Characterizing Provability in BI's Pointer Logic through Resource Graphs

Didier Galmiche and Daniel Méry

LORIA - Université Henri Poincaré
Campus Scientifique, BP 239
Vandœuvre-lès-Nancy, France

Abstract We propose a characterization of provability in BI's Pointer Logic (PL) that is based on semantic structures called resource graphs. This logic has been defined for reasoning about mutable data structures and results about models and verification have been already provided. Here, we define resource graphs that capture PL models by considering heaps as resources and by using a labelling process. We study provability in PL from a new calculus that builds such graphs from which proofs or countermodels can be generated. Properties of soundness and completeness are proved and the countermodel generation is studied.

1 Introduction

Separation logics are logics for reasoning about mutable data structures in which the pre- and postconditions are written in a logic enriched with specific forms of conjunction or implication. In this context, Reynolds has proposed an intuitionistic logic extended with a separation connective $*$ [12] and Ishtiaq and O'Hearn have investigated the same approach from the point of view of the logic of Bunched Implications (BI) [10]. A key point of BI logic is its joint treatment of intuitionistic implication $\rightarrow$ and conjunction $\wedge$ (additive connectives) and linear implication $-\!\!*$ and conjunction $*$ (multiplicative connectives). BI's semantics allows statements to be made using standard connectives and then to combine them in a modular way using $-\!\!*$ and $*$. The resource interpretation of the connectives, where $*$ decomposes the current resource into pieces and $-\!\!*$ talks about new and fresh resource, is central. BI's pointer logic (PL) provides a concrete way of understanding the connectives in the context of program verification apart from logical concerns [6] . It is a possible worlds model of Boolean BI (BBI) where additives are classical that validates the axioms for Hoare triples. PL being used as an assertion language for mutable data structures, it appears essential to provide a proof theory and related proof search methods in order to check PL assertions.

In this paper we propose a characterization of provability in PL that is based on semantic structures, called resource graphs, and a new labelled calculus that builds such graphs from which proofs or countermodels can be generated. A similar approach has been developed for BI with intuitionistic additives [3,4] but it cannot be directly applied to BBI and its classical additives. The key point consists in defining resource graphs that capture PL models by considering heaps as resources and by using an appropriate labelling process. Such graphs allow to capture semantic information essential for the provability analysis [5]. Having defined PL resource graphs, we propose a calculus with labels and constraints for a propositional fragment of PL, called $\mathsf{PL}^{\mapsto}$, in a tableau style that is adapted to countermodel extraction. Soundness and completeness of the related proof search method are proved and countermodel generation from resource graphs is analyzed. Finally, extensions of the calculus and results to PL can be developed.

2 BI's Pointer Logic

BI's Pointer Logic (PL) is a logic for reasoning about mutable data structures [6]. Some problems about pointer management, including aliasing, are difficult to deal with. Recent works have provided logics with a spatial form of conjunction $*$ that splits the heap into distinct subheaps and with a form of assertion, the points-to relation $\mapsto$, to make statements about the contents of heap cells [9,12]. It leads to simple axioms and captures the intuitive operational locality of assignment. They are based on BI logic that includes two implication connectives $\rightarrow$ and $-\!\!*$ with two conjunction connectives $\wedge$ and $*$ [10]. BI's semantics allow statements to be made using standard (additive) connectives and then to combine them using the (multiplicative) connectives.

Here, we consider PL that is a model of Boolean BI (with classical additives) [6] and summarize its key notions and results. First, we have a set of values *Val* that can be integers or locations, a set of variables *Var* and a countable set of locations *Loc*. A stack $s : Var \rightharpoonup_{fin} Val$ is a finite partially defined function that associates values to variables and a heap $h : Loc \rightharpoonup_{fin} Val \times Val$ is a finite partially defined function that associates pairs of values to locations. An expression E can represent a variable, called a stack variable, an integer or a constant that is interpreted w.r.t. a stack s as a value ($[\![E]\!]s \in Val$).

Definition 2.1. *The language of* PL *consists of the predicates $\mapsto$ and $=$, the connectives of* BI*, the existential quantifier and two countable sets of variables of stacks and values. The set of formulae is inductively defined as follows:*

- $At ::= (E \mapsto E_1, E_2) \mid E_1 = E_2$ *where E, E_1 and E_2 are expressions,*
- $\phi ::= At \mid \mathrm{I} \mid \phi * \phi \mid \phi -\!\!* \phi \mid \top \mid \bot \mid \phi \wedge \phi \mid \phi \rightarrow \phi \mid \phi \vee \phi \mid \exists x.\phi.$

Moreover, we can define $\neg\phi$ by $\neg\phi \equiv \phi \rightarrow \bot$ and use it to define connectives rather than taking them as primitives. The predicate $(x \mapsto a, b)$ allows to represent the state of the memory: there exists a variable x in a stack s such that there exists a location l in a heap h such that $[\![E]\!]s = l$ and $h(l) = \langle a, b\rangle$. The actual use of PL consists in describing states about resources with the language and in proving properties about these resources with the logic. The semantics of the formulae is given by a satisfaction relation of the form $s, h \models \phi$ that asserts that ϕ is true in stack $s \in S$ and heap $h \in H$. It is required that the free variables of ϕ are included in the domain of s.

Definition 2.2. *The semantics of the formulae is defined as follows:*

- $s, h \models E_1 = E_2$ *iff* $[\![E_1]\!]s = [\![E_2]\!]s$
- $s, h \models (E \mapsto E_1, E_2)$ *iff* $dom(h) = \{[\![E]\!]s\}$ *and* $h([\![E]\!]s) = \langle [\![E_1]\!]s, [\![E_2]\!]s\rangle$
- $s, h \models \top$ *always*

- $s,h \models \bot$ *never*
- $s,h \models \phi \wedge \psi$ *iff* $s,h \models \phi$ *and* $s,h \models \psi$
- $s,h \models \phi \vee \psi$ *iff* $s,h \models \phi$ *or* $s,h \models \psi$
- $s,h \models \phi \rightarrow \psi$ *iff if* $s,h \models \phi$ *then* $s,h \models \psi$
- $s,h \models \mathrm{I}$ *iff* h *is the empty heap*
- $s,h \models \phi * \psi$ *iff* $\exists h_1, h_2.\ h_1 \# h_2,\ h_1 \cdot h_2 = h,\ s,h_1 \models \phi$ *and* $s,h_2 \models \psi$
- $s,h \models \phi \twoheadrightarrow \psi$ *iff* $\forall h_1.\ if\ h_1 \# h$ *and* $s,h_1 \models \phi$ *then* $s,h_1 \cdot h \models \psi$
- $s,h \models \exists x.\phi$ *iff* $\exists v \in Val.\ [s|x \mapsto v],h \models \phi$

In the previous definition, $h_1 \# h_2$ means that the domains of heaps h_1 and h_2 are disjoint and $h_1 \cdot h_2$ denotes the union of disjoint heaps (union of functions with disjoint domains). Composition of non-disjoint heaps is undefined so that heap composition is only partial. Moreover, the semantic consequence relation $\phi \models \psi$ between formulae holds if and only if for all s,h, if $s,h \models \phi$ then $s,h \models \psi$.

