(v)], r \in \llbracket ho \setminus \{v\} \cup \{\alpha\} \rrbracket^{1'} \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho} f(\alpha') \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} f(\alpha') \text{ since } f(\alpha) = \emptyset \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} f(\alpha') \cup f(v) \text{ since } v \in ho \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} [f \mid \alpha \to f(v)](\alpha') \cup [f \mid \alpha \to f(v)](\alpha) \\ \Leftrightarrow \quad s, h, [f \mid \alpha \to s^+ f(v)], r \in \llbracket ho \rrbracket^{1'} \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} f(\alpha') \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} [f \mid \alpha \to f(v)](\alpha') \\ \Leftrightarrow \quad s, h, [f \mid \alpha \to s^+ f(v)], r \in \llbracket ho \setminus \{\alpha\} \rrbracket^{1'} \\ \Leftrightarrow \quad s, h, [f \mid \alpha \to s^+ f(v)], r \in \llbracket ho \cup \{\alpha\} \rrbracket^{1'} \end{array}\right.$$

- we know that $f(\alpha) = \emptyset$, when $v \notin ho$

$$\left[\begin{array}{l} \quad s, h, f, r \in \llbracket ho \rrbracket^{1'} \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho} f(\alpha') \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} f(\alpha') \text{ since } f(\alpha) = \emptyset \\ \Leftrightarrow \quad dom(h) \subseteq \bigcup_{\alpha' \in ho \setminus \{\alpha\}} [f \mid \alpha \to s^+(v)](\alpha') \\ \Leftrightarrow \quad s, h, [f \mid \alpha \to s^+ f(v)], r \in \llbracket ho \setminus \{\alpha\} \rrbracket^{1'} \end{array}\right.$$

- we know $f(\alpha) = \emptyset$, when $v \in sn$

$$
\left[
\begin{aligned}
&\quad s, h, f, r \in [\![sn]\!]^2 \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus sn. |f(\alpha')| \leq 1 \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus (sn \cup \{\alpha\}). |f(\alpha')| \leq 1 \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus (sn \cup \{\alpha\}). |[f \mid \alpha \to s^+f(v)](\alpha')| \leq 1 \\
\Leftrightarrow\ &\ s, h, [f \mid \alpha \to s^+f(v)], r [\![sn \cup \{\alpha\}]\!]^2
\end{aligned}
\right.
$$

- we know $f(\alpha) = \emptyset$, when $v \notin sn$

$$
\left[
\begin{aligned}
&\quad s, h, f, r \in [\![sn]\!]^2 \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus sn. |f(\alpha')| \leq 1 \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus (sn \setminus \{v\}). |f(\alpha')| \leq 1 \\
\Leftrightarrow\ &\ (\forall \alpha' \in TVar \setminus sn. |f(\alpha')| \leq 1) \wedge (|f(v)| \leq 1) \\
\Leftrightarrow\ &\ (\forall \alpha' \in TVar \setminus (sn \cup \{\alpha\}). |f(\alpha')| \leq 1) \wedge (|f(v)| \leq 1) \\
\Leftrightarrow\ &\ (\forall \alpha' \in TVar \setminus (sn \cup \{\alpha\}). |[f \mid \alpha \to s^+f(v)](\alpha')| \leq 1) \\
&\quad \wedge (|[f \mid \alpha \to s^+f(v)](\alpha)| \leq 1) \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus (sn \setminus \{\alpha\}). |[f \mid \alpha \to s^+f(v)](\alpha')| \leq 1 \\
\Leftrightarrow\ &\ s, h, [f \mid \alpha \to s^+f(v)], r [\![sn \setminus \{\alpha\}]\!]^2
\end{aligned}
\right.
$$

- we know $f(\alpha) = \emptyset$, when $v \in sn^\infty$

$$
\left[
\begin{aligned}
&\quad s, h, f, r \in [\![sn^\infty]\!]^{2'} \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus sn^\infty. f(\alpha') \text{ is finite} \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus (sn^\infty \cup \{\alpha\}). f(\alpha') \text{ is finite} \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus (sn^\infty \cup \{\alpha\}). [f \mid \alpha \to s^+f(v)](\alpha') \text{ is finite} \\
\Leftrightarrow\ &\ s, h, [f \mid \alpha \to s^+f(v)], r [\![sn^\infty \cup \{\alpha\}]\!]^{2'}
\end{aligned}
\right.
$$

- we know $f(\alpha) = \emptyset$, when $v \notin sn^\infty$

$$
\left[
\begin{aligned}
&\quad s, h, f, r \in [\![sn^\infty]\!]^{2'} \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus sn^\infty. f(\alpha') \text{ is finite} \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus (sn^\infty \setminus \{v\}). f(\alpha') \text{ is finite} \\
\Leftrightarrow\ &\ (\forall \alpha' \in TVar \setminus sn^\infty. f(\alpha') \text{ is finite}) \wedge (f(v) \text{ is finite}) \\
\Leftrightarrow\ &\ (\forall \alpha' \in TVar \setminus (sn^\infty \cup \{\alpha\}). f(\alpha') \text{ is finite}) \wedge (f(v) \text{ is finite}) \\
\Leftrightarrow\ &\ (\forall \alpha' \in TVar \setminus (sn^\infty \cup \{\alpha\}). [f \mid \alpha \to s^+f(v)](\alpha') \text{ is finite}) \\
&\quad \wedge ([f \mid \alpha \to s^+f(v)](\alpha) \text{ is finite}) \\
\Leftrightarrow\ &\ \forall \alpha' \in TVar \setminus (sn^\infty \setminus \{\alpha\}). [f \mid \alpha \to s^+f(v)](\alpha') \text{ is finite} \\
\Leftrightarrow\ &\ s, h, [f \mid \alpha \to s^+f(v)], r [\![sn^\infty \setminus \{\alpha\}]\!]^{2'}
\end{aligned}
\right.
$$

- when $v \in TVar$

148

$$
\begin{aligned}
&\quad\; s,h,f,r \in [\![t]\!]^3 \\
&\Leftrightarrow\; \forall \alpha_1, \alpha_2 \in TVar.[\![t(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\Leftrightarrow\; \forall \alpha_1, \alpha_2 \in TVar.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\Leftrightarrow\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}\right. \\
&\Leftrightarrow\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}\right. \\
&\Leftrightarrow\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}\right. \\
&\Leftrightarrow\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}\right. \\
&\Leftrightarrow\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, \emptyset, f(\alpha')]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), \emptyset]\!]^{6'}
\end{aligned}\right. \\
&\Leftrightarrow\;
\left[\begin{aligned}
&\text{when } v = \alpha\;\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, \emptyset, f(\alpha')]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), \emptyset]\!]^{6'}
\end{aligned}\right. \\
&\text{when } v \neq \alpha\;\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, \emptyset, f(\alpha')]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), \emptyset]\!]^{6'}
\end{aligned}\right.
\end{aligned}\right.
\end{aligned}
$$

by domain contrains, we get:

$$
\begin{aligned}
&\Leftrightarrow\;
\left[\begin{aligned}
&\text{when } v = \alpha\;\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, \emptyset, f(\alpha')]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), \emptyset]\!]^{6'}
\end{aligned}\right. \\
&\text{when } v \neq \alpha\;\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'}
\end{aligned}\right.
\end{aligned}\right. \\
&\Leftrightarrow\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(v,\alpha').[\![l, f(v), f(\alpha')]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',v).[\![l, f(\alpha'), f(v)]\!]^{6'}
\end{aligned}\right. \\
&\Leftrightarrow\;
\left[\begin{aligned}
&\bullet\;\; \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in [t \mid (\alpha,\alpha') \to t(v,\alpha') \mid (\alpha',\alpha) \to t(\alpha',v)](\alpha_1,\alpha_2). \\
&\quad\;\; [\, [f \mid \alpha \to s^+ f(v)](\alpha_2)]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in [t \mid (\alpha,\alpha') \to t(v,\alpha') \mid (\alpha',\alpha) \to t(\alpha',v)](\alpha,\alpha'). \\
&\quad\;\; [\, [f \mid \alpha \to s^+ f(v)](\alpha')]\!]^{6'} \\
&\bullet\;\; \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in [t \mid (\alpha,\alpha') \to t(v,\alpha') \mid (\alpha',\alpha) \to t(\alpha',v)](\alpha',\alpha). \\
&\quad\;\; [\, [f \mid \alpha \to s^+ f(v)](\alpha)]\!]^{6'}
\end{aligned}\right. \\
&\Leftrightarrow\; \forall \alpha_1, \alpha_2 \in TVar.\exists l \in [t \mid (\alpha,\alpha') \to t(v,\alpha') \mid (\alpha',\alpha) \to t(\alpha',v)](\alpha_1,\alpha_2). \\
&\quad\;\; [\, [f \mid \alpha \to s^+ f(v)](\alpha_2)]\!]^{6'} \\
&\Leftrightarrow\; s,h,[f \mid \alpha \to s^+ f(v)], r \in [\![[t \mid (\alpha,\alpha') \to t(v,\alpha') \mid (\alpha',\alpha) \to t(\alpha',v)]]\!]^3
\end{aligned}
$$

149

- when $v \notin \mathit{TVar}$, by domain constraints we have $\forall \alpha'.\{\ddagger_{eq}, \dagger_{eq}\} \cap t(\alpha, \alpha') \neq \emptyset \wedge \{\ddagger_{eq}, \dagger_{eq}\} \cap t(\alpha', \alpha) \neq \emptyset$

$$
\begin{aligned}
& \quad s,h,f,r \in [\![t]\!]^3 \\
\Leftrightarrow\ & \quad \forall \alpha_1, \alpha_2 \in TVar.[\![t(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\Leftrightarrow\ & \quad \forall \alpha_1, \alpha_2 \in TVar.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\Leftrightarrow\ & \left[
\begin{aligned}
& \bullet\ \ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \left[
\begin{aligned}
& \bullet\ \ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \left[
\begin{aligned}
& \bullet\ \ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \left[
\begin{aligned}
& \bullet\ \ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, f(\alpha), f(\alpha')]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), f(\alpha)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \left[
\begin{aligned}
& \bullet\ \ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha,\alpha').[\![l, \emptyset, f(\alpha')]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha',\alpha).[\![l, f(\alpha'), \emptyset]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \quad \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
\Leftrightarrow\ & \left[
\begin{aligned}
& \bullet\ \ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}.\exists l \in t(\alpha_1,\alpha_2).[\![l, f(\alpha_1), f(\alpha_2)]\!]^{6'} \\
& \bullet\ \ \exists l \in \{\ddagger_{eq}, =_{eq}\}.[\![l, s(v), s(v)]\!]^{6'} \\
& \bullet\ \ \exists l \in \{\ddagger_{eq}, \dagger_{eq}\}.[\![l, f(v), \emptyset]\!]^{6'} \\
& \bullet\ \ \exists l \in \{\ddagger_{eq}, \dagger_{eq}\}.[\![l, \emptyset, s(v)]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar.\exists l \in CL_{eq}.[\![l, f(\alpha'), s(v)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \left[
\begin{aligned}
& \bullet\ \ \forall \alpha_1, \alpha_2 \in TVar \setminus \{\alpha\}. \\
& \quad \exists l \in t(\alpha_1,\alpha_2).[\, [f \mid \alpha \rightarrow s^+ f(v)](\alpha_2)]\!]^{6'} \\
& \bullet\ \ \exists l \in \{\ddagger_{eq}, =_{eq}\}.[\, [f \mid \alpha \rightarrow s^+ f(v)](\alpha)]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.ad(\alpha') = \otimes \Rightarrow \exists l \in \{\ddagger_{eq}, \dagger_{eq}\}. \\
& \quad [\, [f \mid \alpha \rightarrow s^+ f(v)](\alpha')]\!]^{6'} \\
& \bullet\ \ \exists l \in \{\ddagger_{eq}, \dagger_{eq}\}.[\, [f \mid \alpha \rightarrow s^+ f(v)](\alpha)]\!]^{6'} \\
& \bullet\ \ \forall \alpha' \in TVar \setminus \{\alpha\}.ad(\alpha') \neq \otimes \Rightarrow \exists l \in CL_{eq}. \\
& \quad [\, [f \mid \alpha \rightarrow s^+ f(v)](\alpha)]\!]^{6'}
\end{aligned}
\right. \\
\Leftrightarrow\ & \quad s,h,[f \mid \alpha \rightarrow s^+ f(v)],r \in \left[\!\!\left[
\begin{aligned}
& t \\
& \mid (\alpha,\alpha') \rightarrow (ad(\alpha') = \otimes ? \{\ddagger_{eq}, \dagger_{eq}\} : CL_{eq}) \\
& \mid (\alpha',\alpha) \rightarrow (ad(\alpha') = \otimes ? \{\ddagger_{eq}, \dagger_{eq}\} : CL_{eq}) \\
& \mid (\alpha,\alpha) \rightarrow \{\ddagger_{eq}, =_{eq}\}
\end{aligned}
\right]\!\!\right]^3
\end{aligned}
$$

- we know that $f(\alpha) = \emptyset$, when $v \in Var$

$$
\begin{aligned}
& s, h, f, r \in \llbracket d \rrbracket^7 \\
\Leftrightarrow\ & \exists g \in \llbracket d \rrbracket^{\mathcal{P}}.\ \forall \alpha' \in TVar.\ f(\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\
\Leftrightarrow\ & \exists g \in \llbracket d \rrbracket^{\mathcal{P}}.\ \left[\begin{aligned} &\bullet\quad \forall \alpha' \neq \alpha.\ f(\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\ &\bullet\quad f(\alpha) \cap \mathbb{Z} \subseteq g(\alpha) \end{aligned} \right. \\
\Leftrightarrow\ & \exists g \in \llbracket d \rrbracket^{\mathcal{P}}.\ \forall \alpha' \neq \alpha.\ [f \mid \alpha \to s(v)](\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\
\Leftrightarrow\ & \exists g \in \llbracket d \rrbracket^{\mathcal{P}}.\ \left[\begin{aligned} &\bullet\quad \forall \alpha' \neq \alpha.\ [f \mid \alpha \to s(v)](\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\ &\bullet\quad s(v) \cap \mathbb{Z} \subseteq \mathbb{Z} \end{aligned} \right. \\
\Leftrightarrow\ & \exists g' \in \llbracket top(\alpha, d) \rrbracket^{\mathcal{P}}.\ \left[\begin{aligned} &\bullet\quad \forall \alpha' \neq \alpha.\ [f \mid \alpha \to s(v)](\alpha') \cap \mathbb{Z} \subseteq g'(\alpha') \\ &\bullet\quad s(v) \cap \mathbb{Z} \subseteq g'(\alpha) \end{aligned} \right. \\
\Leftrightarrow\ & \exists g' \in \llbracket top(\alpha, d) \rrbracket^{\mathcal{P}}.\ \forall \alpha' \in TVar.\ [f \mid \alpha \to s(v)](\alpha') \cap \mathbb{Z} \subseteq g'(\alpha') \\
\Leftrightarrow\ & s, h, [f \mid \alpha \to s(v)], r \in \llbracket top(\alpha, d) \rrbracket^d
\end{aligned}
$$

- we know that $f(\alpha) = \emptyset$, when $v \in TVar$

$$
\begin{aligned}
& s, h, f, r \in \llbracket d \rrbracket^7 \\
\Leftrightarrow\ & \exists g \in \llbracket d \rrbracket^{\mathcal{P}}.\ \forall \alpha' \in TVar.\ f(\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\
\Leftrightarrow\ & \exists g \in \llbracket d \rrbracket^{\mathcal{P}}.\ \left[\begin{aligned} &\bullet\quad \forall \alpha' \neq \alpha.\ f(\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\ &\bullet\quad f(\alpha) \cap \mathbb{Z} \subseteq g(\alpha) \end{aligned} \right. \\
\Leftrightarrow\ & \exists g \in \llbracket d \rrbracket^{\mathcal{P}}.\ \forall \alpha' \neq \alpha.\ [f \mid \alpha \to f(v)](\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\
\Leftrightarrow\ & \exists g \in \llbracket d \rrbracket^{\mathcal{P}}.\ \left[\begin{aligned} &\bullet\quad \forall \alpha' \neq \alpha.\ [f \mid \alpha \to f(v)](\alpha') \cap \mathbb{Z} \subseteq g(\alpha') \\ &\bullet\quad f(v) \cap \mathbb{Z} \subseteq g(v) \end{aligned} \right. \\
\Leftrightarrow\ & \exists g' \in \llbracket top(v, copy(v, \alpha, d)) \rrbracket^{\mathcal{P}}.\ \left[\begin{aligned} &\bullet\quad \forall \alpha' \neq \alpha.\ [f \mid \alpha \to f(v)](\alpha') \cap \mathbb{Z} \subseteq g'(\alpha') \\ &\bullet\quad f(v) \cap \mathbb{Z} \subseteq g'(\alpha) \end{aligned} \right. \\
\Leftrightarrow\ & \exists g' \in \llbracket top(v, copy(v, \alpha, d)) \rrbracket^{\mathcal{P}}.\ \forall \alpha' \in TVar.\ [f \mid \alpha \to f(v)](\alpha') \cap \mathbb{Z} \subseteq g'(\alpha') \\
\Leftrightarrow\ & s, h, [f \mid \alpha \to f(v)], r \in \llbracket top(v, copy(v, \alpha, d)) \rrbracket^d
\end{aligned}
$$

$\square$ $\blacksquare$

Proposition 3.41. $\forall \alpha \in TVar.v' \in VAR \setminus \{\alpha\}.S \in PVD^+.(s, h, f, r) \in MFR$ *when*

$\forall vd \in s^+ f(v).\alpha \text{``} \not\in'' vd$

$s, h, [f \mid \alpha \to \emptyset], r \in \llbracket v', S \rrbracket^5 \Leftrightarrow s, h, [f \mid \alpha \to s^+ f(v)], r \in \llbracket v', replace(v, \alpha, S) \rrbracket^5$

Prop. 3.41. $\bullet$ $\left[\begin{aligned} & s, h, [f \mid \alpha \to s^+ f(v)], r \in \llbracket v', replace(v, \alpha, \top) \rrbracket^5 \\ \Leftrightarrow\ & s, h, [f \mid \alpha \to s^+ f(v)], r \in \llbracket v', \top \rrbracket^5 \\ \Leftrightarrow\ & s, h, [f \mid \alpha \to s^+ f(v)], r \in MFR \\ \Leftrightarrow\ & True \end{aligned} \right.$

$$
\bullet \quad
\begin{bmatrix}
& s, h, [f \mid \alpha \to \emptyset], r \in [\![v', \circledast]\!]^5 \\
\Leftrightarrow & (s^+[f \mid \alpha \to \emptyset](v') = \emptyset) \\
\Leftrightarrow & (s^+ f(v') = \emptyset) \\
\Leftrightarrow & (s^+[f \mid \alpha \to s^+ f(v)](v') = \emptyset) \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![v', \circledast]\!]^5 \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![v', replace(v, \alpha, \circledast)]\!]^5
\end{bmatrix}
$$

- when $S \neq \top, \circledast$

$$
\begin{aligned}
& s, h, [f \mid \alpha \to \emptyset], r \in [\![v', S]\!]^5 \\
\Leftrightarrow \quad & s^+[f \mid \alpha \to \emptyset](v') \subseteq \bigcup_{vd \in S} [\![vd, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 \\
\Leftrightarrow \quad & s^+ f(v') \subseteq \bigcup_{vd \in S} [\![vd, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 \\
\Leftrightarrow \quad & \\
\text{by Prop. } 3.43 \quad & s^+ f(v') \subseteq \bigcup_{vd \in S} [\![replace(v, \alpha, vd), (h, [f \mid \alpha \to s^+ f(v)], r)]\!]^8 \\
\Leftrightarrow \quad & s^+[f \mid \alpha \to s^+ f(v)](v') \subseteq \bigcup_{vd \in S} [\![replace(v, \alpha, vd), (h, [f \mid \alpha \to s^+ f(v)], r)]\!]^8 \\
\Leftrightarrow \quad & s^+[f \mid \alpha \to s^+ f(v)](v') \subseteq \bigcup_{vd \in replace(v, \alpha, S)} [\![vd, (h, [f \mid \alpha \to s^+ f(v)], r)]\!]^8 \\
\Leftrightarrow \quad & s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![v', replace(v, \alpha, S)]\!]^5
\end{aligned}
$$

$\square$ $\blacksquare$

Proposition 3.42. $\forall \alpha \in TVar. S \in PVD^+. v \in VAR. (s, h, f, r) \in MFR \quad$ when $\forall vd \in s^+ f(v). \alpha \text{ `` } \not\in'' vd$

$$
s, h, [f \mid \alpha \to \emptyset], r \in [\![v, S]\!]^5 \Leftrightarrow s, h, [f \mid \alpha \to s^+(v)], r \in [\![\alpha, replace(v, \alpha, S)]\!]^5
$$

$$
\text{Prop. } 3.42. \quad \bullet \quad
\begin{bmatrix}
& s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![\alpha, replace(v, \alpha, \top)]\!]^5 \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![\alpha, \top]\!]^5 \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+ f(v)], r \in MFR \\
\Leftrightarrow & True
\end{bmatrix}
$$

$$
\bullet \quad
\begin{bmatrix}
& s, h, [f \mid \alpha \to \emptyset], r \in [\![v, \circledast]\!]^5 \\
\Leftrightarrow & (s^+[f \mid \alpha \to \emptyset](v) = \emptyset) \\
\Leftrightarrow & (s^+ f(v) = \emptyset) \\
\Leftrightarrow & (s^+[f \mid \alpha \to s^+ f(v)](\alpha) = \emptyset) \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![\alpha, \circledast]\!]^5 \\
\Leftrightarrow & s, h, [f \mid \alpha \to s^+ f(v)], r \in [\![\alpha, replace(v, \alpha, \circledast)]\!]^5
\end{bmatrix}
$$

- when $s^+ f(v) \neq \top, \circledast$

$$s, h, [f \mid \alpha \to \emptyset], r \in [\![v, S]\!]^5$$

$$\Leftrightarrow \quad s^+[f \mid \alpha \to \emptyset](v) \subseteq \bigcup_{vd \in S} [\![vd, (h, [f \mid \alpha \to \emptyset], r)]\!]^8$$

$$\Leftrightarrow \quad s^+f(v) \subseteq \bigcup_{vd \in S} [\![vd, (h, [f \mid \alpha \to \emptyset], r)]\!]^8$$

$$\Leftrightarrow$$
$$\text{by Prop. 3.43} \quad s^+f(v) \subseteq \bigcup_{vd \in S} [\![replace(v, \alpha, vd), (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$$

$$\Leftrightarrow \quad s^+[f \mid \alpha \to s^+f(v)](\alpha) \subseteq \bigcup_{vd \in S} replace(v, \alpha, vd), (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$$

$$\Leftrightarrow \quad s^+[f \mid \alpha \to s^+f(v)](\alpha) \subseteq \bigcup_{vd \in replace(v, \alpha, S)} vd, (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$$

$$\Leftrightarrow \quad s, h, [f \mid \alpha \to s^+f(v)], r \in [\![\alpha, replace(v, \alpha, S)]\!]^5$$

□   □

Proposition 3.43. $\forall \alpha \in TVar.l \in Val', vd \in VD, (h, f, r) \in (H \times F \times R).$ when $\alpha \text{``} \notin \text{''} vd$

$$[\![vd, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = [\![replace(v, \alpha, vd), (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$$

Prop. 3.43. We proceed by cases on vd

- $[\![Nilt, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = \{\texttt{nil}\} = [\![Nilt, (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$
  $= [\![replace(v, \alpha, Nilt), (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$

- $[\![True, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = \{True\} = [\![Truet, (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$
  $= [\![replace(v, \alpha, Truet), (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$

- $[\![Falset, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = \{False\} = [\![Falset, (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$
  $= [\![replace(v, \alpha, Falset), (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$

- $[\![Oodt, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = \{\texttt{ood}\} = [\![Oodt, (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$
  $= [\![replace(v, \alpha, Oodt), (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$

- when $[\![Numt, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = \mathbb{Z} = [\![Numt, (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8 =$
  $[\![replace(v, \alpha, Numt), (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$

- $[\![Dangling\_Loc, (h, [f \mid \alpha \to \emptyset], r)]\!]^8 = Loc \setminus dom(h) = [\![Dangling\_Loc, (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8 = [\![replace(v, \alpha, Dangling\_Loc), (h, [f \mid \alpha \to s^+f(v)], r)]\!]^8$

- when $\alpha' \neq \alpha, v$

  $[\![\alpha', (h, [f \mid \alpha \rightarrow \emptyset], r)]\!]^8 = [f \mid \alpha \rightarrow \emptyset](\alpha') = f(\alpha') = [f \mid \alpha \rightarrow s^+f(v)](\alpha') = [\![\alpha', (h, [f \mid \alpha \rightarrow s^+f(v)], r)]\!]^8 = [\![replace(v, \alpha, \alpha'), (h, [f \mid \alpha \rightarrow s^+f(v)], r)]\!]^8$

- $[\![v, (h, [f \mid \alpha \rightarrow \emptyset], r)]\!]^8 = [f \mid \alpha \rightarrow \emptyset](v) = f(v) = [f \mid \alpha \rightarrow s^+f(v)](\alpha) = [\![\alpha, (h, [f \mid \alpha \rightarrow s^+f(v)], r)]\!]^8 = [\![replace(v, \alpha, v), (h, [f \mid \alpha \rightarrow s^+f(v)], r)]\!]^8$

- $\begin{aligned}
  &\quad [\![Loc(A, vd1, vd2), (h, [f \mid \alpha \rightarrow \emptyset], r)]\!]^8 \\
  &= \left\{ l \in dom(h) \;\middle|\; \begin{bmatrix} \bullet \;\; \Pi_1(h(l)) \in [\![vd1, (h, [f \mid \alpha \rightarrow \emptyset], r)]\!]^8 \\ \bullet \;\; \Pi_2(h(l)) \in [\![vd2, (h, [f \mid \alpha \rightarrow \emptyset], r)]\!]^8 \\ \bullet \;\; *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r(l) \\ \bullet \;\; *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r(l) \end{bmatrix} \right\} \\
  &\quad \text{by previous cases, we get:} \\
  &= \left\{ l \in dom(h) \;\middle|\; \begin{bmatrix} \bullet \;\; \Pi_1(h(l)) \in [\![replace(v, \alpha, vd1), (h, [f \mid \alpha \rightarrow s^+f(v)], r)]\!]^8 \\ \bullet \;\; \Pi_2(h(l)) \in [\![replace(v, \alpha, vd2), (h, [f \mid \alpha \rightarrow s^+f(v)], r)]\!]^8 \\ \bullet \;\; *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r(l) \\ \bullet \;\; *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r(l) \end{bmatrix} \right\} \\
  &= [\![replace(v, \alpha, Loc(A, vd1, vd2)), (h, [f \mid \alpha \rightarrow s^+f(v)], r)]\!]^8
  \end{aligned}$

□   □

3.8.4   Union proofs

Corollary 3.4. $\forall ar_1, ar_2 \in AR. \; [\![ar_1]\!] \cup [\![ar_2]\!] \subseteq [\![union(ar_1, ar_2)]\!]$

Cor. 3.4. We only need to prove that: $\forall ar_1, ar_2 \in AR. \; [\![ar_1]\!] \subseteq [\![union(ar_1, ar_2)]\!]$ since *union* is commutive

$s, h \in [\![ar_1]\!] \Rightarrow$ (by def.)$\exists f, r. \; \bar{s}, h, f, r \in [\![ar_1]\!]'$

$\Rightarrow$ (by Prop. 3.3)$\exists f, r. \bar{s}, h, f, r \in [\![union(ar_1, ar_2)]\!]' \Rightarrow s, h \in [\![union(ar_1, ar_2)]\!]$ □   □

Proposition 3.3. $\forall ar_1, ar_2 \in AR. \; [\![ar_1]\!]' \cup [\![ar_2]\!]' \subseteq [\![union(ar_1, ar_2)]\!]'$

Prop. 3.3.   - It's direct to show that the constains on the domain remains.

- $$\begin{aligned}
&\llbracket (ad_1, hu_1, ho_1, sn_1, sn_1^\infty, t_1, d_1) \rrbracket' \\
&= \llbracket ad_1 \rrbracket^4 \cap \llbracket hu_1 \rrbracket^1 \cap \llbracket ho_1 \rrbracket^{1'} \cap \llbracket sn_1 \rrbracket^2 \cap \llbracket sn_1^\infty \rrbracket^{2'} \cap \llbracket t_1 \rrbracket^3 \llbracket d_1 \rrbracket^7 \cap sem* \\
&\subseteq \llbracket ad_1 \dot\sqcup ad_2 \rrbracket^4 \cap \llbracket hu_1 \cap hu_2 \rrbracket^1 \cap \llbracket ho_1 \cup ho_2 \rrbracket^{1'} \cap \llbracket sn_1 \cup sn_2 \rrbracket^2 \cap \llbracket sn_1^\infty \cup sn_2^\infty \rrbracket^{2'} \\
&\quad \cap \llbracket t_1 \dot\cup t_2 \rrbracket^3 \cap \llbracket d_1 \sqcup d_2 \rrbracket^7 \cap sem* \\
&= \llbracket union((ad_1, hu_1, ho_1, sn_1, sn_1^\infty, t_1, d_1), (ad_2, hu_2, ho_2, sn_2, sn_2^\infty, t_2, d_2))
\end{aligned}$$

- $$\begin{aligned}
&\llbracket ad_1 \rrbracket^4 \\
&= \bigcap_{v \in VAR} \llbracket v, ad_1(v) \rrbracket^5 \\
&\subseteq \bigcap_{v \in VAR} \llbracket v, ad_1(v) \sqcup ad_2(v) \rrbracket^5 \quad \text{by Prop. } 3.44 \\
&= \llbracket ad_1 \dot\sqcup ad_2 \rrbracket^4
\end{aligned}$$

- $$\begin{aligned}
&\llbracket hu_1 \rrbracket^1 \\
&= \bigcap_{\alpha \in hu_1} \{s, h, f, r \mid f(\alpha) \cap dom(h) \neq \emptyset\} \\
&\subseteq \bigcap_{\alpha \in hu_1 \cap h_2} \{s, h, f, r \mid f(\alpha) \cap dom(h) \neq \emptyset\} \\
&= \llbracket hu_1 \cap hu_2 \rrbracket^1
\end{aligned}$$

- $$\begin{aligned}
&\llbracket \texttt{full} \rrbracket^{1'} \\
&= MFR \\
&= \llbracket \texttt{full} \cup ho_2 \rrbracket^{1'}
\end{aligned}$$

- $$\begin{aligned}
&\llbracket ho_1 \rrbracket^{1'} \\
&= \{s, h, f, r \mid dom(h) \subseteq \bigcup_{\alpha \in ho_1} f(\alpha)\} \\
&\subseteq \{s, h, f, r \mid dom(h) \subseteq \bigcup_{\alpha \in ho_1 \cup ho_2} f(\alpha)\} \\
&= \llbracket ho_1 \cup ho_2 \rrbracket^{1'}
\end{aligned}$$

- $$\begin{aligned}
&\llbracket sn_1 \rrbracket^2 \\
&= \{s, h, f, r \mid \forall \alpha \in TVar \setminus sn_1. |f(\alpha)| \leq 1\} \\
&\subseteq \{s, h, f, r \mid \forall \alpha \in TVar \setminus (sn_1 \cup sn_2). |f(\alpha)| \leq 1\} \\
&= \llbracket sn_1 \cup sn_2 \rrbracket^2
\end{aligned}$$

- $$\begin{aligned}
&\llbracket sn_1^\infty \rrbracket^{2'} \\
&= \{s, h, f, r \mid \forall \alpha \in TVar \setminus sn_1^\infty. \ f(\alpha) \text{ is finite}\} \\
&\subseteq \{s, h, f, r \mid \forall \alpha \in TVar \setminus (sn_1^\infty \cup sn_2^\infty). \ f(\alpha) \text{ is finite}\} \\
&= \llbracket sn_1^\infty \cup sn_2^\infty \rrbracket^{2'}
\end{aligned}$$

$$
\bullet \quad
\begin{bmatrix}
& & \llbracket t_1 \rrbracket^3 \\
& = & \displaystyle\bigcap_{(\alpha_1,\alpha_2)\in TVar\times TVar} \left\{ s,h,f,r \mid \llbracket t_1(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2) \rrbracket^{6'} \right\} \\
& = & \displaystyle\bigcap_{(\alpha_1,\alpha_2)\in TVar\times TVar} \left\{ s,h,f,r \mid \bigvee_{l\in t_1(\alpha_1,\alpha_2)} \llbracket l, f(\alpha_1), f(\alpha_2) \rrbracket^6 \right\} \\
& \subseteq & \displaystyle\bigcap_{(\alpha_1,\alpha_2)\in TVar\times TVar} \left\{ s,h,f,r \mid \bigvee_{l\in t_1(\alpha_1,\alpha_2)\cup t_2(\alpha_1,\alpha_2)} \llbracket l, f(\alpha_1), f(\alpha_2) \rrbracket^6 \right\} \\
& = & \displaystyle\bigcap_{(\alpha_1,\alpha_2)\in TVar\times TVar} \left\{ s,h,f,r \mid \llbracket t_1(\alpha_1,\alpha_2)\cup t_2(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2) \rrbracket^{6'} \right\} \\
& = & \llbracket t_1 \dot\cup t_2 \rrbracket^3
\end{bmatrix}
$$

$$
\bullet \quad
\begin{bmatrix}
& & \llbracket d_1 \rrbracket^7 \\
& = & \displaystyle\bigcup_{g\in\llbracket d_1\rrbracket^{\mathcal{D}}} \bigcap_{\alpha\in TVar} \{s,h,f,r \mid f(\alpha)\cap\mathbb{Z} \subseteq g(\alpha)\} \\
& \subseteq & \displaystyle\bigcup_{g\in\llbracket d_1\sqcup^{\mathcal{D}} d_2\rrbracket^{\mathcal{D}}} \bigcap_{\alpha\in TVar} \{s,h,f,r \mid f(\alpha)\cap\mathbb{Z} \subseteq g(\alpha)\} \\
& = & \llbracket d_1 \sqcup^{\mathcal{D}} d_2 \rrbracket^7
\end{bmatrix}
$$

$\square$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

Proposition 3.44. $\forall v \in VAR.\ S_1, S_2 \in PVD^+$.

$\llbracket v, S_1 \rrbracket^5 \subseteq \llbracket v, S_1 \sqcup S_2 \rrbracket^5$

Prop. 3.44. $\quad \bullet\ \llbracket v, \top \rrbracket^5 = \llbracket v, \top \sqcup S_2 \rrbracket^5$

- $\llbracket v, S_1 \rrbracket^5 \subseteq MFR = \llbracket v, \top \rrbracket^5 = \llbracket v, S_1 \sqcup \top \rrbracket^5$

- $\llbracket v, S_1 \rrbracket^5 = \llbracket v, S_1 \sqcup \otimes \rrbracket^5$

- when $S_2 \neq \top, \otimes$
$$
\begin{aligned}
& \llbracket v, \otimes \rrbracket^5 \\
=\ & \{s,h,f,r \mid s^+ f(v) = \emptyset\} \\
\subseteq\ & \{s,h,f,r \mid \forall l \in s^+ f(v).\ ...\} \\
=\ & \llbracket v, S_2 \rrbracket^5 \\
=\ & \llbracket v, \otimes \sqcup S_2 \rrbracket^5
\end{aligned}
$$

- when $S_1, S_2 \neq \top, \otimes$
$$
\begin{aligned}
& \llbracket v, S_1 \rrbracket^5 \\
=\ & \{s,h,f,r \mid \forall l \in s^+ f(v).\ \exists vd \in S_1.....\} \\
\subseteq\ & \{s,h,f,r \mid \forall l \in s^+ f(v).\ \exists vd \in S_1 \cup S_2.....\} \\
=\ & \llbracket v, S_1 \cup S_2 \rrbracket^5 \\
=\ & \llbracket v, S_1 \sqcup S_2 \rrbracket^5
\end{aligned}
$$

□ □

3.8.5 Merging proofs

Corollary 3.7. $\forall \alpha_1, \alpha_2 \in TVar.ar \in AR. [\![ar]\!] \subseteq [\![merge(\alpha_1, \alpha_2, ar)]\!]$

Cor. 3.7. $s, h \in [\![ar]\!] \Rightarrow$ (by def.)$\exists f, r. \bar{s}, h, f, r \in [\![ar]\!]'$

$\Rightarrow$ (by Prop. 3.5)$\exists f, r.\bar{s}, h, merge(\alpha_1, \alpha_2, f), r \in [\![merge(\alpha_1, \alpha_2, ar)]\!]'$

$\Rightarrow s, h \in [\![merge(\alpha_1, \alpha_2, ar)]\!]$ □ □

Corollary 3.6. $\forall w \in (\bigcup_{n \in \mathbb{Z}} [0, n] \stackrel{total}{\to} (TVar \times TVar)). \forall ar \in AR. (s, h, f, r \in [\![ar]\!]' \Rightarrow s, h, merge(w, f), r \in [\![merge(w, ar)]\!]')$

Cor. 3.6. Direct by recurrence from Prop. 3.5. □ □

Proposition 3.8. $\forall \alpha_1, \alpha_2 \in TVar.\forall ar \in AR. used(merge(\alpha_1, \alpha_2, ar)) = used(ar) \setminus \{\alpha_1\}$

Prop. 3.8. Direct. □ □

Corollary 3.9. $\forall w \in (\bigcup_{n \in \mathbb{Z}} [0, n] \stackrel{total}{\to} (TVar \times TVar)). \forall ar \in AR. used(merge(w, ar)) = used(ar) \setminus fst(range(w))$

Cor. 3.9. Direct. □ □

Proposition 3.5. $\forall \alpha_1, \alpha_2 \in TVar.\forall ar \in AR. (s, h, f, r \in [\![ar]\!]' \Rightarrow s, h, merge(\alpha_1, \alpha_2, f), r \in [\![merge(\alpha_1, \alpha_2, ar)]\!]')$

Prop. 3.5.   • It's direct to show that the constains on the domain remains.

