Deriving a Complete Type Inference for Hindley-Milner and Vector Sizes using Expansion[☆]

Axel Simon[a]

[a] *Technische Universität München, Lehrstuhl für Informatik 2, Garching b. München, Germany*

Abstract

Type inference and program analysis both infer static properties about a program. Yet, they are constructed using very different techniques. We reconcile both approaches by deriving a type inference from a denotational semantics using abstract interpretation. We observe that completeness results in the abstract interpretation literature can be used to derive type inferences that are backward complete, a property akin to the inference of principal typings. The resulting algorithm is similar to that of Milner-Mycroft, that is, it infers Hindley-Milner types while allowing for polymorphic recursion. Although undecidable, we present a practical check that reliably distinguished typeable from untypeable programs. Instead of type schemes, we use *expansion* to instantiate types. Since our expansion operator is agnostic to the abstract domain, we are able to apply it not only to types. We illustrate this by inferring the size of vector types using systems of linear equalities and present practical uses of polymorphic recursion using vector types.

Keywords: type inference, abstract interpretation, principal typings, polymorphic recursion, expansion, complete analysis, vector size inference, affine relations

1. Introduction

New computer languages are continuously designed, often in the form of domain-specific languages (DSLs). One design aspect of a DSL is if the language should be statically typed and, if so, whether type inference should be used. Type inference is likely to be ruled out since addressing the special language features of a DSL in the type inference may turn into an open-ended research problem. We therefore propose to apply the abstract interpretation framework [2] to constructively derive a type inference algorithm. The motivation is that a variety of abstract domains have been developed over the years that can potentially be re-used for type inference, thereby going beyond Herbrand abstractions (type expressions containing type variables) commonly used in type inference. However, even if a set of domains are chosen and some type inference rules are put forth, different front-ends for the DSL might implement the type inference slightly differently, thus infer different types for the same input and thereby accept a different set of programs. In order to address this issue, we propose to require the type inference to be backward complete, meaning that it infers best (most general) types for every function and expression in the program. The benefit is that it is easy to decide if an implementation of the type inference is incorrect or imprecise (its inferred types are unsound or not the best) and that type annotations are not required since there is no need to refine a type that is best (although the DSL may allow them for documentation purposes). These qualities also apply to inferences that deliver *principal typings* [9], however, this notion requires that best types can be inferred in a bottom-up manner which is not possible for a Hindley-Milner type system [21]. We give a constructive way to create type inferences that infer best types by building on a result from the abstract interpretation literature [15] that shows that, once certain properties hold, a backward complete type inference can be derived by simply abstracting the semantics of the language. However, the presence of any branching construct (an **`if`** or a **`case`** statement) makes the derivation of a backward complete type inference impossible, as illustrated by the following example:

[☆] Supported by DFG Emmy Noether program SI 1579/1 and Microsoft SEIF 2013 grant. This work extends an earlier conference paper [18].
URL: `Axel.Simon@in.tum.de` (Axel Simon)

Example 1. *Consider the semantics of the Haskell expression* `if (f x) then 42 else []` *which is* $\bot$ *if* `(f x)` *does not terminate,* 42 *if it returns* `True` *or* `[]` *(an empty list) if it returns* `False`*. The corresponding type of these results are* a *(a type variable),* `Int` *or* [a]*. In case the evaluation of* `(f x)` *never terminates, then* a *is indeed the best type of the expression. Since determining whether* `(f x)` *may terminate is undecidable, the common approach in type inference is to ignore the outcome of the condition by assuming that both branches of the* `if`*-statement are taken. This is, in fact, an abstraction of the denotational semantics that our approach makes explicit.*

Thus, we derive a backward complete type inference with respect to a semantics that performs a non-deterministic choice between the two branches of a conditional. With respect to this slightly abstracted semantics, we constructively derive a backward complete type inference, meaning that it infers the best type within the universe of types. Deriving this backward complete inference requires an abstraction function α from program values to types that encodes the difference between **let** and λ-bound variables that the Hindley-Milner type system prescribes. Hence, the first contribution of this paper is this derivation including the abstraction map which, to our knowledge, is novel. Previous approaches [1] only used a concretization function γ that is insufficient to show backward completeness [15].