In this model, the worlds are heaps (collections of cons cells in storage) and the conjunction $\phi * \psi$ is true just when the current heap can be split into two components, one of which makes ϕ true and the other which makes ψ true. The implication $\phi \twoheadrightarrow \psi$ talks about new or fresh pieces of heap, disjoint from the current heap. It says that, whenever we have a new heap that makes ϕ true, the combined new and current heap will make ψ true. The other connectives are interpreted pointwise. For instance, the formula $(x \mapsto 3,5) * ((x \mapsto 7,5) \twoheadrightarrow P)$ says that x denotes a cell which holds $(3,5)$ in the current heap and that if we update the car to 7 then P will be true. The semantics of $*$ splits the heap into two parts, one where $(x \mapsto 3,5)$ holds and another where the location x is dangling. Then the semantics of $\twoheadrightarrow$ and $\mapsto$ ensures that P must be true when the second heap is extended by binding x's location to $(7,5)$.

Main concepts and results about the use of PL as an assertion language are given in [6]. As an interesting result, we can mention an operation that disposes of memory, by creating dangling pointers, through the command $dispose(E)$ which deallocates a location. From a semantic point of view, it removes a location from the heap and is defined by the following axiom: $\{P * \exists\, a\, b.(E \mapsto a,b)\}\ dispose(E)\ \{P\}$, where a,b are not free in E. Reasoning backwards from $\top$ we can find cases under which a program is safe to execute. With a double dispose we obtain $\bot$ for the precondition as expected, indicating that the program is not safe to execute for any start state:

$$\begin{array}{c}
\{\bot\} \\
\{\top * \exists\, a\, b.(x \mapsto a,b) * \exists\, c\, d.(x \mapsto c,d)\} \\
dispose(x) \\
\{\top * \exists\, a\, b.(x \mapsto a,b)\} \\
dispose(x) \\
\{\top\}
\end{array}$$

We first study a propositional fragment of PL, denoted $\mathsf{PL}^{\mapsto}$, restricted to atomic formulae $(l \mapsto a,b)$ in which l,a,b are constants, meaning that at location l there is a cell containing (a,b). Then, we can forget the stack variables in the semantic clauses corresponding to this fragment.

3 Heaps, Labels and Resource Graphs

Purely syntactic proof methods (in sequent or natural deduction style) usually deal with a great amount of operational overhead (structural rules, permutabilities of inferences) which is mainly irrelevant w.r.t. the provability of a formula. On the other hand, purely semantic methods often abstract away too much of the operational aspects to be significant and helpful for countermodel construction. In the case of PL, we have a complete semantics based on partial monoids of heaps, and then syntactic and semantic consequence relations (provability and validity) coincide [1]. Therefore, the main properties of heap composition and of PL semantic consequence relation can be reflected at a syntactic level using labels, constraints and a specific closure operator [5] in order to define a resource driven proof method for PL.

3.1 Labels and constraints

Definition 3.1 (labels). *The labelling language consists of a countable set of constant symbols $c_0, c_1, \cdots$ and a binary symbol $\circ$. The set $\mathcal{L}$ of* labels *is the smallest set which contains constants c_i $(i \in \mathbb{N})$ (atomic labels) and such that $x \circ y$ is a (compound) label whenever x, y are labels which do not share any constant symbol.*

The constants are intended to reflect heaps, while $\circ$ is intended to reflect heap composition. We view labels as unordered sequences of symbols, *i.e.*, $\circ$ is interpreted as an associative and commutative operation on labels. For example, we consider the two labels $c_1 \circ (c_2 \circ (c_3 \circ c_4))$ and $(c_2 \circ c_1) \circ (c_4 \circ c_3)$ as equal and we do not distinguish between the two of them. Since order and association are irrelevant, we frequently omit the symbol $\circ$ and simply write xyz instead of $x \circ y \circ z$.

More formally, we first say that a label x is a *sublabel* of a label y if any constant occuring in x also occurs in y, *i.e.*, $x \leq y$ iff $(\forall c_i \in x)(c_i \in y)$. Then, we define $x = y$ iff $x \leq y$ and $y \leq x$. The length $|x|$ of a label x is the number of constants it contains. Moreover, we define $x - y$ as the label z obtained from x by discarding all the constants in y that also occur in x, eventually setting z to c_0 if there is no constant left in x. We then define label constraints as expressions of the form $yz \lhd x$, $\mathrm{Z}x$, $\mathrm{U}x$, $\mathrm{A}x$ that respectively reflect that a heap can be decomposed into two sub-heaps, may contain *no* cell (Zero), may contain *exactly one* cell (Unit) and may contain *at least one* cell (Aggregate).

Definition 3.2 (constraints). *A* label constraint *is an expression of the form $x \lhd y$ or of the form $\mathrm{T}x$, where x and y are labels and $\mathrm{T} \in \{\mathrm{Z}, \mathrm{U}, \mathrm{A}\}$.*

Moreover we need to capture the various resource interactions that occur in PL models and we do it through a closure operator $(\cdot)^\dagger$ on sets of labels and constraints. Let us note that when we write $x \in X^\dagger$, x being a label, we do not intend x to be a member of $X^\dagger$ explicitly, we only require that $X^\dagger$ contains some label y such that $y = x$, *i.e.*, we slightly abuse our notations to work with equivalence classes of labels modulo the equality $=$ induced by the associativity and commutativity of label composition.

[1] This is not the case for Boolean BI, for which there is currently no known complete monoid-based semantics.

Given a constraint k ($y \lhd x$, $\mathrm{Z}x$, $\mathrm{U}x$ or $\mathrm{A}x$), we say that k *holds in* X, written $X \vdash k$, if $k \in X^\dagger$. The domain $\mathcal{D}(X)$ of a set X of labels and constraints is defined as the restriction of $X^\dagger$ to its labels.

Definition 3.3 **($(\cdot)^\dagger$-closure).** *Let X be set of labels and label constraints. We define $X^\dagger$ as the smallest set containing $X \cup \{c_0, \mathrm{Z}c_0\}$ such that*

– $x \in X^\dagger$ *and* $y \leq x \Rightarrow y \in X^\dagger$ *(saturation),*
– $x \lhd y \in X^\dagger \Rightarrow x, y \in X^\dagger$ *($\lhd$-completion),*
– $\mathrm{Z}x \in X^\dagger$ *or* $\mathrm{U}x \in X^\dagger \Rightarrow x \in X^\dagger$ *(ZU-completion),*
– $x \in X^\dagger \Rightarrow x \lhd x \in X^\dagger$ *(reflexivity),*
– $x \lhd y \in X^\dagger$ *and* $y \lhd z \in X^\dagger \Rightarrow x \lhd z \in X^\dagger$ *(transitivity),*
– $x \lhd y \in X^\dagger$ *and* $yz \in X^\dagger \Rightarrow xz \lhd yz \in X^\dagger$ *($\lhd$-propagation),*
– $\mathrm{Z}x \in X^\dagger$ *and* $y \leq x \Rightarrow \mathrm{Z}y \in X^\dagger$ *(Z-decomposition),*
– $\mathrm{U}x \in X^\dagger$ *and* $y \leq x$ *and* $\mathrm{U}y \in X^\dagger \Rightarrow \mathrm{Z}(x-y) \in X^\dagger$ *(U-decomposition),*
– $\mathrm{T}x \in X^\dagger$ *and* $(y \lhd x$ *or* $x \lhd y) \in X^\dagger \Rightarrow \mathrm{T}y \in X^\dagger$ *(T-propagation).*

The reflexivity, transitivity and $\lhd$-propagation conditions simply capture the fact that in PL, the validity relation satisfies $\phi \models \phi$, $\phi \models \psi$ and $\psi \models \chi$ imply $\phi \models \chi$, and $\phi \models \psi$ implies $\phi * \chi \models \psi * \chi$. The Z-decomposition condition reflects that all sub-heaps of an empty heap are empty. The U-decomposition condition reflects that if x stands for a heap having exactly one location and if y represents a sub-heap with only one location also, then any sub-heap of x disjoint from y must be empty. Finally, the T-propagation condition simply explains how the properties of being an empty heap, a one-location heap, or an aggregate heap are propagated through heap composition. For example, let us set $\mathrm{T} = \mathrm{Z}$, $x = c_1$ and $y = c_2c_3$, then, the condition means that if c_1 represents an empty heap ($\mathrm{Z}c_1$) which may be obtained by composition of the two heaps c_2 and c_3 ($c_2c_3 \lhd c_1$), then c_2c_3 also represents an empty heap ($\mathrm{Z}c_2c_3$).