- $$\begin{aligned}
&\llbracket (ad, hu, ho, sn, sn^\infty, t, d) \rrbracket' \\
&= \llbracket ad \rrbracket^4 \cap \llbracket hu \rrbracket^1 \cap \llbracket ho \rrbracket^{1'} \cap \llbracket sn \rrbracket^2 \cap \llbracket sn^\infty \rrbracket^{2'} \cap \llbracket t \rrbracket^3 \cap \llbracket d \rrbracket^7 \cap sem* \\
&\subseteq \left(\begin{array}{ll}
& \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, ad) \rrbracket^4\} \\
\cap & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, hu) \rrbracket^1\} \\
\cap & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, h0) \rrbracket^{1'}\} \\
\cap & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, sn) \rrbracket^2\} \\
\cap & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, sn^\infty) \rrbracket^{2'}\} \\
\cap & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, t) \rrbracket^3\} \\
\cap & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, d) \rrbracket^7\} \\
\cap & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in sem*\}
\end{array} \right) \\
&= \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, (ad, hu, ho, sn, sn^\infty, t, d)) \rrbracket'\}
\end{aligned}$$

- $$\begin{aligned}
&\llbracket ad \rrbracket^4 \\
&= \bigcap_{v \in VAR} \llbracket v, ad(v) \rrbracket^5 \\
&\subseteq \bigcap_{v \in VAR} \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket v, merge(\alpha_1, \alpha_2, ad)(v) \rrbracket^5\} \\
&\quad \text{by Prop. } 3.45, 3.46 \\
&= \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, ad) \rrbracket^4\}
\end{aligned}$$

- $$\begin{aligned}
& \llbracket hu \rrbracket^1 \\
=\ & \bigcap_{\alpha \in hu} \{s,h,f,r \mid f(\alpha) \cap dom(h) \neq \emptyset\} \\
=\ & \bigcap_{\alpha \in hu} \{s,h,f,r \mid f(\alpha) \cap dom(h) \neq \emptyset \wedge \{\alpha_1, \alpha_2\} \not\subseteq hu\} \\
& \cup \bigcap_{\alpha \in hu} \{s,h,f,r \mid f(\alpha) \cap dom(h) \neq \emptyset \wedge \{\alpha_1, \alpha_2\} \subseteq hu\} \\
=\ & \bigcap_{\alpha \in hu} \{s,h,f,r \mid f(\alpha) \cap dom(h) \neq \emptyset \wedge \{\alpha_1, \alpha_2\} \not\subseteq hu\} \\
& \cup \left(\begin{aligned} & \bigcap_{\alpha \in hu \setminus \{\alpha_1,\alpha_2\}} \{s,h,f,r \mid f(\alpha) \cap dom(h) \neq \emptyset \wedge \{\alpha_1,\alpha_2\} \subseteq hu\} \\ & \cap \quad \{s,h,f,r \mid f(\alpha_1) \cap dom(h) \neq \emptyset \wedge \{\alpha_1,\alpha_2\} \subseteq hu\} \\ & \cap \quad \{s,h,f,r \mid f(\alpha_2) \cap dom(h) \neq \emptyset \wedge \{\alpha_1,\alpha_2\} \subseteq hu\} \end{aligned} \right) \\[2ex]
\subseteq\ & \bigcap_{\alpha \in hu \setminus \{\alpha_1,\alpha_2\}} \{s,h,f,r \mid f(\alpha) \cap dom(h) \neq \emptyset \wedge \{\alpha_1,\alpha_2\} \not\subseteq hu\} \\
& \cup \left(\begin{aligned} & \bigcap_{\alpha \in hu \setminus \{\alpha_1,\alpha_2\}} \{s,h,f,r \mid merge(\alpha_1,\alpha_2,f)(\alpha) \cap dom(h) \neq \emptyset \wedge \{\alpha_1,\alpha_2\} \subseteq hu\} \\ & \cap \quad \{s,h,f,r \mid (f(\alpha_1) \cup f(\alpha_2)) \cap dom(h) \neq \emptyset \wedge \{\alpha_1,\alpha_2\} \subseteq hu\} \end{aligned} \right) \\
=\ & \bigcap_{\alpha \in hu \setminus \{\alpha_1,\alpha_2\}} \{s,h,f,r \mid merge(\alpha_1,\alpha_2,f)(\alpha) \cap dom(h) \neq \emptyset \wedge \{\alpha_1,\alpha_2\} \not\subseteq hu\} \\
& \cup \bigcap_{\alpha \in hu \setminus \{\alpha_1\}} \{s,h,f,r \mid merge(\alpha_1,\alpha_2,f)(\alpha) \cap dom(h) \neq \emptyset \wedge \{\alpha_1,\alpha_2\} \subseteq hu\} \\
=\ & \{s,h,f,r \mid s,h,merge(\alpha_1,\alpha_2,f),r \in \llbracket \begin{aligned} & \text{if } \{\alpha_1, \alpha_2\} \not\subseteq hu \\ & \text{then } hu \setminus \{\alpha_1, \alpha_2\} \\ & \text{else } hu \setminus \{\alpha_1\} \end{aligned} \rrbracket^1 \}
\end{aligned}$$

- $$\begin{aligned}
& \llbracket \texttt{full} \rrbracket^{1'} \\
=\ & MFR \\
=\ & \{s,h,f,r \mid s,h,merge(\alpha_1,\alpha_2,f),r \in \llbracket \texttt{full} \rrbracket^{1'}\} \\
=\ & \{s,h,f,r \mid s,h,merge(\alpha_1,\alpha_2,f),r \in \llbracket merge(\alpha_1,\alpha_2,\texttt{full}) \rrbracket^{1'}\}
\end{aligned}$$

- case $\{\alpha_1,\alpha_2\} \cap ho = \emptyset$
$$\begin{aligned}
& \llbracket ho \rrbracket^{1'} \\
=\ & \{s,h,f,r \mid dom(h) \subseteq \bigcup_{\alpha \in ho} f(\alpha)\} \\
\subseteq\ & \{s,h,f,r \mid dom(h) \subseteq \bigcup_{\alpha \in ho} merge(\alpha_1,\alpha_2,f)(\alpha)\} \\
=\ & \{s,h,f,r \mid s,h,merge(\alpha_1,\alpha_2,f),r \in \llbracket ho \rrbracket^{1'}\} \\
=\ & \{s,h,f,r \mid s,h,merge(\alpha_1,\alpha_2,f),r \in \llbracket merge(\alpha_1,\alpha_2,ho) \rrbracket^{1'}\}
\end{aligned}$$

- case $\{\alpha_1,\alpha_2\} \cap ho \neq \emptyset$

$\llbracket ho \rrbracket^{1'}$

$= \{s,h,f,r \mid dom(h) \subseteq \bigcup_{\alpha \in ho} f(\alpha)\}$

$\subseteq \{s,h,f,r \mid dom(h) \subseteq \bigcup_{\alpha \in ho \setminus \{\alpha_1\} \cup \{\alpha_2\}} merge(\alpha_1, \alpha_2, f)(\alpha)\}$

$= \{s,h,f,r \mid s,h, merge(\alpha_1, \alpha_2, f), r \in \llbracket ho \setminus \{\alpha_1\} \cup \{\alpha_2\} \rrbracket^{1'}\}$

$= \{s,h,f,r \mid s,h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, ho) \rrbracket^{1'}\}$

- $\llbracket sn \rrbracket^2$

$= \{s,h,f,r \mid \forall \alpha \in TVar \setminus sn. |f(\alpha)| \leq 1\}$

$\subseteq \{s,h,f,r \mid \forall \alpha \in TVar \setminus (sn \setminus \{\alpha_1\} \cup \{\alpha_2\}). |merge(\alpha_1, \alpha_2, f)(\alpha)| \leq 1\}$

$= \{s,h,f,r \mid s,h, merge(\alpha_1, \alpha_2, f), r \in \llbracket sn \setminus \{\alpha_1\} \cup \{\alpha_2\} \rrbracket^2\}$

- when $\{\alpha_1, \alpha_2\} \cap sn^\infty = \emptyset$

$\llbracket sn^\infty \rrbracket^{2'}$

$= \{s,h,f,r \mid \forall \alpha \in TVar \setminus sn^\infty. \ f(\alpha) \text{ is finite}\}$

$\subseteq \{s,h,f,r \mid \forall \alpha \in TVar \setminus sn^\infty. \ merge(\alpha_1, \alpha_2, f)(\alpha) \text{ is finite}\}$

$= \{s,h,f,r \mid s,h, merge(\alpha_1, \alpha_2, f), r \in \llbracket sn^\infty \rrbracket^{2'}\}$

- when $\{\alpha_1, \alpha_2\} \cap sn^\infty \neq \emptyset$

$\llbracket sn^\infty \rrbracket^{2'}$

$= \{s,h,f,r \mid \forall \alpha \in TVar \setminus sn^\infty. \ f(\alpha) \text{ is finite}\}$

$\subseteq \{s,h,f,r \mid \forall \alpha \in TVar \setminus (sn^\infty \cup \{\alpha_2\}). \ f(\alpha) \text{ is finite}\}$

$= \{s,h,f,r \mid \forall \alpha \in TVar \setminus (sn^\infty \setminus \{\alpha_1\} \cup \{\alpha_2\}). \ merge(\alpha_1, \alpha_2, f)(\alpha) \text{ is finite}\}$

$= \{s,h,f,r \mid s,h, merge(\alpha_1, \alpha_2, f), r \in \llbracket sn^\infty \setminus \{\alpha_1\} \cup \{\alpha_2\} \rrbracket^{2'}\}$

$$
\begin{aligned}
&\llbracket t \rrbracket^3 \\
={}& \bigcap_{(\alpha_1,\alpha_2)\in TVar\times TVar} \big\{s,h,f,r \mid \llbracket t(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)\rrbracket^{6'}\big\} \\
={}& \bigcap_{(\alpha_1,\alpha_2),\,\alpha_1,\alpha_2\notin\{\alpha_1,\alpha_2\}} \big\{s,h,f,r \mid \llbracket t(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1,\alpha_2} \big\{s,h,f,r \mid \llbracket t(\alpha,\alpha_2), f(\alpha), f(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1,\alpha_2} \big\{s,h,f,r \mid \llbracket t(\alpha_2,\alpha), f(\alpha_2), f(\alpha)\rrbracket^{6'}\big\} \\
& \cap\big\{s,h,f,r \mid \llbracket t(\alpha_2,\alpha_2), f(\alpha_2), f(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1} \big\{s,h,f,r \mid \llbracket t(\alpha,\alpha_1), f(\alpha), f(\alpha_1)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1} \big\{s,h,f,r \mid \llbracket t(\alpha_1,\alpha), f(\alpha_1), f(\alpha)\rrbracket^{6'}\big\} \\
& \cap\big\{s,h,f,r \mid \llbracket t(\alpha_1,\alpha_1), f(\alpha_1), f(\alpha_1)\rrbracket^{6'}\big\} \\
\subseteq{}& \bigcap_{(\alpha_1,\alpha_2),\,\alpha_1,\alpha_2\notin\{\alpha_1,\alpha_2\}} \big\{s,h,f,r \mid \llbracket t(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1,\alpha_2} \big\{s,h,f,r \mid \llbracket UR_{eq}(t(\alpha,\alpha_1),t(\alpha,\alpha_2)), f(\alpha), f(\alpha_1)\cup f(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1,\alpha_2} \big\{s,h,f,r \mid \llbracket UL_{eq}(t(\alpha_1,\alpha),t(\alpha_2,\alpha)), f(\alpha_1)\cup f(\alpha_2), f(\alpha)\rrbracket^{6'}\big\}
\end{aligned}
$$

by Prop. 3.11 and 3.12

$$
\begin{aligned}
={}& \bigcap_{(\alpha_1,\alpha_2),\,\alpha_1,\alpha_2\notin\{\alpha_1,\alpha_2\}} \big\{s,h,f,r \mid \llbracket t(\alpha_1,\alpha_2), f(\alpha_1), f(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1,\alpha_2} \big\{s,h,f,r \mid \llbracket UR_{eq}(t(\alpha,\alpha_1),t(\alpha,\alpha_2)), f(\alpha), f(\alpha_1)\cup f(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1,\alpha_2} \big\{s,h,f,r \mid \llbracket UL_{eq}(t(\alpha_1,\alpha),t(\alpha_2,\alpha)), f(\alpha_1)\cup f(\alpha_2), f(\alpha)\rrbracket^{6'}\big\} \\
& \cap\big\{s,h,f,r \mid \llbracket \{\ddagger_{eq},=_{eq}\}, f(\alpha_1)\cup f(\alpha_2), f(\alpha_1)\cup f(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1} \big\{s,h,f,r \mid \llbracket \{\ddagger_{eq},\dagger_{eq}\}, f(\alpha), \emptyset\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1} \big\{s,h,f,r \mid \llbracket \{\ddagger_{eq},\dagger_{eq}\}, \emptyset, f(\alpha),\rrbracket^{6'}\big\} \\
& \cap\big\{s,h,f,r \mid \llbracket \{\ddagger_{eq}\}, \emptyset, \emptyset\rrbracket^{6'}\big\} \\
={}& \bigcap_{(\alpha_1,\alpha_2),\,\alpha_1,\alpha_2\notin\{\alpha_1,\alpha_2\}} \big\{s,h,f,r \mid \llbracket t(\alpha_1,\alpha_2), merge(\alpha_1,\alpha_2,f)(\alpha_1), merge(\alpha_1,\alpha_2,f)(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1,\alpha_2} \big\{s,h,f,r \mid \llbracket UR_{eq}(t(\alpha,\alpha_1),t(\alpha,\alpha_2)), merge(\alpha_1,\alpha_2,f)(\alpha), merge(\alpha_1,\alpha_2,f)(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1,\alpha_2} \big\{s,h,f,r \mid \llbracket UL_{eq}(t(\alpha_1,\alpha),t(\alpha_2,\alpha)), merge(\alpha_1,\alpha_2,f)(\alpha_2), merge(\alpha_1,\alpha_2,f)(\alpha)\rrbracket^{6'}\big\} \\
& \cap\big\{s,h,f,r \mid \llbracket \{\ddagger_{eq},=_{eq}\}, merge(\alpha_1,\alpha_2,f)(\alpha_2), merge(\alpha_1,\alpha_2,f)(\alpha_2)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1} \big\{s,h,f,r \mid \llbracket \{\ddagger_{eq},\dagger_{eq}\}, merge(\alpha_1,\alpha_2,f)(\alpha), merge(\alpha_1,\alpha_2,f)(\alpha_1)\rrbracket^{6'}\big\} \\
& \bigcap_{\alpha\neq\alpha_1} \big\{s,h,f,r \mid \llbracket \{\ddagger_{eq},\dagger_{eq}\}, merge(\alpha_1,\alpha_2,f)(\alpha_1), merge(\alpha_1,\alpha_2,f)(\alpha)\rrbracket^{6'}\big\} \\
& \cap\big\{s,h,f,r \mid \llbracket \{\ddagger_{eq}\}, merge(\alpha_1,\alpha_2,f)(\alpha_1), merge(\alpha_1,\alpha_2,f)(\alpha_1)\rrbracket^{6'}\big\} \\
={}& \left\{ s,h,f,r \;\middle|\; s,h, merge(\alpha_1,\alpha_2,f), r \in \left\llbracket \left[t \;\middle|\;
\begin{array}{l}
\mid (\alpha_2,\alpha) \to UR_{eq}(t(\alpha_1,\alpha),t(\alpha_2,\alpha)) \\
\mid (\alpha,\alpha_2) \to UL_{eq}(t(\alpha,\alpha_1),t(\alpha,\alpha_2)) \\
\mid (\_,\alpha_1) \to \{\ddagger_{eq},\dagger_{eq}\} \\
\mid (\alpha_1,\_) \to \{\ddagger_{eq},\dagger_{eq}\} \\
\mid (\alpha_1,\alpha_1) \to \{\ddagger_{eq}\}
\end{array} \right] \right\rrbracket^3 \right\}
\end{aligned}
$$

162

$$\bullet \quad \begin{cases} & s,h,f,r \in \llbracket d \rrbracket^7 \\ \Leftrightarrow & \exists g \in \llbracket d \rrbracket^{\mathcal{D}}.\ \forall \alpha.\ f(\alpha) \cap \mathbb{Z} \subseteq g(\alpha) \\ \Leftrightarrow & \exists g \in \llbracket d \rrbracket^{\mathcal{D}}. \begin{bmatrix} \bullet & \forall \alpha \neq \alpha_1, \alpha_2.\ f(\alpha) \cap \mathbb{Z} \subseteq g(\alpha) \\ \bullet & f(\alpha_1) \cap \mathbb{Z} \subseteq g(\alpha_1) \\ \bullet & f(\alpha_2) \cap \mathbb{Z} \subseteq g(\alpha_2) \end{bmatrix} \\ \Rightarrow & \exists g' \in \llbracket merge(\alpha_1, \alpha_2, d) \rrbracket^{\mathcal{D}}. \begin{bmatrix} \bullet & \forall \alpha \neq \alpha_1, \alpha_2.\ f(\alpha) \cap \mathbb{Z} \subseteq g'(\alpha) \\ \bullet & (f(\alpha_1) \cup f(\alpha_2)) \cap \mathbb{Z} \subseteq g'(\alpha_2) \end{bmatrix} \\ \Leftrightarrow & \exists g' \in \llbracket merge(\alpha_1, \alpha_2, d) \rrbracket^{\mathcal{D}}. \begin{bmatrix} \bullet & \forall \alpha \neq \alpha_1, \alpha_2.\ f(\alpha) \cap \mathbb{Z} \subseteq g'(\alpha) \\ \bullet & \emptyset \cap \mathbb{Z} \subseteq g'(\alpha_1) \\ \bullet & (f(\alpha_1) \cup f(\alpha_2)) \cap \mathbb{Z} \subseteq g'(\alpha_2) \end{bmatrix} \\ \Leftrightarrow & \exists g' \in \llbracket merge(\alpha_1, \alpha_2, d) \rrbracket^{\mathcal{D}}.\ \forall \alpha.\ merge(\alpha_1, \alpha_2, f)(\alpha) \cap \mathbb{Z} \subseteq g'(\alpha) \\ \Leftrightarrow & s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket merge(\alpha_1, \alpha_2, d) \rrbracket^7 \end{cases}$$

$\square$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

Proposition 3.45. $\forall \alpha_1, \alpha_2 \in TVar. \forall v \in VAR \setminus \{\alpha_1, \alpha_2\}.\ S \in PVD^+$.

$s, h, f, r \in \llbracket v, S \rrbracket^5 \Rightarrow s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket v, replace(\alpha_1, \alpha_2, S) \rrbracket^5$

Prop. 3.45. $\quad \bullet \begin{cases} & \llbracket v, \top \rrbracket^5 \\ = & MFR \\ = & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in MFR\} \\ = & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in \llbracket v, \top \rrbracket^5\} \\ = & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in \llbracket v, replace(\alpha_1, \alpha_2, \top) \rrbracket^5\} \end{cases}$

$\bullet \begin{cases} & \llbracket v, \circledast \rrbracket^5 \\ = & \{s, h, f, r \mid s^+ f(v) = \emptyset\} \\ = & \{s, h, f, r \mid s^+ merge(\alpha_1, \alpha_2, f)(v) = \emptyset\} \\ = & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in \llbracket v, \circledast \rrbracket^5\} \\ = & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in \llbracket v, replace(\alpha_1, \alpha_2, \circledast) \rrbracket^5\} \end{cases}$

$\bullet$ when $S \neq \top, \circledast$

$$\begin{aligned} & \llbracket v, S \rrbracket^5 \\ = & \left\{ s, h, f, r \mid s^+ f(v) \subseteq \bigcup_{vd \in S} . \llbracket vd, (h, f, r) \rrbracket^8 \right\} \\ \subseteq & \left\{ s, h, f, r \mid s^+ f(v) \subseteq \bigcup_{vd \in S} \llbracket replace(\alpha_1, \alpha_2, vd), (h, merge(\alpha_1, \alpha_2, f), r) \rrbracket^8 \right\} \text{by Prop. 3.47} \\ = & \left\{ s, h, f, r \mid s^+ merge(\alpha_1, \alpha_2, f)(v) \subseteq \bigcup_{vd \in replace(\alpha_1, \alpha_2, S)} \llbracket vd, (h, merge(\alpha_1, \alpha_2, f), r) \rrbracket^8 \right\} \\ = & \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in \llbracket v, replace(\alpha_1, \alpha_2, S) \rrbracket^5\} \end{aligned}$$

$\square$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

Proposition 3.46. $\forall \alpha_1, \alpha_2 \in TVar.\ S1, S2 \in PVD^+.$

$s, h, f, r \in [\![\alpha_1, S1]\!]^5 \cap [\![\alpha_2, S2]\!]^5 \Rightarrow s, h, merge(\alpha_1, \alpha_2, f), r \in [\![\alpha_1, replace(\alpha_1, \alpha_2, \circledast)]\!]^5 \cap$

$[\![\alpha_2, replace(\alpha_1, \alpha_2, S1 \sqcup S2)]\!]^5$

Prop. 3.46.   • when $S1 = S2 = \circledast$

$[\![\alpha_1, S1]\!]^5 \cap [\![\alpha_2, S2]\!]^5$

$= [\![\alpha_1, \circledast]\!]^5 \cap [\![\alpha_2, \circledast]\!]^5$

$= \{s, h, f, r \mid s^+f(\alpha_1) = \emptyset \wedge s^+f(\alpha_2) = \emptyset\}$

$= \{s, h, f, r \mid f(\alpha_1) = \emptyset \wedge f(\alpha_2) = \emptyset\}$

$= \{s, h, f, r \mid f(\alpha_2) \cup f(\alpha_1) = \emptyset\}$

$= \{s, h, f, r \mid s^+merge(\alpha_1, \alpha_2, f)(\alpha_1) = \emptyset \wedge s^+merge(\alpha_1, \alpha_2, f)(\alpha_2) = \emptyset\}$

$= \{s, h, f, r \mid s^+merge(\alpha_1, \alpha_2, f)(\alpha_1) = \emptyset \wedge s^+merge(\alpha_1, \alpha_2, f)(\alpha_2) = \emptyset\}$

$= \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in [\![\alpha_1, \circledast]\!]^5 \cap [\![\alpha_2, \circledast]\!]^5\}$

$= \left\{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in \left(\begin{array}{c} [\![\alpha_1, replace(\alpha_1, \alpha_2, \circledast)]\!]^5 \\ \cap [\![\alpha_2, replace(\alpha_1, \alpha_2, S1 \sqcup S2)]\!]^5 \end{array} \right) \right\}$

• when $S1 = \top$ and $S2 = \circledast$

$[\![\alpha_1, S1]\!]^5 \cap [\![\alpha_2, S2]\!]^5$

$= [\![\alpha_1, \top]\!]^5 \cap [\![\alpha_2, \circledast]\!]^5$

$= \{s, f, h, r \mid s^+f(\alpha_2) = \emptyset\}$

$= \{s, f, h, r \mid f(\alpha_2) = \emptyset\}$

$\subseteq MFR$

$= \{s, f, h, r \mid merge(\alpha_1, \alpha_2, f)(\alpha_1) = \emptyset\}$

$= \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in [\![\alpha_1, \circledast]\!]^5 \cap [\![\alpha_2, \top]\!]^5\}$

$= \left\{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in \left(\begin{array}{c} [\![\alpha_1, replace(\alpha_1, \alpha_2, \circledast)]\!]^5 \\ \cap [\![\alpha_2, replace(\alpha_1, \alpha_2, S1 \sqcup S2)]\!]^5 \end{array} \right) \right\}$

• when $S1 \neq \top, \circledast$ and $S2 = \circledast$

164

$$\llbracket \alpha_1, S1 \rrbracket^5 \cap \llbracket \alpha_2, S2 \rrbracket^5$$

$$= \llbracket \alpha_1, S1 \rrbracket^5 \cap \llbracket \alpha_2, \circledast \rrbracket^5$$

$$= \{s, h, f, r \mid s^+ f(\alpha_2) = \emptyset \wedge s^+ f(\alpha_1) \subseteq \bigcup_{vd \in S1} \llbracket vd, (h, f, r) \rrbracket^8 \}$$

$$= \{s, h, f, r \mid f(\alpha_2) = \emptyset \wedge f(\alpha_1) \subseteq \bigcup_{vd \in S1} \llbracket vd, (h, f, r) \rrbracket^8 \}$$

by Prop. 3.47 we get:

$$\subseteq \left\{ \begin{array}{l} s, h, merge(\alpha_1, \alpha_2, f), r \mid \\ f(\alpha_2) \cup f(\alpha_1) \subseteq \bigcup_{vd \in S1} \llbracket replace(\alpha_1, \alpha_2, vd), (h, merge(\alpha_1, \alpha_2, f), r) \rrbracket^8 \end{array} \right\}$$

$$= \left\{ \begin{array}{l} s, h, merge(\alpha_1, \alpha_2, f), r \mid \\ s^+ merge(\alpha_1, \alpha_2, f)(\alpha_1) = \emptyset \\ \wedge s^+ merge(\alpha_1, \alpha_2, f)(\alpha_2) \subseteq \bigcup_{vd \in replace(\alpha_1, \alpha_2, S1)} \llbracket vd, (h, merge(\alpha_1, \alpha_2, f), r) \rrbracket^8 \end{array} \right\}$$

$$= \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in \llbracket \alpha_1, \circledast \rrbracket^5 \cap \llbracket \alpha_2, replace(\alpha_1, \alpha_2, S1) \rrbracket^5 \}$$

$$= \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f)\}, r \in \left(\begin{array}{l} \llbracket \alpha_1, replace(\alpha_1, \alpha_2, \circledast) \rrbracket^5 \\ \cap \llbracket \alpha_2, replace(\alpha_1, \alpha_2, S1 \sqcup S2) \rrbracket^5 \end{array} \right) \}$$

- when $S2 = \top$

$$\llbracket \alpha_1, S1 \rrbracket^5 \cap \llbracket \alpha_2, S2 \rrbracket^5$$

$$\llbracket \alpha_1, S1 \rrbracket^5 \cap \llbracket \alpha_2, \top \rrbracket^5$$

$$\llbracket \alpha_1, S1 \rrbracket^5$$

$$\subseteq MFR$$

$$= \{s, h, f, r \mid merge(\alpha_1, \alpha_2, f)(\alpha_1) = \emptyset \}$$

$$= \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket \alpha_1, \circledast \rrbracket^5 \}$$

$$= \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket \alpha_1, \circledast \rrbracket^5 \cap \llbracket \alpha_2, \top \rrbracket^5 \}$$

$$= \{s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \left(\begin{array}{l} \llbracket \alpha_1, replace(\alpha_1, \alpha_2, \circledast) \rrbracket^5 \\ \cap \llbracket \alpha_2, replace(\alpha_1, \alpha_2, S1 \sqcup S2) \rrbracket^5 \end{array} \right) \}$$

- when $S1 = \circledast$ and $S2 \neq \top, \circledast$

$$\llbracket \alpha_1, S1 \rrbracket^5 \cap \llbracket \alpha_2, S2 \rrbracket^5$$

$$\llbracket \alpha_1, \circledast \rrbracket^5 \cap \llbracket \alpha_2, S2 \rrbracket^5$$

$$= \left\{ s, h, f, r \;\middle|\; \begin{array}{l} s^+f(\alpha_1) = \emptyset \wedge \\ s^+f(\alpha_2) \subseteq \bigcup_{vd \in S2} \llbracket vd, (h, f, r) \rrbracket^8 \end{array} \right\}$$

$$= \left\{ s, h, f, r \;\middle|\; \begin{array}{l} f(\alpha_1) = \emptyset \wedge \\ f(\alpha_1) \cup f(\alpha_2) \subseteq \bigcup_{vd \in S2} \llbracket vd, (h, f, r) \rrbracket^8 \end{array} \right\}$$

by Prop. 3.47, we get:

$$\subseteq \left\{ s, h, f, r \;\middle|\; f(\alpha_1) \cup f(\alpha_2) \subseteq \bigcup_{vd \in S2} \llbracket replace(\alpha_1, \alpha_2, vd), (h, merge(\alpha_1, \alpha_2, f), r) \rrbracket^8 \right\}$$

$$= \left\{ s, h, f, r \;\middle|\; \begin{array}{l} merge(\alpha_1, \alpha_2, f)(\alpha_1) = \emptyset \wedge \\ s^+merge(\alpha_1, \alpha_2, f)(\alpha_2) \subseteq \bigcup_{vd \in replace(\alpha_1, \alpha_2, S2)} \llbracket vd, (h, merge(\alpha_1, \alpha_2, f), r) \rrbracket^8 \end{array} \right\}$$

$$= \{ s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket \alpha_1, \circledast \rrbracket^5 \cap \llbracket \alpha_2, replace(\alpha_1, \alpha_2, S2) \rrbracket^5 \}$$

$$= \left\{ s, h, f, r \;\middle|\; s, h, merge(\alpha_1, \alpha_2, f), r \in \left(\begin{array}{l} \llbracket \alpha_1, replace(\alpha_1, \alpha_2, \circledast) \rrbracket^5 \\ \cap \llbracket \alpha_2, replace(\alpha_1, \alpha_2, S1 \sqcup S2) \rrbracket^5 \end{array} \right) \right\}$$

- when $S1 = \top$ and $S2 \neq \top, \circledast$

$$\llbracket \alpha_1, S1 \rrbracket^5 \cap \llbracket \alpha_2, S2 \rrbracket^5$$

$$\subseteq MFR$$

$$= \{ s, h, f, r \mid merge(\alpha_1, \alpha_2, f)(\alpha_1) = \emptyset \}$$

$$= \{ s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket \alpha_1, \circledast \rrbracket^5 \cap \llbracket \alpha_2, \top \rrbracket^5 \}$$

$$= \left\{ s, h, f, r \;\middle|\; s, h, merge(\alpha_1, \alpha_2, f), r \in \left(\begin{array}{l} \llbracket \alpha_1, replace(\alpha_1, \alpha_2, \circledast) \rrbracket^5 \\ \cap \llbracket \alpha_2, replace(\alpha_1, \alpha_2, S1 \sqcup S2) \rrbracket^5 \end{array} \right) \right\}$$

- when $S1, S2 \neq \top, \circledast$

$$\llbracket \alpha_1, S1 \rrbracket^5 \cap \llbracket \alpha_2, S2 \rrbracket^5$$

$$= \left\{ s, h, f, r \;\middle|\; \begin{array}{l} f(\alpha_1) \subseteq \bigcup_{vd \in S1} \llbracket vd, (h, f, r) \rrbracket^8 \wedge \\ f(\alpha_2) \subseteq \bigcup_{vd \in S2} \llbracket vd, (h, f, r) \rrbracket^8 \end{array} \right\}$$

$$\subseteq \left\{ s, h, f, r \;\middle|\; f(\alpha_1) \cup f(\alpha_2) \subseteq \bigcup_{vd \in S1 \cup S2} \llbracket vd, (h, f, r) \rrbracket^8 \right\}$$

by Prop. 3.47, we get:

$$\subseteq \left\{ s, h, f, r \;\middle|\; f(\alpha_1) \cup f(\alpha_2) \subseteq \bigcup_{vd \in S1 \cup S2} \llbracket replace(\alpha_1, \alpha_2, vd), (h, merge(\alpha_1, \alpha_2, f), r) \rrbracket^8 \right\}$$

$$= \left\{ s, h, f, r \;\middle|\; \begin{array}{l} merge(\alpha_1, \alpha_2, f)(\alpha_1) = \emptyset \wedge \\ s^+merge(\alpha_1, \alpha_2, f)(\alpha_2) \subseteq \bigcup_{vd \in replace(\alpha_1, \alpha_2, S1 \cup S2)} \llbracket vd, (h, merge(\alpha_1, \alpha_2, f), r) \rrbracket^8 \end{array} \right\}$$

$$= \{ s, h, f, r \mid s, h, merge(\alpha_1, \alpha_2, f), r \in \llbracket \alpha_1, \circledast \rrbracket^5 \cap \llbracket \alpha_2, replace(\alpha_1, \alpha_2, S1 \cup S2) \rrbracket^5 \}$$

$$= \left\{ s, h, f, r \;\middle|\; s, h, merge(\alpha_1, \alpha_2, f), r \in \left(\begin{array}{l} \llbracket \alpha_1, replace(\alpha_1, \alpha_2, \circledast) \rrbracket^5 \\ \cap \llbracket \alpha_2, replace(\alpha_1, \alpha_2, S1 \sqcup S2) \rrbracket^5 \end{array} \right) \right\}$$

□   □

Proposition 3.47. $\forall \alpha_1, \alpha_2 \in TVar.vd \in VD.(h, f, r) \in (H \times F \times R).$

$[\![vd, (h, f, r)]\!]^8 \subseteq [\![replace(\alpha_1, \alpha_2, vd), (h, merge(\alpha_1, \alpha_2, f), r)]\!]^8$

Prop. 3.47.   • $[\![replace(\alpha_1, \alpha_2, Nilt), \_]\!]^8 = [\![Nilt, \_]\!]^8$

• $[\![replace(\alpha_1, \alpha_2, Numt), \_]\!]^8 = [\![Numt, \_]\!]^8$

• $[\![replace(\alpha_1, \alpha_2, Dangling\_Loc), (h, \_, \_)]\!]^8 = [\![Dangling\_Loc, (h, \_, \_)]\!]^8$

• when $v \notin \{\alpha_1, \alpha_2\}$
$\quad\quad [\![replace(\alpha_1, \alpha_2, v), (\_, merge(\alpha_1, \alpha_2, f), \_)]\!]^8$
$=\quad [\![v, (\_, merge(\alpha_1, \alpha_2, f), \_)]\!]^8$
$=\quad merge(\alpha_1, \alpha_2, f)(v)$
$=\quad f(v)$
$=\quad [\![v, (\_, f, \_)]\!]^8$

$\bullet \left[\begin{array}{ll} & [\![replace(\alpha_1, \alpha_2, \alpha_2), (\_, merge(\alpha_1, \alpha_2, f), \_)]\!]^8 \\ = & [\![\alpha_2, (\_, merge(\alpha_1, \alpha_2, f), \_)]\!]^8 \\ = & merge(\alpha_1, \alpha_2, f)(\alpha_2) \\ = & f(\alpha_1) \cup f(\alpha_2) \\ \supseteq & f(\alpha_2) \\ = & [\![\alpha_2, (\_, f, \_)]\!]^8 \end{array} \right]$

$\bullet \left[\begin{array}{ll} & [\![replace(\alpha_1, \alpha_2, \alpha_1), (\_, merge(\alpha_1, \alpha_2, f), \_)]\!]^8 \\ = & [\![\alpha_2, (\_, merge(\alpha_1, \alpha_2, f), \_)]\!]^8 \\ = & merge(\alpha_1, \alpha_2, f)(\alpha_2) \\ = & f(\alpha_1) \cup f(\alpha_2) \\ \supseteq & f(\alpha_1) \\ = & [\![\alpha_1, (\_, f, \_)]\!]^8 \end{array} \right]$

$$
\bullet \quad
\begin{bmatrix}
& \llbracket replace(\alpha_1, \alpha_2, Loc(A, vd1, vd2)), (h, merge(\alpha_1, \alpha_2, f), r)\rrbracket^8 \\
= & \llbracket Loc(A, replace(\alpha_1, \alpha_2, vd1), replace(\alpha_1, \alpha_2, vd2)), (h, merge(\alpha_1, \alpha_2, f), r)\rrbracket^8 \\
= & \left\{ l \in dom(h) \middle|
\begin{bmatrix}
\bullet & \Pi_1(h(l)) \in \llbracket replace(\alpha_1, \alpha_2, vd1), (h, merge(\alpha_1, \alpha_2, f), r)\rrbracket^8 \\
\bullet & \Pi_2(h(l)) \in \llbracket replace(\alpha_1, \alpha_2, vd2), (h, merge(\alpha_1, \alpha_2, f), r)\rrbracket^8 \\
\bullet & *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r(l) \\
\bullet & *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r(l)
\end{bmatrix}
\right\} \\
\supseteq & \left\{ l \in dom(h) \middle|
\begin{bmatrix}
\bullet & \Pi_1(h(l)) \in \llbracket vd1, (h, f, r)\rrbracket^8 \\
\bullet & \Pi_2(h(l)) \in \llbracket vd2, (h, f, r)\rrbracket^8 \\
\bullet & *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r(l) \\
\bullet & *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r(l)
\end{bmatrix}
\right\} \\
= & \llbracket Loc(A, vd1, vd2), (h, f, r)\rrbracket^8
\end{bmatrix}
$$

$\square$ $\qquad$ $\square$

Proposition 3.10. $\forall A \in \mathcal{P}(TVar). \forall d' \in \mathcal{D}.\ A$ *is finite* $\Rightarrow \{(ad, hu, ho, sn, sn^\infty, t, d) \in AR \mid d = d' \wedge used(AR) \subseteq A\}$ *is finite*

Prop. 3.10. This is direct by the cardinals of subdomains of AR.