Our second contribution is to derive type inference rules for the special language features occurring in a DSL. In particular, we designed a language to specify instruction decoders [16] which are programs that turn a byte stream into processor instructions. These programs make heavy use of bit-vectors whose type is $|\mathrm{a}|$ where a is a type variable denoting the size of the vector. Consider the decoding of an Atmel AVR microcontroller instruction `ORI` (read "or with immediate") which has the bit pattern `0110kkkkddddkkkk`. Here, each k denotes one bit of a constant and d denotes a register. Note that the constant bits are not contiguous. Our DSL allows an instruction to be specified exactly as this bit pattern. For this to work, each `k` is in fact a function that reads one bit and appends it to a bit vector `ks` that is initially empty. Analogous for `d` and `ds`. These bit vectors are later used to construct the appropriate arguments of the instruction. Assuming that the bits read by the function `d` are `d0`, `d1`, `d2`, and `d3`, the following calculation is performed internally (here `''` denotes the empty bit-vector and `^` denotes the concatenation of bit vectors):

```
let ds0 = ''; ds1 = ds0 ^ d1; ds2 = ds1 ^ d2; ds3 = ds2 ^ d3; ds4 = ds3 ^ d4 in
case ds4 of '0000' -> R0; -- turn the bit vector into a datatype representing registers
            '0001' -> R1;
            ...
```

Since `ds4` is matched against several 4-bit vectors, it is clear that `ds4` : $|4|$. Furthermore, `ds0` : $|0|$. However, inferring the sizes of the other bit vectors requires the instantiation of the concatenation function `^` whose type can be given as $|\mathrm{a}| \to |\mathrm{b}| \to |\mathrm{c}|$ with the additional arithmetic constraint that $\mathrm{a} + \mathrm{b} = \mathrm{c}$. Intuitively, it seems clear that calculating an instance of this type, say $|d| \to |\mathrm{e}| \to |\mathrm{f}|$, also needs to duplicate the size information to $\mathrm{d} + \mathrm{e} = \mathrm{f}$. In general, this task might be more complicated if several equalities over $\mathrm{a}, \mathrm{b}, \mathrm{c}$ exist, so the question arises how calculating an instance can be performed in a principled manner. To this end, we observe that a relational *expand* operation [17] can be used for instantiating both, the type and the size information. Expansion replaces the syntactic operation of instantiating a type scheme with a semantic operation on the abstract domain of Herbrand abstractions (which is used to track the types of variables). Due to its semantic nature, we can derive how to apply expansion as part of deriving the type inference. While an expand operator has been proposed for instantiating types in System F [12], it was specific to types and its correctness was shown with respect to a universe of types containing type schemes. Our derived type inference is correct by construction [2], thereby making a proof with respect to inference rules using type schemes unnecessary. Analogously, the use of the expand operator on the domain of affine equations [10] can also be derived from the semantics, thereby suggesting that the expand operation can be useful when enhancing type systems with other abstract domains. Demonstrating the utility of expansion is our third contribution.

The derived type inference algorithm corresponds to the algorithm proposed by Mycroft [14], extended by the inference of vectors sizes. Mycroft's (and our) algorithm allows for polymorphic recursion, that is, given a definition $\mathbf{let}\ f\ x = e_1\ \mathbf{in}\ e_2$, the expression e_1 may call f with different types. While polymorphic recursion is hardly required in everyday programs, we show that it has interesting applications in a type system with vector sizes. Henglein showed that polymorphic recursion is undecidable [8] which translates into possible non-termination of our type inference. In practice, non-termination has only been observed for trivially incorrect functions [7, 8]. We present a novel test that recognizes this kind of recursion by turning them into an occurs-check. Our fourth contribution is thus to present a practical method for inferring polymorphic recursive types.

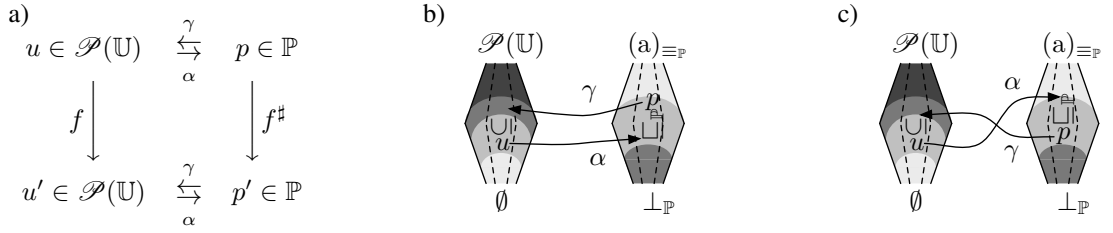

Figure 1: concrete/abstract transformers and monotone versus antitone abstraction