3.2 Resource graphs

In this subsection, we explain how labels and constraints give rise to a specific semantic structure, called a *resource graph*, which is a graphical representation of the $(\cdot)^\dagger$-closure operator.

Definition 3.4. *Let X be set of labels and constraints, the* resource graph *associated to X and denoted $\mathcal{G}(X)$ is a directed graph $[N, E]$.*
The set of nodes *N is derived from the labels occurring in $X^\dagger$ (the domain of X) by decorating each label x with a* tag Γ *which is a (possibly empty) subset of* $\{\mathrm{Z}, \mathrm{U}, \mathrm{A}\}$ *such that, for any* $\mathrm{T} \in \{\mathrm{Z}, \mathrm{U}, \mathrm{A}\}$*, if $X \vdash \mathrm{T}x$ then* $\mathrm{T} \in \Gamma$.
The set of arrows *E is such that there is an arrow $\Gamma x \twoheadrightarrow \Delta y$ from the node Γx to the node Δy iff the constraint $x \lhd y$ holds in X,* i.e.*, $X \vdash x \lhd y$.*

Let us illustrate the previous definition with some examples. Firstly, we consider the set $X = \{\mathrm{Z}c_0, c_2c_3 \lhd c_1, \mathrm{U}c_2, \mathrm{U}c_2c_3\}$. In order to obtain $X^\dagger$, we must add the constraint $\mathrm{Z}c_3$ to satisfy the *(U-decomposition)* of Definition 3.3 since we have $\mathrm{U}c_2$, $\mathrm{U}c_2c_3$ and

$c_3 \leq c_2c_3$. Furthermore, due to the presence of $c_2c_3 \lhd c_1$ and $\mathrm{U}c_2c_3$, *(*U*-propagation)* leads to the addition of $\mathrm{U}c_1$ so that $X^\dagger = \{\mathrm{Z}c_0, c_2c_3 \lhd c_1, \mathrm{U}c_2, \mathrm{U}c_2c_3, \mathrm{Z}c_3, \mathrm{U}c_1\}$. Secondly, beginning with the set $Y = \{\mathrm{Z}c_0, c_2c_3 \lhd c_1, \mathrm{U}c_2, \mathrm{U}c_3, \mathrm{Z}c_1\}$, the application of *(*Z*-propagation)* and *(*Z*-decomposition)* on the constraints $c_2c_3 \lhd c_1$ and $\mathrm{Z}c_1$ lead to $Y^\dagger = \{\mathrm{Z}c_0, c_2c_3 \lhd c_1, \mathrm{U}c_2, \mathrm{U}c_3, \mathrm{Z}c_1, \mathrm{Z}c_2c_3, \mathrm{Z}c_2, \mathrm{Z}c_3\}$. The resource graphs $\mathcal{G}_0(X)$ and $\mathcal{G}_0(Y)$, respectively associated to X and Y, then look as follows [2]:

$$\mathcal{G}_0(X): \begin{tikzcd} \boxed{\mathrm{Z}c_0} & \boxed{\mathrm{Z}c_3} & \boxed{\mathrm{U}c_1} & \boxed{\mathrm{U}c_2} \\ & & \boxed{\mathrm{U}c_2c_3} \arrow[u] & \end{tikzcd} \qquad \mathcal{G}_0(Y): \begin{tikzcd} \boxed{\mathrm{Z}c_0} & \boxed{\mathrm{ZU}c_3} & \boxed{\mathrm{Z}c_1} & \boxed{\mathrm{ZU}c_2} \\ & & \boxed{\mathrm{Z}c_2c_3} \arrow[u] & \end{tikzcd}$$

3.3 Normalizing resource graphs

In order to reduce the size of the resource graphs, we need to remove nodes that are semantically redundant. For that, we define a normalization process on resource graphs that takes into account that there is only one empty heap which is moreover the unit of heap composition. Therefore, we only keep the node $\mathrm{Z}c_0$ to represent the empty heap and discard all other nodes of the form $\mathrm{Z}x$. Let $\mathcal{G}(X) = [N, E]$ be the resource graph associated with a set X of labels and constraints. Labels $x \in \mathcal{D}(X)$ such that $X \vdash \mathrm{Z}x$ are called Z-labels, or Z-constants when x is a constant symbol. Since such labels are assumed to behave as units w.r.t. label composition $\circ$, we define an equivalence relation $\simeq$ on X such that $x \simeq y$ iff $(\forall c_i)(X \nvdash \mathrm{Z}c_i \Rightarrow (c_i \in y \Leftrightarrow c_i \in x))$, *i.e.*, the labels x and y are equivalent upto $\simeq$ if they have the same non-zero constants. We write $X \vdash x \simeq y$ to mean that $x \simeq y$ holds in X. In terms of the resource graph $\mathcal{G}(X)$, $\Gamma x \simeq \Delta y$ holds in $\mathcal{G}(X)$, written $\mathcal{G}(X) \vdash \Gamma x \simeq \Delta y$, iff $x \simeq y$ holds in X[3]. For example, $\mathrm{U}c_2c_3 \simeq \mathrm{U}c_2$ holds in $\mathcal{G}_0(X)$ because $\mathrm{Z}c_3$ holds in $\mathcal{G}_0(X)$.

In order to put a resource graph $\mathcal{G}(X) = [N, E]$ in normal form, the first step is to gather all labels that are equivalent upto $\simeq$, *i.e.*, for all labels x, its equivalence class modulo $\simeq$ is given by $\overline{x} = \{y \mid X \vdash x \simeq y\}$. For the equivalence classes of labels modulo $\simeq$ to give rise to the nodes $\overline{N}$ of a resource graph in normal form, we need to decorate them with appropriate tags. Therefore, the second step of the normalization process is to merge the tags associated to the labels populating an equivalence class using set union. Thus, for all classes $\overline{x}$, we compute the set $\varphi = \bigcup_{\Delta y \in N, y \in \overline{x}} \Delta$. The tag associated to $\overline{x}$, denoted Φ, is then obtained from φ by discarding the A letter whenever the U letter is already a member of φ, *i.e.*, $\Phi = \varphi - \{\mathrm{A}\}$ if $\{\mathrm{U}, \mathrm{A}\} \subseteq \phi$ and $\Phi = \varphi$ otherwise [4]. Finally, in order to give rise to the edges $\overline{E}$ of the normal resource graph associated to $\mathcal{G}(X)[N, E]$, we add an arrow $\Phi\overline{x} \twoheadrightarrow \Psi\overline{y}$ going from a node $\Phi\overline{x}$ to a node $\Psi\overline{y}$ iff $\Gamma u \twoheadrightarrow \Delta v \in E$ for some $u \in \overline{x}$ and $v \in \overline{y}$. Keeping the label with a minimum length as the

---

[2] For simplicity, we do not explicitly represent reflexive and transitive arrows.