- $|VD1| \le 6 + |A|$

- $|VD| \le |VD1| + 4|VD1|^2$

- $|PVD^+| \le 1 + 2^{|VD|}$

- $|AD| \le |A| \times |PVD^+|$

- $|CL_{eq}| \le 128$

- $|TB| \le |CL_{eq}| \times |A|^2$

- $|AR| \le |AD| \times 2^{|A|} \times (1 + 2^{|A|}) \times 2^{|A|} \times 2^{|A|} \times \times |TB| \times |\mathcal{D}|$

So $\forall$ $\square$ $\qquad$ $\square$

3.8.6 Functions on CL_{eq} proofs

Proposition 3.11. $\forall S_{eq}^1, S_{eq}^2 \in CL_{eq}.\ \forall S, S^1, S^2 \in \mathcal{P}(Val').\ \llbracket S_{eq}^1, S, S^1\rrbracket^{6'} \wedge \llbracket S_{eq}^2, S, S^2\rrbracket^{6'} \Rightarrow \llbracket UR_{eq}(S_{eq}^1, S_{eq}^2), S, S^1 \cup S^2\rrbracket^{6'}$

$$\llbracket S^1_{eq}, S, S^1 \rrbracket^{6'} \wedge \llbracket S^2_{eq}, S, S^2 \rrbracket^{6'}$$

$$= \left(\bigvee_{l \in S^1_{eq}} \llbracket l, S, S^1 \rrbracket^6 \right) \wedge \left(\bigvee_{l \in S^2_{eq}} \llbracket l, S, S^2 \rrbracket^6 \right)$$

$$= \bigvee_{l_1,l_2 \in S^1_{eq} \times S^2_{eq}} \llbracket l_1, S, S^1 \rrbracket^6 \wedge \llbracket l_2, S, S^2 \rrbracket^6$$

$$\underset{\text{Prop. 3.11.}}{\Rightarrow} \bigvee_{l_1,l_2 \in S^1_{eq} \times S^2_{eq}} \bigvee_{l \in ur_{eq}(l_1,l_2)} \llbracket l, S, S^1 \cup S^2 \rrbracket^6$$

by construction of ur_{eq}

$$= \bigvee_{l \in \bigcup_{l_1,l_2 \in S^1_{eq} \times S^2_{eq}} ur_{eq}(l_1,l_2)} \llbracket l, S, S^1 \cup S^2 \rrbracket^6$$

$$= \bigvee_{l \in UR_{eq}(S^1_{eq}, S^2_{eq})} \llbracket l, S, S^1 \cup S^2 \rrbracket^6$$

$$= \llbracket UR_{eq}(S^1_{eq}, S^2_{eq}), S, S^1 \cup S^2 \rrbracket^{6'}$$

□ □

Proposition 3.12. $\forall S^1_{eq}, S^2_{eq} \in CL_{eq}. \ \forall S, S^1, S^2 \in \mathcal{P}(Val'). \ \llbracket S^1_{eq}, S^1, S \rrbracket^{6'} \wedge \llbracket S^2_{eq}, S^2, S \rrbracket^{6'} \Rightarrow \llbracket UL_{eq}(S^1_{eq}, S^2_{eq}), S^1 \cup S^2, S \rrbracket^{6'}$

Prop. 3.12. Same as for Prop. 3.12.□ □

3.8.7 Widening proofs

Proposition 3.16. $\forall w \in \mathbb{N} \overset{total}{\to} AR. \ \exists A \in \mathcal{P}(AR). \ (A \text{ is finite}) \wedge \exists i \in \mathbb{N}. \forall i' \geq i. \ \nabla^{AR}(w \upharpoonright_{[0,i']^{total} \to AR}) \in A$

Prop. 3.16. Let $(ad'_i, hu'_i, ho'_i, sn'_i, sn'^{\infty}_i, t'_i, d'_i) = \nabla^{AR}(w \upharpoonright_{[0,i]^{total} \to AR})$.

Let $(adm_i, hum_i, hom_i, snm_i, snm^{\infty}_i, tm_i, dm_i) = merge(\nabla^{merge}(w \upharpoonright_{[0,i]^{total} \to AR}), w(i))$.

We have $(adm_i, hum_i, hom_i, snm_i, snm^{\infty}_i, tm_i, dm_i) = (ad'_i, hu'_i, ho'_i, sn_i, sn'^{\infty}_i, t'_i, dm_i)$.

Remember that we have :

$$\forall w \in \mathbb{N} \overset{total}{\to} \mathcal{D}. \ \exists i_1 \in \mathbb{N}. \forall i' \geq i_1. \ \nabla^{\mathcal{D}}(w \upharpoonright_{[0,i']^{total} \to \mathcal{D}}) = \nabla^{\mathcal{D}}(w \upharpoonright_{[0,i]^{total} \to \mathcal{D}})$$

so

$$\forall w \in \mathbb{N} \overset{total}{\to} AR. \ \exists i_1 \in \mathbb{N}. \forall i' \geq i_1. \ dm'_i = dm_i$$

and

$$\forall w \in \mathbb{N} \overset{total}{\to} AR. \ \exists A \in \mathcal{P}(TVar). \ (A \text{ is finite}) \wedge \exists i_2 \in \mathbb{N}. \forall i' \geq i_2.$$
$$(used(merge(\nabla^{merge}(w \upharpoonright_{[0,i']^{total} \to AR}), w(i'))) \subseteq A)$$

Thus we have (for $i = max(i_1, i_2)$)

$$\forall w \in \mathbb{N} \stackrel{total}{\rightarrow} AR. \; \exists A \in \mathcal{P}(TVar). \; (A \text{ is finite}) \wedge \exists i \in \mathbb{N}. \forall i' \geq i.$$
$$(dm'_i = dm_i) \wedge (used(ad'_i, hu'_i, ho'_i, sn_i, sn'^{\infty}_i, t'_i, dm_i) \subseteq A)$$

So by Prop. 3.10, we have that

$$\forall w \in \mathbb{N} \stackrel{total}{\rightarrow} AR. \; \exists i \in \mathbb{N}. \{(ad'_{i'}, hu'_{i'}, ho'_{i'}, sn_{i'}, sn'^{\infty}_{i'}, t'_{i'}, dm_{i'}) | i' \geq i\} \text{ is finite}$$

$\square$ $\blacksquare$

Proposition 3.17. $\forall w \in \mathbb{N} \stackrel{total}{\rightarrow} AR. \; \forall i \in \mathbb{N}. \; \forall s, h, f, r. \exists g. \; s, h, f, r \in [\![w(i)]\!]' \Rightarrow s, h, g(f), r \in$ $[\![\nabla^{AR}(w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR})]\!]'$

Prop. 3.17. Let $(ad, hu, ho, sn, sn^{\infty}, t, d) = w(i)$,

let $(ad', hu', ho', sn', sn'^{\infty}, t', d') = \nabla^{AR}(w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR})$.

We want $\forall s, h, f, r. \exists g.$

$s, h, f, r \in [\![w(i)]\!]' \Rightarrow s, h, g(f), r \in [\![ad']\!]^4 \cap [\![hu']\!]^1 \cap [\![ho']\!]^{1'} \cap [\![sn']\!]^2 \cap [\![sn'^{\infty}]\!]^{2'} \cap [\![t']\!]^3 \cap [\![d']\!]^7 \cap$

$sem*$

Recall :

$$\nabla^{AR}(w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR})$$
$$= \nabla^{\mathcal{D}}_{AR} \left(\left[w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR} | \; i \rightarrow merge(\nabla^{merge}(w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR}), w(i)) \right] \right)$$
$$= set\_d(merge(\nabla^{merge}(w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR}), w(i)),$$
$$\nabla^{\mathcal{D}}(give\_d(\left[w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR} | \; i \rightarrow merge(\nabla^{merge}(w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR}), w(i)) \right])))$$

Let $(ad_m, hu_m, ho_m, sn_m, sn^{\infty}_m, t_m, d_m) = merge(\nabla^{merge}(w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR}), w(i))$.

We have $(ad_m, hu_m, ho_m, sn_m, sn^{\infty}_m, t_m, d_m) = (ad', hu', ho', sn, sn'^{\infty}, t', d_m)$.

By Cor. 3.6 we have

$s, h, f, r \in [\![w(i)]\!]' \Rightarrow s, h, merge(\nabla^{merge}(w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR}), f), r \in [\![merge(\nabla^{merge}(w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR}), w(i))]\!]'$

which is

$s, h, f, r \in [\![w(i)]\!]' \Rightarrow s, h, merge(\nabla^{merge}(w \upharpoonright_{[0,i] \stackrel{total}{\rightarrow} AR}), f), r \in [\![ad']\!]^4 \cap [\![hu']\!]^1 \cap [\![ho']\!]^{1'} \cap$

$[\![sn']\!]^2 \cap [\![sn'^{\infty}]\!]^{2'} \cap [\![t']\!]^3 \cap [\![d_m]\!]^7 \cap sem*$

We now prove that $[\![d_m]\!]^7 \subseteq [\![d']\!]^7$:

We have $d' = \nabla^{\mathcal{D}}(give\_d(\left[w \upharpoonright_{[0,i]}{}^{total}_{\to}{}_{AR}\mid i \to merge(\nabla^{merge}(w \upharpoonright_{[0,i]}{}^{total}_{\to}{}_{AR}), w(i))\right]))$

by constrains on $\nabla^{\mathcal{D}}$,

$$\forall g_1 \in [\![give\_d(merge(\nabla^{merge}(w \upharpoonright_{[0,i]}{}^{total}_{\to}{}_{AR}), w(i)))]\!]^{\mathcal{D}}. \exists g_2 \in [\![d']\!]^{\mathcal{D}}. \forall \alpha.g_1(\alpha) \subseteq g_2(\alpha)$$

which is

$$\forall g_1 \in [\![d_m]\!]^{\mathcal{D}}. \exists g_2 \in [\![d']\!]^{\mathcal{D}}. \forall \alpha.g_1(\alpha) \subseteq g_2(\alpha)$$

We can now prove that $[\![d_m]\!]^7 \subseteq [\![d']\!]^7$

$$
\begin{aligned}
& s, h, f, r \in [\![d_m]\!]^7 \\
\Leftrightarrow\ & \exists g_1 \in [\![d_m]\!]^{\mathcal{D}}. \forall \alpha \in TVar.f(\alpha) \cap \mathbb{Z} \subseteq g_1(\alpha) \\
\Leftrightarrow\ & \exists g_1 \in [\![d_m]\!]^{\mathcal{D}}. \left[
\begin{aligned}
&\bullet\ \ \forall \alpha \in TVar.f(\alpha) \cap \mathbb{Z} \subseteq g_1(\alpha) \\
&\bullet\ \ \exists g_2 \in [\![d']\!]^{\mathcal{D}}. g_1(\alpha) \subseteq g_2(\alpha)
\end{aligned}
\right. \\
\Leftrightarrow\ & \exists g_1 \in [\![d_m]\!]^{\mathcal{D}}. \left[
\begin{aligned}
&\bullet\ \ \forall \alpha \in TVar.f(\alpha) \cap \mathbb{Z} \subseteq g_1(\alpha) \\
&\bullet\ \ \exists g_2 \in [\![d']\!]^{\mathcal{D}}. g_1(\alpha) \subseteq g_2(\alpha) \\
&\bullet\ \ \exists g_2 \in [\![d']\!]^{\mathcal{D}}. \forall \alpha \in TVar.f(\alpha) \cap \mathbb{Z} \subseteq g_2(\alpha)
\end{aligned}
\right. \\
\Rightarrow\ & \exists g_2 \in [\![d']\!]^{\mathcal{D}}. \forall \alpha \in TVar.f(\alpha) \cap \mathbb{Z} \subseteq g_2(\alpha) \\
\Leftrightarrow\ & s, h, f, r \in [\![d']\!]^7
\end{aligned}
$$

□   □

3.8.8 Basic ast proofs

Proposition 3.21.

$$\bigcup_{vd \in vd_0} [\![vd, (s, h_0, f_0, r_0)]\!]^8 \cap \bigcup_{vd \in vd_1} [\![vd, (s, h_1, f_1, r_1)]\!]^8 \subseteq \bigcup_{vd \in basic\_ast(vd_0, vd_1)} [\![vd, (s, h, f, r)]\!]^8$$

if we define $\bigcup_{vd \in \Omega} [\![vd, (s, h, f, r)]\!]^8 \triangleq \emptyset$ *and* $\bigcup_{vd \in \top} [\![vd, (s, h, f, r)]\!]^8 \triangleq MFR$

Prop. 3.21. If we write the proof for $basic\_ast(S_1, S_2)$ then we will not write the proof for $basic\_ast(S_2, S_1)$, also if we prove $basic\_ast(S, \{A\})$ then we do not prove $basic\_ast(S, \{B\})$ except for $S = \{A\}$.

- Cases $\{A\}$ with

  - $\{A\}$

    from the definition of $\llbracket \cdot \rrbracket^8$ we have $\llbracket A, (s_0, h_0, f_0, r_0) \rrbracket^8 = \llbracket A, (s_1, h_1, f_1, r_1) \rrbracket^8 = \llbracket A, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in basic\_ast(\{A\}, \{A\})} \llbracket vd, (s, h, f, r) \rrbracket^8$

  - S for other cases not $\top$

    from the definition of $\llbracket \cdot \rrbracket^8$ we have $\llbracket A, (s_0, h_0, f_0, r_0) \rrbracket^8 \cap \llbracket B, (s_1, h_1, f_1, r_1) \rrbracket^8 = \emptyset = \bigcup_{vd \in \Omega} \llbracket vd, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in basic\_ast(\{A\}, S)} \llbracket vd, (s, h, f, r) \rrbracket^8$

  - $\top$

    $\llbracket A, (s_0, h_0, f_0, r_0) \rrbracket^8 \cap \bigcup_{vd \in \top} \llbracket vd, (s_1, h_1, f_1, r_1) \rrbracket^8 = \llbracket A, (s_0, h_0, f_0, r_0) \rrbracket^8 = \llbracket A, (s, h, f, r) \rrbracket^8 = basic\_ast(\{A\}, \top) \llbracket basic\_ast(\{A\}, \{A\}), (s, h, f, r) \rrbracket^8$

- $\{Dgt\}$ with

  - $\{Dgt\}$

    $\bigcup_{vd \in \{Dgt\}} \llbracket vd, (s, h_0, f_0, r_0) \rrbracket^8 \cap \bigcup_{vd \in \{Dgt\}} \llbracket vd, (s, h_1, f_1, r_1) \rrbracket^8 = \llbracket Dgt, (s, h_0, f_0, r_0) \rrbracket^8 \cap \llbracket Dgt, (s, h_1, f_1, r_1) \rrbracket^8 = (Loc \setminus dom(h_0)) \cap (Loc \setminus dom(h_1)) = Loc \setminus (dom(h_0) \cup dom(h_1)) = Loc \setminus dom(h) = \llbracket Dgt, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in \{Dgt\}} \llbracket vd, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in basic\_ast(\{Dgt\}, \{Dgt\})} \llbracket vd, (s, h, f, r) \rrbracket^8$

  - $\{Loct\}$

    $\bigcup_{vd \in \{Dgt\}} \llbracket vd, (s, h_0, f_0, r_0) \rrbracket^8 \cap \bigcup_{vd \in \{Loct\}} \llbracket vd, (s, h_1, f_1, r_1) \rrbracket^8 = \llbracket Dgt, (s, h_0, f_0, r_0) \rrbracket^8 \cap \llbracket Loct, (s, h_1, f_1, r_1) \rrbracket^8 = (Loc \setminus dom(h_0)) \cap dom(h_1) = dom(h_1)$ since $h_0 \sharp h_1 \subseteq dom(h) = \llbracket Loct, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in \{Loct\}} \llbracket vd, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in basic\_ast(\{Dgt\}, \{Loct\})} \llbracket vd, (s, h, f, r) \rrbracket^8$

– $\{Dgt, Loct\}$

$$\bigcup_{vd \in \{Dgt\}} [\![vd, (s, h_0, f_0, r_0)]\!]^8 \cap \bigcup_{vd \in \{Dgt, Loct\}} [\![vd, (s, h_1, f_1, r_1)]\!]^8 = [\![Dgt, (s, h_0, f_0, r_0)]\!]^8 \cap$$

$$([\![Dgt, (s, h_1, f_1, r_1)]\!]^8 \cup [\![Loct, (s, h_1, f_1, r_1)]\!]^8) = (Loc \backslash dom(h_0)) \cap (Loc \backslash dom(h_1) \cup$$

$$dom(h_1)) = (Loc \backslash dom(h_0)) \subseteq Loc = (Loc \backslash dom(h) \cup dom(h)) = ([\![Dgt, (s, h, f, r)]\!]^8 \cup$$

$$[\![Loct, (s, h, f, r)]\!]^8) = \bigcup_{vd \in \{Dgt, Loct\}} [\![vd, (s, h, f, r)]\!]^8$$

$$= \bigcup_{vd \in basic\_ast(\{Dgt\}, \{Dgt, Loct\})} [\![vd, (s, h, f, r)]\!]^8$$

– $\{Loc(A, vd1, vd2)\}$

$$\bigcup_{vd\in\{Dgt\}} [\![vd,(s,h_0,f_0,r_0)]\!]^8$$

$$\cap \bigcup_{vd\in\{Loc(A,vd1,vd2)\}} [\![vd,(s,h_1,f_1,r_1)]\!]^8$$

$$= [\![Dgt,(s,h_0,f_0,r_0)]\!]^8$$

$$\cap [\![Loc(A,vd1,vd2),(s,h_1,f_1,r_1)]\!]^8$$

$$= (Loc\setminus dom(h_0))$$

$$\cap \left\{ l\in dom(h_1) \;\middle|\; \begin{bmatrix} \bullet & \Pi_1(h_1(l))\in[\![vd1,(h_1,f_1,r_1)]\!]^8 \\ \bullet & \Pi_2(h_1(l))\in[\![vd2,(h_1,f_1,r_1)]\!]^8 \\ \bullet & *1\in A\wedge\Pi_1(h_1(l))\in Loc\Rightarrow\Pi_1(h_1(l))\in r_1(l) \\ \bullet & *2\in A\wedge\Pi_2(h_1(l))\in Loc\Rightarrow\Pi_2(h_1(l))\in r_1(l) \end{bmatrix} \right\}$$

since $h=h_0\cdot h_1$, we have:

$$= \left\{ l\in dom(h_1) \;\middle|\; \begin{bmatrix} \bullet & \Pi_1(h(l))\in[\![vd1,(h_1,f_1,r_1)]\!]^8 \\ \bullet & \Pi_2(h(l))\in[\![vd2,(h_1,f_1,r_1)]\!]^8 \\ \bullet & *1\in A\wedge\Pi_1(h(l))\in Loc\Rightarrow\Pi_1(h(l))\in r_1(l) \\ \bullet & *2\in A\wedge\Pi_2(h(l))\in Loc\Rightarrow\Pi_2(h(l))\in r_1(l) \end{bmatrix} \right\}$$

from Lem. 3.22, we get:

$$\subseteq \left\{ l\in dom(h_1) \;\middle|\; \begin{bmatrix} \bullet & \Pi_1(h(l))\in[\![vd1,(h,f,r)]\!]^8 \\ \bullet & \Pi_2(h(l))\in[\![vd2,(h,f,r)]\!]^8 \\ \bullet & *1\in A\wedge\Pi_1(h(l))\in Loc\Rightarrow\Pi_1(h(l))\in r_1(l) \\ \bullet & *2\in A\wedge\Pi_2(h(l))\in Loc\Rightarrow\Pi_2(h(l))\in r_1(l) \end{bmatrix} \right\}$$

$$\subseteq \left\{ l\in dom(h) \;\middle|\; \begin{bmatrix} \bullet & \Pi_1(h(l))\in[\![vd1,(h,f,r)]\!]^8 \\ \bullet & \Pi_2(h(l))\in[\![vd2,(h,f,r)]\!]^8 \\ \bullet & *1\in A\wedge\Pi_1(h(l))\in Loc\Rightarrow\Pi_1(h(l))\in r(l) \\ \bullet & *2\in A\wedge\Pi_2(h(l))\in Loc\Rightarrow\Pi_2(h(l))\in r(l) \end{bmatrix} \right\}$$

$$= [\![Loc(A,vd1,vd2),(s,h,f,r)]\!]^8$$

$$= \bigcup_{vd\in\{Loc(A,vd1,vd2)\}} [\![vd,(s,h,f,r)]\!]^8$$

$$= \bigcup_{vd\in basic\_ast(\{Dgt\},\{Loc(A,vd1,vd2)\})} [\![vd,(s,h,f,r)]\!]^8$$

$- \top$

$$\bigcup_{vd\in\{Dgt\}} [\![vd,(s,h_0,f_0,r_0)]\!]^8 \cap \bigcup_{vd\in\top} [\![vd,(s,h_1,f_1,r_1)]\!]^8 = [\![Dgt,(s,h_0,f_0,r_0)]\!]^8 = Loc\setminus$$

$$dom(h_0)\subseteq Loc = (Loc\setminus dom(h)\cup dom(h)) = ([\![Dgt,(s,h,f,r)]\!]^8\cup[\![Loct,(s,h,f,r)]\!]^8) =$$

$$\bigcup_{vd\in\{Dgt,Loct\}} [\![vd,(s,h,f,r)]\!]^8 = \bigcup_{vd\in basic\_ast(\{Dgt\},\top)} [\![vd,(s,h,f,r)]\!]^8$$

- $\{Loct\}$ with

- $\{Loct\}$

$$\bigcup_{vd\in\{Loct\}}[\![vd,(s,h_0,f_0,r_0)]\!]^8 \cap \bigcup_{vd\in\{Loct\}}[\![vd,(s,h_1,f_1,r_1)]\!]^8 = [\![Loct,(s,h_0,f_0,r_0)]\!]^8 \cap$$
$$[\![Loct,(s,h_1,f_1,r_1)]\!]^8 = dom(h_0)\cap dom(h_1) = \emptyset = \bigcup_{vd\in\Omega}[\![vd,(s,h,f,r)]\!]^8$$
$$= \bigcup_{vd\in basic\_ast(\{Loct\},\{Loct\})}[\![vd,(s,h,f,r)]\!]^8$$

- $\{Dgt,Loct\}$

$$\bigcup_{vd\in\{Loct\}}[\![vd,(s,h_0,f_0,r_0)]\!]^8\cap \bigcup_{vd\in\{Dgt,Loct\}}[\![vd,(s,h_1,f_1,r_1)]\!]^8 = [\![Loct,(s,h_0,f_0,r_0)]\!]^8\cap$$
$$([\![Dgt,(s,h_1,f_1,r_1)]\!]^8\cup[\![Loct,(s,h_1,f_1,r_1)]\!]^8) = dom(h_0)\cap(Loc\setminus dom(h_1)\cup dom(h_1)) =$$
$$dom(h_0) \subseteq dom(h) = [\![Loct,(s,h,f,r)]\!]^8 = \bigcup_{vd\in\{Loct\}}[\![vd,(s,h,f,r)]\!]^8$$
$$= \bigcup_{vd\in basic\_ast(\{Loct\},\{Dgt,Loct\})}[\![vd,(s,h,f,r)]\!]^8$$

- $\{Loc(A_1,l_1^1,l_1^2)\}$

$$\bigcup_{vd\in\{Loct\}}[\![vd,(s,h_0,f_0,r_0)]\!]^8\cap \bigcup_{vd\in\{Loc(A_1,l_1^1,l_1^2)\}}[\![vd,(s,h_1,f_1,r_1)]\!]^8 = [\![Loct,(s,h_0,f_0,r_0)]\!]^8\cap$$
$$[\![\{Loc(A_1,l_1^1,l_1^2)\},(s,h_1,f_1,r_1)]\!]^8 \subseteq dom(h_0)\cap dom(h_1) = \emptyset = \bigcup_{vd\in\Omega}[\![vd,(s,h,f,r)]\!]^8$$
$$= \bigcup_{vd\in basic\_ast(\{Loct\},\{Loc(A_1,l_1^1,l_1^2)\})}[\![vd,(s,h,f,r)]\!]^8$$

- $\top$

$$\bigcup_{vd\in\{Loct\}}[\![vd,(s,h_0,f_0,r_0)]\!]^8 \cap \bigcup_{vd\in\top}[\![vd,(s,h_1,f_1,r_1)]\!]^8 = [\![Loct,(s,h_0,f_0,r_0)]\!]^8 =$$
$$dom(h_0) \subseteq dom(h) = [\![Loct,(s,h,f,r)]\!]^8 = \bigcup_{vd\in\{Loct\}}[\![vd,(s,h,f,r)]\!]^8$$
$$= \bigcup_{vd\in basic\_ast(\{Loct\},\top)}[\![vd,(s,h,f,r)]\!]^8$$

- $\{Dgt,Loct\}$ with

- $\{Dgt,Loct\}$

$$\bigcup_{vd\in\{Dgt,Loct\}}[\![vd,(s,h_0,f_0,r_0)]\!]^8\cap \bigcup_{vd\in\{Dgt,Loct\}}[\![vd,(s,h_1,f_1,r_1)]\!]^8 = ([\![Dgt,(s,h_0,f_0,r_0)]\!]^8\cup$$
$$[\![Loct,(s,h_0,f_0,r_0)]\!]^8) \cap ([\![Dgt,(s,h_1,f_1,r_1)]\!]^8 \cup [\![Loct,(s,h_1,f_1,r_1)]\!]^8) = (Loc \setminus$$
$$dom(h_0) \cup dom(h_0)) \cap (Loc \setminus dom(h_1) \cup dom(h_1)) = Loc = Loc \setminus dom(h) \cup$$

$$dom(h) = \llbracket Dgt, (s, h, f, r) \rrbracket^8 \cup \llbracket Loct, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in \{Dgt, Loct\}} \llbracket vd, (s, h, f, r) \rrbracket^8 =$$

$$\bigcup_{vd \in basic\_ast(\{Dgt, Loct\}, \{Dgt, Loct\})} \llbracket vd, (s, h, f, r) \rrbracket^8$$

$$- \ \{Loc(A_1, l_1^1, l_1^2)\}$$

$$\bigcup_{vd \in \{Dgt, Loct\}} \llbracket vd, (s, h_0, f_0, r_0) \rrbracket^8$$

$$\cap \bigcup_{vd \in \{Loc(A, vd1, vd2)\}} \llbracket vd, (s, h_1, f_1, r_1) \rrbracket^8$$

$$= \ (\llbracket Dgt, (s, h_0, f_0, r_0) \rrbracket^8 \cup \llbracket Loct, (s, h_0, f_0, r_0) \rrbracket^8)$$

$$\cap \llbracket Loc(A, vd1, vd2), (s, h_1, f_1, r_1) \rrbracket^8$$

$$= \ (Loc \setminus dom(h_0) \cup dom(h_0)$$

$$\cap \left\{ l \in dom(h_1) \ \middle| \ \begin{bmatrix} \bullet & \Pi_1(h_1(l)) \in \llbracket vd1, (h_1, f_1, r_1) \rrbracket^8 \\ \bullet & \Pi_2(h_1(l)) \in \llbracket vd2, (h_1, f_1, r_1) \rrbracket^8 \\ \bullet & *1 \in A \wedge \Pi_1(h_1(l)) \in Loc \Rightarrow \Pi_1(h_1(l)) \in r_1(l) \\ \bullet & *2 \in A \wedge \Pi_2(h_1(l)) \in Loc \Rightarrow \Pi_2(h_1(l)) \in r_1(l) \end{bmatrix} \right\}$$

since $h = h_0 \cdot h_1$, we have:

$$= \ \left\{ l \in dom(h_1) \ \middle| \ \begin{bmatrix} \bullet & \Pi_1(h(l)) \in \llbracket vd1, (h_1, f_1, r_1) \rrbracket^8 \\ \bullet & \Pi_2(h(l)) \in \llbracket vd2, (h_1, f_1, r_1) \rrbracket^8 \\ \bullet & *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r_1(l) \\ \bullet & *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r_1(l) \end{bmatrix} \right\}$$

from Lem. 3.22, we have:

$$\subseteq \ \left\{ l \in dom(h_1) \ \middle| \ \begin{bmatrix} \bullet & \Pi_1(h(l)) \in \llbracket vd1, (h, f, r) \rrbracket^8 \\ \bullet & \Pi_2(h(l)) \in \llbracket vd2, (h, f, r) \rrbracket^8 \\ \bullet & *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r_1(l) \\ \bullet & *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r_1(l) \end{bmatrix} \right\}$$

$$\subseteq \ \left\{ l \in dom(h) \ \middle| \ \begin{bmatrix} \bullet & \Pi_1(h(l)) \in \llbracket vd1, (h, f, r) \rrbracket^8 \\ \bullet & \Pi_2(h(l)) \in \llbracket vd2, (h, f, r) \rrbracket^8 \\ \bullet & *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r(l) \\ \bullet & *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r(l) \end{bmatrix} \right\}$$

$$= \ \llbracket Loc(A, vd1, vd2), (s, h, f, r) \rrbracket^8$$

$$= \ \bigcup_{vd \in \{Loc(A, vd1, vd2)\}} \llbracket vd, (s, h, f, r) \rrbracket^8$$

$$= \ \bigcup_{vd \in basic\_ast(\{Dgt, Loct\}, \{Loc(A, vd1, vd2)\})} \llbracket vd, (s, h, f, r) \rrbracket^8$$

$$- \ \top$$

$$\bigcup_{vd\in\{Dgt,Loct\}} [\![vd,(s,h_0,f_0,r_0)]\!]^8 \cap \bigcup_{vd\in\top} [\![vd,(s,h_1,f_1,r_1)]\!]^8 = [\![Dgt,(s,h_0,f_0,r_0)]\!]^8 \cup$$

$$[\![Loct,(s,h_0,f_0,r_0)]\!]^8 = Loc \setminus dom(h_0) \cup dom(h_0) = (Loc \setminus dom(h) \cup dom(h)) =$$

$$([\![Dgt,(s,h,f,r)]\!]^8 \cup [\![Loct,(s,h,f,r)]\!]^8) = \bigcup_{vd\in\{Dgt,Loct\}} [\![vd,(s,h,f,r)]\!]^8$$

$$= \bigcup_{vd\in basic\_ast(\{Dgt,Loct\},\top)} [\![vd,(s,h,f,r)]\!]^8$$

- $\{Loc(A_0,l_0^1,l_0^2)\}$ with

  - $\{Loc(A_1,l_1^1,l_1^2)\}$

    $$\bigcup_{vd\in\{Loc(A_0,l_0^1,l_0^2)\}} [\![vd,(s,h_0,f_0,r_0)]\!]^8 \cap \bigcup_{vd\in\{Loc(A_1,l_1^1,l_1^2)\}} [\![vd,(s,h_1,f_1,r_1)]\!]^8$$

    $$= [\![Loc(A_0,l_0^1,l_0^2),(s,h_0,f_0,r_0)]\!]^8 \cap [\![Loc(A_1,l_1^1,l_1^2),(s,h_1,f_1,r_1)]\!]^8 \subseteq dom(h_0) \cap$$

    $$dom(h_1) = \emptyset = \bigcup_{vd\in\Omega} [\![vd,(s,h,f,r)]\!]^8$$

    $$= \bigcup_{vd\in basic\_ast(\{Loc(A_0,l_0^1,l_0^2)\},\{Loc(A_1,l_1^1,l_1^2)\})} [\![vd,(s,h,f,r)]\!]^8$$

  - $\top$

$$\bigcup_{vd \in \{Loc(A_0, l_0^1, l_0^2)\}} \llbracket vd, (s, h_0, f_0, r_0) \rrbracket^8 \cap \bigcup_{vd \in \top} \llbracket vd, (s, h_1, f_1, r_1) \rrbracket^8$$

$$= \llbracket Loc(A_0, l_0^1, l_0^2) (s, h_0, f_0, r_0) \rrbracket^8$$

since $h = h_0 \cdot h_1$, we have:

$$= \left\{ l \in dom(h_1) \middle| \begin{array}{l} \bullet \quad \Pi_1(h(l)) \in \llbracket l_0^1, (h_1, f_1, r_1) \rrbracket^8 \\ \bullet \quad \Pi_2(h(l)) \in \llbracket l_0^2, (h_1, f_1, r_1) \rrbracket^8 \\ \bullet \quad *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r_1(l) \\ \bullet \quad *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r_1(l) \end{array} \right\}$$

from Lem. 3.22, we get:

$$\subseteq \left\{ l \in dom(h_1) \middle| \begin{array}{l} \bullet \quad \Pi_1(h(l)) \in \llbracket l_0^1, (h, f, r) \rrbracket^8 \\ \bullet \quad \Pi_2(h(l)) \in \llbracket l_0^2, (h, f, r) \rrbracket^8 \\ \bullet \quad *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r_1(l) \\ \bullet \quad *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r_1(l) \end{array} \right\}$$

$$\subseteq \left\{ l \in dom(h) \middle| \begin{array}{l} \bullet \quad \Pi_1(h(l)) \in \llbracket l_0^1, (h, f, r) \rrbracket^8 \\ \bullet \quad \Pi_2(h(l)) \in \llbracket l_0^2, (h, f, r) \rrbracket^8 \\ \bullet \quad *1 \in A \wedge \Pi_1(h(l)) \in Loc \Rightarrow \Pi_1(h(l)) \in r(l) \\ \bullet \quad *2 \in A \wedge \Pi_2(h(l)) \in Loc \Rightarrow \Pi_2(h(l)) \in r(l) \end{array} \right\}$$

$$= \llbracket Loc(A_0, l_0^1, l_0^2), (s, h, f, r) \rrbracket^8$$

$$= \bigcup_{vd \in \{Loc(A_0, l_0^1, l_0^2)\}} \llbracket vd, (s, h, f, r) \rrbracket^8$$

$$= \bigcup_{vd \in basic\_ast(\{Loc(A_0, l_0^1, l_0^2)\}, \top)} \llbracket vd, (s, h, f, r) \rrbracket^8$$

- $\top$ with

  - $\top$

$$\bigcup_{vd \in \top} \llbracket vd, (s, h_0, f_0, r_0) \rrbracket^8 \cap \bigcup_{vd \in \top} \llbracket vd, (s, h_1, f_1, r_1) \rrbracket^8 = MFR = \bigcup_{vd \in \top} \llbracket vd, (s, h, f, r) \rrbracket^8 =$$
$$\bigcup_{vd \in basic\_ast(\top, \top)} \llbracket vd, (s, h, f, r) \rrbracket^8$$

- $\emptyset$ or Ω

$$\bigcup_{vd \in \top} \llbracket vd, (s, h_0, f_0, r_0) \rrbracket^8 \cap A = \bigcup_{vd \in \Omega} \llbracket vd, (s, h_0, f_0, r_0) \rrbracket^8 \cap A = \emptyset = \bigcup_{vd \in \Omega} \llbracket vd, (s, h, f, r) \rrbracket^8 =$$
$$\bigcup_{vd \in basic\_ast(\emptyset, \_)} \llbracket vd, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in basic\_ast(\_, \emptyset)} \llbracket vd, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in basic\_ast(\Omega, \_)} \llbracket vd, (s, h, f, r) \rrbracket^8$$
$$= \bigcup_{vd \in basic\_ast(\_, \Omega)} \llbracket vd, (s, h, f, r) \rrbracket^8$$

$\square$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

3.8.9 Extra ast proofs

Proposition 3.48. $[\![sn_0]\!]^2 \cap [\![sn_1]\!]^2 \subseteq [\![sn_{01}]\!]^2$ and $[\![sn_0^\infty]\!]^{2'} \cap [\![sn_1^\infty]\!]^{2'} \subseteq [\![sn_{01}^\infty]\!]^{2'}$

Prop. 3.48. Direct from definition of $f_0 \dot{\cup} f_1$ and $[\![\cdot]\!]^2$ and $[\![\cdot]\!]^{2'}$ □ □

Proposition 3.49.

$$
If \quad
\begin{aligned}
&\bullet \quad s, h_0, f_0, r_0 \in [\![hu_0]\!]^1 \\
&\bullet \quad s, h_1, f_1, r_1 \in [\![hu_1]\!]^1 \quad then \; s, h_0 \cdot h_1, f_0 \dot{\cup} f_1, r_0 \dot{\cup} r_1 \in [\![hu_0 \cup hu_1]\!]^1 \\
&\bullet \quad h_0 \sharp h_1
\end{aligned}
$$

Prop. 3.49.

$$
\Leftrightarrow
\left[
\begin{aligned}
&\bullet \quad s, h_0, f_0, r_0 \in [\![hu_0]\!]^1 \\
&\bullet \quad s, h_1, f_1, r_1 \in [\![hu_1]\!]^1 \\
&\bullet \quad h_0 \sharp h_1 \\
&\bullet \quad \forall \alpha \in hu_0. \; f_0(\alpha) \cap dom(h_0) \neq \emptyset \\
&\bullet \quad \forall \alpha \in hu_1. \; f_1(\alpha) \cap dom(h_1) \neq \emptyset
\end{aligned}
\right.
$$

$$
\Leftrightarrow
\left[
\begin{aligned}
&\bullet \quad h_0 \sharp h_1 \\
&\bullet \quad \forall \alpha \in hu_0. \; f(\alpha) \cap dom(h_0) \neq \emptyset \\
&\bullet \quad \forall \alpha \in hu_1. \; f(\alpha) \cap dom(h_1) \neq \emptyset
\end{aligned}
\right.
$$

$$
\Rightarrow
\left[
\begin{aligned}
&\bullet \quad h_0 \sharp h_1 \\
&\bullet \quad \forall \alpha \in hu_0 \cup hu_1. \; f(\alpha) \cap (dom(h_0) \cup dom(h_1)) \neq \emptyset \\
&\bullet \quad h_0 \sharp h_1
\end{aligned}
\right.
$$

$$
\Leftrightarrow \quad s, h_0 \cdot h_1, f_0 \dot{\cup} f_1, r_0 \dot{\cup} r_1 \in [\![hu_0 \cup hu_1]\!]^1
$$

□ □

Proposition 3.50.

$$
If \quad
\begin{aligned}
&\bullet \quad s, h_0, f_0, r_0 \in [\![ho_0]\!]^{1'} \\
&\bullet \quad s, h_1, f_1, r_1 \in [\![ho_1]\!]^{1'} \quad then \; s, h_0 \cdot h_1, f_0 \dot{\cup} f_1, r_0 \dot{\cup} r_1 \in [\![ho_0 \cup ho_1]\!]^{1'} \\
&\bullet \quad h_0 \sharp h_1
\end{aligned}
$$

Prop. 3.50.

$$\Leftrightarrow \left[\begin{array}{l} \bullet \quad s, h_0, f_0, r_0 \in [\![ho_0]\!]^{1'} \\ \bullet \quad s, h_1, f_1, r_1 \in [\![ho_1]\!]^{1'} \\ \bullet \quad h_0 \sharp h_1 \\ \bullet \quad dom(h_0) \subseteq \bigcup_{\alpha \in ho_0} f_0(\alpha) \\ \bullet \quad dom(h_1) \subseteq \bigcup_{\alpha \in ho_1} f_1(\alpha) \\ \bullet \quad h_0 \sharp h_1 \end{array}\right.$$

$$\Leftrightarrow \left[\begin{array}{l} \bullet \quad dom(h_0) \subseteq \bigcup_{\alpha \in ho_0} f(\alpha) \\ \bullet \quad dom(h_1) \subseteq \bigcup_{\alpha \in ho_1} f(\alpha) \\ \bullet \quad h_0 \sharp h_1 \end{array}\right.$$

$$\Rightarrow \left[\begin{array}{l} \bullet \quad dom(h) \subseteq \bigcup_{\alpha \in ho_0 \cup ho_1} f(\alpha) \\ \bullet \quad h_0 \sharp h_1 \end{array}\right.$$

$$\Leftrightarrow \quad s, h_0 \cdot h_1, f_0 \dot{\cup} f_1, r_0 \dot{\cup} r_1 \in [\![ho_0 \cup ho_1]\!]^{1'}$$

□   ◻

Proposition 3.51. $s, h_0, f_0, r_0 \in sem * \wedge s, h_1, f_1, r_1 \in sem* \Rightarrow s, h, f, r_0 \dot{\cup} r_1 \in sem*$

Prop. 3.51. We have by hypothesis:

- $\forall l_0 \in dom(r_0).$ $\left[\begin{array}{l} \bullet \quad \boldsymbol{dom(r_0) \cup codom(r_0) \subseteq dom(h_0)} \\ \bullet \quad l_0 \notin r_0(l_0) \\ \bullet \quad \forall l_0' \in r_0(l_0) \cap dom(r_0).r_0(l_0') \subseteq r_0(l_0) \end{array}\right.$

- $\forall l_1 \in dom(r_1).$ $\left[\begin{array}{l} \bullet \quad \boldsymbol{dom(r_1) \cup codom(r_1) \subseteq dom(h_1)} \\ \bullet \quad l_1 \notin r_1(l_1) \\ \bullet \quad \forall l_1' \in r_1(l_1) \cap dom(r_1).r_1(l_1') \subseteq r_1(l_1) \end{array}\right.$

So $\forall l \in dom(r). (l \in dom(h_0) \wedge \forall l' \in r(l).ll' \in dom(h_0)) XOR(l \in dom(h_1) \wedge \forall l' \in r(l).ll' \in dom(h_1)).$

We prove what we want only in the case where $l \in dom(r_0)$.