Overall, this paper makes the following four contributions:

- It observes that type inferences that are backward complete abstract interpretations exhibit the advantages associated with the inference of principal typings, but that they are more general. We derive a backward complete type inference for the Hindley-Milner type system for which no principal typings exist by calculating $f^\sharp = \alpha \circ f \circ \gamma$ where f is the denotational semantics of an ML-like language, lifted to sets and $f^\sharp$ is the corresponding type inference rule. We define the novel abstraction map α from value environments to Hindley-Milner type environments that encodes how λ- and **let**-bound variables differ in the way they are abstracted.
- We present a type inference for vector sizes that demonstrates how type inference can be extended by classic abstract domains found in program analysis.
- Our derivation instantiates function types using a domain-agnostic expand operator rather than type schemes. We illustrate the universality of this expand operator by extending the derived type system with the inference of vector sizes.
- We present a novel test that turns all non-terminating behaviors that we have observed into an occurs check, thereby presenting a practical approach to inferring polymorphic recursive types.

After a primer on abstractions to types, Sect. 3 defines the denotational semantics that Sect. 4 abstracts to types, yielding a type inference algorithm. Section 5 sketches the inference of bit vectors sizes and presents an example for polymorphic recursion (Sect. 5.2). Finally, Sect. 6 discusses our implementation and related work.

2. Overview of Abstracting into Types

We review the basics of backward completeness and how type inference can be described as an abstract interpretation. We then show how polymorphic types can be represented without type schemes.

2.1. Completeness in Abstract Interpretation

Figure 1a) shows how the abstract interpretation framework [2] proposes to relate some abstract property $p \in \mathbb{P}$ with several concrete properties $u \subseteq \mathbb{U}$ using an abstraction function $\alpha : \mathscr{P}(\mathbb{U}) \to \mathbb{P}$ and/or a concretization function $\gamma : \mathbb{P} \to \mathscr{P}(\mathbb{U})$. Given a complete partial order $\sqsubseteq_\mathbb{P}$ on $\mathbb{P}$, the tuple $\langle \alpha, \gamma \rangle$ forms a monotone Galois connection with $\alpha(u) \sqsubseteq_\mathbb{P} p$ iff $u \subseteq \gamma(p)$ as illustrated in Fig. 1b). We lift the program semantics that operates on the universe of values $\mathbb{U}$ to $\mathscr{P}(\mathbb{U})$ and use it as the collecting semantics f. A monotone abstract transfer function $f^\sharp : \mathbb{P} \to \mathbb{P}$ is a sound approximation of $f : \mathscr{P}(\mathbb{U}) \to \mathscr{P}(\mathbb{U})$ iff $\alpha \circ f \sqsubseteq_\mathbb{P} f^\sharp \circ \alpha$ where $(f \circ g) = \lambda x \,.\, f(g(x))$. If $\alpha \circ f = f^\sharp \circ \alpha$ then $f^\sharp$ is so-called backwards-complete [15] and the least fixpoint of $f^\sharp$ is the best possible ($\sqsubseteq_\mathbb{P}$-smallest) abstract property $p \in \mathbb{P}$, that is $\alpha(\mathrm{lfp}^{\subseteq}_{\emptyset}(f)) = \mathrm{lfp}^{\sqsubseteq_\mathbb{P}}_{\bot_\mathbb{P}}(f^\sharp)$ where $\mathrm{lfp}^{\leqslant}_{S}$ is the $\leq$-smallest fixpoint starting at S. Such an analysis is called (abstract) complete [5, 15]. This paper derives a type inference for Hindley-Milner types that is abstract complete by calculating $f^\sharp = \alpha \circ f \circ \gamma$ where f is the denotational semantics of an ML-like language, lifted to sets and $f^\sharp$ is the corresponding type inference rule. The so derived rules are backward complete if the additional property $\forall u_1, u_2 \in \mathscr{P}(\mathbb{U}) \,.\, \alpha(u_1) = \alpha(u_2) \Rightarrow \alpha(f(u_1)) = \alpha(f(u_2))$ holds [15]. The latter property states that two inputs u_1, u_2 to a semantic rule f have no effect on its resulting type $\alpha(f(u_i))$ as long as their type $\alpha(u_i)$ remains the same or, more intuitively, that the type system is not a dependent type system. The obtained backward complete inference rules

ensure that the inferred type for any program variable and expression is the most general type within the universe of types $\mathbb{P}$ which is often referred to as *principal typing* [9]. In our related work section, we will detail the difference between backward complete type inference and other definitions of principal typings [21].