[3] In the rest of the paper, we define relations α either in terms of labels, or in terms of nodes, the link between the two forms is such that $X \vdash x \,\alpha\, y$ iff $\mathcal{G}(X) \vdash \Gamma x \,\alpha\, \Delta y$.

[4] This is semantically justified by the fact that it is no use knowing that a heap contains at least one cell when one already knows it contains exactly one.

witness for its equivalence class modulo $\simeq$, the previous examples lead to the following normal resource graphs:

$$\begin{tikzcd}
\mathcal{G}_n(X): & \boxed{\mathrm{Z}c_0} & \boxed{\mathrm{U}c_1} & \mathcal{G}_n(Y): & \boxed{\mathrm{ZU}c_0} \\
& & \boxed{\mathrm{U}c_2} \arrow[u] & &
\end{tikzcd}$$

3.4 Points-to predicate distributions

Given a resource graph $\mathcal{G}(X) = [N, E]$, we consider distributions of points-to predicates over nodes of the form Ux. For example, in the previous resource graph $\mathcal{G}_n(X)$, we associate $(l \mapsto a, b)$ and $(k \mapsto c, d)$ to the node Uc_2. Such a distribution is denoted as follows: $Pto(\mathrm{U}c_2) = \{(l \mapsto a, b), (k \mapsto c, d)\}$.

$$\begin{tikzcd}
\mathcal{G}_n(X): & \boxed{\mathrm{Z}c_0} & \boxed{\mathrm{U}c_1} & & \mathcal{G}_n(Y): & \boxed{\mathrm{ZU}c_0} & \begin{array}{l} k \mapsto c, d \\ l \mapsto a, b \end{array} \\
& & \boxed{\mathrm{U}c_2} \arrow[u] & \begin{array}{l} k \mapsto c, d \\ l \mapsto a, b \end{array} & & &
\end{tikzcd}$$

Let us define the new relation $\approx$ as the smallest equivalence on X generated by $(\lhd \cup \simeq)$. Then, in order to keep track of which locations are defined in a given heap, we associate *multisets* of points-to predicates called *loc-sets* to the nodes of a resource graph. The loc-set $Loc(\Gamma x)$ associated to a node Γx is such that $Pto(\Gamma x) \subseteq Loc(\Gamma x)$ and if $\mathcal{G}(X) \vdash \Gamma x \approx \Delta yz$ for some $\Sigma_1 y$, $\Sigma_2 z \in N$, then $Loc(\Sigma_1 y) \cup Loc(\Sigma_2 z) \subseteq Loc(\Gamma x)$. In other words, loc-sets inherit points-to predicates via label composition and arrows of the resource graphs. Finally, we say that $Loc(\Gamma x)$ and $Loc(\Delta y)$ are *compatible*, denoted $Loc(\Gamma x) \# Loc(\Delta y)$, or more shortly $x \# y$, if they share no location l.
Let us illustrate the previous notions with the following resource graph:

$$\begin{tikzcd}
\begin{array}{l} l \mapsto a, b \\ k \mapsto c, d \end{array} & \boxed{\Delta x} & \begin{array}{l} l \mapsto a, b \\ k \mapsto e, f \end{array} \\
\boxed{\mathrm{U}y} & \boxed{\Gamma yz} \arrow[u] & \boxed{\mathrm{U}z}
\end{tikzcd}$$

Here, the points-to distribution is such that $Pto(\mathrm{U}y) = \{(l \mapsto a, b), (k \mapsto c, d)\}$ and $Pto(\mathrm{U}z) = \{(l \mapsto a, b), (k \mapsto e, f)\}$. The loc-set associated with the node Δx is such that $Loc(\Delta x) = \{(l \mapsto a, b), (l \mapsto a, b), (k \mapsto c, d), (k \mapsto e, f)\}$. Notice that $(l \mapsto a, b)$ occurs twice in $Loc(\Delta x)$, the first occurence coming from $Pto(\mathrm{U}y)$ and the second coming from $Pto(\mathrm{U}z)$. Therefore $Loc(\mathrm{U}y)$ and $Loc(\mathrm{U}z)$ are not compatible since they share the location l (and the location k too).

3.5 Structural consistency

Let us define the notion of a structural consistency for (normal) resource graphs, which intuitively means that a resource graph indeed represents a "real" model of PL. We first introduce the notion of *well-formed* resource graphs.

Definition 3.5. *A normal resource graph $\mathcal{G}(X) = [N,E]$ is* well-formed *iff for all $\Gamma x \in N$, the set Γ contains at most one element.*

Then, we introduce the notion of *complexity measure* on the nodes of a resource graph. The role of such a notion is to unambiguously determine how many cells are assumed to be in the heap represented by a given node. The information conveyed by the constraints of type $\mathrm{A}x$ is only a rough abstraction (at least one cell) that needs to be completed if we have to extract countermodels from resource graphs as explained later in Section 5.

Definition 3.6. *Given a well-formed normal resource graph $\mathcal{G}(X) = [N,E]$, a* complexity measure *on $\mathcal{G}(X)$ is a total function $Comp : N \to \mathbb{N}$ such that*

- $Comp(\mathrm{Z}x) = 0$, $Comp(\mathrm{U}x) = 1$, $Comp(\mathrm{A}x) \geq 1$,
- $Comp(\Gamma xy) = Comp(\Delta x) + Comp(\Sigma y)$,
- $\mathcal{G}(X) \vdash \Gamma x \approx \Delta y \Rightarrow Comp(\Gamma x) = Comp(\Delta y)$.

Definition 3.7. *Given a measure of complexity $Comp$ on a well-formed resource graph $\mathcal{G}(X)$ and a distribution Pto, $\mathcal{G}(X)$ is* structurally consistent *w.r.t. $(Pto, Comp)$ iff:*

– *$\forall \mathrm{Z}x \in N$, $Loc(\mathrm{Z}x)$ is the empty set (SC1),*
– *$\forall \mathrm{U}x \in N$, $Loc(\mathrm{U}x)$ is a singleton set (SC2),*
– *$\forall \Gamma x, \Delta y \in N$, $\Sigma xy \in N \Rightarrow Loc(\Gamma x) \# Loc(\Delta y)$ (SC3),*
– *$\forall \Gamma x$, $Cardinal(Loc(\Gamma x)) \leq Comp(\Gamma x)$ (SC4).*

$\mathcal{G}(X)$ is structurally consistent w.r.t. a distribution Pto iff there exists some complexity measure $Comp$ for which it is structurally consistent w.r.t. $(Pto, Comp)$.

The previous conditions simply reflect that a heap with no cell should have no location, that a heap with one cell should have exactly one location, that heaps must not share locations if they are to be composed and that there should not be more locations in a heap than there are cells. In the rest of the paper, since a label x uniquely determines a node Γx in a resource graph, we shall sometimes write $Loc(x)$ and $Comp(x)$ instead of $Loc(\Gamma x)$ and $Comp(\Gamma x)$.

4 Resource Graphs and Provability

In this section, we propose a tableau-based calculus for PL that builds resource graphs. The choice of a tableau proof-search method is motivated by its well-known ability to propose countermodel extraction facilities [2], but our notions of labels, constraints and resource graphs can be integrated to connection-based, or sequent-based calculi [3], in order to characterize provability in PL.