- since $dom(r_0) \cup codom(r_0) \subseteq dom(h_0)$ and $dom(r_1) \cup codom(r_1) \subseteq dom(h_1)$ we have
  $dom(r) \cup codom(r) \subseteq dom(h)$

- $l \in dom(h_0)$ thus $l \notin codom(r_1)$ and by hypothesis $l \notin r_0(l)$ thus $l \notin r(l)$

- $\forall l' \in r_0(l) \cap dom(r_0). r_0(l') \subseteq r_0(l)$ thus $\forall l' \in r(l) \cap dom(r). r(l') \subseteq r(l)$

□   ◻

180

3.8.10 Basic equal proofs

Proposition 3.30. *If we define* $\bigcup_{vd \in \Omega} \llbracket vd, (s, h, f, r) \rrbracket^8 \triangleq \emptyset$ *and* $\bigcup_{vd \in \top} \llbracket vd, (s, h, f, r) \rrbracket^8 \triangleq MFR$

If

- $\forall (vd_0, vd_1) \in eq. \; vd_0 \in TVar \Rightarrow f(vd_0) \in \llbracket vd_1, (s, h, f, r) \rrbracket^8$

- $\forall (vd_0, vd_1) \in eq. \; vd_1 \in TVar \Rightarrow f(vd_1) \in \llbracket vd_0, (s, h, f, r) \rrbracket^8$

- *if* vd_0 *or* $vd_1 = \{Loc(A, l^1, l^2)\}$ *then* $l^1, l^2 \notin sn$

- $basic\_equal(vd_0, vd_1, eq) = (vd_{01}, eq')$

then

- $\bigcup_{vd \in vd_0} \llbracket vd, (s, h, f, r) \rrbracket^8 \cap \bigcup_{vd \in vd_1} \llbracket vd, (s, h, f, r) \rrbracket^8 \subseteq \bigcup_{vd \in vd_{01}} \llbracket vd, (s, h, f, r) \rrbracket^8$

- $\forall (vd_0, vd_1) \in eq'. \; vd_0 \in TVar \Rightarrow f(vd_0) \in \llbracket vd_1, (s, h, f, r) \rrbracket^8$

- $\forall (vd_0, vd_1) \in eq'. \; vd_1 \in TVar \Rightarrow f(vd_1) \in \llbracket vd_0, (s, h, f, r) \rrbracket^8$

Prop. 3.30. If we write the proof for $basic\_equal(vd_0, vd_1)$ then we will not write the proof for $basic\_equal(vd_1, vd_0)$, also if we prove $basic\_equal(vd_0, \{A\})$ then we do not prove $basic\_ast(vd_0, \{B\})$ except for $vd_0 = \{A\}$.

- First, we prove the case where $vd_0 = vd_1$:

  then $vd_{01} = vd_0$ and $eq' = eq$ and the proposition is obvious.

- Case with Ω

  $\bigcup_{vd \in \Omega} \llbracket vd, (s, h, f, r) \rrbracket^8 \cap \bigcup_{vd \in vd_1} \llbracket vd, (s, h, f, r) \rrbracket^8 = \emptyset \cap \bigcup_{vd \in vd_1} \llbracket vd, (s, h, f, r) \rrbracket^8 = \emptyset$
  $= \bigcup_{vd \in \Omega} \llbracket vd, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in basic\_equal(\Omega, vd_1)} \llbracket vd, (s, h, f, r) \rrbracket^8$

- Case with $\top$

  $\bigcup_{vd \in \top} \llbracket vd, (s, h, f, r) \rrbracket^8 \cap \bigcup_{vd \in vd_1} \llbracket vd, (s, h, f, r) \rrbracket^8 = MFR \cap \bigcup_{vd \in vd_1} \llbracket vd, (s, h, f, r) \rrbracket^8$
  $= \bigcup_{vd \in vd_1} \llbracket vd, (s, h, f, r) \rrbracket^8 = \bigcup_{vd \in basic\_equal(\top, vd_1)} \llbracket vd, (s, h, f, r) \rrbracket^8$

181

- Case with $\emptyset$

$$\bigcup_{vd \in \emptyset} [\![vd, (s, h, f, r)]\!]^8 \cap \bigcup_{vd \in vd_1} [\![vd, (s, h, f, r)]\!]^8 = \emptyset \cap \bigcup_{vd \in vd_1} [\![vd, (s, h, f, r)]\!]^8 = \emptyset$$
$$= \bigcup_{vd \in \Omega} [\![vd, (s, h, f, r)]\!]^8 = \bigcup_{vd \in basic\_equal(\emptyset, vd_1)} [\![vd, (s, h, f, r)]\!]^8$$

- when $vd_0 \neq vd_1$ and they are not $\Omega, \emptyset, \top$

  - Cases $\{A\}$ with S:

    $$\bigcup_{vd \in \{A\}} [\![vd, (s, h, f, r)]\!]^8 \cap \bigcup_{vd \in S} [\![vd, (s, h, f, r)]\!]^8 = [\![A, (s, h, f, r)]\!]^8 \cap \bigcup_{vd \in S} [\![vd, (s, h, f, r)]\!]^8 =$$
    $$\emptyset = \bigcup_{vd \in \Omega} [\![vd, (s, h, f, r)]\!]^8 = \bigcup_{vd \in basic\_equal(\{A\}, S)} [\![vd, (s, h, f, r)]\!]^8$$

  - Cases $\{Dgt\}$ with

    * $\{Loct\}$

      $$\bigcup_{vd \in \{Dgt\}} [\![vd, (s, h, f, r)]\!]^8 \cap \bigcup_{vd \in \{Loct\}} [\![vd, (s, h, f, r)]\!]^8$$
      $$= [\![Dgt, (s, h, f, r)]\!]^8 \cap [\![Loct, (s, h, f, r)]\!]^8 = (Loc \setminus dom(h)) \cap dom(h) = \emptyset =$$
      $$\bigcup_{vd \in \Omega} [\![vd, (s, h, f, r)]\!]^8 = \bigcup_{vd \in basic\_equal(\{Dgt\}, \{Loct\})} [\![vd, (s, h, f, r)]\!]^8$$

    * $\{Dgt, Loct\}$

      $$\bigcup_{vd \in \{Dgt\}} [\![vd, (s, h, f, r)]\!]^8 \cap \bigcup_{vd \in \{Dgt, Loct\}} [\![vd, (s, h, f, r)]\!]^8 = [\![Dgt, (s, h, f, r)]\!]^8 \cap$$
      $$([\![Dgt, (s, h, f, r)]\!]^8 \cup [\![Loct, (s, h, f, r)]\!]^8) = (Loc \setminus dom(h)) \cap ((Loc \setminus dom(h)) \cup$$
      $$dom(h)) = (Loc \setminus dom(h)) = \bigcup_{vd \in \{Dgt\}} [\![vd, (s, h, f, r)]\!]^8$$
      $$= \bigcup_{vd \in basic\_equal(\{Dgt\}, \{Dgt, Loct\})} [\![vd, (s, h, f, r)]\!]^8$$

    * $\{Loc(A_1, l_1^1, l_1^2)\}$

      $$\bigcup_{vd \in \{Dgt\}} [\![vd, (s, h, f, r)]\!]^8 \cap \bigcup_{vd \in \{Loc(A_1, l_1^1, l_1^2)\}} [\![vd, (s, h, f, r)]\!]^8 = [\![Dgt, (s, h, f, r)]\!]^8 \cap$$
      $$[\![Loc(A_1, l_1^1, l_1^2), (s, h, f, r)]\!]^8 \subseteq (Loc \setminus dom(h)) \cap dom(h) = \emptyset = \bigcup_{vd \in \Omega} [\![vd, (s, h, f, r)]\!]^8 =$$
      $$\bigcup_{vd \in basic\_equal(\{Dgt\}, \{Loc(A_1, l_1^1, l_1^2)\})} [\![vd, (s, h, f, r)]\!]^8$$

  - Cases $\{Loct\}$ with

    * $\{Dgt, Loct\}$

      $$\bigcup_{vd \in \{Loct\}} [\![vd, (s, h, f, r)]\!]^8 \cap \bigcup_{vd \in \{Dgt, Loct\}} [\![vd, (s, h, f, r)]\!]^8 = [\![Loct, (s, h, f, r)]\!]^8 \cap$$

182

$$([\![Dgt, (s, h, f, r)]\!]^8 \cup [\![Loct, (s, h, f, r)]\!]^8) = dom(h) \cap ((Loc \backslash dom(h)) \cup dom(h)) =$$

$$dom(h) = \bigcup_{vd \in \{Loct\}} [\![vd, (s, h, f, r)]\!]^8 = \bigcup_{vd \in basic\_equal(\{Loct\}, \{Dgt, Loct\})} [\![vd, (s, h, f, r)]\!]^8$$

* $\{Loc(A_1, l_1^1, l_1^2)\}$

$$\bigcup_{vd \in \{Loct\}} [\![vd, (s, h, f, r)]\!]^8 \cap \bigcup_{vd \in \{Loc(A_1, l_1^1, l_1^2)\}} [\![vd, (s, h, f, r)]\!]^8$$

$$= [\![Loct, (s, h, f, r)]\!]^8 \cap [\![Loc(A_1, l_1^1, l_1^2), (s, h, f, r)]\!]^8$$

$$= dom(h) \cap [\![Loc(A_1, l_1^1, l_1^2), (s, h, f, r)]\!]^8 = [\![Loc(A_1, l_1^1, l_1^2), (s, h, f, r)]\!]^8$$

$$= \bigcup_{vd \in \{Loc(A_1, l_1^1, l_1^2)\}} [\![vd, (s, h, f, r)]\!]^8$$

$$= \bigcup_{vd \in basic\_equal(\{Loct\}, \{Loc(A_1, l_1^1, l_1^2)\})} [\![vd, (s, h, f, r)]\!]^8$$

– Cases $\{Dgt, Loct\}$ with

  * $\{Loc(A_1, l_1^1, l_1^2)\}$

$$\bigcup_{vd \in \{Dgt, Loct\}} [\![vd, (s, h, f, r)]\!]^8 \cap \bigcup_{vd \in \{Loc(A_1, l_1^1, l_1^2)\}} [\![vd, (s, h, f, r)]\!]^8 = ([\![Dgt, (s, h, f, r)]\!]^8 \cup$$

$$[\![Loct, (s, h, f, r)]\!]^8) \cap [\![Loc(A_1, l_1^1, l_1^2), (s, h, f, r)]\!]^8 = Loc \cap [\![Loc(A_1, l_1^1, l_1^2), (s, h, f, r)]\!]^8 =$$

$$[\![Loc(A_1, l_1^1, l_1^2), (s, h, f, r)]\!]^8 = \bigcup_{vd \in \{Loc(A_1, l_1^1, l_1^2)\}} [\![vd, (s, h, f, r)]\!]^8$$

$$= \bigcup_{vd \in basic\_equal(\{Dgt, Loct\}, \{Loc(A_1, l_1^1, l_1^2)\})} [\![vd, (s, h, f, r)]\!]^8$$

– Case [*] when $vd_0 \neq vd_1$

$$\bigcup_{vd\in\{Loc(A_1,l_1^1,l_1^2)\}} [\![vd,(s,h,f,r)]\!]^8 \cap \bigcup_{vd\in\{Loc(A_2,l_2^1,l_2^2)\}} [\![vd,(s,h,f,r)]\!]^8$$

$$= [\![Loc(A_1,l_1^1,l_1^2),(s,h,f,r)]\!]^8 \cap [\![Loc(A_2,l_2^1,l_2^2),(s,h,f,r)]\!]^8$$

$$= \left\{ l\in dom(h) \;\middle|\; \begin{array}{l} \bullet \quad \Pi_1(h(l))\in[\![l_1^1,(h,f,r)]\!]^8 \\ \bullet \quad \Pi_2(h(l))\in[\![l_1^2,(h,f,r)]\!]^8 \\ \bullet \quad *1\in A_1 \wedge \Pi_1(h(l))\in Loc \Rightarrow \Pi_1(h(l))\in r(l) \\ \bullet \quad *2\in A_1 \wedge \Pi_2(h(l))\in Loc \Rightarrow \Pi_2(h(l))\in r(l) \end{array} \right\}$$

$$\cap \left\{ l\in dom(h) \;\middle|\; \begin{array}{l} \bullet \quad \Pi_1(h(l))\in[\![l_2^1,(h,f,r)]\!]^8 \\ \bullet \quad \Pi_2(h(l))\in[\![l_2^2,(h,f,r)]\!]^8 \\ \bullet \quad *1\in A_2 \wedge \Pi_1(h(l))\in Loc \Rightarrow \Pi_1(h(l))\in r(l) \\ \bullet \quad *2\in A_2 \wedge \Pi_2(h(l))\in Loc \Rightarrow \Pi_2(h(l))\in r(l) \end{array} \right\}$$

$$= \left\{ l\in dom(h) \;\middle|\; \begin{array}{l} \bullet \quad \Pi_1(h(l))\in[\![l_1^1,(h,f,r)]\!]^8 \\ \bullet \quad \Pi_1(h(l))\in[\![l_2^1,(h,f,r)]\!]^8 \\ \bullet \quad \Pi_2(h(l))\in[\![l_1^2,(h,f,r)]\!]^8 \\ \bullet \quad \Pi_2(h(l))\in[\![l_2^2,(h,f,r)]\!]^8 \\ \bullet \quad *1\in A_1 \wedge \Pi_1(h(l))\in Loc \Rightarrow \Pi_1(h(l))\in r(l) \\ \bullet \quad *2\in A_1 \wedge \Pi_2(h(l))\in Loc \Rightarrow \Pi_2(h(l))\in r(l) \\ \bullet \quad *1\in A_2 \wedge \Pi_1(h(l))\in Loc \Rightarrow \Pi_1(h(l))\in r(l) \\ \bullet \quad *2\in A_2 \wedge \Pi_2(h(l))\in Loc \Rightarrow \Pi_2(h(l))\in r(l) \end{array} \right\}$$

$$= \left\{ l\in dom(h) \;\middle|\; \begin{array}{l} \bullet \quad \Pi_1(h(l))\in[\![l_1^1,(h,f,r)]\!]^8 \\ \bullet \quad \Pi_1(h(l))\in[\![l_2^1,(h,f,r)]\!]^8 \\ \bullet \quad \Pi_2(h(l))\in[\![l_1^2,(h,f,r)]\!]^8 \\ \bullet \quad \Pi_2(h(l))\in[\![l_2^2,(h,f,r)]\!]^8 \\ \bullet \quad *1\in(A_1\cup A_2) \wedge \Pi_1(h(l))\in Loc \Rightarrow \Pi_1(h(l))\in r(l) \\ \bullet \quad *2\in(A_1\cup A_2) \wedge \Pi_2(h(l))\in Loc \Rightarrow \Pi_2(h(l))\in r(l) \end{array} \right\}$$

————————— - to cross this line see below ———————————-

$$\subseteq \left\{ l\in dom(h) \;\middle|\; \begin{array}{l} \bullet \quad \Pi_1(h(l))\in[\![l_{12}^1,(h,f,r)]\!]^8 \\ \bullet \quad \Pi_2(h(l))\in[\![l_{12}^2,(h,f,r)]\!]^8 \\ \bullet \quad *1\in(A_1\cup A_2) \wedge \Pi_1(h(l))\in Loc \Rightarrow \Pi_1(h(l))\in r(l) \\ \bullet \quad *2\in(A_1\cup A_2) \wedge \Pi_2(h(l))\in Loc \Rightarrow \Pi_2(h(l))\in r(l) \end{array} \right\}$$

$$= \bigcup_{vd\in\{Loc(A_1\cup A_2,l_{12}^1,l_{12}^2)\}} [\![vd,(s,h,f,r)]\!]^8$$

We prove by cases, as the definition is done, we need to proof the inequality and the "eq" part.

First, if $eq' = eq$, then we have the "eq" part.

Then: (we write only the proof for the ^{1} part, the ^{2} part being similar)

* if $l_1^1 = l_2^1$ then we have $l_{12}^1 = l_1^1$ and we have $\Pi_1(h(l)) \in [\![l_1^1, (h, f, r)]\!]^8 \wedge$

  $\Pi_1(h(l)) \in [\![l_2^1, (h, f, r)]\!]^8 \Leftrightarrow \Pi_1(h(l)) \in [\![l_{12}^1, (h, f, r)]\!]^8$

* elsif $l_1^1 \in TVar$ then $l_{12}^1 = l_1^1$ and we have $\Pi_1(h(l)) \in [\![l_1^1, (h, f, r)]\!]^8 \wedge$

  $\Pi_1(h(l)) \in [\![l_2^1, (h, f, r)]\!]^8 \Rightarrow \Pi_1(h(l)) \in [\![l_1^1, (h, f, r)]\!]^8 \Leftrightarrow \Pi_1(h(l)) \in [\![l_{12}^1, (h, f, r)]\!]^8$

  since $l_1^1 \in TVar \setminus sn$, we have $\Pi_1(h(l)) = f(l_1^1)$ thus $f(l_1^1) \in [\![l_2^1, (h, f, r)]\!]^8$

  and we can update eq' as defined.

* elsif $l_2^1 \in TVar$ then $l_{12}^1 = l_2^1$ and we have $\Pi_1(h(l)) \in [\![l_1^1, (h, f, r)]\!]^8 \wedge$

  $\Pi_1(h(l)) \in [\![l_2^1, (h, f, r)]\!]^8 \Rightarrow \Pi_1(h(l)) \in [\![l_2^1, (h, f, r)]\!]^8 \Leftrightarrow \Pi_1(h(l)) \in [\![l_{12}^1, (h, f, r)]\!]^8$

  since $l_2^1 \in TVar \setminus sn$, we have $\Pi_1(h(l)) = f(l_2^1)$ thus $f(l_2^1) \in [\![l_1^1, (h, f, r)]\!]^8$

  and we can update eq' as defined.

* else

  this is in fact the case where $l_1^1 \neq l_2^1$ and $l_1^1, l_2^1 \in \{Numt, Truet, Falset, Oodt, Nilt\}$

  then $\Pi_1(h(l)) \in [\![l_1^1, (h, f, r)]\!]^8 \wedge \Pi_1(h(l)) \in [\![l_2^1, (h, f, r)]\!]^8 \Leftrightarrow false$

  thus the set before the line is equal to $\emptyset$ which is equal to $\bigcup_{vd \in \Omega} [\![vd, (s, h, f, r)]\!]^8$

  thus the result of $basic\_equal$ is Ω

$\square$ $\blacksquare$

3.8.11 Reach functions proofs

Proposition 3.24. $\forall G \in \mathcal{P}(TVar \times PVD^+)$. $\forall s, h, f, r$. $(\forall (\alpha, S) \in G.\ s, h, f, r \in [\![\alpha, S]\!]^5) \Rightarrow$

$\forall \alpha \in TVar.\ \forall (\alpha', S) = reach(\alpha).$

$$\left(f(\alpha) \subseteq f(\alpha') \wedge \left((\alpha', S) \in G \vee \left(\exists \beta_1, \beta_2. \begin{array}{l} S = (\beta_1, \beta_2) \wedge \\ (\alpha', \{\beta_1\}) \in G \wedge \\ (\beta_2, \{\alpha'\}) \in G \wedge \\ f(\alpha') = f(\beta_1) = f(\beta_2) \end{array} \right) \right) \right)$$

$\wedge (\alpha \notin sn_{01} \Rightarrow \alpha' \notin sn_{01})$

Prop.3.24. **Correctness**

The property $\wedge (\alpha \notin sn_{01} \Rightarrow \alpha' \notin sn_{01})$ is obvious.

- if $\alpha \notin dom(G)$ then $\alpha' = \alpha$ and it's ok

- if $(\alpha, \{vd\}) \in dom(G)$ and $vd \notin TVar$ then $\alpha' = \alpha$ and it's ok

- if $\alpha \in dom(G)$ then since $\forall \beta \in \{\alpha\}.\ f(\alpha) \subseteq f(\beta) \subseteq f(\alpha) \wedge \alpha \in \{\alpha\} \wedge (\beta = \alpha \vee \exists \beta' \in$
  $\{\alpha\}.\ (\beta, \{\beta'\}) \in G)$ we will just prove the proposition for *reachrec* with hypothesis

  if $\forall \beta \in V.\ f(\alpha) \subseteq f(\beta) \subseteq f(\alpha') \wedge \alpha' \in V \wedge (\beta = \alpha' \vee \exists \beta' \in V.\ (\beta, \{\beta'\}) \in G)$ and if
  $(\delta, S') = reachrec(G, V, \alpha').$ then

$$
\left(f(\alpha) \subseteq f(\delta) \wedge \left((\delta, S') \in G \vee \left(\exists \delta_1, \delta_2. \begin{array}{c} S' = (\delta_1, \delta_2) \wedge \\ (\delta, \{\delta_1\}) \in G \wedge \\ (\delta_2, \{\delta\}) \in G \wedge \\ f(\delta) = f(\delta_1) = f(\delta_2) \end{array} \right) \right) \right)
$$

  - if $(\alpha', \{\beta\}) \in G$ and $\beta \in V$ and $(\beta, \{\beta'\}) \in G$,

    then $\delta = \beta$ and $S' = (\beta', \alpha')$ and $\delta_1 = \beta'$ and $\delta_2 = \alpha'$

    * by hypothesis we have $f(\alpha) \subseteq f(\beta)$ since $\beta \in V$ thus $f(\alpha) \subseteq f(\delta)$

    * $(\beta, \{\beta'\} \in G$ thus $(\delta, \{\delta_1\}) \in G$

      which also implies that $f(\delta) \subseteq f(\delta_1)$

    * $(\alpha', \{\beta\}) \in G$ thus $(\delta_2, \{\delta\}) \in G$

      which also implies that $f(\delta_2) \subseteq f(\delta)$

    * by hypothesis we have $f(\beta) \subseteq f(\alpha')$ since $\beta \in V$ thus $f(\delta) \subseteq f(\delta_2)$ thus we
      have $f(\delta) = f(\delta_2)$

    * by hypothesis, since $\beta \in V$, either $\beta = \alpha'$ thus $\beta = \beta'$ and $f(\delta) = f(\delta_1)$,
      either $\beta' \in V$ thus $f(\beta') \subseteq f(\alpha')$ which is also $f(\delta_1) \subseteq f(\delta_2)$ then we have
      $f(\delta_1) = f(\delta)$

  - when $(\alpha', \{\beta\}) \in G$ and $\beta \notin V \cup sn_{01}$ and $\beta \in TVar$

    we have $f(\alpha') \subseteq f(\beta) \subseteq f(\beta)$, $\beta \in V \cup \{\beta\}$, and we had

    $\forall \delta \in V.\ (\delta = \alpha') \vee (\exists \beta' \in V.\ (\delta, \{\beta'\}) \in G)$

    thus we have to check that for α' we have $\exists \beta' \in V \cup \{\beta\}.\ (\alpha', \{\beta'\}) \in G$

186

which is the case for $\beta' = \beta$

and then we respect the hypothesis of the recurrence and then we have if $\forall \delta \in V \cup \{\beta\}. f(\alpha) \subseteq f(\delta) \subseteq f(\beta) \wedge \beta \in V \wedge (\delta = \beta \vee \exists \delta' \in V. (\delta, \{\beta'\}) \in G)$ and if $(\delta, S') = reachrec(G, V, \beta)$. then

$$\left(f(\alpha) \subseteq f(\delta) \wedge \left((\delta, S') \in G \vee \left(\exists \delta_1, \delta_2. \begin{array}{l} S' = (\delta_1, \delta_2) \wedge \\ (\delta, \{\delta_1\}) \in G \wedge \\ (\delta_2, \{\delta\}) \in G \wedge \\ f(\delta) = f(\delta_1) = f(\delta_2) \end{array} \right) \right) \right)$$

thus we have if $\forall \delta \in V. f(\alpha) \subseteq f(\delta) \subseteq f(\alpha') \wedge \alpha' \in V \wedge (\delta = \alpha' \vee \exists \beta' \in V. (\delta, \{\beta'\}) \in G)$ and if $(\delta, S') = reachrec(G, V, \alpha')$. then

$$\left(f(\alpha) \subseteq f(\delta) \wedge \left((\delta, S') \in G \vee \left(\exists \delta_1, \delta_2. \begin{array}{l} S' = (\delta_1, \delta_2) \wedge \\ (\delta, \{\delta_1\}) \in G \wedge \\ (\delta_2, \{\delta\}) \in G \wedge \\ f(\delta) = f(\delta_1) = f(\delta_2) \end{array} \right) \right) \right)$$

– when $(\alpha', S) \in G$

then $\delta = \alpha'$ and $S' = S$

  * by hypo. $f(\alpha) \subseteq f(\alpha')$ thus $f(\alpha) \subseteq f(\delta)$

  * $(\alpha', S) \in G$ thus $(\delta, S') \in G$

Termination This comes from the fact that we suppose that G has a finite domain (or we could only suppose that there are no infinite unlooping path of non-summary variables). $\square$

Proposition 3.26. $\forall g \in \mathcal{P}(Var \times PVD^+). G \in \mathcal{P}(TVar \times PVD^+). \forall s, h, f, r. (\forall (v, S) \in g \uplus G. s, h, f, r \in [\![v, S]\!]^5) \Rightarrow \forall (v, S) \in reach2(g, G, x).$

$$\left(s(x) = s^+ f(v) \wedge \left(S = \top \vee (v, S) \in g \uplus G \vee \left(\exists \beta_1, \beta_2. \begin{array}{l} S = (\beta_1, \beta_2) \wedge \\ (v, \{\beta_1\}) \in G \wedge \\ (\beta_2, \{v\}) \in G \wedge \\ s(x) = f(\delta_1) = f(\delta_2) \end{array} \right) \right) \right)$$

Prop.3.26. • when $(x, vd) \notin g$

then $v = x$ and $S = \top$

then $s^+ f(v) = s(x)$ and it's ok

- when $(x, vd) \in g$ and $\forall \alpha. \, \alpha \notin vd$

  then $v = x$ and $S = vd$

  then $s^+ f(v) = s(x)$ and it's ok

- when $(x, vd) \in g$ and $\exists \alpha. \, \alpha \in vd \cap sn_{01}$

  then $v = x$ and $S = vd$

  then $s^+ f(v) = s(x)$ and it's ok

- when $(x, \{\alpha\}) \in g$ and $\alpha \notin sn_{01}$

  then $(v, S) = reach(\alpha, G)$

  we have by $(x, \{\alpha\}) \in g$ that $s(x) \in f(\alpha)$, but since we use Prop. 3.24,

  we then have $f(\alpha) \subseteq f(v)$, thus $s(x) \in f(v)$

  by $\alpha \notin sn_{01}$ and Prop. 3.24 we have $v \notin sn_{01}$ thus $|f(v)| \leq 1$ and thus $s(x) = f(v)$

  and the rest comes also from Prop. 3.24.

$\square$

3.8.12 Class proofs

Proposition 3.52. $\forall ar \in AR, x \in Var. \, [\![class(x, ar)]\!]' \supseteq class(x, [\![ar]\!]')$

Prop. 3.52. When $ad(x) = \emptyset$, $[\![class(x, ar)]\!]' = \emptyset = class(x, [\![ar]\!]')$.

When $ad(x) \neq \emptyset$

$$
\begin{aligned}
& [\![class(x, (ad, hu, ho, sn, sn^\infty, t, d))]\!]' \\
= \quad & [\![([ad|x \rightarrow \top], hu, ho, sn, sn^\infty, t, d))]\!]' \\
= \quad & \left(\begin{array}{l} \bigcap_{v \neq x} [\![v, ad(v)]\!]^5 \cap \\ [\![hu]\!]^1 \cap [\![ho]\!]^{1'} \cap [\![sn]\!]^2 \cap [\![sn^\infty]\!]^{2'} \cap \\ [\![t]\!]^3 \cap [\![d]\!]^7 \cap sem* \end{array} \right) \\
\supseteq \quad & \left(\begin{array}{l} \bigcap_{v \neq x} [\![v, ad(v)]\!]^5 \cap \\ [\![hu]\!]^1 \cap [\![ho]\!]^{1'} \cap [\![sn]\!]^2 \cap [\![sn^\infty]\!]^{2'} \cap \\ [\![t]\!]^3 \cap [\![d]\!]^7 \cap sem* \cap \\ class\,(x, [\![x, ad(x)]\!]^5) \end{array} \right) \\
= \quad & class \left(x, \begin{array}{l} \bigcap_{v \neq x} [\![v, ad(v)]\!]^5 \cap [\![x, ad(x)]\!]^5 \cap \\ [\![hu]\!]^1 \cap [\![ho]\!]^{1'} \cap [\![sn]\!]^2 \cap [\![sn^\infty]\!]^{2'} \cap \\ [\![t]\!]^3 \cap [\![d]\!]^7 \cap sem* \end{array} \right) \quad \text{by Prop. 3.57 and 3.58} \\
= \quad & class(x, [\![(ad, hu, ho, sn, sn^\infty, t, d)]\!]')
\end{aligned}
$$

$\square$ $\blacksquare$

Proposition 3.53. $\forall ar \in AR.ad \in PVD^+.\forall x \in Var.\ [\![x, ad]\!]^5 \cap [\![class(x, ar)]\!]' = [\![[ar \mid x \rightarrow ad]]\!]'$

Prop. 3.53. Direct from definition. $\square$ $\blacksquare$

Proposition 3.54. $\forall x \in Var, S_1, S_2 \in \mathcal{P}(MFR).\ S_1 \subseteq S_2 \Rightarrow class(x, S_1) \subseteq class(x, S_2)$

Prop. 3.54. Direct from definition. $\square$ $\blacksquare$

Proposition 3.55. $\forall x \in Var, S \in \mathcal{P}(MFR).\ S \subseteq class(x, S)$

Prop. 3.55. Direct from definition. $\square$ $\blacksquare$

Proposition 3.56. $\forall x \in Var, S \in \mathcal{P}(MFR).\ x \notin noclass(S) \Rightarrow S = class(x, S)$

Prop. 3.56. Direct from definition. $\square$ $\blacksquare$

Proposition 3.57. $\forall x \in Var, S_1, S_2 \in \mathcal{P}(MFR).\ x \notin noclass(S_1) \Rightarrow class(x, S_1 \cap S_2) = S_1 \cap class(x, S_2)$

Prop. 3.57.

$$s, h, f, r \in S_1 \cap class(x, S_2)$$

$$iff \quad \exists s'. \left[\begin{array}{l} \bullet \quad s, h, f, r \in S_1 \\ \bullet \quad [s' \mid x \to \mathsf{ood}] = [s \mid x \to \mathsf{ood}] \\ \bullet \quad s', h, f, r \in S_2 \end{array} \right.$$

$$iff \quad \exists s'. \left[\begin{array}{l} \bullet \quad s', h, f, r \in S_1 \\ \bullet \quad [s' \mid x \to \mathsf{ood}] = [s \mid x \to \mathsf{ood}] \\ \bullet \quad s', h, f, r \in S_2 \end{array} \right. \quad \text{by hyp.}$$

$$iff \quad s, h, f, r \in class(x, S_1 \cap S_2)$$

$\square$ $\square$

Proposition 3.58. *We give noclass for all subsemantics of* $\llbracket \cdot \rrbracket'$

- $noclass(sem*) = \emptyset$

- $\forall hu \in \mathcal{P}(TVar).\ noclass(\llbracket hu \rrbracket^1) = \emptyset$

- $\forall ho \in \mathcal{P}(TVar).\ noclass(\llbracket ho \rrbracket^{1'}) = \emptyset$

- $\forall sn \in \mathcal{P}(TVar).\ noclass(\llbracket sn \rrbracket^2) = \emptyset$

- $\forall sn^\infty \in \mathcal{P}(TVar).\ noclass(\llbracket sn^\infty \rrbracket^{2'}) = \emptyset$

- $\forall v \in VAR.\ noclass(\llbracket v, \top \rrbracket^5) = \emptyset$

- $\forall \alpha \in TVar.\ noclass(\llbracket \alpha, \circledast \rrbracket^5) = \emptyset$

- $\forall \alpha \in TVar, S \neq \top, \circledast\ .\ noclass(\llbracket \alpha, S \rrbracket^5) = \emptyset$

- $\forall x \in Var, S \neq \top, \circledast\ .\ noclass(\llbracket x, S \rrbracket^5) \subseteq \{x\}$

- $\forall t \in TB.\ noclass(\llbracket t \rrbracket^3) = \emptyset$

- $\forall d \in \mathcal{D}.\ noclass(\llbracket d \rrbracket^7) = \emptyset$

Prop. 3.58. • case $sem*$, direct from definition of $sem*$

190

- $\begin{aligned}
& noclass(\llbracket hu \rrbracket^1) \\
&= \{x \in Var \mid \exists s,h,f,r,i.\ s,h,f,r \in \llbracket hu \rrbracket^1 \wedge [s \mid x \to i],h,f,r \notin \llbracket hu \rrbracket^1\} \\
&= \{x \in Var \mid \exists s,h,f,r,i.\ (\forall \alpha \in hu.f(\alpha) \cap dom(h) \neq \emptyset) \wedge (\exists \alpha \in hu.f(\alpha) \cap dom(h) = \emptyset)\} \\
&= \emptyset
\end{aligned}$

- $\begin{aligned}
& noclass(\llbracket ho \rrbracket^{1'}) \\
&= \{x \in Var \mid \exists s,h,f,r,i.\ s,h,f,r \in \llbracket ho \rrbracket^{1'} \wedge [s \mid x \to i],h,f,r \notin \llbracket ho \rrbracket^{1'}\} \\
&= \{x \in Var \mid \exists s,h,f,i.\ dom(h) \subseteq \bigcup_{\alpha \in ho} f(\alpha) \wedge dom(h) \not\subseteq \bigcup_{\alpha \in ho} f(\alpha)\} \\
&= \emptyset
\end{aligned}$

- case sn, sn^∞, direct from definition of $\llbracket sn, sn^\infty \rrbracket^2$

- $noclass(\llbracket v, \top \rrbracket^5) = noclass(MFR) = \emptyset$

- when $\alpha \in TVar$

  $noclass(\llbracket \alpha, \circledast \rrbracket^5) = noclass(\{s,h,f,r \mid s^+ f(\alpha) = \emptyset\}) = noclass(\{s,h,f,r \mid f(\alpha) = \emptyset\}) = \emptyset$

- when $x \in Var$, does not happen !