2.2. Abstract Interpretation for Types

The abstract interpretation framework is usually used for *may*-analyses that express possible behaviors of a program. For instance, the abstract domain of intervals infers the *may* information $x \in [2, 5]$ when approximating the possible results of the Haskell expression `if (f x) then 2 else 5`. In contrast, type inferences are *must*-analyses that gather constraints that exclude certain program behaviors. For instance, inferring the type of the expression `if (f x) then '11' else '10101'` yields a type error since the bit vectors `'11'` and `'10101'` give rise to two constraints, namely that the expression *must* be of type $|2|$ and of type $|5|$, which is inconsistent. Note that both analyses may internally represent the states of the two branches with $a = 2$ and $a = 5$. However, when *may*-facts are inferred, the states are joined (yielding $[2, 5]$ for the interval domain) whereas in the case where *must*-facts are inferred, the states are intersected (leading to an inconsistent constraint system). In other words, the union of two program behaviors corresponds to an intersection in the abstract type domain. Analogously, the semantics of loops that is characterized by a least-fixpoint computation corresponds to a greatest-fixpoint computation on types. This is reflected by using an antitone Galois connection $\langle \alpha, \gamma \rangle$ where $p \sqsubseteq_{\mathbb{P}} \alpha(u)$ iff $u \subseteq \gamma(p)$ as illustrated in Fig. 1c). The antitone relation between concrete and abstract domain is chosen to reflect the difference between type systems and standard program analysis [1]. It is also possible to simply flip the abstract lattice upside down in which case the number of constraints grows with the number of program behaviors. This view has also been taken in the literature [14]. In this paper we chose the antitone view since commonly known abstract domains can be reused for type inference without flipping their join and meet operations. For instance, Herbrand structures that we use for types have previously been used to infer possible instantiations in Prolog programs [6]. Moreover, for vector sizes we use the affine domain [10] that has previously been used to infer equality relations between numeric program variables.

2.3. Representing Polymorphism without Type Schemes

Let $\mathbb{X}$ denote the set of program variables. For simplicity, we assume that variables at binding sites (λ- and $\mathbf{let}$-bindings) are pair-wise different. The Hindley-Milner type system assigns, to each variable $x \in \mathbb{X}$, a type term ranging over a binary function symbol $\cdot \to \cdot$, data constructors and type variables. Additionally, variables at $\mathbf{let}$-declarations are usually associated with a type scheme which states which type variables may be instantiated. Due to the ability to instantiate $\mathbf{let}$-bound variables, these are called "polymorphic" whereas λ-bound variables are called "monomorphic". However, this is overly simplistic, as illustrated by our running example, an elaborate identity function:

Example 2. *The program* $\mathbf{let}\ f = \lambda x\,.\,\mathbf{let}\ g = \lambda y\,.\,x\ \mathbf{in}\ g\,g\ \mathbf{in}\ f$ *yields the following type environment when analyzing* x *in* $\lambda y\,.\,x$*:*

$$f \mapsto \forall \mathrm{a}\,.\,\mathrm{a} \to \mathrm{a}, \quad x \mapsto \mathrm{a}, \quad g \mapsto \forall \mathrm{b}\,.\,\mathrm{b} \to \mathrm{a}, \quad y \mapsto \mathrm{b}$$

Here, the type variable a *is instantiated when using* f *(thus* f *is "polymorphic") but not instantiated when using* g*.*

Instead of type schemes that specify which type variables to instantiate, we will define an abstraction map $\alpha^{\mathcal{X}}_{\mathbb{M}}$ for whole environments in which all type variables are implicitly $\forall$-quantified, resulting in the following types for Ex. 2:

$$f \mapsto \mathrm{a} \to \mathrm{a}, \quad x \mapsto \mathrm{a}, \quad g \mapsto \mathrm{b} \to \mathrm{a}, \quad y \mapsto \mathrm{b}$$