Definition 4.1. *A* signed formula *is an expression $\mathrm{S}\,\phi : x$, where $\mathrm{S} \in \{\mathrm{F}, \mathrm{T}\}$ is a sign, ϕ is a $\mathsf{PL}^{\mapsto}$ formula and x is a label.*

Definition 4.2. *Let χ be a formula in $\mathsf{PL}^{\mapsto}$.* A tableau for χ *is a binary tree $\mathcal{T}$ whose root node is labelled with the signed formula $\mathrm{F}\,\chi : c_0$, all other nodes being either labelled with a signed formula, or with a constraint, and which is built (respecting the structure of χ) according to the rules depicted on Figure 1.*

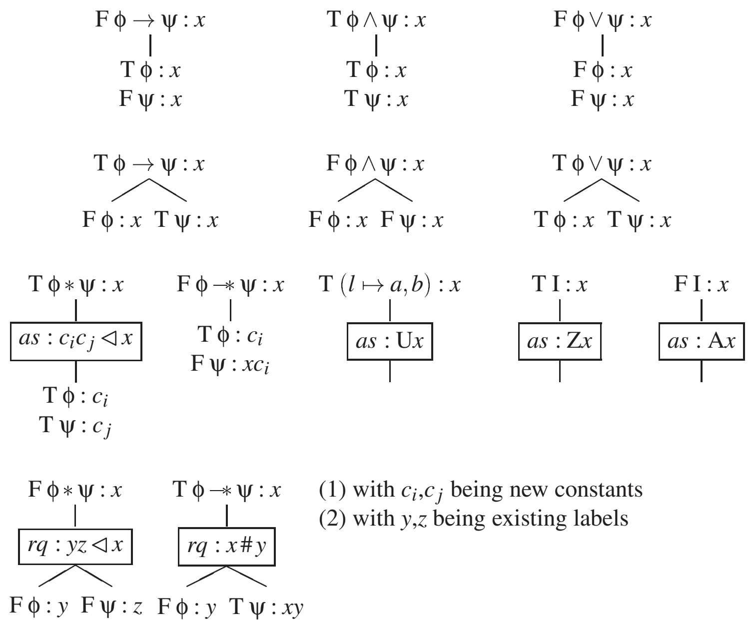

Figure1. Tableaux rules for $\mathsf{TPL}^{\mapsto}$

Given a branch $\mathcal{B}$, $As(\mathcal{B})$ is the set of its *assertions* and $Rq(\mathcal{B})$ the set of its *requirements*. The domain of a branch $\mathcal{B}$, denoted $\mathcal{D}(\mathcal{B})$, is the restriction of the set $As(\mathcal{B})^{\dagger}$ to its labels. We shall see that assertions behave as known facts while requirements behave as goals that must be achieved using assertions. The application of expansion rules of Figure 1 leads to the (incremental) construction of a resource graph, denoted $\mathcal{G}(\mathcal{B})$, that is induced by the $(\cdot)^{\dagger}$-closure of the assertions occuring in $\mathcal{B}$. Moreover, the points-to predicate distribution is given by the signed formulae of the branch with sign T, *i.e.*, $\mathcal{G}(\mathcal{B})$ is the resource graph $\mathcal{G}(As(\mathcal{B})^{\dagger})[N,E]$ such that, for all nodes $\mathrm{U}x \in N$, $Pto(\mathrm{U}x) = \{\, (l \mapsto a,b) \,|\, \mathrm{T}\,(l \mapsto a,b) : x \in \mathcal{B}\,\}$.

Having resource graphs associated to tableau branches and built from the assertions we relate them to the requirements generated by the $\mathrm{F}\,\phi * \psi : x$ and $\mathrm{T}\,\phi -\!\!* \psi : x$ signed formulae. This leads to the notion of *admissible tableau*.

Definition 4.3. *A requirement is* admissible *in a branch $\mathcal{B}$ of a tableau $\mathcal{T}$ if it holds in the $((\cdot)^{\dagger}$-closure of the) assertions that occur (in $\mathcal{B}$) closer to the root of $\mathcal{T}$ than this requirement. A branch $\mathcal{B}$ is* admissible *if all its requirements are admissible in $\mathcal{B}$ and a tableau $\mathcal{T}$ is admissible if all its branches are admissible.*

Before we proceed with the notion of *logical consistency*, which intuitively means that a formula of $\mathsf{PL}^{\mapsto}$ can be falsified, we need to introduce an equivalence relation on the nodes of a resource graph. Given a complexity measure *Comp* on the resource graph $\mathcal{G}(\mathcal{B})$ of a branch $\mathcal{B}$, we say that the relation $\Gamma x \sim \Delta y$ holds in $\mathcal{G}(\mathcal{B})$, written $\mathcal{G}(\mathcal{B}) \vdash \Gamma x \sim \Delta y$ (or more shortly $\mathcal{B} \vdash x \sim y$), iff either $\mathcal{G}(\mathcal{B}) \vdash \Gamma x \approx \Delta y$ or $Comp(\Gamma x) =$

$$\sqrt{}_1\ \mathrm{F}\ ((l \mapsto a,b) * ((l \mapsto a,b) \multimap (k \mapsto c,d))) \multimap (h \mapsto e,f) : c_0$$

$$\sqrt{}_2\ \mathrm{T}\ (l \mapsto a,b) * ((l \mapsto a,b) \multimap (k \mapsto c,d)) : c_1$$

$$\mathrm{F}\ (h \mapsto e,f) : c_1 (= c_0 c_1)$$

$$\boxed{as_1 : c_2 c_3 \lhd c_1}$$

$$\mathrm{T}\ (l \mapsto a,b) : c_2$$

$$\sqrt{}_3\ \mathrm{T}\ (l \mapsto a,b) \multimap (k \mapsto c,d) : c_3$$

$$\begin{array}{cc} \mathrm{F}\ (l \mapsto a,b) : c_2 & \mathrm{T}\ (k \mapsto c,d) : c_2 c_3 \\ \times & \times \end{array}$$

Figure2. Tableau for $((l \mapsto a,b) * ((l \mapsto a,b) \multimap (k \mapsto c,d))) \multimap (h \mapsto e,f)$.

$Comp(\Delta y)$, $Loc(\Gamma x) = Loc(\Delta y)$ and $Cardinal(Loc(\Gamma x)) = Comp(\Gamma x)$. In other words, two nodes are equivalent upto $\sim$ if they represent heaps having the same cells (with the same content) and the same locations.

Definition 4.4. *A branch $\mathcal{B}$ is* logically inconsistent *w.r.t. to a complexity measure Comp on $\mathcal{G}(\mathcal{B})$ if it satisfies one of the following conditions:*

– $\mathrm{T}\ (l \mapsto a,b) : x$, $\mathrm{F}\ (l \mapsto a,b) : y \in \mathcal{B}$ *and* $\mathcal{B} \vdash x \sim y$ *(CL1);*
– $\mathrm{F}\ \mathrm{I} : x \in \mathcal{B}$ *and* $Comp(\Gamma x) = 0$ *(CL2);*
– $\mathrm{F}\ \top : x \in \mathcal{B}$ *(CL3);*
– $\mathrm{T}\ \bot : x \in \mathcal{B}$ *(CL4).*

A tableau is logically inconsistent *w.r.t. Comp if it has a branch that is logically inconsistent w.r.t. Comp.*

Definition 4.5. *A branch $\mathcal{B}$ is* open *if it is logically consistent and if its resource graph $\mathcal{G}(\mathcal{B})$ is structurally consistent w.r.t. some complexity measure Comp on $\mathcal{G}(\mathcal{B})$, otherwise it is* closed*. A tableau $\mathcal{T}$ is open if it has an open branch.*