  $noclass(\llbracket x, \circledast \rrbracket^5) = noclass(\{s,h,f,r \mid s^+ f(x) = \emptyset\}) = noclass(\{s,h,f,r \mid x \notin dom(s)\}) = \{x\}$

- when $\alpha \in TVar, S \neq \top, \circledast$

  $noclass(\llbracket \alpha, S \rrbracket^5) = noclass(\{s,h,f,r \mid f(\alpha) \subseteq \bigcup_{vd \in S} \llbracket vd, (h,f,r) \rrbracket^8\}) = \emptyset$

- when $x \in Var, S \neq \top, \circledast$

  $noclass(\llbracket \alpha, S \rrbracket^5) = noclass(\{s,h,f,r \mid x \notin dom(s) \vee \exists vd \in S.\ s(x) \in \llbracket vd, (h,f,r) \rrbracket^8\}) \subseteq \{x\}$

- $\begin{aligned}
& noclass(\llbracket t \rrbracket^3) \\
&= noclass(\bigcap_{(v_1,v_2)} \{s,h,f,r \mid \llbracket t(v_1,v_2), f(v_1), f(v_2) \rrbracket^{6'}\}) \\
&= \left\{ x \in Var \;\middle|\; \exists s,h,f,r,i. \begin{array}{l} \bullet\ \ \forall v_1,v_2.\ \llbracket t(v_1,v_2), f(v_1), f(v_2) \rrbracket^{6'} \\ \bullet\ \ \exists v_1,v_2.\ \neg \llbracket t(v_1,v_2), f(v_1), f(v_2) \rrbracket^{6'} \end{array} \right\} \\
&= \emptyset
\end{aligned}$

- $noclass(\llbracket d \rrbracket^7) = noclass(\bigcup\limits_{g \in \llbracket d \rrbracket^{\mathcal{D}}} \bigcap\limits_{\alpha \in TVar} \{s, h, f, r \mid f(\alpha) \cap \mathbb{Z} \subseteq g(\alpha)\}) = \emptyset$

□   ■

Proposition 3.59. $A \subseteq \llbracket ar \rrbracket' \Rightarrow class(x, A) \subseteq \llbracket class(x, ar) \rrbracket'$

Prop. 3.59. Direct from Prop. 3.52 and 3.54. □   ■

3.8.13   Exists proofs

Th. 3.31 for Def. 3.37. Recall :
$$
\begin{aligned}
\llbracket \exists x.\ P \rrbracket &= \{s, h \mid \exists v.\ [s \mid x \rightarrow v], h \in \llbracket P \rrbracket\} \\
&= \{s, h \mid \exists v.\ [s \mid x \rightarrow v], h \in \llbracket x = x \wedge P \rrbracket\} \\
&= class(x, \llbracket x = x \wedge P \rrbracket)
\end{aligned}
$$

We want to prove that $\forall s, h, f, r.\exists g.$

$$s, h, f, r \in \llbracket ar \rrbracket' \wedge \bar{s}, h \in class(x, \llbracket x = x \wedge P \rrbracket) \Rightarrow s, h, g(f), r \in \llbracket class(x, T(class(x, ar), x = x \wedge P)) \rrbracket'$$

By recurrence we have $\forall s, h, f, r.\ \exists g.$

$$s, h, f, r \in \llbracket class(x, ar) \rrbracket' \wedge \bar{s}, h \in \llbracket x = x \wedge P \rrbracket \Rightarrow s, h, g(f), r \in \llbracket T(class(x, ar), x = x \wedge P) \rrbracket'$$

So $\forall s, h, f, r.\ \exists g.$

$$
\begin{aligned}
& s, h, f, r \in \llbracket class(x, ar) \rrbracket' \wedge s, h, f, r \in \{s, h, f, r \mid \bar{s}, h \in \llbracket x = x \wedge P \rrbracket\} \\
& \Rightarrow s, h, f, r \in \{s, h, f, r \mid s, h, g(f), r \in \llbracket T(class(x, ar), x = x \wedge P) \rrbracket'\}
\end{aligned}
$$

Which is

$$\llbracket class(x, ar) \rrbracket' \cap \{s, h, f, r \mid \bar{s}, h \in \llbracket x = x \wedge P \rrbracket\} \subseteq \{s, h, f, r \mid s, h, g(f), r \in \llbracket T(class(x, ar), x = x \wedge P) \rrbracket'\}$$

By Prop. 3.54,

$$
\begin{aligned}
& class(x, \llbracket class(x, ar) \rrbracket' \cap \{s, h, f, r \mid \bar{s}, h \in \llbracket x = x \wedge P \rrbracket\}) \\
& \subseteq class(x, \{s, h, f, r \mid s, h, g(f), r \in \llbracket T(class(x, ar), x = x \wedge P) \rrbracket'\})
\end{aligned}
$$

We have $x \notin noclass(\llbracket class(x, ar) \rrbracket')$, so by Prop. 3.57

$$
\begin{aligned}
& \llbracket class(x, ar) \rrbracket' \cap class(x, \{s, h, f, r \mid \bar{s}, h \in \llbracket x = x \wedge P \rrbracket\}) \\
& \subseteq class(x, \{s, h, f, r \mid s, h, g(f), r \in \llbracket T(class(x, ar), x = x \wedge P) \rrbracket\})
\end{aligned}
$$

192

Which is

$$s, h, f, r \in \llbracket class(x, ar) \rrbracket' \wedge \bar{s}, h \in class(x, \llbracket x = x \wedge P \rrbracket))$$
$$\Rightarrow s, h, g(f), r \in class(x, \llbracket T(class(x, ar), x = x \wedge P) \rrbracket')$$

By Prop. 3.52 $class(x, \llbracket T(class(x, ar), x = x \wedge P) \rrbracket) \subseteq \llbracket class(x, T(class(x, ar), x = x \wedge P)) \rrbracket$ so

$$s, h, f, r \in \llbracket class(x, ar) \rrbracket' \wedge \bar{s}, h \in class(x, \llbracket x = x \wedge P \rrbracket))$$
$$\Rightarrow s, h, g(f), r \in \llbracket class(x, T(class(x, ar), x = x \wedge P)) \rrbracket'$$

By Prop. 3.55:

$$s, h, f, r \in \llbracket ar \rrbracket' \wedge \bar{s}, h \in class(x, \llbracket x = x \wedge P \rrbracket))$$
$$\Rightarrow s, h, g(f), r \in \llbracket class(x, T(class(x, ar), x = x \wedge P)) \rrbracket'$$

□ ∎

3.8.14 Why using $\llbracket \cdot \rrbracket'$?

Because we want our semantic to be a conjunction, and we cannot express $\llbracket \cdot \rrbracket$ as a conjunction:

Case of F

Imagine we want the semantics of $ad = [ad_i | x \to \{\alpha\} | y \to \{\alpha\} | \alpha \to \{Numt\}]$. If the semantics would not keep f, we would have $\llbracket [ad_i | y \to \{\alpha\} | \alpha \to \{Numt\}] \rrbracket = \{s, h \mid s(y) \in \mathbb{Z}\}$ then to take into account that $x \to \{\alpha\}$ we do not know which value has be chosen for α...

Case of R

Imagine we want the semantics of $ad = [ad_i | x \to \{Loc(\{*2\}, Numt, \alpha_1)\} | z \to \{\alpha_1\} \mid y \to \{\alpha_2\} | \alpha_1 \to \{Loc(\{*2\}, Numt, \alpha_2)\} | \alpha_2 \to \{Loc(\emptyset, Nilt, Nilt)\}]$

if we take the semantics of it without the assignment for x, we get $\{s, h | \exists l_1, l_2.\ s(z) = l_1 \wedge s(y) = l_2 \wedge h(l_1) = \langle i, l_2 \rangle \wedge h(l_2) = \langle \texttt{nil}, \texttt{nil} \rangle \wedge l_1 \neq l_2\}$,

but now, with $x \to \{Loc(\{*2\}, Numt, \alpha_1)\}$, we know that we must also have $s(x) = l \wedge h(l) = \langle i, l' \rangle$ with $l' \neq l_1$, but we don't know that l should be different from l_2...

In other words...without beeing precise we would say that for $\alpha \rightarrow \{Loc(\{*2\}, Numt, \alpha)\}$, the $\{*2\}$ does not only say that we cannot have $h(l) = \langle i, l \rangle$ but also we want $\forall n.\ h(l_0) = \langle i, l_1 \rangle, ..., h(l_n) = \langle i, l_{n+1} \rangle$, $l_0 \neq l_n + 1$.

And of course, we want the same property for $\alpha_1 \rightarrow \{Loc(\{*2\}, Numt, \alpha_2)\}$, $\alpha_2 \rightarrow \{Loc(\{*2\}, Numt, \alpha_1)\}$

Chapter 4

Implementation

In this chapter, we describe a prototype implementation of our static analysis techniques for separation logic.

4.1 Introduction

The implementation reads a program (C) from a file and reads a formula in separation logic (P) from a file. From C and P, the implementation can compute the weakest precondition $(wlp(P, C))$ and the strongest postcondition $(sp(P, C))$ of the program and the formula in separation logic. The tool can also compute an overapproximating translation of a separation logic formula into an element of our abstract language $(T(P))$. In this case, the implementation produces an executable program which reads a program from a file, prints out the safety precondition for this program $(wlp(\texttt{true}, C))$, the postcondition for the safety precondition $(sp(wlp(\texttt{true}, C), C))$, and a translation of this postcondition into our abstract language$(T(sp(wlp(\texttt{true}, C), C)))$. There is also an executable program which reads a formula and prints out its translation into our abstract language.

The translation can sometimes find that the formula P was equivalent to $\texttt{true}$ or $\texttt{false}$, otherwise it gives an overapproximation of the formula.

For a program C, if the translation of the postcondition for the safety precondition $(T(sp(wlp(\texttt{true}, C), C)))$ returns $\texttt{false}$, then we know that for any input entry, the program fails. If the translation of the safety precondition $(T(wlp(\texttt{true}, C)))$ returns $\texttt{true}$, then we

know that the program never fails (even if we overapproximate, because `true` is a particular value of our output). Otherwise, we cannot decide about all errors of the original program, but we do know some properties of the final states if the program terminates without error. For example, if in our result we have that x and y reach the same auxiliary variable and only that then we know that if the program terminates, x and y are aliased. Or, if x reaches only $\{Nilt\}$, then we know that at termination, $x = $ `nil`.

Organisation of the chapter: In Sect. 4.2, we present the tool's software architecture and what are the inputs, the outputs, the executable programs produced. In Sect. 4.3, we describe the syntax of the inputs and the data structures manipulated. In Sect. 4.4, we present the main part of the implementation, the translation of formula into the abstract language. In Sect. 4.5, we discuss the design choices we made for the implementation.

4.2 Software architecture

The implementation uses modules which define types and functions on those types. There are some options for compiling, but everything can be compiled by the command, `make`.

4.2.1 Reading from files

First, as shown in Fig. 4.1, we have a compiler in `program_read.ml` that reads a program in a file (called in the figure `prog_example.p`) using the lexer in the file `prog_lexer.mll` and the parser in the file `prog_parser.mly` (both the lexer and the parser, are using the file `support.ml` which is principally used for defining the type `info` which records informations about where in the original file a token is read). In the same way as the compiler for programs, we have a compiler that reads a formula in a file as shown in Fig. 4.2.

The syntaxes of the input files of programs and formulae and the data structures produced are presented in Sect. 4.3.
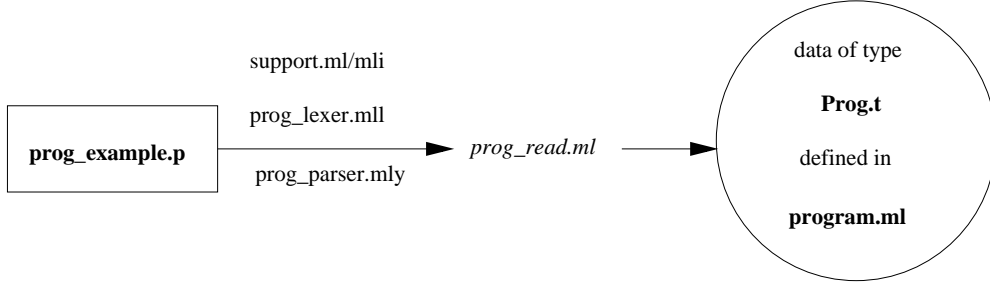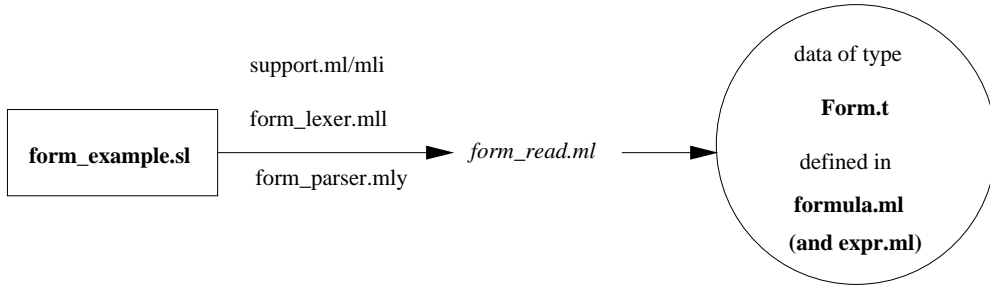
Figure 4.1: *Reading a program in a file*

Figure 4.2: *Reading a formula in a file*

4.2.2 Computing informations

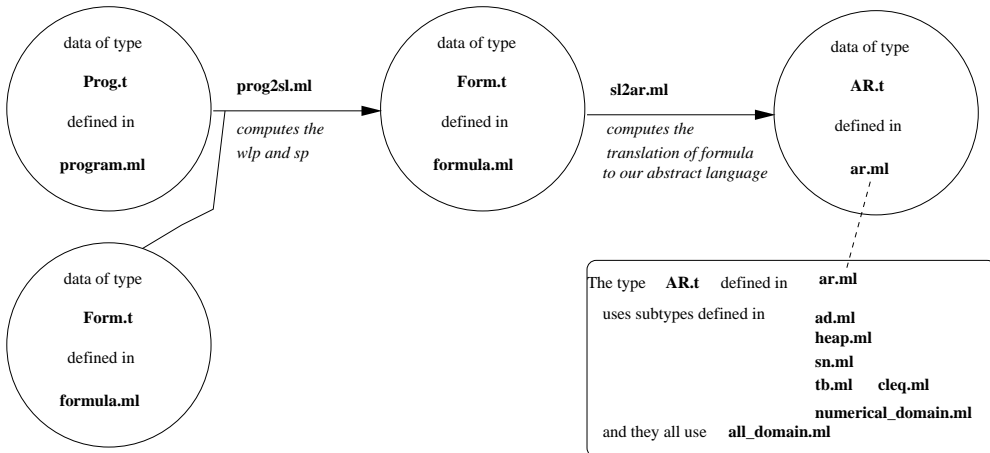In Fig. 4.3, is a schema of the programs which manipulate data.

Figure 4.3: *Computing pre- and post-conditions, computing the abstraction*

In the file `prog2sl.ml` are implemented the functions `wlp` and `sp` which both take as in-

put a program of data type `Prog.t` and a formula of data type `Form.t` and return the pre- or post-conditions of data type `Form.t`. The computation of pre- and post-conditions have no interest by itself since it is straightforward and there is no approximation or implementation choice of any kind. (We use, for example, fixpoints for the while-loops

$$wlp(P, \texttt{while } E \texttt{ do } C_1) = \nu X_v.((E = \texttt{true} \wedge wlp(X_v, C_1)) \vee (E = \texttt{false} \wedge P))$$

and

$$sp(P, \texttt{while } E \texttt{ do } C_1) = (\mu X_v.sp(X_v \wedge E = \texttt{true}, C_1) \vee P) \wedge (E = \texttt{false})$$

.)

In the file `sl2ar.ml` is implemented the function `sl2ar` which takes a formula of data type `Form.t` and returns its overapproximation in the abstract language as a data of type `AR.t`. The computation of the abstraction is presented in Sect. 4.4.

4.2.3 Building executables

We have those files used to produce executable programs, as presented in Table 4.1. The files from the first column (`mk_foo.ml`) are the last files in the dependency chain, they are the last ones to be compiled and the one producing the executables in the third column (`foo`). They print things (like "The program read is :") and call functions from other files (like the function `prog_read` from the file `prog_read.ml`), mainly from the files in the second column. The use of the executable programs from the third column is presented in Fig. 4.4.

4.3 Syntaxes of inputs and data structures

In this section, we define the syntax of the input files which can contain programs and the data structures produced by the compiler. We present the syntax of input files for formulae and data structures produced by the compiler. We then present the data structure of the abstract domain.

Files to compile	Main files used	Executables produced
mk_prog_read.ml	prog_read.ml	prog_read
mk_form_read.ml	form_read.ml	form_read
mk_prog2sl.ml	prog2sl.ml	prog2sl
mk_sl2ar.ml	sl2ar.ml	sl2ar
mk_prog2sl2ar.ml	prog2sl.ml, sl2ar.ml	prog2sl2ar

Table 4.1: *Table about the executable files*

The type `info` is just a type defined in the file `support.ml` that record where in the file a token was read by the compiler. (We later use it for heuristics in choosing which nodes to summarise.)

4.3.1 Program syntax

Syntax of the input file of programs

We define as a grammar the syntax of a program P in the input file: (We allow more diversity in the syntax, for example the keyword can have different capitalizations. We reject some badly formed expressions like `x:= true + 3;` or we allow `x != 2` for `not (x = 2)` but this is common and not expressed here for simplicity)

$$
\begin{aligned}
n & := && \text{any string} \\
E & := && (E) \mid nil \mid n \mid \mathbb{Z} \mid true \mid false \mid E \text{ and } E \mid E \text{ or } E \mid not\ E \\
& && \mid E = E \mid E < E \mid E + E \mid E - E \mid E * E \mid E/E \mid -E \mid E => E \mid E <= E \\
P & := && P \mid C \mid P; C \\
C & := && call\ n\ (n_1, ..., n_n) \mid fun\ n\ (n_1, ..., n_n)\ P \mid \texttt{while}\ E\ \texttt{do}\ P\ \texttt{done} \mid \texttt{skip} \\
& && \mid \texttt{if}\ E\ \texttt{do}\ P\ \texttt{else}\ P\ \texttt{fi} \mid \texttt{dispose}\ E \mid n := \texttt{cons}(E, E) \mid n := E \\
& && \mid n := n \cdot 1 \mid n := n \cdot 2 \mid n \cdot 1 := E \mid n \cdot 2 := E
\end{aligned}
$$

The program reader reads programs with declarations and calls of functions, but we do not treat them in the current version of the analyser.

Syntax of the data structure for program produced by the compiler

The syntax of the program is defined in the file `program.ml` as the type `t` from module `Prog` as seen in the following :
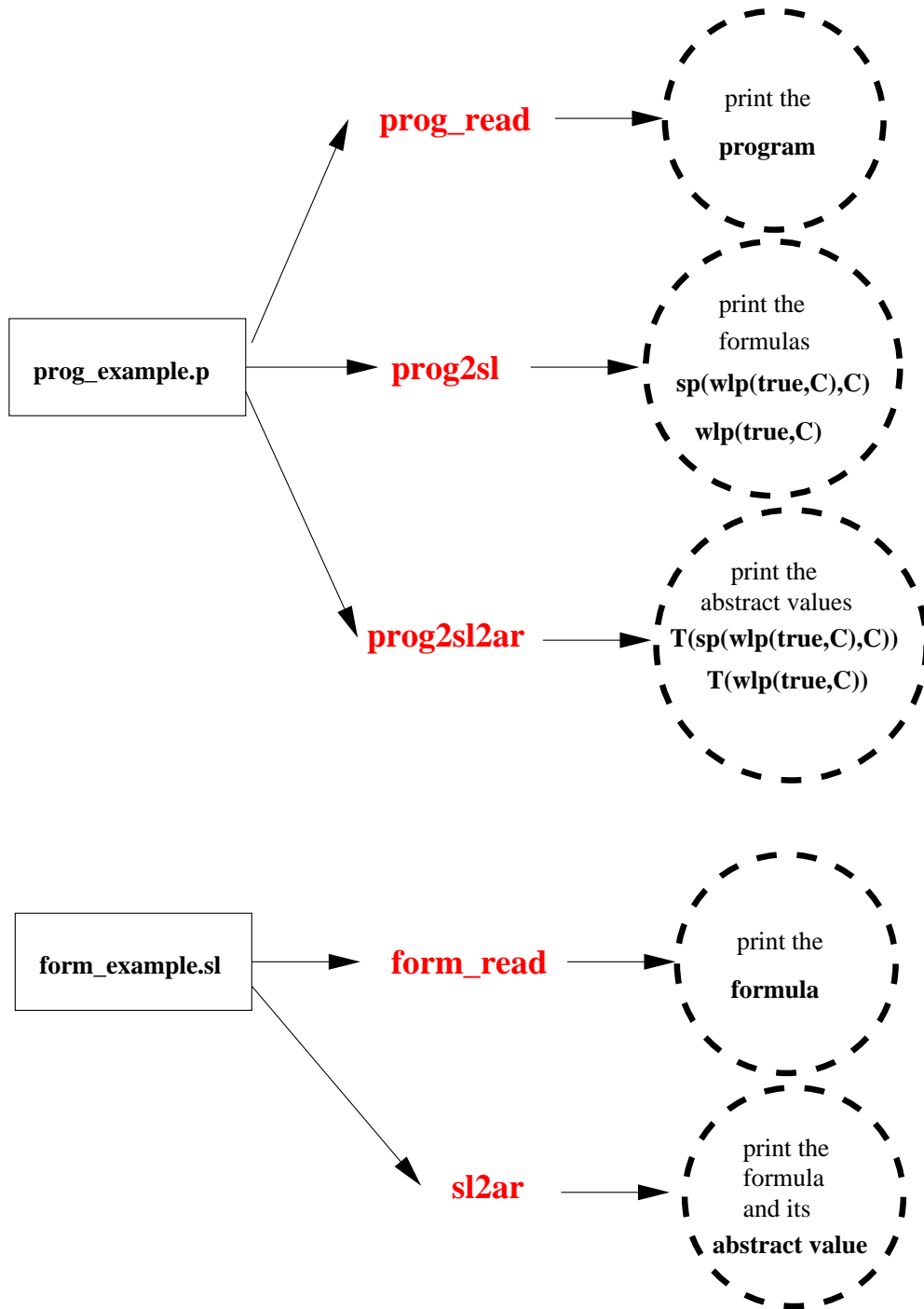
Figure 4.4: *The executables produces by the analyses*

```
module Prog =                          module Field =

struct                                 struct
```

```
    type fun_var = string                          type t = Fst | Snd | Field of string

    type t =                                        [...]

        AssSS of info * string * Expr.t             end

      | AssSH of info * string * string * Field.t

      | AssHS of info * string * Field.t * Expr.t

      | Cons_p of info * string * Expr.t * Expr.t

      | Dispose of info * string

      | Comp of info * t * t

      | IFT of info * Expr.t * t

      | IFTE of info * Expr.t * t * t

      | Skip of info

      | W of info * Expr.t * t

      | Fun of info * fun_var * (info  * string) list * t

      | Call of info * fun_var * (info * string) list
[...]
end
```

4.3.2   Formula syntax

Syntax of the input file of formula

As for programs, we allow some variations on the syntax which are not presented here:

$$
\begin{aligned}
n \quad &:= \quad any\ string \\
E \quad &:= \quad (E) \mid nil \mid n \mid \mathbb{Z} \mid true \mid false \mid E\ and\ E \mid E\ or\ E \mid not\ E \\
&\qquad \mid E = E \mid E < E \mid E + E \mid E - E \mid E * E \mid E/E \mid -E \mid E => E \mid E <= E \\
F \quad &:= \quad (F) \mid true \mid false \mid (n \mid - > E, E) \mid (n \backslash - > E, E) \mid E == E \mid F <==> F \\
&\qquad \mid F ==> F \mid F <== F \mid F\ *\ F \mid F\ -* F \mid \backslash\backslash e\ n \cdot\ F \mid \backslash\backslash f\ n \cdot\ F \mid emp \\
&\qquad \mid E\backslash/E \mid E/\backslash E \mid not\ E \mid Mu\ n\ \cdot F \mid Nu\ n\ \cdot F \mid F\ [E/n]
\end{aligned}
$$

We have "$\backslash\backslash e$" for existantial quantifier and "$\backslash\backslash f$" for universal quantifier.

Syntax of the data structure for formula produced by the compiler

The syntax of the formulae is defined in the file `formula.ml` as the type `t` from module `Form` as seen in the following :

```
module Form =
struct

  type form_var = string

  type t =

    | Exists of info * string * t  | Forall of info * string * t

    | Impl of info * t * t  | ImplEq of info * t * t  | Not of info * t

    | Equ of info * Expr.t * Expr.t

    | Wedge of info * t * t  | Vee of info * t * t

    | Mag of info * t * t  | Ast of info * t * t  | Emp of info

    | Subst of info * t * Expr.t * string  | FVar of info * form_var

    | Mu of info * form_var * t  | Nu of info * form_var * t

    | Cons of info * string * Expr.t * Expr.t

    | Conshook of info * string * Expr.t * Expr.t

    | False of info  | True of info
[...]
end
```

using the type `t` from module `Expr` from file `expr.ml`

```
module Expr =
struct

  type t =

      Nil of info | Var of info * string | Int of info * int | Bool of info * bool

    | Add of info*t*t | Sub of info*t*t | Mul of info*t*t | Div of info*t*t

    | GT  of info*t*t | LT  of info*t*t | GEq of info*t*t | LEq of info*t*t
```

```
    | Or  of info*t*t | And of info*t*t | Imp of info*t*t | Neg of info * t
    | Eq  of info*t*t
[...]
end
```

4.3.3   Abstract data syntax

We wrote the domains and subdomains using modules which define types and functions on
the types. Here we only present the types.

To define a module `FooSet` for sets of elements of type `foo`, we do :

```
module PreFooSet = Set.Make
  (struct
    type t = foo
    let compare=compare
  end)
module FooSet =
struct
  include PreFooSet
[...]
end
```

For mappings, we build functors. For example, we define the type `Environment` which
is used to create maps from elements of type `Var.t` to elements of another type. We
use the module `Map2` which already implements mappings. We first define what is a
`PrintableDomain` for the codomain of the mapping:

```
module type PrintableDomain =
  sig
    type t
    val print: t -> unit
```

```
      end
```

Then, we define the functor `Environment` :

```
module PreEnv = Map2.Make(
  struct
    type t=Var.t
    let compare=compare
  end )
module Environment =
  functor (V: PrintableDomain) ->
  struct
    type t = V.t PreEnv.t
    [...]
  end
```

The abstract result is of type

$$AR.t = \text{Top of bool | Bot | V of ar\_dom}$$

where `Top (true)` corresponds to the formula *true* (so `Not (Top true))` is `Bot`), and `Top (false)` corresponds to the domain's overall `Top`, meaning that we know nothing about the value. (So `Not (Top false))` is `Top false`.)

`ar_dom` is the domain AR defined in Chap. 3 :

```
 ar_dom = {
    stack : ST.t;
    ad : AD.t;
    hu : Heap.hu;
```

204

	Auxiliary variables	Formula variables	Variables used for fixpoints
Variable type	`Var.t =` $\left\{ \begin{array}{l} name : string; \\ time : float; \\ info : info; \end{array} \right\}$	`StackVar.t = string`	`FormVar.t = string`
Sets of variables	`VarSet.t`	`StackVarSet.t`	`FormVarSet.t`
Pairs	`VarPair.t = Var.t * Var.t`		
Sets of pairs	`VarPairSet.t`		
Mapping functors	`Environment : Var.t →` ? `Tabular : VarPair.t →` ?	`ST.t : StackVar.t → Var.t`	`FormEnvironment :` `FormVar.t →` ?

```
    ho : Heap.ho;

    sn : SN.all;

    sninf : SN.inf;

    tb : TB.t;

    d : NumDom.t;

    clean: bool;

  }
```

The field `clean` is just a flag that remembers if a value has been "cleaned" or not (to be "clean" has a meaning in the implementation and we do not claim that cleaning is a normalization).

If we unfold the definitions of subdomains of **ar_dom** we get Fig. 4.5. You can notice that (in `LocSet.element`), we only allow locations of two variables for simplicity in the implementation.

The type `NumDom.t` so far is only `Top | Bot | D` but `D` could be parametrized by an existing numerical domain.

Examples

In the implementation, the arrows from variables in the formulae are registered in the field `stack`. The arrows from auxiliary variables are registered in `ad`; all the arrows from one variable are registered in one arrow in `ad`; the set of pointed values are split into sets of variables through the field `variables`; a set of values in {Numt, Truet, Falset, Nilt} in the field `values`; some boolean fields which indicate if the values are in the set or not `dangling, ood, loctop`; and the last field `loc` is again a set of all the locations possible but those locations can only be of two auxiliary variables. A location has 4 fields: `ast1` and `ast2` are the booleans' meaning if we have ∗1 and ∗2, and `car` and `cdr` are auxiliary variables for the first and second fields of the location. The fields `hu` and `ho` are as expected for the heap bounds and `sn` and `sninf` for the summary nodes. `tb` is the table encoding

206

```
 ar_dom = {
    stack : StackVar.t -> Var.t;
    ad : Var.t -> (Top | Bot | Set of { variables : VarSet.t;
                                        values : ValSet.t;
                                        dangling : bool;
                                        ood: bool;
                                        loctop : bool;
                                        loc : LocSet.t; } )
    hu : VarSet.t;
    ho : H of VarSet.t | Full | Bot;
    sn : VarSet.t ;
    sninf : VarSet.t;
    tb : (Var.t * Var.t) -> (Top | Bot | Set of CleqSet.t);
    d : NumDom.t;
    clean: bool;
  }
with
ValSet.element = Numt  | Truet  | Falset  | Nilt
LocSet.element = {ast1: bool; ast2: bool; car : Var.t; cdr : Var.t}
ClEqSet.element = DD_eq | D_eq | Eq_eq | Sub_eq | Sup_eq | Sharp_eq | Circ_eq
```

Figure 4.5: *The result domain structure*

the domain CL_{eq}. The implementation tries to avoid recording some redundances: in the first example, the obvious values like `(v3,v3) -> DD_eq, Eq_eq` are not registered, and `(v4,v3) -> Set D_eq` is deduced from `(v3,v4) -> Set D_eq` (if there was a `Sup_eq` it would become a `Sub_eq`). The numerical information is recorded in the field `d`, which values so far can only be `Top, Bot` and `D`. But `D` should be specialized with the advancement of the implementation.

Example 1: The formula $(x = y \land y = \mathtt{nil}) \lor (y = \mathtt{true})$ translates to
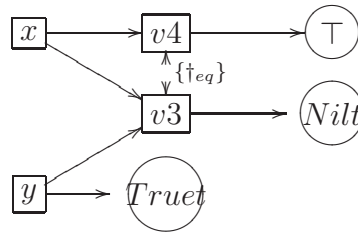
and in a non-graphical representation to

$$
\left(
\begin{array}{l}
[ar_i | x \to \{v3, v4\} | y \to \{v3, Truet\} | v3 \to \{Nilt\} | v4 \to \top], \\
\emptyset, TVar, \emptyset, \emptyset, \\
\left[t_i \left|
\begin{array}{c|c|c}
 & v3 & v4 \\
\hline
v3 & \{\ddagger_{eq}, =_{eq}\} & \{\dagger_{eq}\} \\
\hline
v4 & \{\dagger_{eq}\} & \{\ddagger_{eq}, =_{eq}\}
\end{array}
\right. \right], \top^{\mathcal{D}}
\end{array}
\right)
$$

which in the implementation would be

```
V {stack= <(x -> v1),(y -> v2)> ;

   ad= <(v1 -> Set {variables: {v3,v4};

                    values: ValSet.empty;

                    dangling: false;

                    ood:false;

                    loctop:false;

                    loc: LocSet.empty;}),

        (v2 -> Set {variables: {v3};

                    values: {Truet};

                    dangling: false;

                    ood:false;

                    loctop:false;

                    loc: LocSet.empty;}),

        (v3 -> Set {variables: VarSet.empty;

                    values: {Nilt};

                    dangling: false;

                    ood:false;

                    loctop:false;

                    loc: LocSet.empty;}),

        (v4 -> Top)>;

   hu= VarSet.empty;
```

```
    ho= Full;

    sn= VarSet.empty;

    sninf= VarSet.empty;

    tb = <( (v3,v4) -> Set {D_eq} )>;

    d = Top;

    clean = true;

}
```
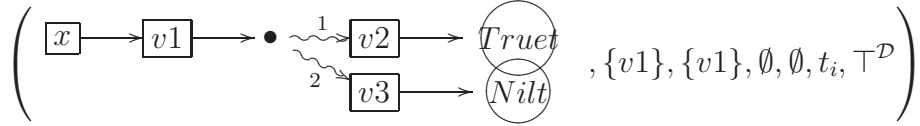
Example 2: If we want to translate the formula $(x \mapsto \mathtt{true}, \mathtt{nil})$ we would have the result

$$\left(\quad , \{v1\}, \{v1\}, \emptyset, \emptyset, t_i, \top^{\mathcal{D}} \right)$$

which is

```
V {stack= <(x -> v1)> ;

    ad= <(v1 -> Set {variables: VarSet.empty;

                     values: ValSet.empty;

                     dangling: false;

                     ood:false;

                     loctop:false;

                     loc: { {ast1:false;

                             ast2:false;

                             car: v2;

                             cdr: v3;} };}),

         (v2 -> Set {variables: VarSet.empty;

                     values: {Truet};

                     dangling: false;

                     ood:false;

                     loctop:false;
```

```
                  loc: LocSet.empty;}),
        (v3 -> Set {variables: VarSet.empty;
                    values: {Nilt};
                    dangling: false;
                    ood:false;
                    loctop:false;
                    loc: LocSet.empty;})>;
    hu= {v1};
    ho= {v1};
    sn= VarSet.empty;
    sninf= VarSet.empty;
    tb = <>;
    d = Top;
    clean = true;
}
```

4.4 The translation of formula into elements of the domain: sl2ar.ml

The translation (function `sl2ar` in file `sl2ar.ml`) takes a formula of type `Form.t` and returns an overapproximation of it in the abstract language of type `AR.t`.

First, the formula is "canonized" (by calling the function `Form.canonize`), which is not a real canonicalization but a function which changes the formula into an equivalent formula which improve its translation (for example, it pushes the ¬ as deap as possible). This function is not fully implemented yet, and probably can be specialised many times to gain more and more precision.

Then, we call a recursive function `sl2ar_rec` whose first argument is an element of the abstract language of type `AR.t` (which for the first call is `Top true` meaning it's the top of the

abstract language plus we know we did not do yet any approximation), second argument is a formula of type `Form.t` and third argument is an environment of type `FixPtEnv.t` which maps variable for fixpoints formula into values of type `AR.t`. (For the first call, it is `FixPtEnv.top` which is an empty environment.) The function will return a value of type `AR.t` which is an overapproximation of the intersection between the first argument and an overapproximation of the translation of the second element taking the third argument (the environment) for interpreting (some of) the fixpoint variables in the formula. The function will also return an updated environment of type `FixPtEnv.t`.

Before, after and during the loop, the function `AR.clean` is called on the current element of type `AR.t`. Again, this is not a canonicalization but a function which transforms an element of the abstract language into an equivalent one which makes the analysis more efficient (for example, to detect that the element is in fact equivalent to `AR.bot`).

Then, the function `sl2ar_rec` splits cases on the connective of the formula and it respects the translation presented in Sect. 3.

When treating a variable, we usually have to "initialize" it if it is not done yet, that is picking a fresh auxiliary variable and set that variable pointing to this fresh auxiliary one in the stack.

The equalities: For equality of expressions with boolean connectives we use rewriting rules (like $C == A \lor B$ is transformed into $(\texttt{true} == C \land \texttt{true} == A) \lor (\texttt{true} == C \land \texttt{true} == B) \lor (\texttt{false} == C \land \texttt{false} == A \land \texttt{false} == B)$).
The numerical equalities are treated to rewrite expressions to get a boolean equal to a numerical expression, then we call a function `evaluate_numerical_expression` on E or $\neg E$ depending on the boolean. This function `evaluate_numerical_expression` principally relies on a function of the numerical domain (not implemented yet) which takes a numerical expression, a stack (mapping formula variable to auxiliary variables) and a value of the numerical domain and returns an intersection of this value for the cases where the numerical expression holds, and also returns an updated stack and the list of the variables

211

which are numerical. This function is called at the deepest level, so it won't have to evaluate numerical expressions containing $\lor$ or $\land$ or other complicated expressions. We then update the numerical part with the updated numerical value and then we intersect in the domain each numerical variable with the singleton containing `Numt` by calling the function `AD.inter_var_pvd`.