As indicated in the example, using g requires that only the type variable b is instantiated to a fresh variable. In order to clarify how to instantiate the types above, we define a pre-order on $\mathbb{X}$, with $x \preccurlyeq y$ if x is in scope of y, that is, when y is bound by $\lambda y\,.\,e_1$ or by $\mathbf{let}\ y = e_2\ \mathbf{in}\ e_3$ then e_i may contain x. In the more general case of allowing binding *groups* of the form $\mathbf{let}\, x = \ldots; y = \ldots \mathbf{in} \ldots$, both $x \preccurlyeq y$ and $y \preccurlyeq x$ hold since they are visible in each other's body. Let $x \prec y$ abbreviate $y \not\preccurlyeq x \wedge x \neq y$. In the example, the ordering of the variables is $f \prec x \prec g \prec y$. Furthermore, let $\mathbb{X}^{\lambda} \subseteq \mathbb{X}$ denote the set of λ-bound variables, here $x, y \in \mathbb{X}^{\lambda}$, $f, g \in \mathbb{X} \setminus \mathbb{X}^{\lambda}$. Instantiating the type of $u \in \mathbb{X} \setminus \mathbb{X}^{\lambda}$ substitutes fresh variables for all variables in the type of u that do not also occur in $v \prec u$. The type schemes can thus be recovered through a traversal starting at the $\prec$-smallest variable and generating type schemes at each variable $u \in \mathbb{X} \setminus \mathbb{X}^{\lambda}$ for all as-of-yet unquantified variables. Note, though, that the analysis we derive never creates any type schemes and but computes which variables to instantiate on-the-fly using $\preccurlyeq$ and $\mathbb{X}^{\lambda}$.

This concludes the preliminaries for deriving a complete type inference. We now define the concrete semantics.

$$
\begin{array}{rcl}
\mathcal{S}[\![\cdot]\!] & : & \mathbb{E} \to (\mathbb{X} \to \mathbb{U}_\perp) \to \mathbb{U}_\perp \\
\mathcal{S}[\![x]\!]\ \rho & = & \rho(x) \\
\mathcal{S}[\![\lambda x\,.\,e]\!]\ \rho & = & \uparrow^{\mathbb{U}}_{\mathbb{F}}(\lambda v\,.\,\mathcal{S}[\![e]\!]\ (\rho[x \mapsto v])) \\
\mathcal{S}[\![e_1\,e_2]\!]\ \rho & = & (v_1 \trianglelefteq \mathbb{F} \stackrel{?}{:} (v_2 \trianglelefteq \mathbb{W} \stackrel{?}{:} \Omega \stackrel{:}{:} \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2) \stackrel{:}{:} \Omega) \text{ where } v_i = \mathcal{S}[\![e_i]\!]\ \rho \text{ for } i = 1,2 \\
\mathcal{S}[\![\mathbf{let}\ x = e\ \mathbf{in}\ e']\!]\ \rho & = & (v \trianglelefteq \mathbb{W} \stackrel{?}{:} \Omega \stackrel{:}{:} \mathcal{S}[\![e']\!]\ (\rho[x \mapsto v])) \text{ where } v = \mathrm{lfp}^{\preceq}_{\perp_{\mathbb{U}}} \lambda v\,.\,\mathcal{S}[\![e]\!]\ (\rho[x \mapsto v]) \\
\mathcal{S}[\![C\ e_1 \ldots e_n]\!]\ \rho & = & (v_1 \trianglelefteq \mathbb{W} \stackrel{?}{:} \Omega \stackrel{:}{:} \ldots (v_n \trianglelefteq \mathbb{W} \stackrel{?}{:} \Omega \stackrel{:}{:} \text{if } \delta_{\mathbb{U}}(C\,v_1 \ldots v_n) \neq \emptyset \\
 & & \text{then } \uparrow^{\mathbb{U}}_{\mathbb{A}}(C\ v_1 \ldots v_n) \text{ else } \Omega) \ldots) \qquad \text{where } v_i = \mathcal{S}[\![e_i]\!]\ \rho \text{ for } i \in [1,n] \\
\mathcal{S}[\![\mathbf{case}\ e_s\ \mathbf{of}\ C\ x_1 \ldots x_n : e_t; e_e]\!]\ \rho & = & (v_s \trianglelefteq \mathbb{A} \stackrel{?}{:} \text{if } \delta_{\mathbb{U}}(\downarrow^{\mathbb{U}}_{\mathbb{A}}(v_s)) \cap (\cup\{\delta_{\mathbb{U}}(C\,v'_1 \ldots v'_n) \mid v'_i \in \mathbb{U}\}) = \emptyset \text{ then } \Omega \text{ else} \\
 & & \text{if } \downarrow^{\mathbb{U}}_{\mathbb{A}}(v_s) = C\ v_1 \ldots v_n \text{ then } \mathcal{S}[\![e_t]\!]\ \rho[x_1 \mapsto v_1, \ldots x_n \mapsto v_n] \text{ else } \mathcal{S}[\![e_e]\!]\ \rho \\
 & & \stackrel{:}{:} \Omega) \qquad \text{where } v_s = \mathcal{S}[\![e_s]\!]\ \rho
\end{array}
$$