Definition 4.6. *Let ϕ be a formula of $\mathsf{PL}^{\mapsto}$, a tableau $\mathcal{T}$ is a* TPL*-proof of ϕ if there is a sequence of tableaux $(\mathcal{T}_i)_{1 \le i \le n}$ such that*
1. $\mathcal{T}_1$ is the one-node tableau the root of which is labelled with $\mathrm{F}\ \phi : c_0$,
2. $\mathcal{T}_{i+1}$ is obtained from $\mathcal{T}_i$ by a decomposition rule of Figure 1,
3. $\mathcal{T}_n = \mathcal{T}$, $\mathcal{T}$ is closed and admissible.
A formula ϕ is provable in TPL *if there exists a* TPL*-proof of ϕ.*

Let us illustrate these closure conditions. The tableau of Figure 2 has a left branch which is not logically consistent (condition *(CL1)* on $(l \mapsto a,b)$ and c_2). In the right branch, we have to capture the semantics of $\mapsto$. For instance, if a heap forces $(l \mapsto a,b)$ then this heap has only one cell that is at location l and contains the values $\langle a,b \rangle$. The resource graph associated to this branch is $\mathcal{G}_0(X)$, given in subsection 3.2, that after normalization is not structurally consistent because the loc-set $Loc(\mathrm{U}c_2) =$

$\{(l \mapsto a,b),(k \mapsto c,d)\}$ is not a singleton set as required by condition *(SC2)* of Definition 3.7. Then this branch is closed (marked with a $\times$).

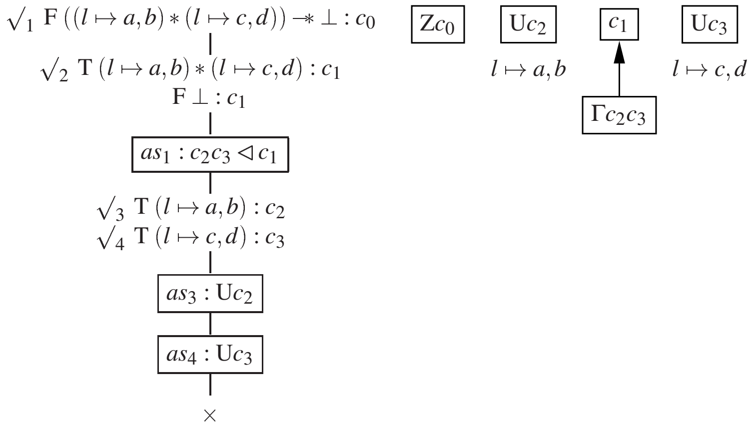

Figure3. Tableau for $((l \mapsto a,b) * (l \mapsto c,d)) -\!\!* \perp$.

Figure 3 gives a tableau for the formula $((l \mapsto a,b) * (l \mapsto c,d)) -\!\!* \perp$ that means that we cannot have two cells in a heap that are at the same location. We could deduce the existence of two heaps (represented by c_2 and c_3) that respectively force $(l \mapsto a,b)$ and $(l \mapsto c,d)$, but then their composition (represented by c_2c_3) is not defined since they are not disjoint and thus cannot force $(l \mapsto a,b) * (l \mapsto c,d)$. We observe that the associated resource graph in normalized form is not structurally consistent since $Loc(\mathrm{U}c_2) = \{(l \mapsto a,b)\}$ and $Loc(\mathrm{U}c_3) = \{(l \mapsto c,d)\}$, so that $Loc(c_2c_3) = Loc(c_1) = \{(l \mapsto a,b),(l \mapsto c,d)\}$, which contradicts condition *(SC3)* of Definition 3.7 because the loc-sets $Loc(\mathrm{U}c_2)$ and $Loc(\mathrm{U}c_3)$ are not compatible (they share location l). Then this branch is closed.

5 Countermodel Construction and Completeness

Before we study the completeness we prove the soundness of TPL by introducing the size of a heap h, denoted $Size(h)$, that is the number of locations it contains.

Definition 5.1 (realization). *Let $\mathcal{B}$ be a branch. A* realization of $\mathcal{B}$ *is a mapping* $||-|| : \mathcal{D}(\mathcal{B}) \to Heaps$ *that satisfies:*
– $||x \circ y|| = ||x|| \cdot ||y||$, $\mathcal{B} \vdash \mathrm{Z}x \Rightarrow ||x|| = e$,
– $\mathcal{B} \vdash \mathrm{U}x \Rightarrow Size(||x||) = 1$, $\mathcal{B} \vdash \mathrm{A}x \Rightarrow Size(||x||) \geq 1$,
– $yz \lhd x \in As(\mathcal{B}) \Rightarrow ||x|| \cdot ||y|| = ||z||$, $\mathcal{B} \vdash x \approx y \Rightarrow ||x|| = ||y||$,
– $\mathrm{T}\,\phi : x \in \mathcal{B} \Rightarrow ||x|| \models \phi$ *and* $\mathrm{F}\,\phi : x \in \mathcal{B} \Rightarrow ||x|| \not\models \phi$.
A branch $\mathcal{B}$ is realizable *if there exists a realization of $\mathcal{B}$. A tableau $\mathcal{T}$ is* realizable *if it contains a realizable branch.*

Lemma 5.1. *A closed tableau is not realizable.*

Proof. By analysis of the $(\cdot)^{\dagger}$-closure rules.

Lemma 5.2. *The rules of* TPL *preserve realizability.*

Proof. We show that, for all realizable tableaux $\mathcal{T}$, if $\mathcal{T}'$ is a tableau obtained from $\mathcal{T}$ by application of a TPL rule, then $\mathcal{T}'$ is realizable.

Theorem 5.1 (soundness). *Let* ϕ *be a formula of* $\mathsf{PL}^{\mapsto}$. *If there exists a* TPL-*proof of* ϕ *then* ϕ *is valid in* $\mathsf{PL}^{\mapsto}$ *semantics.*

Proof. Let $(\mathcal{T}_i)_{1\leq i\leq n}$ be a TPL-proof of ϕ. Suppose that ϕ does not hold in $\mathsf{PL}^{\mapsto}$ semantics, then $e \not\models \phi$. Consequently, $\|c_0\| = e$ is a trivial realization of $\mathcal{T}_0$. Lemma 5.2 then entails that all tableaux in $(\mathcal{T}_i)_{1\leq i\leq n}$ are realizable. This is a contradiction because, by definition of a TPL-proof, $\mathcal{T}_n$ is closed, which implies, by Lemma 5.1, that $\mathcal{T}_n$ is not realizable.

Definition 5.2. *A signed formula* $\mathrm{S}\,\phi : x$ *is* analyzed *in a branch* $\mathcal{B}$, *denoted* $\mathcal{B} \succ \mathrm{S}\,\phi : x$, *iff* $\mathrm{S}\,\phi : y \in \mathcal{B}$ *for some label* y *such that* $\mathcal{B} \vdash x \approx y$.