A variable equal to `Nil` mainly consists of calling the function `AD.inter_var_pvd` for it and for the singleton containing `Nilt`. A variable equal to `true` is the same but with `Truet`.

The equality of two variables is treated by the function `AR.inter_stackvar_stackvar` which in a simple view consists of having the two variables pointing to a same variable and making the intersection of what they were pointing to.

The equality of equalities are split (`true` $== (A = B)$ become $(A == B)$). The case `false` $== (A = B)$ is a bit different and is kind of a rewriting of the equality cases but it tries to push the negation as deep as possible. This phase obliges us to treat the intersection twice because if we have `false` $== (x = Nil)$, we will do some kind of intersection of x with "`not`" `Nilt`, which for simplicity, we do not express if x used to be anything. Our domain can express only $x \neq Nil$ by setting a fresh auxiliary variable to `Nilt` then setting that x is not equal to that auxiliary variable, but this would be quite costly, and would increase the size of the data we manipulate, so, to keep some precision we decide that the translation of conjunction instead of being $T(ar, A \land B) \triangleq T(T(ar, A), B)$ to be $T(ar, A \land B) \triangleq T(T(T(ar, A), B), A)$.

The theoretical translation of $\exists$ is $T(ar, \exists x.\ P) \triangleq class(x, T(class(x, ar), x = x \land P))$, so we have a function `AR.fun_class` which implement $class$ and we in fact do at the end another intersection with ar.

Fixpoints: The computation of $\mu X_v.\ P$ is correct only if the formula is equivalent to the infinite union of the iterations. (This is the case for the formula produced by wlp or sp and any of the "useful" formula one would write). We first set the value of X_v in

the fixpoint-environment to `Bot`, then we call `sl2ar_rec` for $ar\_\top$ and P and we make a union of the results at each step, then we call the widening, and we test for stopping if the new result is equal or smaller than the previous one. For simplicity we implemented a weak test in `AR.weak_knowing_A_is_bigger_than_B_then_is_A_smaller` which is a comparison of two elements of the domain, knowing that A is already bigger than B, it says if A is also smaller than B but sometimes says no when it is. This weak implementation is not proved to terminate but we believe it does and at least it is safe and efficient. Then, when the fixpoint is computed, we call again the function `sl2ar_rec` but taking into account ar (the fixpoint has to be computed independently from ar).

Currently, the widening is what is presented in the previous chapter, what can be changed is the merging strategy which chose which auxiliary variables can be merged. The current strategy consists of merging new auxiliary variables with the previous ones (if existing) which have the same info, which means come from the same part of the formula. In practice, only the $\exists$ will produce infinitely many auxiliary variables and they are the ones which will be merged. This part can be changed to have more specific strategy of merging but we did not yet implement this.

The computation of ν is always correct and does not ask that the fixpoint is an intersection, because we have $\nu X_v.\ P = P\{(\nu X_v.\ P)/X_v\}$. As for μ, we can not take ar into account before we compute the approximation of the fixpoint. The procedure is a refinement; first we overapproximate the value of the fixpoint to `Top false`, setting in the fixpoint-environment X_v to `Top false` and translating P in this environment. We repeat the operation many times with the new value. We stop like for μ using

`AR.weak_knowing_A_is_bigger_than_B_then_is_A_smaller`.

We improve the computation of ν when we know that it is a big intersection, because then we can already start with `Top true` (which we indeed do if translating wlp or sp).

4.5 Analysis

The implementation is still not finished, as some functions have not been implemented yet (in this case, they return the safest and simplest value). All the data structures are defined and the main algorithm of translating formulae into elements of the domain is implemented.

The major task which remains to be implemented is the translation of numerical expressions. This is not specific to our domain or our implementation or to separation logic, but is a major goal to be implemented since we want the tool to be generic enough for when we plug the numerical module with existing numerical domains.

Another important point of the implementation is the rewriting of the formulae to improve the precision of the implementation. This is the case as explained earlier for negation. Another important example, since the translation is an overapproximation, while translating $F\!\rightarrow\!\!*G$, we do not have precise information about what is F, then we have to suppose that it might have been equivalent to `false` thus we have to translate $F\!\rightarrow\!\!*G$ to `Top (false)`. To gain precision there, we could split the cases of F or we could also improve the way we record "when and where we do overapproximate" (now, only `Top (true)` allows to say we did not overapproximate) and then write a function for $\rightarrow\!\!*$ directly in the abstract language (like we did for $*$). Another example is that we treat $(a \mapsto b, c) * \mathtt{true}$ as $a \hookrightarrow b, c$ and treat $\hookrightarrow$ as a specific connective which improves the analysis. During a previous implementation, we had an intermediate domain which were sets of $(a \mapsto b, c)$, so we could make a more precise translation if we were first collecting the locations (as much as possible). We did not do this here since the implementation is not finished and we went for simplicity but this point relates to the question of working directly with a logic which divides the separation information from the other informations as is now common (as seen in the next chapter about comparison with other works). It would be hazardous to compare, during other analysis, the cost to keep the formulae within a subset of the logic which keeps the separation information separately and the cost in our analysis to rewrite formulae but here we believe there is a link between those two "expenses".

We already talked about fixpoints, when the implementation is finished, this is the major place for tuning the precision of the analysis by making a more clever choice for merging nodes. So far, we merge nodes built from translating the same part of the formula, in practice that means we merge all the new nodes created by $\exists$ inside a fixpoint. We could think of merging nodes only if they have some other common properties. (For example, if the numerical domain could record the parity of numbers, we could think of cases where the parity would also be a criterion for merging.)

We could add some sugared structures like "list" or "tree" and treat them specifically, or more ambitiously add some mechanisms to build those structures by giving them some definitions in the existing data structures and the way the analysis will treat them (folding/unfolding for example).

The major point of interest of the analysis is the way we use auxiliary variables, the way we added them directly in the definition of our domain and gave them a semantics came from a previous implementation where we found them useful.

The structure of the implementation with modules is convenient to modify different parts of the analysis without having to make changes everywhere.

The test of inclusion for any element of our language would be very costly, and in general, the language was not designed to be a usual abstract domain with all the functions coming along but was designed to have a precise semantics and to fit the implementation requirements. So, for example, to implement the test of inclusion, we only implemented a weak test of inclusion, which is always safe, but precise only when we use it during the fixpoint computation. (That is, if an auxiliary variable has the same name in the two arguments, it comes from the same part of the translated formula, that is, we would be precise for the intersection of $x \rightarrow v1 \rightarrow Nilt$ with $x \rightarrow v1 \rightarrow Top$ or even with $x \rightarrow v2 \rightarrow Top$ but not with $x \rightarrow v1 \rightarrow Nilt$ and $y \rightarrow v1 \rightarrow Falst$ which would never happen in our implementation since we take care about taking fresh variables.)

Chapter 5

Comparison with other works

There exist many papers that use concepts similar to ours, but we will focus on two active lines of work which have been operating for serveral years with many people: "smallfoot", the work from London which is the most involved with separation logic, and "TVLA", a well established work for analysing storage structure.

5.1 Smallfoot

In this section, we give some comparison with the work from the "East London Massive", as they call themselves. They are the major group involved in separation logic.

We will talk about 4 of their articles. The first, "Smallfoot: Modular Automatic Assertion Checking with Separation Logic",[20] was published in 2005. This was followed by, "A local shape analysis based on separation logic"[21], which introduced the Space Invader tool. The two most recent works are "Shape analysis for composite data structures"[22] and "Footprint Analysis: A Shape Analysis that Discovers Preconditions"[23].

5.1.1 Smallfoot: Modular Automatic Assertion Checking with Separation Logic (FMCO'05)

This paper introduced the smallfoot analysis tool, which implements a program checker for a subset of separation logic.

Applications of the tool

Smallfoot is made for **checking** specifications of programs dealing with pointers, in particular with lists or trees.

Characteristics/techniques

- The pre and post conditions for procedures and loop invariants must be specified with the program to check.

- The specification must be written in a **subset of separation logic** where pure boolean conditions (Π) are separated from the separation information (Σ).
  The syntax of the separation logic subset is

$$B ::= E = F \mid E \neq F$$
$$\Pi ::= \texttt{true} \mid \texttt{false} \mid B_1 \wedge \cdots \wedge B_n$$
$$H ::= E \mapsto (t_1 : E_1, ..., t_n : E_n) \mid \textbf{tree}(E) \mid \textbf{ls}(E, E) \mid \textbf{xlseg}(E, E, E, E)$$
$$\Sigma ::= \texttt{emp} \mid H_1 * \cdots \wedge H_n$$
$$P ::= \Pi \wedge \Sigma \mid \text{ if } B \text{ then } P \text{ else } P$$

A program assertion is a P-phrase.

- The tool includes several built-in recursive predicates (**tree**, **ls** and **xlseg**) but no mechanism for arbitrary inductive definitions of data structures. For example, **tree**(x) asserts that x points to a tree in a heap and **xlseg**(x, y) asserts that there is a xor-linked list between x and y.

- The tool does symbolic execution of assertions and generates a set of verification conditions that must be solved by a theorem prover.

- The tool includes an extension to handle parallelism expressed with conditional critical regions (CCRs).

Example: We have a procedure, **disp_tree**, which disposes the cells of a binary tree:

217

```
procedure disp_tree (p) {

    i := p -> l;

    j := p -> r;

    dispose_tree(i);

    dispose_tree(j);

    dispose(p)

}
```

If we supply the Hoare triple assumption for the procedure: $\{\textbf{tree}(p)\}\textbf{disp\_tree}(p)\{\texttt{emp}\}$, then smallfoot generates this proof:

$$[(p \mapsto l : x, r : y) * \textbf{tree}(x) * \textbf{tree}(y)]$$
$$i := p \rightarrow l; j := p \rightarrow r;$$
$$[(p \mapsto l : i, r : j) * \textbf{tree}(i) * \textbf{tree}(j)]$$
$$\textbf{disp\_tree}(i);$$
$$[(p \mapsto l : i, r : j) * \textbf{tree}(j)]$$
$$\textbf{disp\_tree}(j);$$
$$[(p \mapsto l : i, r : j)]$$
$$\textbf{dispose}(p);$$
$$[\texttt{emp}]$$

Advantages/drawbacks

Smallfoot is fundamentally a tool for checking, that is, the user has to give the loop invariants and procedure pre- and post-conditions. For its purpose, the tool is good, since separation logic is nice for handwriting specifications. But specification checking implies that one already knows the properties of the program to be checked. It is unclear whether the tool will scale to large programs, but this is a standard problem of the general technique of checking.

Finally, the subset of separation logic implemented by smallfoot is not enough for expressing pre- and post-conditions for arbitrary commands, for example, for while loops or for when the wlp needs $-\!\!*$. (But since the tool does checking, the question of the paper is

218

to be able to check formulae from the specified subset and not to be able to express the *wlp* and *sp* for every comand.

Relationship to our work

The smallfoot research is not directly comparable with our work since the goals are different. Smallfoot is for checking programs and we cannot do that. To do so, we would need some way to check the implications of the pre- or post-conditions given with the wlp or sp, or we should write a procedure that checks inclusion of elements of our domain. (We do use such a procedure when translating fixpoints, but it is an overapproximating procedure). Still, since the primary goal of our work is to build an intermediate language that allow interactions between different analyses, we could do this with our tool. Also, the subset of separation logic used with smallfoot is different from ours since we use any separation logic formula. It could be interesting to build a specific translation of the formulae from the subset accepted by smallfoot into our domain.

5.1.2   A local shape analysis based on separation logic (TACAS'06)

This paper presents the "Space invader" tool.

Applications of the tool

The tool is dedicated to doing **shape analysis of linked list programs**, where Hoare triples are calculated.

Characteristics/techniques

- As with smallfoot, the space invader tool employs a **subset of separation logic** which is the domain of analysis. This subset separates the pure part (Π) and the spatial part (Σ) of the formula as in smallfoot.

The syntax of assertions goes as follows:

$$\Pi ::= \{\} \mid \{E = F\} \mid \Pi_1 \wedge \Pi_2$$
$$\Sigma ::= \emptyset \mid \{E \mapsto F\} \mid \{\mathbf{ls}(E, F)\} \mid \{\mathbf{junk}\} \mid \Sigma_1 * \Sigma_2$$
$$P ::= \Pi, \Sigma \mid P_1 \vee P_2$$

Note that syntax of the heap structure is greatly restricted, limited to the three primitive assertion forms:

  i $E \mapsto F$ (E points to F)

  ii $\mathbf{ls}(E, F)$ (there is a linear list from E to F)

  iii $\mathbf{junk}$ (anything at all)

For this reason, space invader can deduce only basic properties of lists.

- The tool does not deal with numericals.

- The analysis uses symbolic execution of assertions but attempts to make an "abstract domain" of canonical forms of assertions, which are "widened" using rewriting rules.

In the canonisation rules below, $P(E, F)$ stands for $E \mapsto F$ or $\mathbf{ls}(E, F)$

Table 2 Abstraction Rules

$$\frac{}{E = x' \wedge \Pi \mid \Sigma \rightsquigarrow (\Pi \mid \Sigma)[E/x']} \text{ (St1)} \qquad \frac{}{x' = E \wedge \Pi \mid \Sigma \rightsquigarrow (\Pi \mid \Sigma)[E/x']} \text{ (St2)}$$

$$\frac{x' \notin \mathsf{Vars}'(\Pi, \Sigma)}{\Pi \mid \Sigma * P(x', E) \rightsquigarrow \Pi \mid \Sigma \cup \mathsf{junk}} \text{ (Gb1)} \qquad \frac{x', y' \notin \mathsf{Vars}'(\Pi, \Sigma)}{\Pi \mid \Sigma * P_1(x', y') * P_2(y', x') \rightsquigarrow \Pi \mid \Sigma \cup \mathsf{junk}} \text{ (Gb2)}$$

$$\frac{x' \notin \mathsf{Vars}'(\Pi, \Sigma, E, F) \qquad \Pi \vdash F = \mathsf{nil}}{\Pi \mid \Sigma * P_1(E, x') * P_2(x', F) \rightsquigarrow \Pi \mid \Sigma * \mathsf{ls}(E, \mathsf{nil})} \text{ (Abs1)}$$

$$\frac{x' \notin \mathsf{Vars}'(\Pi, \Sigma, E, F, G, H) \qquad \Pi \vdash F = G}{\Pi \mid \Sigma * P_1(E, x') * P_2(x', F) * P_3(G, H) \rightsquigarrow \Pi \mid \Sigma * \mathsf{ls}(E, F) * P_3(G, H)} \text{ (Abs2)}$$

Figure 5.1: *Table from the article*

The laws (Gb1) and (Gb2) lose precision because they remove information about some variables which correspond to unreachable nodes (garbage). The laws (Abs1) and (Abs2) make abstraction by ignoring facts that depend on a midpoint in a list segment, unless it is named by a program variable.

The rules are applied to canonicalise post-condition assertions, to ensure termination of an analysis.

- The tool uses auxiliary variables and must check equalities, in particular when it uses the rearrangement rule: $\frac{\Pi_0, \Sigma_0 * P(E,G)}{\Pi_0, \Sigma_0 * P(F,G)} \Pi_0 \models E = F$, which says that if the pure part of a formula Π_0 implies an expression equality $E = F$, then the formula implies the formula where in the spatial part $*P(E,G)$ is replaced by $*P(F,G)$.

Example: In this reverse-list example, given a linked list as a precondition, the analysis calculates that the postcondition might be a linked list or a single points-to fact, and the analysis calculates also the loop invariant. Note that the assertions are disjunctions of pairs of form (Π, Σ):

Program:	$p := \texttt{nil}; \texttt{while } (c \neq \texttt{nil}) \texttt{ do } (n := c \to tl; c \to tl := p; p := c; c := n)$
Pre:	$\{\}, \{\textbf{ls}(c, \texttt{nil})\}$
Post:	$\{c = \texttt{nil} \wedge c = n \wedge n = \texttt{nil}\}, \{\textbf{ls}(p, \texttt{nil})\}$
	$\vee \{c = \texttt{nil} \wedge c = n \wedge n = \texttt{nil}\}, \{p \mapsto \texttt{nil}\}$
Inv:	$\{p = \texttt{nil}\}, \{\textbf{ls}(c, \texttt{nil})\}$
	$\vee \{c = n \wedge n = \texttt{nil}\}, \{p \mapsto \texttt{nil}\}$
	$\vee \{c = n \wedge n = \texttt{nil}\}, \{\textbf{ls}(p, \texttt{nil})\}$
	$\vee \{c = n\}, \{p \mapsto \texttt{nil} * \textbf{ls}(n, \texttt{nil})\}$
	$\vee \{c = n\}, \{\textbf{ls}(p, \texttt{nil}) * \textbf{ls}(n, \texttt{nil})\}$

Advantages/drawbacks

The major deficiency of space invader is the need to check equalities among auxiliary variables, but the article does not give details on performing that task. We found it interesting that in our domain, auxiliary variables have a precise semantics and also that we do not have to check equalities on variables. Another drawback of the approximation used in space invader is that, for each new data structure, it must employ a new analysis and rules to perform analysis for loops. Also, the analysis does not allow other abstractions than the ones for predicates or for considering part of the heap as junk.

Relationship to our work

The goal in both cases is to do shape analysis. Beyond this common goal, it is hard to compare our work to space invader, since space invader is primarily a "list-shape" detector - only lists can be analysed, no numerical, and for other data structures, there must be extra predicates defined along with some rules to perform abstraction (folding/unfoldings for examples) and canonicalization to analyse loops. Still, the idea of using a subset of separation logic is good.

The use of predicates for the data structures that we know are used in the programs is a good one. But our approach was to use an general approach for abstraction (with summary nodes, like in TVLA, for example), and then plug into the analysis some "sugared structures" and a system of folding/unfolding.

5.1.3 Shape analysis for composite data structures (CAV'07)

The third tool in the series from the East London Massive analyzes programs with doubly-linked lists.

Applications of the tool

The proposed approach gives a shape analysis for doubly-linked lists but here, **some predicates** for different data structures **can be found** by the analysis.

Characteristics/techniques

- The analysis uses symbolic execution with focus and canonicalization steps (like TVLA).

- The analysis uses a "generic higher-order inductive predicate" ($\mathbf{ls}\ \Lambda$) which is specialized during canonicalization. $\mathbf{ls}\ \Lambda\ (x, y, z, w)$ describes a (possibly empty, cyclic or doubly-) linked list segment where each node in the segment itself is a data structure described by Λ.

  The syntax of assertions goes as follow:

$$E := x \mid x' \mid \texttt{nil}$$
$$\Pi ::= \texttt{true} \mid E = E \mid E \neq E \mid \Pi \wedge \Pi$$
$$\Sigma ::= \texttt{emp} \mid \texttt{true} \mid \Sigma * \Sigma \mid E \mapsto T(\vec{f} : \vec{E}) \mid \textbf{ls} \; \Lambda \; (E, E, E, E)$$
$$P ::= \Pi \wedge \Sigma$$
$$\Lambda ::= \lambda[x', x', x', \vec{x'}] \cdot P$$

Unfortunately, the article does not provide a concrete semantics of the language and in particular of **ls** and λ, which makes the approach difficult to understand.

- The analysis depends heavily on a theorem prover.

- The analysis uses heuristics to find predicates because the authors state that the usual predicates are not enough for real programs.

Because the paper is a proposal and it does not document an implemented tool, there are no simple examples that we can present that illustrate the synthesis of instances of the doubly-linked list predicate, **ls** Λ.

Advantages/drawbacks

Creating predicates seems to be a better approach than asking the user to define all the predicates.

But the current approach has one big disadvantage: the **ls** Λ (x, y, z, w) predicate is complex (and unfortunately, the article does not provide the formal semantics of that predicate) as well as the predicate discovery algorithm, which requires a theorem prover. It would seem that the approach is not likely to work well with large programs unless a user gives annotations.

Relationship to our work

In our work, we do not have special predicates (or abstract-data structures like lists) that would be treated directly by the analysis, but we have summary nodes for abstraction.

With our abstract domain, we could not express doubly-linked lists which are not cyclic, because our separation system of $*1$ and $*2$ does not allow one to say there is no cycle through paths of "1" and not through paths of "2" while there could be through mixed paths.

A good improvement in our domain, would be to allow labels for noncycling. This would require adding[1]

- a set of labels, *Labels*

- $Fields := \{1, 2\}$ (we could also in some version express more field possibilities)

  $VD := VD1 \mid Dangling\_Loc \mid Loc(\mathcal{P}(Fields \times Labels) \times VD1 \times VD1)$

  (This is used for the syntax of the graph part of our abstract domain, here we replaced the actual $\mathcal{P}(\{*1, *2\})$ by $\mathcal{P}(Fields \times Labels)$.)

- $R \triangleq Labels \rightharpoonup Loc \rightharpoonup \mathcal{P}(Loc)$

  (This domain is used in the concrete semantics of our abstract domain to record the sets of reachable locations from each location.)

- $[\![Loc(A, vd1, vd2), (h, f, r)]\!]^8 \triangleq$
  $$\left\{ l \in dom(h) \;\middle|\; \begin{array}{l} \bullet \quad \Pi_1(h(l)) \in [\![vd1, (h, f, r)]\!]^8 \\ \bullet \quad \Pi_2(h(l)) \in [\![vd2, (h, f, r)]\!]^8 \\ \bullet \quad (f, i) \in A \wedge \Pi_f(h(l)) \in Loc \Rightarrow \Pi_f(h(l)) \in (r(i))(l) \end{array} \right\}$$
  (It is in this rule that the semantics of $*i$ were taken into account in the last bullet, now we record that a location is reachable by another but for a given label.)

- $sem* \triangleq$
  $$\left\{ s, h, f, r \;\middle|\; \forall i \in Labels. \; \forall l \in dom(r(i)). \begin{array}{l} \bullet \quad l \notin (r(i))(l) \\ \bullet \quad \forall l' \in (r(i))(l) \cap dom(r(i)). \\ \qquad (r(i))(l') \subseteq (r(i))(l) \end{array} \right\}$$
  (This part of the semantics of our abstract domain which was insuring that there was no cycle and that reachability is transitive. The new semantics is also the same but it specifies that those two constraints should be satisfied for any label.)

---

[1] if you have a colored version, the changes are written in blue

Then we could express uncyclic doubly-linked lists.

5.1.4 Footprint Analysis: A Shape Analysis that Discovers Preconditions (SAS'07)

The final paper we examine concentrates on precondition discovery.

Applications of the tool

The tool is made for **finding a procedure's safety preconditions** (they call this "footprint analysis") in separation logic by a forward analysis. The basic idea is: assume the heap is "empty"; perform a forwards shape analysis on the procedure; each dereference to the heap adds a cell to the "empty" heap, building a "footprint" that will be necessary for sound execution of the procedure.

Characteristics/techniques

- Footprint analysis uses the same subset of separation logic, which splits the pure and spatial parts of the formula into a Π-part and a Σ-part.

  The syntax of assertions goes as follow:

$$E := x \mid x' \mid \texttt{nil} \mid ...$$
$$\Pi ::= \texttt{true} \mid E = E \wedge \Pi \mid E \neq E \wedge \Pi$$
$$\Sigma ::= \texttt{true} \mid \texttt{emp} \mid E \mapsto E \mid \Sigma * \Sigma \mid \textbf{lseg}(E, E)$$
$$H ::= \Pi \wedge \Sigma$$

  As before, $\textbf{lseg}(E, E')$ denotes a linear list from address E to E'.

- They do not have assertions with fixpoints.

- They use a theorem prover for proving entailments between symbolic heaps.

The work has a peculiar aspect: it is an imitation of a backward analysis by going forwards, so they find a set of safety preconditions but they do an overapproximation analysis. This means the preconditions are not necessarily safe ! Thus they need to use another

forward shape analysis to discover post-conditions and check for errors that might arise if the calculated precondition is unsound.

Here is their explaination (in [23] page 2):

> "We say ideally here because there is a complication. In order to stop the footprint assertion from growing forever it is periodically abstracted. The abstraction we use is an overapproximation and, usually in shape analysis, this leads to incompleteness while maintaining soundness. But, abstracting the footprint assertion is tantamount to weakening a precondition, and so for us is a potentially unsound step. As a result, we also use a post-analysis phase, where we run a standard forwards shape analysis to filter out the unsafe preconditions that have been discovered. For each of the safe preconditions, we also generate a corresponding postcondition."

Here is an example, taken from their paper:

```
1: while (c!=0) {
2: t=c;
3: c=c->tl;
4: free(t);
5: }
```

Discovered Precondition: $c == c \land \mathbf{lseg}(c, 0)$

Fig. 1. A program to delete a list and the discovered precondition when run in the start state, $(c == c \land \mathtt{emp}, \mathtt{emp})$.

Their analysis returns what is in Fig. 5.2; again, the table is taken from their paper. Within the table, it is hard to understand what is the "Current Heap", "Footprint Heap" and their relationship. My understanding is that the "Current Heap" corresponds to the post-conditions, and the "Footprint Heap" corresponds to the pre-conditions. When in the

	Current Heap	Footprint Heap
First iteration		
pre:	c!=0 $\wedge$ c==c_ $\wedge$ t==c_ $\wedge$ emp	emp
post:	t!=0 $\wedge$ c==c1_ $\wedge$ t==c_ $\wedge$ c_ $\mapsto$ c1_	c_ $\mapsto$ c1_
Second Iteration		
pre:	c!=0 $\wedge$ c==c1_ $\wedge$ t==c1_ $\wedge$ emp	c_ $\mapsto$ c1_
post:	t!=0 $\wedge$ c==c2_ $\wedge$ t==c1_ $\wedge$ c1_ $\mapsto$ c2_	c_ $\mapsto$ c1_ * c1_ $\mapsto$ c2_
abs post:	t!=0 $\wedge$ c==c2_ $\wedge$ t==c1_ $\wedge$ c1_ $\mapsto$ c2_	lseg(c_,c2_)
Third Iteration		
pre:	c!=0 $\wedge$ c==c2_ $\wedge$ t==c2_ $\wedge$ emp	lseg(c_,c2_)
post:	t!=0 $\wedge$ c==c3_ $\wedge$ t==c2_ $\wedge$ c2_ $\mapsto$ c3_	lseg(c_,c2_) * c2_ $\mapsto$ c3_
abs post:	t!=0 $\wedge$ c==c3_ $\wedge$ t==c2_ $\wedge$ c2_ $\mapsto$ c3_	lseg(c_,c3_)

Fig. 2. Pre and Post States at line 3 during footprint analysis of `delete_list`

Figure 5.2: *Return of example for SAS'07*

figure it is said "Pre" and "Post", I interpret that not as a pre-condition and post-condition but as "Current Heap + Pre = post-condition between line 2 and 3", "Current Heap + Post = post-condition between line 3 and 4", "Footprint Heap + Pre = pre-condition between line 2 and 3" and "Footprint Heap + Post = pre-condition between line 3 and 4".

So "Footprint Heap" is an attempt to compute a pre-condition in a forward way which leads to unsoundness which needs to be checked afterward.

Advantages/drawbacks

Again, using a subset of the logic seems a good idea. On the positive side, the approach appears to be useful for modular analysis. On the negative side, the generated preconditions are not expressive enough for realistic applications.

Relationship to our work

This can not be compared to our work since the goals are not common. On the other hand, the proposed analysis technique can combine to any other forward analysis based on separation logic, so that's where our work could be useful and why they cite us in their paper.

5.1.5 Conclusions about smallfoot/space invader

The use of a subset of separation logic, which splits the pure assertion part from the separation part, is a recent, common approach. I believe it is quite a good approach but it requires more work during analysis. One important thing that could be noticed is that the subsets do not use the connective $\rightarrow\!\!*$, which is indeed a problem for us in our translation. I believe the work that we could have done in transforming a formula with $\rightarrow\!\!*$ into one free of that connective, corresponds to the work done in their analysis for staying free of that connective.

A drawback of their analyses, is that they use theorem provers. It is not clear that the theorem provers always terminate, and the work sometimes requires the use of heuristics. But in counterpart, we use heuristics for choosing the way of doing summary nodes, which affects the precision of our analysis.

Again, our primary goal was to use separation logic as an intermediate language from different shape analysis, and it seems that this could be usuful for example for combining it with the work presented in the SAS'07 paper.

5.2 TVLA

In this section, we give some comparisons with the work from TVLA ("Three-valued logical analyser") which can be seen in "Parametric Shape Analysis via 3-Valued Logic", POPL'99[4] and "TVLA: A System for Implementing Static Analyses", SAS'00[24].

Applications of the tool

The tool is a framework for shape analysis. It is made to find invariants of the shape of the storage heap used by a program.

Characteristics/techniques

The domain of the analysis, is based on a 3-Valued logic, that is a formula is either true (1), false (0), or "we don't know" (1/2).

The domain has some abstract elements (can be seen as nodes of a shape graph), some predicates, and it associates for each predicate its value (0, 1, 1/2) for any abstract element (or tuple of elements).

The predicates are either "core-predicates" like in Fig. 5.3 where sm corresponds to

Predicate	Intended Meaning
$x(v)$	Does pointer variable **x** point to element v?
$sm(v)$	Does element v represent more than one concrete element?
$n(v_1, v_2)$	Does the **n** field of v_1 point to v_2?

Table 1: The core predicates that correspond to the `List` data-type declaration from Figure 1(a).

Figure 5.3: *TVLA's core predicates*

summary nodes. Or the predicates can be "instrumentation predicates" that can be defined so to improve the analysis, for example, they could be the ones in Fig. 5.4 .

The instrumentation predicates are formulae which syntax is given in Fig. 5.5 and their semantics are given by Fig. 5.6 which shows the 3-valued interpretation of the connectives.

In Fig. 5.7 is a program for list reversal and an example of the analysis for the core predicates for this program is in Fig. 5.8. The program uses three variables, x, y and n. The core predicates are $x(v)$, $z(v)$, $n(v)$, and $sm(v)$. Fig. 5.8 shows how each statement, **st**, updates the 3-valued interpretation of each core predicate.

Abstraction in TVLA comes from the 3-valued semantics but also from the core predicate sm which says if an abstract element can represent several concrete elements or not. Those are the summary nodes. (This is what inspired us to talk also of summary nodes.)

The user has to define the instrumentation predicates that describe shape properties of

Pred.	Intended Meaning	Purpose
$is(v)$	Do two or more fields of heap elements point to v?	lists and trees
$r_x(v)$	Is v (transitively) reachable from pointer variable **x**?	separating disjoint data structures
$r(v)$	Is v reachable from some pointer variable (i.e., is v a non-garbage element)?	compile-time garbage collection
$c(v)$	Is v on a directed cycle?	ref. counting
$c_{f.b}(v)$	Does a field-**f** dereference from v, followed by a field-**b** dereference, yield v?	doubly-linked lists
$c_{b.f}(v)$	Does a field-**b** dereference from v, followed by a field-**f** dereference, yield v?	doubly-linked lists

Table 2: Examples of instrumentation predicates

Figure 5.4: *some TVLA's instrumentation predicates*

Definition 3.2 *A* **formula** *over a* **vocabulary** $\mathcal{P} = \{p_1, \ldots, p_n\}$ *is defined inductively, as follows:*

Atomic Formulae *The* **logical-literals 0**, **1**, *and* **1/2** *are atomic formulae with no free variables.*

For every predicate symbol $p \in \mathcal{P}$ of arity k, $p(v_1, \ldots, v_k)$ is an atomic formula with free variables $v_1, \ldots, v_k$.

The formula $(v_1 = v_2)$ is an atomic formula with free variables v_1 and v_2.

Logical Connectives *If φ_1 and φ_2 are formulae whose sets of free variables are V_1 and V_2, respectively, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, and $(\neg\varphi_1)$ are formulae with free variables $V_1 \cup V_2$, $V_1 \cup V_2$, and V_1, respectively.*

Quantifiers *If φ is a formula with free variables $v_1, v_2, \ldots, v_k$, then $(\exists v_1 : \varphi)$ and $(\forall v_1 : \varphi)$ are both formulae with free variables $v_2, v_3, \ldots, v_k$.*

Transitive Closure *If φ is a formula with free variables V such that $v_1, v_2 \in V$ and $v_3, v_4 \notin V$, then $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$ is a formula with free variables $(V - \{v_1, v_2\}) \cup \{v_3, v_4\}$.*

Figure 5.5: *TVLA's formula syntax*

230

Atomic *For a logical-literal* $l \in \{0, 1, 1/2\}$, $[\![l]\!]_3^S(Z) = l$
(where $l \in \{0, 1, 1/2\}$*).*

For an atomic formula $p(v_1, \dots, v_k)$,

$$[\![p(v_1, \dots, v_k)]\!]_3^S(Z) = \iota^S(p)(Z(v_1), \dots, Z(v_k))$$

For an atomic formula $(v_1 = v_2)$,

$$[\![v_1 = v_2]\!]_3^S(Z) = \begin{cases} 0 & Z(v_1) \neq Z(v_2) \\ 1 & \begin{array}{l} Z(v_1) = Z(v_2) \\ \land \ \iota^S(sm)(Z(v_1)) = 0 \end{array} \\ 1/2 & \text{otherwise} \end{cases}$$

Logical Connectives *For logical formulae* φ_1 *and* φ_2

$$[\![\varphi_1 \land \varphi_2]\!]_3^S(Z) = \min([\![\varphi_1]\!]_3^S(Z), [\![\varphi_2]\!]_3^S(Z))$$
$$[\![\varphi_1 \lor \varphi_2]\!]_3^S(Z) = \max([\![\varphi_1]\!]_3^S(Z), [\![\varphi_2]\!]_3^S(Z))$$
$$[\![\neg\varphi_1]\!]_3^S(Z) = 1 - [\![\varphi_1]\!]_3^S(Z)$$

Quantifiers *If* φ *is a logical formula,*

$$[\![\forall v_1 : \varphi]\!]_3^S(Z) = \min_{u \in U^S} [\![\varphi_1]\!]_3^S(Z[v_1 \mapsto u])$$
$$[\![\exists v_1 : \varphi]\!]_3^S(Z) = \max_{u \in U^S} [\![\varphi_1]\!]_3^S(Z[v_1 \mapsto u])$$

Transitive Closure *For* $(TC \ v_1, v_2 : \varphi)(v_3, v_4)$,

$$[\![(TC \ v_1, v_2 : \varphi)(v_3, v_4)]\!]_3^S(Z) =$$
$$\max_{\substack{u_1, \dots, u_n \in U, \\ Z(v_3) = u_1, Z(v_4) = u_n}} \min_{i=1}^{n-1} [\![\varphi]\!]_3^S(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

Figure 5.6: *TVLA's formula semantics*

the memory and provide the transfer function which describes how the command transforms the value of those predicates. These must be defined in the same style as seen in Fig. 5.8.

In the case of loops, the user must choose which predicates are relevant or not, then if two abstract elements have the same value for all relevant predicates, they are merged to a summary node.

You can see the example with a graphical representation in Fig. 5.9. In fact, the domain used in TVLA is not "one graph" but a set of graphs. To improve precision, the analysis can apply a function called "focus" such that if a predicate has a value "1/2" for some

```
                          /* reverse.c */
                          #include ''list.h''
                          List reverse(List x) {
                            List y, t;
                            assert(acyclic_list(x));
/* list.h */              y = NULL;
typedef struct node {     while (x != NULL) {
  struct node *n;           t = y;
  int data;                 y = x;
} *List;                    x = x->n;
                            y->n = t;
                          }
                          return y;
                        }
        (a)                          (b)
```

Figure 1: (a) Declaration of a linked-list data type in C. (b) A C function that uses destructive updating to reverse the list pointed to by parameter x.

Figure 5.7: *TVLA's list-reversal program*

abstract element, it will be split into several structures for each case that correspond. (In some works this is called "specialization".) In particular, a summary node splits into several nodes. The analysis also has a "coerce" algorithm which sharpens the structures using some compatibility constraints.

Advantages/drawbacks

The main advantage to TVLA is that the user can create new instrumentation predicates, write the transfer functions, choose which predicates are relevant, and then use the already implemented tool.