Definition 5.3. *A signed formula* $\mathrm{S}\,\phi : x$ *is* completely analyzed *or* fulfilled *in a branch* $\mathcal{B}$, *denoted* $\mathcal{B} \Vdash \mathrm{S}\,\phi : x$, *if it matches one of the following cases:*

- $\mathcal{B} \Vdash \mathrm{T\,I} : x$ *iff* $\mathcal{B} \succ \mathrm{T\,I} : x$ *and* $\mathcal{B} \vdash \mathrm{Z}x$,
- $\mathcal{B} \Vdash \mathrm{F\,I} : x$ *iff* $\mathcal{B} \succ \mathrm{F\,I} : x$ *and* $\mathcal{B} \vdash \mathrm{A}x$,
- $\mathcal{B} \Vdash \mathrm{T}\,(l \mapsto a,b) : x$ *iff* $\mathcal{B} \vdash \mathrm{U}x$,
- $\mathcal{B} \Vdash \mathrm{F}\,\psi \wedge \chi : x$ *iff* $\mathcal{B} \succ \mathrm{F}\,\psi : x$ *or* $\mathcal{B} \succ \mathrm{F}\,\chi : x$,
- $\mathcal{B} \Vdash \mathrm{T}\,\psi \wedge \chi : x$ *iff* $\mathcal{B} \succ \mathrm{T}\,\psi : x$ *and* $\mathcal{B} \succ \mathrm{T}\,\chi : x$,
- $\mathcal{B} \Vdash \mathrm{F}\,\psi \vee \chi : x$ *iff* $\mathcal{B} \succ \mathrm{F}\,\psi : x$ *and* $\mathcal{B} \succ \mathrm{F}\,\chi : x$,
- $\mathcal{B} \Vdash \mathrm{T}\,\psi \vee \chi : x$ *iff* $\mathcal{B} \succ \mathrm{T}\,\psi : x$ *or* $\mathcal{B} \succ \mathrm{T}\,\chi : x$,
- $\mathcal{B} \Vdash \mathrm{F}\,\psi \rightarrow \chi : x$ *iff* $\mathcal{B} \succ \mathrm{T}\,\psi : x$ *and* $\mathcal{B} \succ \mathrm{F}\,\chi : x$,
- $\mathcal{B} \Vdash \mathrm{T}\,\psi \rightarrow \chi : x$ *iff* $\mathcal{B} \succ \mathrm{F}\,\psi : x$ *or* $\mathcal{B} \succ \mathrm{T}\,\chi : x$,
- $\mathcal{B} \Vdash \mathrm{F}\,\psi * \chi : x$ *iff* $(\forall y,z \in \mathcal{D}(\mathcal{B}))(\mathcal{B} \vdash yz \approx x \Rightarrow (\mathcal{B} \succ \mathrm{F}\,\psi : y$ *or* $\mathcal{B} \succ \mathrm{F}\,\chi : z))$,
- $\mathcal{B} \Vdash \mathrm{T}\,\psi * \chi : x$ *iff* $(\exists y,z \in \mathcal{D}(\mathcal{B}))(\mathcal{B} \vdash yz \approx x$ *and* $\mathcal{B} \succ \mathrm{T}\,\psi : y$ *and* $\mathcal{B} \succ \mathrm{T}\,\chi : z)$,
- $\mathcal{B} \Vdash \mathrm{F}\,\psi \mathrel{-\!\!*} \chi : x$ *iff* $(\exists y \in \mathcal{D}(\mathcal{B}))(xy \in \mathcal{D}(\mathcal{B})$ *and* $\mathcal{B} \succ \mathrm{T}\,\psi : y$ *and* $\mathcal{B} \succ \mathrm{F}\,\chi : xy)$,
- $\mathcal{B} \Vdash \mathrm{T}\,\psi \mathrel{-\!\!*} \chi : x$ *iff* $(\forall y \in \mathcal{D}(\mathcal{B}))(xy \in \mathcal{D}(\mathcal{B}) \Rightarrow (\mathcal{B} \succ \mathrm{F}\,\psi : y$ *or* $\mathcal{B} \succ \mathrm{T}\,\chi : xy))$.
- *for all other cases,* $\mathcal{B} \Vdash \mathrm{S}\,\phi : x$ *iff* $\mathcal{B} \succ \mathrm{S}\,\phi : x$.

Definition 5.4. *A branch* $\mathcal{B}$ *is* complete *iff it is open and all signed formulae in* $\mathcal{B}$ *are fulfilled. A tableau* $\mathcal{T}$ *is* complete *iff it contains a complete branch.*

It is standard to define a tableau construction procedure that builds either a closed tableau or a complete tableau [2]. Let us now explain how to construct a countermodel from an complete branch using the tableau depicted on Figure 4. In this example, the resource graph induces a complexity mesure such that $Comp(\mathrm{U}c_3) = Comp(\mathrm{U}c_5) = 1$. On the other hand, we have $Loc(c_1) = Loc(c_2c_3) = Loc(c_4c_5) = \{\,(l \mapsto a,b),(k \mapsto a,b)\,\}$, which implies $Comp(c_1) = Comp(c_2c_3) = Comp(c_4c_5) \geq 2$.

$$\mathrm{F}\ (\top * (l \mapsto a,b)) \wedge (\top * (k \mapsto a,b)) \twoheadrightarrow (l \mapsto a,b) : c_0$$

$$\mathrm{T}\ (\top * (l \mapsto a,b)) \wedge (\top * (k \mapsto a,b)) : c_1$$
$$\mathrm{F}\ (l \mapsto a,b) : c_1$$

$$\mathrm{T}\ \top * (l \mapsto a,b) : c_1$$
$$\mathrm{T}\ \top * (k \mapsto a,b) : c_1$$

$$\boxed{as : c_2c_3 \lhd c_1}$$

$$\mathrm{T}\ \top : c_2$$
$$\mathrm{T}\ (l \mapsto a,b) : c_3$$

$$\boxed{as : c_4c_5 \lhd c_1}$$

$$\mathrm{T}\ \top : c_4$$
$$\mathrm{T}\ (k \mapsto a,b) : c_5$$

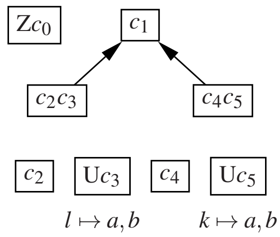

Figure4. Open tableau for $(\top * (l \mapsto a,b)) \wedge (\top * (k \mapsto a,b)) \twoheadrightarrow (l \mapsto a,b)$

After having solved the linear system of equations induced by the definition of a complexity measure, we can deduce $Comp(c_2) = Comp(c_4) = 1$. Moreover, we have $Loc(c_2) = Loc(c_4) = \emptyset$, so that $Cardinal(Loc(c_2)) = Cardinal(Loc(c_4)) = 0$. Then the branch is logically and structurally consistent. However, condition *(SC4)* of Definition 3.7 only requires $Cardinal(Loc(\Gamma x)) \leq Comp(\Gamma x)$ for all nodes Γx. Therefore, in order to obtain a countermodel, we need to find a completion of all loc-sets such that for all nodes Γx, $Cardinal(Loc(\Gamma x)) = Comp(\Gamma x)$.

A completion of all loc-sets can be achieved using set unification on the equations induced by the resource graph and the definition of *Loc*. Here, we obtain four equations:

- $Loc(c_2) \cup Loc(\mathrm{U}c_3) = Loc(c_2c_3) = \{(l \mapsto a,b),(k \mapsto a,b)\}$,
- $Loc(c_4) \cup Loc(\mathrm{U}c_5) = Loc(c_4c_5) = \{(l \mapsto a,b),(k \mapsto a,b)\}$,
- $Loc(\mathrm{U}c_3) = \{(l \mapsto a,b)\}$,
- $Loc(\mathrm{U}c_5) = \{(k \mapsto a,b)\}$.

From this, we get $Loc(c_2) = \{(k \mapsto a,b)\}$ and $Loc(c_4) = \{(l \mapsto a,b)\}$, so that all locations in $Loc(c_2)$ and $Loc(c_4)$ are fully determined by the system of *Loc* equations, but in the general case it is sometimes necessary to add special locations, not occurring in the resource graph and pointing to any content. For example, if we replace $(k \mapsto a,b)$ by $(l \mapsto a,b)$ in the formula of Figure 4, then we must complete $Loc(c_2)$ and $Loc(c_4)$ with a location that does not occur in the heap, for instance, $Loc(c_2) = Loc(c_4) = \{(m \mapsto ?,?)\}$. The question marks mean that the content of the cell associated to the additional location does not matter.

The last step of the countermodel construction process is to derive a partial monoid of heaps from the labels in the resource graph. We proceed as follows: for all nodes Γx, we define a heap $h_x : Loc \to Val \times Val$ such that $h_x(l) = \langle a,b \rangle$ iff $(l \mapsto a,b) \in Loc(\Gamma x)$. Then, we define $\mathcal{H} = (H, \cdot, h_{c_0})$ as the structure such that $H = \{h_x \mid \Gamma x \in \mathcal{G}(\mathcal{B})\}$, knowing that heap composition is given by the union of disjoint partial functions.

Lemma 5.3. *If $\mathcal{B}$ is a complete branch then $\mathcal{H} = (H, \cdot, h_{c_0})$ is a $\mathsf{PL}^{\mapsto}$-model such that a) if $\mathcal{B} \succ \mathrm{T}\,\phi : x$, then $(\exists \Gamma z \in \mathcal{G}(\mathcal{B}))(\mathcal{B} \vdash z \sim x$ and $h_z \models \phi)$ and b) if $\mathcal{B} \succ \mathrm{F}\,\phi : x$, then $(\exists \Gamma z \in \mathcal{G}(\mathcal{B}))(\mathcal{B} \vdash z \sim x$ and $h_z \not\models \phi)$.*

Proof. We show that condition *(SC1)* of Definition 3.7 implies that h_{c_0} is the empty heap and that condition *(SC3)* implies that whenever there exists Δz in the resource graph such that $\mathcal{B} \vdash z \sim xy$ then $h_x \cdot h_y = h_z$. The two properties a) and b) are deduced by induction on ϕ w.r.t. conditions of Definition 5.3.

Theorem 5.2 (completeness). *Let ϕ be a formula of $\mathsf{PL}^{\mapsto}$. If ϕ is valid in $\mathsf{PL}^{\mapsto}$ semantics, then there exists a TPL-proof of ϕ.*

Proof. Let ϕ be a valid $\mathsf{PL}^{\mapsto}$ formula. Suppose that ϕ has no TPL-proof, then, there exists no sequence of tableaux $(\mathcal{T}_i)_{1 \leq i \leq n}$ such that $\mathcal{T}_n$ is closed. Therefore, any (fair) tableau construction procedure results in a tableau containing a complete branch $\mathcal{B}$ from which we can build the structure $\mathcal{H} = (H, \cdot, h_{c_0})$. Since $\mathcal{B} \succ \mathrm{F}\,\phi : c_0$, Lemma 5.3 entails that $\mathcal{H}$ is a $\mathsf{PL}^{\mapsto}$-model such that $h_{c_0} \not\models \phi$, which contradicts the validity of ϕ.

6 Extension to PL

We can extend these results for $\mathsf{PL}^{\mapsto}$ in order to deal with the predicate $E_1 = E_2$ and the formula $\exists x.\phi$. Here, $\mapsto$ associates a cell to a stack variable and such variables are in the scope of quantifiers and then the formulae are closed.

In order to deal with $\exists$, we can use a standard technique that eliminates existential quantifiers by instantiating the variables with constants depending on the sign of the signed formulae [2]. When the sign is T one generates a new constant and when it is F we reuse a constant already generated. Here, the constants are locations and we need to memorize the variable instantiations for countermodel construction. The corresponding expansion rules look as follows:

$$
\begin{array}{cc}
* \,\mathrm{T}\, \exists x.\phi(x) : y & \mathrm{F}\, \exists x.\phi(v) : y \\
| & | \\
\mathrm{T}\, \phi(c) : y & \mathrm{F}\, \phi(l) : y
\end{array}
$$

*:c is a new constant.

Notice that the previous expansion rules do not concern labels because, from the semantics, we observe that we consider stack variables and not heaps that are the actual resources. Coming back to the example about double dispose in Section 2, we can use the tableau method to prove that $\perp \leftrightarrow \top * \exists\, a\, b.(x \mapsto a,b) * \exists\, c\, d.(x \mapsto c,d)$ and then to conclude that the program is not safe to execute for any start state.

Finally, in order to deal with the predicate $=$ we have the same rules but need to extend the tableau closure conditions. Having a signed formula $\mathrm{T}\, X = Y : x$ with constants X and Y syntactically equal, because of semantics $\mathrm{S}\, (X \mapsto a,b) : x$ is duplicated as $\mathrm{S}\, (Y \mapsto a,b) : x$. For $\mathrm{F}\, X = Y : x$, we need to check if constants X and Y are syntactically equal. In this case we have a contradiction with the semantics since the formula is signed by F.

7 Conclusions and Perspectives

Separation logics provide verification formalisms for pointer programs and allow to express properties about data structures with shared mutable state [6,12]. We study proof-theoretic foundations for such logics, by focusing first on PL, the BI's pointer logic [6]. We mainly define a characterization of provability in PL through so-called resource graphs and provide a new calculus with labels and constraints that builds resource graphs from which countermodels can be extracted. We expect to develop the same approach for the affine variant of PL [6] with intuitionistic additives that allows to prove interesting properties about sharing. Some spatial logics for trees or graphs [1] are related to PL as extensions of Boolean BI and we aim to study these logics from our proof-theoretic perspective. Moreover, comparisons with existing works on theorem proving dedicated to pointer programs [7,8,11] could be fruitful for some refinements.

References

1. L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *Int. Conference on Automata, Langages and Programming, ICALP'02, LNCS 2380*, pages 597–610, 2002.
2. M. Fitting. *First-Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer Verlag, 1990.
3. D. Galmiche and D. Méry. Connection-based proof search in propositional BI logic. In *18th Int. Conference on Automated Deduction, CADE-18, LNAI 2392*, pages 111–128, 2002. Copenhagen, Danemark.
4. D. Galmiche and D. Méry. Semantic labelled tableaux for propositional BI without bottom. *Journal of Logic and Computation*, 13(5):707–753, 2003.
5. D. Galmiche and D. Méry. Resource graphs and countermodels in resource logics. *Electronic Notes in Theoretical Computer Science*, 125(3):117–135, 2005.
6. S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM Symposium on Principles of Programming Languages, POPL 2001*, pages 14–26, London, UK, 2001.
7. J. Jenson, M. Jorgensen, N. Klarkund, and M. Schwartzback. Automatic verification of pointer programs using monadic second-order logic. In *Conf. on Programming Language Design and Implementation, PLDI'97*, pages 225–236, 1997.
8. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In *Int. Conference on Automated Deduction, CADE-19, LNCS 2741*, pages 121–135, Miami, USA, July 2003.
9. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th Int. Workshop on Computer Science Logic, CSL 2001, LNCS 2142*, pages 1–19, Paris, France, 2001.
10. P.W. O'Hearn and D. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.
11. S. Ranise and D. Deharbe. Applying light-weight theorem proving to debugging and verifying pointer programs. *Electronic Notes in Theoretical Computer Science*, 86(1), 2003.
12. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Danemark, July 2002.