The drawbacks are

- the analysis is not compositional

- the user must know in advance the kind of properties he wants to find, and he should know the type of data structures involved in the program

st	φ_p^{st}
`x = NULL`	$\varphi_x^{st}(v) \overset{\text{def}}{=} \mathbf{0}$ $\varphi_z^{st}(v) \overset{\text{def}}{=} z(v)$, for each $z \in (PVar - \{x\})$ $\varphi_n^{st}(v_1, v_2) \overset{\text{def}}{=} n(v_1, v_2)$ $\varphi_{sm}^{st}(v) \overset{\text{def}}{=} sm(v)$
`x = t`	$\varphi_x^{st}(v) \overset{\text{def}}{=} t(v)$ $\varphi_z^{st}(v) \overset{\text{def}}{=} z(v)$, for each $z \in (PVar - \{x\})$ $\varphi_n^{st}(v_1, v_2) \overset{\text{def}}{=} n(v_1, v_2)$ $\varphi_{sm}^{st}(v) \overset{\text{def}}{=} sm(v)$
`x = t->n`	$\varphi_x^{st}(v) \overset{\text{def}}{=} \exists v_1 : t(v_1) \wedge n(v_1, v)$ $\varphi_z^{st}(v) \overset{\text{def}}{=} z(v)$, for each $z \in (PVar - \{x\})$ $\varphi_n^{st}(v_1, v_2) \overset{\text{def}}{=} n(v_1, v_2)$ $\varphi_{sm}^{st}(v) \overset{\text{def}}{=} sm(v)$
`x->n = t`	$\varphi_z^{st}(v) \overset{\text{def}}{=} z(v)$, for each $z \in PVar$ $\varphi_n^{st}(v_1, v_2) \overset{\text{def}}{=} \begin{array}{l}(n(v_1, v_2) \wedge \neg x(v_1)) \\ \vee\ (x(v_1) \wedge t(v_2))\end{array}$ $\varphi_{sm}^{st}(v) \overset{\text{def}}{=} sm(v)$
`x = malloc()`	$\varphi_x^{st}(v) \overset{\text{def}}{=} new(v)$ $\varphi_z^{st}(v) \overset{\text{def}}{=} \begin{array}{l}z(v) \wedge \neg new(v), \\ \text{for each } z \in (PVar - \{x\})\end{array}$ $\varphi_n^{st}(v_1, v_2) \overset{\text{def}}{=} \begin{array}{l}n(v_1, v_2) \\ \wedge\ \neg new(v_1) \wedge \neg new(v_2)\end{array}$ $\varphi_{sm}^{st}(v) \overset{\text{def}}{=} sm(v) \wedge \neg new(v)$

Table 5: Predicate-update formulae for the core predicates for List and reverse.

Figure 5.8: *TVLA's list-reversal result of analysis*

- there is no extra assistance for analyzing loops; the user just chooses the relevant predicates

- the tool does not deal with numerical information

S	Structure	Graphical Representation
S_0	**unary predicates:** **indiv.** x y t sm is **binary predicates:** **n**	
S_1	**unary predicates:** **indiv.** x y t sm is u_1 1 0 0 0 0 **binary predicates:** **n** u_1 u_1 0	$x \longrightarrow (u_1)$
S_2	**unary predicates:** **indiv.** x y t sm is u_1 1 0 0 0 0 u 0 0 0 1/2 0 **binary predicates:** **n** u_1 u u_1 0 1/2 u 0 1/2	$x \longrightarrow (u_1) \cdots\!\!\xrightarrow{n}\!\!\cdots ((u)) \circlearrowright n$

Figure 2: The three-valued logical structures that describe all possible acyclic inputs to **reverse**.

Figure 5.9: *TVLA's example*

Relationship to our work

Both TVLA and our tool are for making shape analysis. Our work was designed primarily to make an interface between separation logic and other domains for shape analysis, and in particular with TVLA. So, the basic shape graphs of TVLA are quite similar to ours.

When we have a single arrow from an element in our graph, we can see it as a "1" in the 3-valued domain, while when we do not have an arrow from an element, we can see it as a "0", then when we have two arrows from an element, we can be see them as "1/2" but in addition, we know that at least one exists. We would think that in TVLA, to obtain similar information, we would need to add a specific predicate which would be an $\lor$.

I can not say if adding more complicated structures to our domain, like some sugared syntax for lists and then some system for treating them in the stabilization differently (like

234

for example with folding or unfolding), would be more or less work than adding a new predicate for TVLA and writting the transfer functions.

It might be interesting to perform some kind of TVLA-based analysis where the original logic would have separation connectives (or just $E \mapsto F$, $\mathbf{ls}(E, F)$ and $\mathbf{junk}$).

5.3   others

- Rinard in "Compositional Pointer and Escape Analysis for Java Programs"[25] and "Incrementalized Pointer and Escape Analysis"[26] build precondition graphs like the one in footprint analysis.

- Yang & al. in "Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis"[17] use grammars to add recursivity to separation logic.

- Rival & al. in "Shape Analysis with Structural Invariant Checkers"[27] are using inductive predicates with a separation connective and with a system of folding/unfolding.

- Magill & al. in "Inferring Invariants in Separation Logic for Imperative List-processing Programs"[28] are doing similar work as smallfoot but they deal with numericals.

- There were attempts to do pointer analysis (alias analysis) using languages from Deutsch[29] or more recently "Storeless Semantics and Alias Logic"[30].

- Salcianu, Andersen, Steensgaard, Heintze, Tzolovski, Foster Aiken, Ryder Landi, Emilianov, Jonkers, MOller, Reddy,...

5.4   Conclusion

We finish this chapter with some perspectives on key aspects of the tools we have surveyed.

5.4.1 Modularity

A major attraction of separation logic is its support of modular reasoning — a property about a command can be proved with only that part of the heap used by the command, and the result is preserved when the command is used with a heap larger than the one used in the proof. This concept is supported by the *frame rule* seen in Fig. 5.10

$$\frac{\{\phi_1\}C\{\phi_2\}}{\{\phi_1 * \phi'\}C\{\phi_2 * \phi'\}} \text{ where } \phi'\text{'s variables are not modified by } C$$

Figure 5.10: *Frame rule*

The frame rule supports modular reasoning in that, when distinct properties are proved of commands that use disjoint parts of the heap, the commands and their properties can be composed without interference.

When we examine each of the tools that implement subsets of separation logic, we see that the frame rule is used implicitly by the theorem provers. It is usually done by pattern matching, but the details of the theorem provers are not clear. Also, doing pattern matching implies the need for a subset of the logic which helps the use of heuristics.

Other systems, like TVLA, work with a global heap and do not employ modular reasoning in any explicit way. For example, "when one updates a single abstract heap cell this may require also the updating of properties assiociated with all other cells. Furthermore, each update of another cell might itself depend on the whole heap." [21].

In our own work, we do not concern ourselves with an explicit or implicit representation of the frame rule because we do not work with inference-rules for high-level commands.

5.4.2 Expressibility of heap logic

Each of the systems presented in this chapter uses its own subset of separation logic (or, in TVLA's case, a subset of three-valued predicate logic selected by the user). Some subsets are chosen because the formulae have a convenient presentation or allow useful heuristics in the system's implementation. In particular, if you separate the boolean conditions from the

separation information, applying certains rules are easier, in particular, for pattern-matching in the theorem provers.

A disadvantage of using a subset logic is loss of expressibility. For example, none of the subsets of separation logic implemented in the tools from O'Hearn's group can express weakest liberal preconditions or strongest postconditions of commands in a while-loop language. Or, the subsets selected are so limited in expressibility that when a tool calculates a postcondition, the postcondition formula falls outside the subset, and an information losing abstraction is needed to bring the formula back into the subset logic. Theorem provers are typically used to do this.

5.4.3 Folding/unfolding

All of the systems use unfolding and folding to operate on recursively defined data types. Since linear-list and tree types are recursively defined, this means the interpretation of recursion is crucial to the correctness of the tool.

Many times this interpretation is hidden within the semantics of unfold/fold, which can cause confusion.

Within our system, the semantics of recursive formulae (types) is precisely defined, and there are precise, sound laws for manipulating the formulae. It would be interesting to extend our system with "built-in" instances (like list, tree, etc.) that employ unfolding and folding while computing the accelerations. This extension would be precise and sound because it is built on top of a precise semantics.

The drawback of this system is that it asks the user to already know the kind of data structures used in the program, and it might be nicer to have some general "heuristics" used for acceleration instead of asking someone to know the data structure (that is, finding some "general pattern" of folding/unfolding which is common to lists and trees instead of having to define those data structures.

The work on grammars in [17] for example is in some way an attempt to generalise this.

5.4.4 Theorem provers

Most of the systems described in this section rely crucially on theorem provers to solve subgoals and do simplification. The theorem prover is typically treated as a "black box," and the success of the overall system in calculating a result can depend on the power of the theorem prover connected to it. In some cases, this affects the utility of the tool.

Related to this is the choice of abstraction of the formulae; the abstraction can be stated "manually", as in TVLA, where the user selects the primitive predicates used within the three-valued predicate logic, or automatically, as in the systems that define a subset logic, where the abstraction is "built in" through the recursive predicates.

The major problem of using theorem provers is that they do not insure termination.

5.4.5 Auxiliary variables

A standard difficulty with analysis tools, like the ones presented in this chapter, is the treatment of auxiliary variables.

In most published papers, the auxiliary variables (like ours) are not presented in depth in the paper. But they must exist in some way when variable equality is checked (for example, in the rearrangement rules in[20]) or when entailments are proved between symbolic heaps (for example, in[23]). The matter of the semantics of auxiliary variables is often postponed to an "implementation problem".

In contrast, our system has a precise semantics for auxiliary variables which allow us to prove operations that can be directly implemented.

5.4.6 Analysis versus verification

Although all the tools presented in this chapter are meant to help one verify a correctness property of a program, almost all do an *analysis* of the heap that one must use to build a correctness proof.

For smallfoot, the paper from FMCO'05 checks whether Hoare triples are correct. The

SAS'07 paper finds pre-conditions that might be unsound, and the TACAS'06 paper does shape analysis by finding overapproximations of post-conditions from a given pre-state. So, in those 3 papers, the safety of a program is not insured.

TVLA only finds shapes, but does not insure safety of the program. One might encode a safety property as an instrumentation predicate.

Our work is also not meant to prove safety of a program. If we combine the computation of the wlp or sp and the translation to our abstract domain, we have a shape analysis but usually not verification. We can sometimes say that a program has no error if we compute the safety precondition formula $(wlp(\texttt{true}, C))$ and then translate it to our abstract domain and find that the formula is $\texttt{true}$. We can find that a program has for sure an error (or does not terminate), if we compute the $sp(wlp(\texttt{true}, C), C)$ and then translate it to our abstract domain and find that the formula is $\texttt{false}$.

5.4.7 What is distinctive about our system

Our work was primarily designed to use separation logic as an interface between different analyses. That is, a function could be characterized by formulae in separation logic, and this information would be plugged into another analysis of a program which would make calls to that function. So, our abstract domain takes ideas from the existing ones (in particular, summary nodes).

The current work on separation logic often takes the path of choosing a subset of the logic, adding some specific recursive predicates (list, tree, etc.), and computing the pre- or post-conditions by making fixpoint computation with some folding/unfolding system. In contrast, we added fixpoints to the logic itself, so there is no work to find pre- or post-conditions in all cases. But the work of finding pre-/post-conditions which falls within separation logic subsets is, I believe, equivalent to the efforts we make when we compute the abstract value in our domain from a formula and we want it to be precise. We can improve the precision of a translation if we rewrite the formula. The major advantage of our system

would be that we can do some abstraction which does not depend on knowledge about the data structure involved.

Chapter 6

Conclusion

If we may summarize the main accomplishments of the thesis, these would be:

- adding fixpoints to separation logic, which provides a way to express recursive formulae, expressing preconditions for while-loops, and expressing all post-conditions, letting us prove useful properties about the extended logic

- giving a precise semantics of the abstract domain of separation formulae in terms of sets of memory

- designing the abstract language as a partially reduced product of subdomains

- giving a semantics to auxiliary variables and not leaving this as an implementation design question

- combining the domain's heap analysis with a numerical domain which could be chosen from existing ones (e.g. polyhedra, octogons)

- designing the novel tabular data structure which allows extra precision by using a graph of sets instead of sets of graphs

The primary work to continue this thesis would be to finish the prototype implementation, to continue profiling studies with standard example programs, and to experiment with strategies for building summary nodes.

The major improvement to the abstract language would be the addition of labels that mark which information is overapproximate and which is not, instead of having the entire graph and the entire answer being or not an overapproximation.

Most static analyses do predicate abstraction, or analysis that requires one to know in advance which data structures the program deals with. We believe it is a great advantage to have the approximations for translating fixpoints as we do, and building summary nodes whose strategy can be changed to gain precision; but indeed, there are many programs for which we know that they deal with lists, trees or other data structures, and it is also advantageous if the analysis can incorporate that information. So in our work, it would be nice to include the possibility of adding "sugared structures", with probably the usual rules for folding/unfolding, so that we would first try to apply those rules to keep those "sugared structures" in the result, instead of using their equivalent in our abstract language, but still enable the analysis to give information and termination without relying on the "sugared structure".

The way we allow separation information with $*1$ and $*2$ is still quite primitive since we have only one family of edges among which cycles are forbidden. We could think to add some labels to those $*1$ and $*2$ to have families of uncycling edges.

It would also be fun to design an analysis of programs directly in our abstract language. But since the pre- and post-conditions characterise the program with no abstraction, it would not directly bring something new. But as we noted earlier, with formula, we can gain precision when we distinguish several subcases of a same connective, so there might be some possibility to gain precision if we wrote the analysis directly on the program.

Sagiv's TVLA project is quite fashionable at this point in time, even though it does not scale well. One project, which might just be exploring a dead-end but could be fun, would be to add separation connectives to the current logic used by TVLA.

If we try to be objective and optimistic, we would like to think that the way we built the semantics of the abstraction language and in particular the semantics for the auxiliary

242

variables (which was quite complicated to match all the properties we wanted, in particular in proofs of the translations) could give ideas to people who would do the same. We also believe the way we used the new tabular data structure to cope with the imprecision from not having sets of graphs might be useful in other forms of program analysis.

Bibliography

[1] P. Cousot., *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.*, Université scientifique et médicale de Grenoble, Grenoble, France.

[2] P. Cousot and R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977, ACM Press, New York, NY.

[3] P. Cousot and R. Cousot, Systematic design of program analysis frameworks, in *POPL'79*, pages 269–282, San Antonio, Texas, 1979, ACM Press, New York, NY.

[4] M. Sagiv, T. Reps, and R. Wilhelm., Parametric shape analysis via 3-valued logic, in *POPL'99*, 1999.

[5] P. Lam, V. Kuncak, and M. Rinard, Generalized typestate checking for data structure consistency, in *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[6] J. C. Reynolds, Separation logic : A logic for shared mutable data structures, in *LICS'02*, pages 55–74, Denmark, 2002, IEEE Computer Society.

[7] C. A. R. Hoare, Comm. ACM **12**, 576 (1969).

[8] E. W. Dijkstra, *A Discipline of Programming*, Prentice Halll, Englewood Cliffs, NJ, 1976.

[9] J. C. Reynolds, Syntactic control of interference, in *POPL'78*, pages 39–46, ACM Press, New York, NY, 1978.

[10] S. Ishtiaq and P. O'Hearn, BI as an assertion language for mutable data structures, in *POPL'01*, pages 14–26, 2001.

[11] H. Yang and P. O'Hearn, A semantic basis for local reasoning, in *FoSSaCS'02*, Lecture Notes in Computer Science, pages 402–416, Springer, 2002.

[12] E. Moggi, Information and Computation **93**, 55 (1991).

[13] É.-J. Sims, Extending separation logic with fixpoints and postponed substitution., in *AMAST*, edited by C. Rattray, S. Maharaj, and C. Shankland, volume 3116 of *Lecture Notes in Computer Science*, pages 475–490, Springer, 2004.

[14] É.-J. Sims, Theoretical Computer Science **351**, 258 (2006).

[15] J. W. de Bakker, *Mathematical Theory of Program Correctness*, Prentice Hall, Englewood Cliffs, NJ, 1980.

[16] H. Y. P. O'Hearn and J. Reynolds, Syntactic control of interference, in *POPL'04*, Italy, 2004, ACM Press, New York, NY.

[17] H. Y. O. Lee and K. Yi, Automatic verification of pointer programs using grammar-based shape analysis, in *ESOP'05*, Edinburgh, 2005.

[18] L. B. Bodil Biering and N. Torp-Smith, Bi hyperdoctrines and separation logic, in *ESOP'05*, Edinburgh, 2005.

[19] A. Miné, A new numerical abstract domain based on difference-bound matrices, in *PADO II*, volume 2053 of *LNCS*, pages 155–172, Springer-Verlag, 2001.

[20] C. C. J Berdine and P. O'Hearn, Smallfoot: Modular automatic assertion checking with separation logic, in *FMCO'05*, 2005.

[21] P. O. D Distefano and H. Yang, A local shape analysis based on separation logic, in *TACAS'06*, 2006.

[22] B. C. . a. J Berdine, C. Calcagno, Shape analysis for composite data structures, in *CAV'07*, 2007.

[23] P. O. C Calcagno, D Distefano and H. Yang, Footprint analysis: A shape analysis that discovers preconditions, in *SAS'07*, 2007.

[24] T. Lev-Ami and M. Sagiv, Tvla: A system for implementing static analyses, in *SAS'00*, 2000.

[25] J. Whaley and M. Rinard, Compositional pointer and escape analysis for java programs, in *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, CO, 1999.

[26] F. Vivien and M. Rinard, Incrementalized pointer and escape analysis, in *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, Utah, 2001.

[27] X. R. B. Chang and G. Necula, Shape analysis with structural invariant checkers, in *SAS'07*, 2007.

[28] E. C. S. Magill, A. Nanevski and P. Lee, Inferring invariants in separation logic for imperative list-processing programs, in *SPACE'06*, 2006.

[29] A. Deutsch, A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations, in *Conference on Computer Languages*, 1992.

[30] R. I. M. Bozga and Y. Lakhnech, Storeless semantics and alias logic, in *Workshop on Partial Evaluation and Semantics Based Program Manipulation*, 2003.

Appendix A

Junk: ast algorithm

The algorithm works with 7 sets

- $\boldsymbol{g_0}$: set of pairs $(x, S) \in Var \times PVD^+$ such that $ar_0(x) = S$ and $S \neq \top$, that is
  $$s(x) \subseteq \bigcup_{vd \in S} [\![vd, (h_0, f_0, r_0)]\!]^8$$

- $\boldsymbol{G_0}$: set of pairs $(\alpha, S) \in TVar \times PVD^+$ such that $ar_0(x) = S$ and $S \neq \otimes$, that is
  $$f(\alpha) \subseteq \bigcup_{vd \in S} [\![vd, (h_0, f_0, r_0)]\!]^8$$

- $\boldsymbol{g_1}$: set of pairs $(x, S) \in Var \times PVD^+$ such that $ar_1(x) = S$ and $S \neq \top$, that is
  $$s(x) \subseteq \bigcup_{vd \in S} [\![vd, (h_1, f_1, r_1)]\!]^8$$

- $\boldsymbol{G_1}$: set of pairs $(\alpha, S) \in TVar \times PVD^+$ such that $ar_1(x) = S$ and $S \neq \otimes$, that is
  $$f(\alpha) \subseteq \bigcup_{vd \in S} [\![vd, (h_1, f_1, r_1)]\!]^8$$

- $\boldsymbol{g_{01}}$: set of pairs $(x, S) \in Var \times PVD^+$ such that $s, h, f_1 \,\dot{\cup}\, f_2, r_0 \dot{\cup} r_1 \in [\![x, S]\!]^5$

- $\boldsymbol{G_{01}}$: set of pairs $(\alpha, S) \in TVar \times PVD^+$ such that $s, h, f_1 \,\dot{\cup}\, f_2, r_0 \dot{\cup} r_1 \in [\![\alpha, S]\!]^5$

- $\boldsymbol{equal}$: set of pairs such that

  - $\forall (vd_0, vd_1) \in eq'. \; vd_0 \in TVar \Rightarrow f(vd_0) \in [\![vd_1, (s, h, f, r)]\!]^8$

  - $\forall (vd_0, vd_1) \in eq'. \; vd_1 \in TVar \Rightarrow f(vd_1) \in [\![vd_0, (s, h, f, r)]\!]^8$

(remark, for $\boldsymbol{g_0}$ and $\boldsymbol{g_1}$, S can never be $\otimes$ by domain counstraints, by default, S is $\top$ for variables in Var and $\otimes$ for those in $TVar$)

The algorithm has 3 steps:

1. we built $\boldsymbol{G_{01}}$ roughly by just pasting the two graphs $\boldsymbol{G_0}$ and $\boldsymbol{G_1}$ together

2. we build $\boldsymbol{g_{01}}$ and update the ast information coming from the combination of $\boldsymbol{g_0}$ and $\boldsymbol{g_1}$, the equalities coming from it are postponed for the next step and registed in $\boldsymbol{equal}$

3. we resolve the equalities in $\boldsymbol{equal}$

We have $ast(ar_0, ar_1)$ is such that

- $\forall v \in VAR$. if $(v, S) \in \boldsymbol{g_{01}} \cup \boldsymbol{G_{01}}$ then $ast(ar_0, ar_1)(v) = S$

- $\forall x \in Var$ if $\neg \exists S. (x, S) \in \boldsymbol{g_{01}}$ then $ast(ar_0, ar_1)(x) = \top$

- $\forall \alpha \in TVar$ if $\neg \exists S. (\alpha, S) \in \boldsymbol{G_{01}}$ then

    - if $\alpha \notin used(ar_0) \cup used(ar_1)$ then $ast(ar_0, ar_1)(\alpha) = \otimes$

    - if $\alpha \in used(ar_0) \cup used(ar_1)$ then $ast(ar_0, ar_1)(\alpha) = \top$

We will define some rules to modify the tuples of the 7 sets and we will prove that those rules keep the properties satisfied by the sets.

We will have some rule such which will answer Ω, wich means that $ast(ar_0, ar_1)$ does not exists, this means it was not possible that we had all the hypothesis at the same time.

We make the proofs for unobvious rules.

We will always have $\forall v \in VAR.\forall S_1, S_2. ((v, S_1) \in \boldsymbol{g_{01}} \cup \boldsymbol{G_{01}} \wedge (v, S_2) \in \boldsymbol{g_{01}} \cup \boldsymbol{G_{01}}) \Rightarrow (S_1 = S_2)$. We do not write the proof, it is at the begining and it is obvious that all rules conserve this property.

STEP 1 : Initialization of G_{01}

We present the case of G_0, it is the same for G_1

1. For all α such that

   - $(\alpha, S) \in G_0$

   - $Dangling\_Loc \in S$

   then update

   - G_{01} to $G_{01} \cup \{(\alpha, S \cup \{Loct\})\}$

2. For all α such that

   - $(\alpha, S) \in G_0$

   - $Dangling\_Loc \notin S$

   then update

   - G_{01} to $G_{01} \cup \{(\alpha, S\}$

Proof. **Tobeprecised** As for the cases of g_0, because of the auxiliary variables constraints we took it is ok when we have auxiliary variables. For the case with $Dangling\_Loc$, this comes because if α can be dangling in h_0, then it is for sure a location but might not be dangling in h. $\square$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

THIS PART COULD BE REWRITEN SAYING THAT WE SET

$G_{01} = \bigcup\limits_{\alpha \in dom(G_0) \cup dom(G_1)} (\alpha, basic\_ast(G_0(\alpha), G_1(\alpha))$ with extending $G_0(\alpha) = \top$ if $\alpha \in dom(G_1) \backslash$
$dom(G_0)$ (and the same for G_1).

249

STEP 2 : Elimination of g_0 and g_1, creation of g_{01}, update of G_{01}, creation of equal

We repeat the procedure until g_0 and g_1 are empty.

Let (v_0, vd_0) be $reach2(g_0, G_0, x)$ and (v_1, vd_1) be $reach2(g_1, G_1, x)$.

1. if $vd_0 = \emptyset$ or $vd_1 = \emptyset$ then Ω.

   (just remember that if x was not in the stack, it should have assigned $Oodt$, the empty set is satified by nothing)

2. If vd_0 and vd_1 are not pairs and $\forall \alpha \in TVar.\ \alpha \notin vd_0 \wedge \alpha \notin vd_1$ then

   (cases with no cycle and no summary nodes)

   (a) if $basic\_ast(vd_0, vd_1) = \Omega$ then Ω

   (b) if $basic\_ast(vd_0, vd_1) \neq \Omega$ then

      i. if $v_0 = v_1 = x$ then update

         - g_0 to $g_0 \setminus \{(x, \{vd_0\})\}$
         - g_1 to $g_1 \setminus \{(x, \{vd_1\})\}$
         - g_{01} to $g_{01} \cup \{(x, basic\_ast(vd_0, vd_1))\}$

      ii. if $v_0 = x$ and $v_1 \in TVar$ then let vd_{01} be such that $(v_1, vd_{01}) \in G_{01}$

         - g_0 to $g_0 \setminus \{(x, \{vd_0\})\}$
         - g_1 to $g_1 \setminus \{(x, \{vd_1\})\}$
         - g_{01} to $g_{01} \cup \{(x, g_1(x))\}$
         - G_{01} to $G_{01} \setminus \{v_1, vd_{01}\} \cup \{v_1, basic\_equal(basic\_ast(vd_0, vd_1), vd_{01}, equal)\}$

      iii. if $v_0 \in TVar$ and $v_1 = x$ then let vd_{01} be such that $(v_0, vd_{01}) \in G_{01}$

         - g_0 to $g_0 \setminus \{(x, \{vd_0\})\}$
         - g_1 to $g_1 \setminus \{(x, \{vd_1\})\}$
         - g_{01} to $g_{01} \cup \{(x, g_0(x))\}$

- $\bullet$ $\boldsymbol{G_{01}}$ to $\boldsymbol{G_{01}} \backslash \{v_0, vd_{01}\} \cup \{v_0, basic\_equal(basic\_ast(vd_0, vd_1), vd_{01}, \boldsymbol{equal})\}$

    iv. if $v_0, v_1 \in TVar$ then let vd_{01}^0 be such that $(v_0, vd_{01}^0) \in \boldsymbol{G_{01}}$ let vd_{01}^1 be such that $(v_1, vd_{01}^1) \in \boldsymbol{G_{01}}$

- $\bullet$ $\boldsymbol{g_0}$ to $\boldsymbol{g_0} \backslash \{(x, \{vd_0\})\}$
- $\bullet$ $\boldsymbol{g_1}$ to $\boldsymbol{g_1} \backslash \{(x, \{vd_1\})\}$
- $\bullet$ $\boldsymbol{g_{01}}$ to $\boldsymbol{g_{01}} \cup \{(x, \boldsymbol{g_0}(x))\}$
- $\bullet$ $\boldsymbol{G_{01}}$ to $\boldsymbol{G_{01}} \backslash \{v_0, vd_{01}^0\} \backslash \{v_1, vd_{01}^1\} \cup \{v_0, basic\_equal(basic\_ast(vd_0, vd_1), vd_{01}^0, \boldsymbol{equal})\}$ $\{v_1, basic\_equal(basic\_ast(vd_0, vd_1), vd_{01}^1, \boldsymbol{equal})\}$
- $\bullet$ $\boldsymbol{equal}$ to $\boldsymbol{equal} \cup \{(v_0, v_1)\}$

3. (Cases with cycles)

    (a) If $\exists \alpha, \alpha' \in TVar. \; vd_0 = (\alpha, \alpha')$ and $\exists \beta, \beta' \in TVar. \; vd_1 = (\beta, \beta')$ then an implementation would make something more precise here to keep a cycle, but we don't write it here and go to the following cases

    (b) If $\exists \alpha, \alpha' \in TVar. \; vd_0 = (\alpha, \alpha')$ then

- $\bullet$ if $(\alpha', v_0) \in \boldsymbol{G_{01}}$ then update $\boldsymbol{G_{01}}$ to $\boldsymbol{G_{01}} \backslash \{(\alpha', v_0)\}$
- $\bullet$ update $\boldsymbol{G_0}$ to $\boldsymbol{G_0} \backslash \{(\alpha', v_0)\}$

    (c) If $\exists \alpha, \alpha' \in TVar. \; vd_1 = (\alpha, \alpha')$ then

- $\bullet$ if $(\alpha', v_1) \in \boldsymbol{G_{01}}$ then update $\boldsymbol{G_{01}}$ to $\boldsymbol{G_{01}} \backslash \{(\alpha', v_1)\}$
- $\bullet$ update $\boldsymbol{G_0}$ to $\boldsymbol{G_0} \backslash \{(\alpha', v_1)\}$

4. In case of summary node, use the rule defined in Sect. 2b.

STEP 3 : elimination of $\boldsymbol{equal}$, update of $\boldsymbol{G_{01}}$

We repeat the procedure until $\boldsymbol{equal}$ is empty.

By construction $equal \subseteq \mathcal{P}(TVar \times VD1) \cup \mathcal{P}(VD1 \times TVar)$. ($equal$ is updated in STEP 2 when applying $basic\_equal$ which updates it only when encoutering two $Loc(...)$ and also in STEP 2 rule 2.b.iv).

1. $(vd, vd) \in \textbf{\textit{equal}}$ then update $\textbf{\textit{equal}}$ to $\textbf{\textit{equal}} \setminus \{(vd, vd)\}$

2. cases with one variable and one not variable

   - if

     - $(\alpha_0, vd_1) \in \textbf{\textit{equal}}$ with $\alpha_0 \in TVar$, $vd_1 \notin TVar$
     - $(\beta_0, vd_0) = reach(\textbf{\textit{G}}_{\textbf{01}}, \alpha_0)$
     - $vd_0 \notin sn$, vd_0 not a pair
     - $basic\_equal(vd_0, vd_1, equal) = (vd_{01}, eq') \neq \Omega$

     then update

     - $\textbf{\textit{equal}}$ to $eq' \setminus \{(\alpha_0, vd_1)\}$
     - $\textbf{\textit{G}}_{\textbf{01}}$ to $\textbf{\textit{G}}_{\textbf{01}} \setminus \{(\beta_0, vd_0)\} \cup \{(\beta_0, vd_{01})\}$

   - if

     - $(\alpha_0, vd_1) \in \textbf{\textit{equal}}$ with $\alpha_0 \in TVar$, $vd_1 \notin TVar$
     - $(\beta_0, vd_0) = reach(\textbf{\textit{G}}_{\textbf{01}}, \alpha_0)$
     - $vd_0 \notin sn$, vd_0 not a pair
     - $basic\_equal(vd_0, vd_1, equal) = \Omega$

     then Ω

   - if

     - $(\alpha_0, vd_1) \in \textbf{\textit{equal}}$ with $\alpha_0 \in TVar$, $vd_1 \notin TVar$
     - $(\beta_0, vd_0) = reach(\textbf{\textit{G}}_{\textbf{01}}, \alpha_0)$
     - $vd_0 \in sn$

     then use the rule defined in Sect. 2b.

   - if

     - $(\alpha_0, vd_1) \in \textbf{\textit{equal}}$ with $\alpha_0 \in TVar$, $vd_1 \notin TVar$

– $(\beta_0, (vd_0, vd_0')) = reach(\mathbf{G_{01}}, \alpha_0)$

then update

– $\mathbf{equal}$ to $\mathbf{eq'} \setminus \{(\alpha_0, vd_1)\}$

– $\mathbf{G_{01}}$ to $\mathbf{G_{01}} \setminus \{(vd_0', \beta_0)\} \cup \{(vd_0', vd_1)\}$

- if $(vd_0, \alpha_1) \in \mathbf{equal}$ with $vd_0 \notin TVar$, $\alpha_1 \in TVar$
  similar to previous cases

3. Cases with two variables, if $(\alpha_0, \alpha_1) \in \mathbf{equal}$ with $\alpha_0, \alpha_1 \in TVar$

   Let (v_0, vd_0) be $reach(\mathbf{G_{01}}, \alpha_0)$ and (v_1, vd_1) be $reach(\mathbf{G_{01}}, \alpha_1)$.

   - if $vd_0 \in sn_{01}$ or $vd_1 \in sn_{01}$ then use the rule defined in Sect. 2b.

   - if $vd_0 = (vd_0', vd_0'')$ and $vd_1 = (vd_1', vd_1'')$ then

     – $\mathbf{G_{01}}$ to $\mathbf{G_{01}} \setminus \{(vd_0'', \{v_0\}), (vd_1, \{vd_1'\})\} \cup \{(vd_0'', \{vd_1'\}), (vd_1, \{vd_0\})\}$

     – $\mathbf{equal}$ to $\mathbf{equal} \setminus \{(\alpha_0, \alpha_1)\}$

   - if $vd_0 = (vd_0', vd_0'')$ then $\mathbf{G_{01}}$ to $\mathbf{G_{01}} \setminus \{(vd_0'', \{v_0\})\} \cup \{(vd_0'', \top)\}$

   - if $vd_1 = (vd_1', vd_1'')$ then $\mathbf{G_{01}}$ to $\mathbf{G_{01}} \setminus \{(vd_1'', \{v_1\})\} \cup \{(vd_1'', \top)\}$

   - if vd_0, vd_1 are not pairs and $\notin sn_{01}$ (i.e. $\notin TVar$) then

     – if $basic\_equal(\{vd_0\}, \{vd_1\}, equal) = \Omega$ then Ω

     – if $basic\_equal(\{vd_0\}, \{vd_1\}, equal) = (vd_{01}, eq')$ then update

       * $\mathbf{G_{01}}$ to $\mathbf{G_{01}} \setminus \{(v_0, \{vd_0\}), (v_1, \{vd_1\})\} \cup \{(v_0, \{v_1\}), (v_1, vd_{01})\}$

       * $\mathbf{equal}$ to $\mathbf{eq'} \setminus \{(\alpha_0, \alpha_1)\}$

Appendix B

Intersection

Recall $ar_\top \triangleq (ad\_top, \emptyset, \texttt{full}, TVar, TVar, t\_top)$

$$
\begin{array}{rcl}
inter & : & (\boldsymbol{AR} \times \boldsymbol{AR}) \to \boldsymbol{AR}
\end{array}
$$

1. $\quad inter(ar_\top, ar_2) \triangleq ar_2$

2. $\quad inter(ar_1, ar_\top) \triangleq ar_1$

3. $\displaystyle inter\left(\begin{array}{l}(ad_1, hu_1, ho_1, sn_1, sn_1^\infty, t_1), \\ (ad_2, hu_2, ho_2, sn_2, sn_2^\infty, t_2)\end{array}\right) \triangleq inter\left(\begin{array}{l}(ad_1, \emptyset, ho_1, sn_1, sn_1^\infty, t_1), \\ (ad_2, hu_1 \cup hu_2, ho_2, sn_2, sn_2^\infty, t_2)\end{array}\right)$

4. $\displaystyle inter\left(\begin{array}{l}([ad_1 \mid v \to S], hu_1, ho_1, sn_1, sn_1^\infty, t_1), \\ ([ad_2 \mid v \to \top], hu_2, ho_2, sn_2, sn_2^\infty, t_2)\end{array}\right) \triangleq inter\left(\begin{array}{l}([ad_1 \mid v \to \top], hu_1, ho_1, sn_1, sn_1^\infty, t_1), \\ ([ad_2 \mid v \to S], hu_2, ho_2, sn_2, sn_2^\infty, t_2)\end{array}\right)$

5. $\displaystyle inter\left(\begin{array}{l}([ad_1 \mid \alpha \to \circledast], hu_1, ho_1, sn_1, sn_1^\infty, t_1), \\ (ad_2, hu_2, ho_2, sn_2, sn_2^\infty, t_2)\end{array}\right) \triangleq inter\left(\begin{array}{l}([ad_1 \mid \alpha \to \top], hu_1, ho_1, sn_1, sn_1^\infty, t_1), \\ (ad_2, hu_2, ho_2, sn_2, sn_2^\infty, t_2)\end{array}\right)$

6. $\displaystyle inter\left(\begin{array}{l}(ad_1, hu_1, ho_1, sn_1, sn_1^\infty, t_1), \\ ([ad_2 \mid \alpha \to \circledast], hu_2, ho_2, sn_2, sn_2^\infty, t_2)\end{array}\right) \triangleq inter\left(\begin{array}{l}([ad_1 \mid \alpha \to \top], hu_1, ho_1, sn_1, sn_1^\infty, t_1), \\ ([ad_2 \mid \alpha \to ad_1(\alpha)], hu_2, ho_2, sn_2, sn_2^\infty, t_2\end{array}\right.
$$

Appendix C

Intersection proof

Proposition 3.27. $\forall ar_1, ar_2 \in AR.\ [\![ar_1]\!]' \cap [\![ar_2]\!]' \subseteq [\![inter(ar_1, ar_2)]\!]'$

Prop. 3.27. We do the proof by recursion on the difference between ar_1 and $ar_\top$ (in other words we do the proof for each rule). Recall that $ar_\top \triangleq (ad_\top, \emptyset, \texttt{full}, TVar, TVar, t_\top, \top^\mathcal{D})$

We note $\sqcap$ the polymophique definition of *inter*.

We suppose that we have an total order $<$ on the variables such that $\forall \alpha, \beta \in VAR.\ \alpha < \beta \vee \beta < \alpha \vee \alpha = \beta)$ and $\exists \alpha \in VAR.\ \forall \beta \in VAR.\alpha < \beta$.

We suppose that the number of differences in ad_1 and ad_2 is finite.

1. $[\![ar_\top \sqcap ar_2]\!]' \triangleq [\![ar_2]\!]' = [\![ar_\top]\!]' \cap [\![ar_2]\!]'$

2. $[\![ar_1 \sqcap ar_\top]\!]' \triangleq [\![ar_1]\!]' = [\![ar_1]\!]' \cap [\![ar_\top]\!]'$

3.
$$
\begin{aligned}
&\quad [\![(ad_1, hu_1, ho_1, sn_1, sn_1^\infty, t_1, d_1) \sqcap (ad_2, hu_2, ho_2, sn_2, sn_2^\infty, t_2, d_2)]\!]' \\
&\triangleq [\![(ad_1 \sqcap ad_2, hu_1 \sqcap hu_2, ho_1 \sqcap ho_2, sn_1 \sqcap sn_2, sn_1^\infty \sqcap sn_2^\infty, t_1 \sqcap t_2, d_1 \sqcap d_2)]\!]' \\
&= [\![ad_1 \sqcap ad_2]\!]^4 \cap [\![hu_1 \sqcap hu_2]\!]^1 \cap [\![ho_1 \sqcap ho_2]\!]^{1'} \cap [\![sn_1 \sqcap sn_2]\!]^2 \cap [\![sn_1^\infty \sqcap sn_2^\infty]\!]^{2'} \\
&\quad \cap [\![t_1 \sqcap t_2]\!]^3 \cap [\![d_1 \sqcap d_2]\!]^7 \cap sem* \\
&\supseteq [\![ad_1]\!]^4 \cap [\![ad_2]\!]^4 \cap [\![hu_1]\!]^1 \cap [\![hu_2]\!]^1 \cap [\![ho_1]\!]^{1'} \cap [\![ho_2]\!]^{1'} \cap [\![sn_1]\!]^2 \cap [\![sn_2]\!]^2 \\
&\quad \cap [\![sn_1^\infty]\!]^{2'} \cap [\![sn_2^\infty]\!]^{2'} \cap [\![t_1]\!]^3 \cap [\![t_2]\!]^3 \cap [\![d_1]\!]^7 \cap [\![d_2]\!]^7 \cap sem* \\
&= [\![(ad_1, hu_1, ho_1, sn_1, sn_1^\infty, t_1, d_1)]\!]' \cap [\![(ad_2, hu_2, ho_2, sn_2, sn_2^\infty, t_2, d_2)]\!]'
\end{aligned}
$$

4.
$$\begin{array}{rl}
& [\![hu_1 \sqcap hu_2]\!]^1 \\
\triangleq & [\![hu_1 \cup hu_2]\!]^1 \\
= & \bigcap_{\alpha \in hu_1 \cup hu_2} \{s, h, f, r \mid f(\alpha) \cap dom(h) \neq \emptyset\} \\
= & \bigcap_{\alpha \in hu_1} \{s, h, f, r \mid f(\alpha) \cap dom(h) \neq \emptyset\} \cap \bigcap_{\alpha \in hu_2} \{s, h, f, r \mid f(\alpha) \cap dom(h) \neq \emptyset\} \\
= & [\![hu_1]\!]^1 \cap [\![hu_2]\!]^1
\end{array}$$

5.
$$\begin{array}{rl}
& [\![ho_1 \sqcap \mathtt{full}]\!]^{1'} \\
\triangleq & [\![ho_1]\!]^{1'} \\
= & [\![ho_1]\!]^{1'} \cap [\![\mathtt{full}]\!]^{1'}
\end{array}$$

6.
$$\begin{array}{rl}
& [\![\mathtt{full} \sqcap ho_2]\!]^{1'} \\
\triangleq & [\![ho_2]\!]^{1'} \\
= & [\![\mathtt{full}]\!]^{1'} \cap [\![ho_2]\!]^{1'}
\end{array}$$

7.
$$\begin{array}{rl}
& [\![ho_1 \sqcap ho_2]\!]^{1'} \\
\triangleq & [\![ho_2]\!]^{1'} \\
= & \{s, h, f, r \mid dom(h) \subseteq \bigcup_{\alpha \in ho_2} f(\alpha)\} \\
\supseteq & \{s, h, f, r \mid dom(h) \subseteq (\bigcup_{\alpha \in ho_1} f(\alpha)) \cap (\bigcup_{\alpha \in ho_2} f(\alpha))\} \\
= & \{s, h, f, r \mid dom(h) \subseteq \bigcup_{\alpha \in ho_1} f(\alpha)\} \cap \{s, h, f, r \mid dom(h) \subseteq \bigcup_{\alpha \in ho_2} f(\alpha)\} \\
= & [\![ho_1]\!]^{1'} \cap [\![ho_2]\!]^{1'}
\end{array}$$

Notice that this approximation is a choice, we could chose also $ho_1 \sqcap ho_2 \triangleq ho_1$.

8.
$$\begin{array}{rl}
& [\![ad \sqcap ad]\!]^4 \\
\triangleq & [\![ad]\!]^4 \\
= & [\![ad]\!]^4 \cap [\![ad]\!]^4
\end{array}$$

9.
$$\begin{array}{rl}
& [\![ad_1 \sqcap ad_2]\!]^4 \\
\triangleq & gfp[\![ad_1 \sqcap ad_2]\!]^4
\end{array}$$

10.
$$\begin{array}{rll}
& [\![[ad_1 \mid v \to \circledast] \sqcap [ad_2 \mid v \to S]]\!]^4 \\
\triangleq & [\![[ad_1 \mid v \to S] \sqcap [ad_2 \mid v \to S]]\!]^4 \\
= & [\![[ad_1 \mid v \to S]]\!]^4 \cap [\![[ad_2 \mid v \to S]]\!]^4 & \text{by rec.} \\
= & \bigcap_{v' \neq v} [\![v', ad_1(v')]\!]^5 \cap [\![v, S]\!]^5 \cap \bigcap_{v' \neq v} [\![v', ad_2(v')]\!]^5 \\
\supseteq & \bigcap_{v' \neq v} [\![v', ad_1(v')]\!]^5 \cap [\![v, \circledast]\!]^5 \cap [\![v, S]\!]^5 \cap \bigcap_{v' \neq v} [\![v', ad_2(v')]\!]^5 \\
= & [\![[ad_1 \mid v \to \circledast]]\!]^4 \cap [\![[ad_2 \mid v \to S]]\!]^4
\end{array}$$

11.
$$
\begin{array}{rl}
& \llbracket [ad_1 \mid v \rightarrow S] \sqcap [ad_2 \mid v \rightarrow \circledast] \rrbracket^4 \\
\triangleq & \llbracket [ad_1 \mid v \rightarrow S] \sqcap [ad_2 \mid v \rightarrow S] \rrbracket^4 \\
= & \llbracket [ad_1 \mid v \rightarrow S] \rrbracket^4 \cap \llbracket [ad_2 \mid v \rightarrow S] \rrbracket^4 \qquad \text{by rec.} \\
= & \bigcap_{v' \neq v} \llbracket v', ad_1(v') \rrbracket^5 \cap \llbracket v, S \rrbracket^5 \cap \bigcap_{v' \neq v} \llbracket v', ad_2(v') \rrbracket^5 \\
\supseteq & \bigcap_{v' \neq v} \llbracket v', ad_1(v') \rrbracket^5 \cap \llbracket v, S \rrbracket^5 \cap \llbracket v, \circledast \rrbracket^5 \cap \bigcap_{v' \neq v} \llbracket v', ad_2(v') \rrbracket^5 \\
= & \llbracket [ad_1 \mid v \rightarrow S] \rrbracket^4 \cap \llbracket [ad_2 \mid v \rightarrow \circledast] \rrbracket^4
\end{array}
$$

12.
$$
\begin{array}{rl}
& \llbracket [ad_1 \mid v \rightarrow \top] \sqcap [ad_2 \mid v \rightarrow S] \rrbracket^4 \\
\triangleq & \llbracket [ad_1 \mid v \rightarrow S] \sqcap [ad_2 \mid v \rightarrow S] \rrbracket^4 \\
= & \llbracket [ad_1 \mid v \rightarrow S] \rrbracket^4 \cap \llbracket [ad_2 \mid v \rightarrow S] \rrbracket^4 \qquad \text{by rec.} \\
= & \bigcap_{v' \neq v} \llbracket v', ad_1(v') \rrbracket^5 \cap \llbracket v, S \rrbracket^5 \cap \bigcap_{v' \neq v} \llbracket v', ad_2(v') \rrbracket^5 \\
= & \llbracket [ad_1 \mid v \rightarrow \top] \rrbracket^4 \cap \llbracket [ad_2 \mid v \rightarrow S] \rrbracket^4
\end{array}
$$

13.
$$
\begin{array}{rl}
& \llbracket [ad_1 \mid v \rightarrow S] \sqcap [ad_2 \mid v \rightarrow \top] \rrbracket^4 \\
\triangleq & \llbracket [ad_1 \mid v \rightarrow S] \sqcap [ad_2 \mid v \rightarrow S] \rrbracket^4 \\
= & \llbracket [ad_1 \mid v \rightarrow S] \rrbracket^4 \cap \llbracket [ad_2 \mid v \rightarrow S] \rrbracket^4 \qquad \text{by rec.} \\
= & \bigcap_{v' \neq v} \llbracket v', ad_1(v') \rrbracket^5 \cap \llbracket v, S \rrbracket^5 \cap \bigcap_{v' \neq v} \llbracket v', ad_2(v') \rrbracket^5 \\
= & \llbracket [ad_1 \mid v \rightarrow S] \rrbracket^4 \cap \llbracket [ad_2 \mid v \rightarrow \top] \rrbracket^4
\end{array}
$$

14. when $S_1, S_2 \neq \top, \circledast$ and $S_1 \subseteq S_2$
$$
\begin{array}{rl}
& \llbracket [ad_1 \mid v \rightarrow S_1] \sqcap [ad_2 \mid v \rightarrow S_2] \rrbracket^4 \\
\triangleq & \llbracket [ad_1 \mid v \rightarrow S_1] \sqcap [ad_2 \mid v \rightarrow S_1] \rrbracket^4 \\
= & \llbracket [ad_1 \mid v \rightarrow S_1] \rrbracket^4 \cap \llbracket [ad_2 \mid v \rightarrow S_1] \rrbracket^4 \qquad \text{by rec.} \\
= & \bigcap_{v' \neq v} \llbracket v', ad_1(v') \rrbracket^5 \cap \llbracket v, S_1 \rrbracket^5 \cap \bigcap_{v' \neq v} \llbracket v', ad_2(v') \rrbracket^5 \\
= & \bigcap_{v' \neq v} \llbracket v', ad_1(v') \rrbracket^5 \cap \bigcap_{v' \neq v} \llbracket v', ad_2(v') \rrbracket^5 \\
& \cap \left\{ s, h, f, r \mid s^+ f(v) \subseteq \bigcup_{vd \in S_1} \llbracket vd, (h, f, r) \rrbracket^8 \right\} \\
= & \bigcap_{v' \neq v} \llbracket v', ad_1(v') \rrbracket^5 \cap \bigcap_{v' \neq v} \llbracket v', ad_2(v') \rrbracket^5 \\
& \cap \left\{ s, h, f, r \mid s^+ f(v) \subseteq \bigcup_{vd \in S_1} \llbracket vd, (h, f, r) \rrbracket^8 \right\} \\
& \cap \left\{ s, h, f, r \mid s^+ f(v) \subseteq \bigcup_{vd \in S_2} \llbracket vd, (h, f, r) \rrbracket^8 \right\} \\
= & \bigcap_{v' \neq v} \llbracket v', ad_1(v') \rrbracket^5 \cap \llbracket v, S_1 \rrbracket^5 \cap \llbracket v, S_2 \rrbracket^5 \cap \bigcap_{v' \neq v} \llbracket v', ad_2(v') \rrbracket^5 \\
= & \llbracket [ad_1 \mid v \rightarrow S_1] \rrbracket^4 \cap \llbracket [ad_2 \mid v \rightarrow S_2] \rrbracket^4
\end{array}
$$

15. when $S_1, S_2 \neq \top, \circledast$ and $S_2 \subseteq S_1$

$$
\begin{aligned}
&\quad [\![[ad_1 \mid v \to S_1] \sqcap [ad_2 \mid v \to S_2]]\!]^4 \\
&\triangleq\ [\![[ad_1 \mid v \to S_2] \sqcap [ad_2 \mid v \to S_2]]\!]^4 \\
&=\ [\![[ad_1 \mid v \to S_2]]\!]^4 \cap [\![[ad_2 \mid v \to S_2]]\!]^4 \qquad\qquad \text{by rec.}\\
&=\ \bigcap_{v'\neq v} [\![v', ad_1(v')]\!]^5 \cap [\![v, S_2]\!]^5 \cap \bigcap_{v'\neq v} [\![v', ad_2(v')]\!]^5 \\
&=\ \bigcap_{v'\neq v} [\![v', ad_1(v')]\!]^5 \cap \bigcap_{v'\neq v} [\![v', ad_2(v')]\!]^5 \\
&\qquad \cap \left\{ s,h,f,r \mid s^+f(v) \subseteq \bigcup_{vd\in S_2} [\![vd, (h,f,r)]\!]^8 \right\} \\
&=\ \bigcap_{v'\neq v} [\![v', ad_1(v')]\!]^5 \cap \bigcap_{v'\neq v} [\![v', ad_2(v')]\!]^5 \\
&\qquad \cap \left\{ s,h,f,r \mid s^+f(v) \subseteq \bigcup_{vd\in S_1} [\![vd, (h,f,r)]\!]^8 \right\} \\
&\qquad \cap \left\{ s,h,f,r \mid s^+f(v) \subseteq \bigcup_{vd\in S_2} [\![vd, (h,f,r)]\!]^8 \right\} \\
&=\ \bigcap_{v'\neq v} [\![v', ad_1(v')]\!]^5 \cap [\![v, S_1]\!]^5 \cap [\![v, S_2]\!]^5 \cap \bigcap_{v'\neq v} [\![v', ad_2(v')]\!]^5 \\
&=\ [\![[ad_1 \mid v \to S_1]]\!]^4 \cap [\![[ad_2 \mid v \to S_2]]\!]^4
\end{aligned}
$$

16. when $S_1, S_2 \neq \top, \otimes$ (and $S_1 \not\subseteq S_2$ and $S_2 \not\subseteq S_1$)

let's define $NoVar = gfp\{Numt, Truet, Falset, Oodt, Nilt, Dangling\_Loc\} \cup \{Loc(A, vd_1, vd_2) \in VD \mid vd_1, vd_2 \in NoVar\}$.

(a) when $S_1 = \{vd_1\}$, $S_2 = \{vd_2\}$ with $vd_1, vd_2 \in NoVar$

   i. if $v = x \in Var$

$$
\begin{aligned}
&\quad [\![[ad_1 \mid x \to S_1] \sqcap [ad_2 \mid x \to S_2]]\!]^4 \\
&\triangleq\ [\![ar_\perp]\!]^4 \\
&=\ \emptyset \\
&=\ \bigcap_{v' \neq x} [\![v', ad_1(v')]\!]^5 \\
&\quad \cap \{s, h, f, r \mid \{s(x)\} = \emptyset\} \\
&\quad \cap \bigcap_{v' \neq x} [\![v', ad_2(v')]\!]^5 \\
&=\ \bigcap_{v' \neq x} [\![v', ad_1(v')]\!]^5 \\
&\quad \cap \{s, h, f, r \mid \{s(x)\} \subseteq [\![vd_1, (h, f, r)]\!]^8 \cap [\![vd_2, (h, f, r)]\!]^8\} \\
&\quad \cap \bigcap_{v' \neq x} [\![v', ad_2(v')]\!]^5 \\
&=\ \bigcap_{v' \neq x} [\![v', ad_1(v')]\!]^5 \\
&\quad \cap \{s, h, f, r \mid \{s(x)\} \subseteq [\![vd_1, (h, f, r)]\!]^8\} \\
&\quad \cap \{s, h, f, r \mid \{s(x)\} \subseteq [\![vd_2, (h, f, r)]\!]^8\} \\
&\quad \cap \bigcap_{v' \neq x} [\![v', ad_2(v')]\!]^5 \\
&=\ \bigcap_{v' \neq x} [\![v', ad_1(v')]\!]^5 \cap [\![x, S_1]\!]^5 \cap [\![x, S_2]\!]^5 \cap \bigcap_{v' \neq x} [\![v', ad_2(v')]\!]^5 \\
&=\ [\![[ad_1 \mid x \to S_1]]\!]^4 \cap [\![[ad_2 \mid x \to S_2]]\!]^4
\end{aligned}
$$

ii. if $v = \alpha \in TVar$

$$
\begin{aligned}
&\quad [\![[ad_1 \mid \alpha \rightarrow S_1]\sqcap[ad_2 \mid \alpha \rightarrow S_2]]\!]^4 \\
&\triangleq\ [\![clean([ad_1 \mid \alpha \rightarrow\circledast]\sqcap[ad_2 \mid \alpha \rightarrow\circledast])]\!]^4 \quad \text{(in practice we clean on-the-fly)} \\
&=\ [\![[ad_1 \mid \alpha \rightarrow\circledast]\sqcap[ad_2 \mid \alpha \rightarrow\circledast]]\!]^4 \quad \text{(by Prop. D.2)} \\
&=\ [\![[ad_1 \mid \alpha \rightarrow\circledast]]\!]^4 \cap [\![[ad_2 \mid \alpha \rightarrow\circledast]]\!]^4 \quad \text{(by rec.)} \\
&=\ \bigcap_{v' \neq \alpha} [\![v', ad_1(v')]\!]^5 \\
&\quad \cap [\![\alpha, \circledast]\!]^4 \\
&\quad \cap \bigcap_{v' \neq \alpha} [\![v', ad_2(v')]\!]^5 \\
&=\ \bigcap_{v' \neq \alpha} [\![v', ad_1(v')]\!]^5 \\
&\quad \cap \{s, h, f, r \mid f(\alpha) = \emptyset\} \\
&\quad \cap \bigcap_{v' \neq \alpha} [\![v', ad_2(v')]\!]^5 \\
&=\ \bigcap_{v' \neq \alpha} [\![v', ad_1(v')]\!]^5 \\
&\quad \cap \{s, h, f, r \mid f(\alpha) \subseteq [\![vd_1, (h, f, r)]\!]^8 \cap [\![vd_2, (h, f, r)]\!]^8\} \\
&\quad \cap \bigcap_{v' \neq \alpha} [\![v', ad_2(v')]\!]^5 \\
&=\ \bigcap_{v' \neq \alpha} [\![v', ad_1(v')]\!]^5 \\
&\quad \cap \{s, h, f, r \mid f(\alpha) \subseteq [\![vd_1, (h, f, r)]\!]^8\} \\
&\quad \cap \{s, h, f, r \mid f(\alpha) \subseteq [\![vd_2, (h, f, r)]\!]^8\} \\
&\quad \cap \bigcap_{v' \neq \alpha} [\![v', ad_2(v')]\!]^5 \\
&=\ \bigcap_{v' \neq \alpha} [\![v', ad_1(v')]\!]^5 \cap [\![\alpha, S_1]\!]^5 \cap [\![\alpha, S_2]\!]^5 \cap \bigcap_{v' \neq \alpha} [\![v', ad_2(v')]\!]^5 \\
&=\ [\![[ad_1 \mid \alpha \rightarrow S_1]]\!]^4 \cap [\![[ad_2 \mid \alpha \rightarrow S_2]]\!]^4
\end{aligned}
$$

(b) when $S_1 = \{vd\}$ with $vd \in NoVar$, $S_2 = \{Loc(A, vd_1, vd_2)\}$

Same treatment as for case 16a.

(c) when $S_2 = \{vd\}$ with $vd \in NoVar$, $S_1 = \{Loc(A, vd_1, vd_2)\}$

Same treatment as for case 16a.

(d) when $S_1 = \{\beta\}$ and $S_2 \subseteq NoVar$

Same treatment as for case 16a.

(e) ...

(f)

17. ....

18. by domain countrains we get that $\alpha \in TVar$

$$[\![ad_1 \mid v \to \otimes]\!]^4 \cap [\![ad_2]\!]^4$$
$$= \bigcap_{v' \neq \alpha} [\![v', ad_1(v')]\!]^5 \cap [\![\alpha, \otimes]\!]^5 \cap [\![ad_2]\!]^4$$
$$\subseteq \bigcap_{v' \neq \alpha} [\![v', ad_1(v')]\!]^5 \cap [\![ad_2]\!]^4$$
$$= \bigcap_{v' \neq \alpha} [\![v', ad_1(v')]\!]^5 \cap [\![v, \top]\!]^5 \cap [\![ad_2]\!]^4$$
$$= [\![ad_1 \mid v \to \top]\!]^4 \cap [\![ad_2]\!]^4$$

19. by domain countrains we get that $\alpha \in TVar$

$$[\![ad_1]\!]^4 \cap [\![ad_2 \mid \alpha \to \otimes]\!]^4$$
$$= \bigcap_{v' \neq \alpha} [\![v', ad_1(v')]\!]^5 \cap [\![\alpha, ad_1(\alpha)]\!]^5 \bigcap_{v' \neq \alpha} [\![v', ad_2(v')]\!]^5 \cap [\![\alpha, \otimes]\!]^5$$
$$\subseteq \bigcap_{v' \neq \alpha} [\![v', ad_1(v')]\!]^5 \cap [\![\alpha, ad_1(\alpha)]\!]^5 \bigcap_{v' \neq \alpha} [\![v', ad_2(v')]\!]^5$$
$$= [\![ad_1 \mid v \to \top]\!]^4 \cap [\![ad_2 \mid \alpha \to ad_1(\alpha)]\!]^4$$

———————————————————————————————

I want to prove that $\forall s, h, f_1, f_2, r, ad_1, ad_2. \exists f_{12}$.

- $s, h, f_1, r \in [\![ad_1]\!]^4$
- $s, h, f_2, r \in [\![ad_2]\!]^4$ $\Rightarrow s, h, f_{12}, r \in [\![ad_1 \sqcap ad_2]\!]^4$

We define $AD_{TPL} \triangleq \mathcal{P}(VAR \times PVD^+)$. and

$$tpl \in AD \overset{total}{\to} AD_{TPL}$$
$$tpl(ad) \triangleq \bigcup_{v \in VAR} \{(v, ad(v))\}$$

$\forall ad_{tpl} \in AD_{TPL}$. if $\forall v \in VAR. \exists! S \in PVD^+. (v, S) \in ad_{tpl}$ then we define $tpl^{-1}(ad_{tpl}) \triangleq ad \in AD$ such that $tpl(ad) = ad_{tpl}$.

Recall that $[\![ad]\!]^4 \triangleq \bigcap_{v \in VAR} [\![v, ad(v)]\!]^5$

We define
$$[\![\cdot]\!]^{4'} \in AD_{TPL} \to MFR$$
$$[\![ad_{tpl}]\!]^{4'} \triangleq \bigcap_{v \in VAR} [\![v, ad_{tpl}(v)]\!]^5$$

By construction we have $\forall ad. [\![tpl(ad)]\!]^{4'} = [\![ad]\!]^4$ and $\forall ad_{tpl}$ such that $tpl^{-1}(ad_{tpl})$ exists $[\![tpl^{-1}(ad_{tpl})]\!]^4 = [\![ad_{tpl}]\!]^{4'}$.

We define $ad_1 \sqcap ad_2 \triangleq tpl^{-1}(\sqcap(tpl(ad_1) \cup tpl(ad_2)))$

We have to prove that it exists and that

- $s, h, f_1, r \in [\![tpl(ad_1)]\!]^{4'}$
- $s, h, f_2, r \in [\![tpl(ad_2)]\!]^{4'}$

$\Rightarrow s, h, f_1 \dot\sqcap f_2, r \in [\![\sqcap(tpl(ad_1) \cup tpl(ad_2))]\!]'^4$

We will first prove that:

- $s, h, f_1, r \in [\![ad_1]\!]^4$
- $s, h, f_2, r \in [\![ad_2]\!]^4$

$\Rightarrow s, h, f_1 \dot\sqcap f_2, r \in [\![ad_1]\!]^4 \cap [\![ad_2]\!]^4$

and then prove that

$$\forall ad_{tpl} \in AD_{TPL}. \; [\![ad_{tpl}]\!]'^4 \subseteq [\![\sqcap(ad_{tpl})]\!]'^4$$

Let's proceed the first proof: We want :

- $s, h, f_1, r \in \bigcap_{v \in VAR} [\![v, ad_1(v)]\!]^5$
- $s, h, f_2, r \in \bigcap_{v \in VAR} [\![v, ad_2(v)]\!]^5$

$\Rightarrow$

- $s, h, f_1 \dot\sqcap f_2, r \in \bigcap_{v \in VAR} [\![v, ad_1(v)]\!]^5$
- $s, h, f_1 \dot\sqcap f_2, r \in \bigcap_{v \in VAR} [\![v, ad_2(v)]\!]^5$

it's symetrical for ad_1 and ad_2 so we will prove:

- $s, h, f_1, r \in \bigcap_{v \in VAR} [\![v, ad_1(v)]\!]^5$
- $s, h, f_2, r \in \bigcap_{v \in VAR} [\![v, ad_2(v)]\!]^5$

$\Rightarrow$ - $s, h, f_1 \dot\sqcap f_2, r \in \bigcap_{v \in VAR} [\![v, ad_1(v)]\!]^5$

by recurrence on the definition of $[\![\cdot]\!]^5$.

- case $ad_1(v) = \top$, we directly have $s, h, f_1 \dot\sqcap f_2, r \in [\![v, \top]\!]^5$

- $ad_1(v) = \otimes$, by the domain constraints, we have $v \in TVar$, and by hypothesis we have $s, h, f_1, r \in [\![v, \otimes]\!]^5$ so we have $f_1(v) = \emptyset$ thus $f_1 \dot\sqcap f_2(v) = \emptyset$ and then as expected $s, h, f_1 \dot\sqcap f_2, r \in [\![v, \otimes]\!]^5$

- $ad_1(v) = S$, with $S \neq \top, \otimes$, by hypothesis we have : $s^+ f_1(v) \subseteq \bigcup_{vd \in S} [\![vd, (h, f_1, r)]\!]^8$, we have by construction $s^+(f_1 \dot\sqcap f_2)(v) \subseteq s^+ f_1(v)$, so it remains to prove that:

$$\bigcup_{vd \in S} [\![vd, (h, f_1, r)]\!]^8 \subseteq \bigcup_{vd \in S} [\![vd, (h, f_1, r)]\!]^8$$

262

_____

1. Case $s, h, f_1, r \in [\![\alpha \rightarrow \{\beta\}$

2. Case $v \notin sn_1 \cap sn_2$

   (we have then $|s^+f_1(v)| \leq 1$) :

$$s, h, f_1, r \in [\![[v \rightarrow \{vd_1\} \cup S_1 \mid ad_1]\!]^4$$
$$\wedge \quad s, h, f_2, r \in [\![[v \rightarrow \{vd_2\} \cup S_2 \mid ad_2]\!]^4$$

$$\equiv \quad s, h, f_1, r \in [\![v, \{vd_1\} \cup S_1]\!]^5$$
$$\wedge \quad s, h, f_1, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_1(v')]\!]^5$$
$$\wedge \quad s, h, f_2, r \in [\![v, \{vd_2\} \cup S_2]\!]^5$$
$$\wedge \quad s, h, f_2, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_2(v')]\!]^5$$

$$\equiv \quad s^+f_1(v) \subseteq [\![vd_1, (h, f_1, r)]\!]^8 \cup \bigcup_{vd_1' \in S_1} [\![vd_1', (h, f_1, r)]\!]^8$$
$$\wedge \quad s, h, f_1, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_1(v')]\!]^5$$
$$\wedge \quad s^+f_2(v) \subseteq [\![vd_2, (h, f_2, r)]\!]^8 \cup \bigcup_{vd_2' \in S_2} [\![vd_2', (h, f_2, r)]\!]^8$$
$$\wedge \quad s, h, f_2, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_2(v')]\!]^5$$

$$\equiv \quad (s^+f_1(v) \subseteq [\![vd_1, (h, f_1, r)]\!]^8$$
$$\vee s^+f_1(v) \subseteq \bigcup_{vd_1' \in S_1} [\![vd_1', (h, f_1, r)]\!]^8)$$
$$\wedge \quad s, h, f_1, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_1(v')]\!]^5$$
$$\wedge \quad (s^+f_2(v) \subseteq [\![vd_2, (h, f_2, r)]\!]^8$$
$$\vee s^+f_2(v) \subseteq \bigcup_{vd_2' \in S_2} [\![vd_2', (h, f_2, r)]\!]^8)$$
$$\wedge \quad s, h, f_2, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_2(v')]\!]^5$$

$$
\begin{aligned}
\equiv \quad & (s^+ f_1(v) \subseteq [\![vd_1, (h, f_1, r)]\!]^8 \\
& \wedge s, h, f_1, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_1(v')]\!]^5 \\
& \wedge s^+ f_2(v) \subseteq [\![vd_2, (h, f_2, r)]\!]^8 \\
& \wedge s, h, f_2, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_2(v')]\!]^5) \\
\vee \quad & (s^+ f_1(v) \subseteq \bigcup_{vd'_1 \in S_1} [\![vd'_1, (h, f_1, r)]\!]^8 \\
& \wedge s, h, f_1, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_1(v')]\!]^5 \\
& \wedge s^+ f_2(v) \subseteq [\![vd_2, (h, f_2, r)]\!]^8 \\
& \wedge s, h, f_2, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_2(v')]\!]^5) \\
\vee \quad & (s^+ f_1(v) \subseteq [\![vd_1, (h, f_1, r)]\!]^8 \\
& \wedge s, h, f_1, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_1(v')]\!]^5 \\
& \wedge s^+ f_2(v) \subseteq \bigcup_{vd'_2 \in S_2} [\![vd'_2, (h, f_2, r)]\!]^8 \\
& \wedge s, h, f_2, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_2(v')]\!]^5) \\
\vee \quad & (s^+ f_1(v) \subseteq \bigcup_{vd'_1 \in S_1} [\![vd'_1, (h, f_1, r)]\!]^8 \\
& \wedge s, h, f_1, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_1(v')]\!]^5 \\
& \wedge s^+ f_2(v) \subseteq \bigcup_{vd'_2 \in S_2} [\![vd'_2, (h, f_2, r)]\!]^8 \\
& \wedge s, h, f_2, r \in \bigcap_{v' \in VAR \neq v} [\![v', ad_2(v')]\!]^5)
\end{aligned}
$$

$$\equiv \qquad (s, h, f_1, r \in [\![v \to \{vd_1\} \mid ad_1]\!]^4$$
$$\wedge s, h, f_2, r \in [\![v \to \{vd_2\} \mid ad_2]\!]^4)$$
$$\vee \qquad (s, h, f_1, r \in [\![v \to S_1 \mid ad_1]\!]^4$$
$$\wedge s, h, f_2, r \in [\![v \to \{vd_2\} \mid ad_2]\!]^4)$$
$$\vee \qquad (s, h, f_1, r \in [\![v \to \{vd_1\} \mid ad_1]\!]^4$$
$$\wedge s, h, f_2, r \in [\![v \to S_2 \mid ad_2]\!]^4)$$
$$\vee \qquad (s, h, f_1, r \in [\![v \to S_1 \mid ad_1]\!]^4$$
$$\wedge s, h, f_2, r \in [\![v \to S_2 \mid ad_2]\!]^4)$$

$$\Rightarrow \quad \text{(by rec)} \quad \exists f.\ s, h, f, r \in [\![v \to \{vd_1\} \mid ad_1] \sqcap [v \to \{vd_2\} \mid ad_2]\!]^4$$
$$\vee \qquad \exists f'.\ s, h, f', r \in [\![v \to S_1 \mid ad_1] \sqcap [v \to \{vd_2\} \mid ad_2]\!]^4$$
$$\vee \qquad \exists f''.\ s, h, f'', r \in [\![v \to \{vd_1\} \mid ad_1] \sqcap [v \to S_2 \mid ad_2]\!]^4$$
$$\vee \qquad \exists f'''.\ s, h, f''', r \in [\![v \to S_1 \mid ad_1] \sqcap [v \to S_2 \mid ad_2]\!]^4$$

$$\Rightarrow \quad \text{(by proof of Prop. 3.3)} \quad \exists g.\ s, h, g, r \in \left[\!\!\left[\begin{array}{l} ([v \to \{vd_1\} \mid ad_1] \sqcap [v \to \{vd_2\} \mid ad_2]) \\ \dot{\sqcup}\ \ ([v \to S_1 \mid ad_1] \sqcap [v \to \{vd_2\} \mid ad_2]) \\ \dot{\sqcup}\ \ ([v \to \{vd_1\} \mid ad_1] \sqcap [v \to S_2 \mid ad_2]) \\ \dot{\sqcup}\ \ ([v \to S_1 \mid ad_1] \sqcap [v \to S_2 \mid ad_2]) \end{array} \right]\!\!\right]^4$$

So we can say that we have :

$$s, h, f_1, r \in [\![v \to S_1 \mid ad_1]\!]^4$$
$$\wedge \quad s, h, f_2, r \in [\![v \to S_2 \mid ad_2]\!]^4$$

$$\Rightarrow \qquad \exists f'.\ s, h, f', r \in \left[\!\!\left[\dot{\bigsqcup}_{vd_1 \in S_1, vd_2 \in S_2} ([v \to \{vd_1\} \mid ad_1] \sqcap [v \to \{vd_2\} \mid ad_2]) \right]\!\!\right]^4$$

$\square$ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad $\square$

Appendix D

Clean proof

We define $\forall \alpha \in \mathit{TVar}.\ VD_\alpha \triangleq \{\alpha\} \cup \{Loc(A, vd_1, vd_2) \in VD \mid vd_1 = \alpha \vee vd_2 = \alpha\}.$ and $\forall V \in \mathcal{P}(\mathit{TVar}).\ VD_V \triangleq \bigcup_{\alpha \in V} VD_\alpha.$

Definition D.1.   *1. When $\forall v.VD_V \cap (ad_1(v) \cup ad_2(v)) = \emptyset$*

$$clean(V, ad_1, ad_2) \triangleq (ad_1, ad_2)$$

2. *when $VD_V \cap S_1 \neq \emptyset$ and $S_1 \setminus VD_V \neq \emptyset$*

$$clean(V, [ad_1 \mid v \rightarrow S_1], [ad_2 \mid v \rightarrow S_2]) \triangleq clean(V, [ad_1 \mid v \rightarrow S_1 \setminus VD_V], [ad_2 \mid v \rightarrow S_2])$$

3. *when $VD_V \cap S_1 \neq \emptyset$ and $S_1 \setminus VD_V = \emptyset$*

    (a) when $x \in Var$

$$clean(V, [ad_1 \mid x \rightarrow S_1], [ad_2 \mid x \rightarrow S_2]) \triangleq (ar_\perp, [ad_2 \mid x \rightarrow S_2])$$

    (b) when $\alpha \in TVar$

$$clean(V, [ad_1 \mid \alpha \rightarrow S_1], [ad_2 \mid \alpha \rightarrow S_2]) \triangleq clean(V \cup \{\alpha\}, [ad_1 \mid \alpha \rightarrow \otimes], [ad_2 \mid \alpha \rightarrow \otimes])$$

4. *when $VD_V \cap S_1 = \emptyset$ and $VD_V \cap S_2 \neq \emptyset$ and $S_2 \setminus VD_V \neq \emptyset$*

$$clean(V, [ad_1 \mid v \rightarrow S_1], [ad_2 \mid v \rightarrow S_2]) \triangleq clean(V, [ad_1 \mid v \rightarrow S_1], [ad_2 \mid v \rightarrow S_2 \setminus VD_V])$$

5. *when $VD_V \cap S_1 = \emptyset$ and $VD_V \cap S_2 \neq \emptyset$ and $S_2 \setminus VD_V = \emptyset$*

    (a) *when $x \in Var$*

$$clean(V, [ad_1 \mid x \rightarrow S_1], [ad_2 \mid x \rightarrow S_2]) \triangleq ([ad_1 \mid x \rightarrow S_1], ar_\perp)$$

    (b) *when $\alpha \in TVar$*

$$clean(V, [ad_1 \mid \alpha \rightarrow S_1], [ad_2 \mid \alpha \rightarrow S_2]) \triangleq clean(V \cup \{\alpha\}, [ad_1 \mid \alpha \rightarrow \circledast], [ad_2 \mid \alpha \rightarrow \circledast])$$

Proposition D.2.

$\forall \alpha \in TVar.\ ad_1, ad_2 \in AD$

* $[\![fst(clean(\alpha, [ad_1 \mid \alpha \rightarrow \circledast], [ad_2 \mid \alpha \rightarrow \circledast]))]\!]^4 = [\![[ad_1 \mid \alpha \rightarrow \circledast]]\!]^4$
* $[\![snd(clean(\alpha, [ad_2 \mid \alpha \rightarrow \circledast], [ad_2 \mid \alpha \rightarrow \circledast]))]\!]^4 = [\![[ad_2 \mid \alpha \rightarrow \circledast]]\!]^4$
* *the number of difference between* $fst(clean(\alpha, [ad_1 \mid \alpha \rightarrow \circledast], [ad_2 \mid \alpha \rightarrow \circledast])$ *and* $snd(clean(\alpha, [ad_1 \mid \alpha \rightarrow \circledast], [ad_2 \mid \alpha \rightarrow \circledast])$ *is inferior or equal to the one between* $[ad_1 \mid \alpha \rightarrow \circledast]$ *and* $[ad_2 \mid \alpha \rightarrow \circledast]$

Prop. D.2. We suppose that $\{v \mid ad_1(v) \neq \circledast, \top \vee ad_2(v) \neq \circledast, \top\}$ is finite.

Let $VD_\alpha \triangleq \{\alpha\} \cup \{Loc(A, vd_1, vd_2) \in VD \mid vd_1 = \alpha \vee vd_2 = \alpha\}$

1. When $\forall v.VD_\alpha \cap (ad_1(v) \cup ad_2(v)) = \emptyset$

$$clean(\alpha, ad_1, ad_2) \triangleq (ad_1, ad_2)$$

2. when $VD_\alpha \cap S_1 \neq \emptyset$ and $S_1 \setminus VD_\alpha \neq \emptyset$

$$clean(\alpha, [ad_1 \mid v \rightarrow S_1], [ad_2 \mid v \rightarrow S_2]) \triangleq clean(\alpha, [ad_1 \mid v \rightarrow S_1 \setminus VD_\alpha], [ad_2 \mid v \rightarrow S_2])$$

    ...

$\square$