Figure 2: standard denotational semantics for $e \in \mathbb{E}$

3. Syntax and Semantics

Before we give a semantic meaning to types, we commence by defining the syntax and semantics of an extended λ-calculus. We mainly follow Milner [13] but add algebraic data types. Let $x \in \mathbb{X}$ and let $C \in \mathbb{C}$ be the n-ary constructor of some algebraic data type. The set of expressions $\mathbb{E}$ is given by the following grammar:

$$
\begin{array}{rcl}
e & \in & \mathbb{E} \\
e & ::= & x \mid \lambda x\,.\,e \mid e_1\,e_2 \mid \mathbf{let}\ x = e\ \mathbf{in}\ e' \\
 & \mid & C\ e_1 \ldots e_n \mid \mathbf{case}\ e_s\ \mathbf{of}\ C\ x_1 \ldots x_n : e_t; e_e
\end{array}
$$

Here, the standard constructs of the λ-calculus are extended by a **let**-construct that allows x to be **recursively** used in e (x is in scope in e). The second line defines values in form of constructors with n arguments and a test that matches the value of the scrutinee e_s against the constructor C and, if successful, evaluates the "then"-branch e_t in an environment where x_i is bound to the ith value of the constructor. If the match fails, the "else"-branch e_e is evaluated. For the sake of presentation, we set $\mathbb{C} = \{\mathbf{Cons}, \mathbf{Nil}\} \cup \mathbb{Z}$, for lists and integers.

Evaluating an expression may terminate with a type error. For instance, it is a type error if $e_1 \in \mathbb{E}$ in the application $e_1\,e_2$ does not evaluate to a λ-expression. In contrast, evaluating $\mathbf{let}\ f = \lambda x\,.\,f\ (\mathbf{Cons}\ 0\ x)\ \mathbf{in}\ f\,\mathbf{Nil}$ may result in a heap overflow error. The aim in type checking a program is to guarantee the absence of type errors but the ambition is not as high as proving the absence of other run-time errors. We now give a formal semantics to $e \in \mathbb{E}$.

3.1. Semantics

Let $S_\perp := S \cup \{\perp_S\}$ where $\perp_S \notin S$ denotes an undefined value or a non-terminating evaluation. The denotational standard semantics evaluates $e \in \mathbb{E}$ to a program value $\mathbb{U}_\perp$ where $\mathbb{U}$ is a sum of constructors terms $\mathbb{A}$, functions $\mathbb{F}$ and the set $\{\omega\}$ where ω, the wrong value, denotes a type error. It is defined as follows:

$$
\begin{array}{rcl}
\mathbb{U} & := & \mathbb{A} + \mathbb{F} + \mathbb{W} \\
\mathbb{F} & := & \mathbb{U} \to \mathbb{U}_\perp \\
\mathbb{A} & := & \{C\ v_1 \ldots v_n \mid v_i \in \mathbb{U}\} \\
\mathbb{W} & := & \{\omega\}
\end{array}
$$

We construct values of type $\mathbb{U}$ from a value in $\mathbb{D} = \mathbb{A}, \mathbb{F}, \mathbb{W}$ using the injection functions $\uparrow^{\mathbb{U}}_{\mathbb{D}}(\cdot) : \mathbb{D} \to \mathbb{U}$ (thus, $\uparrow^{\mathbb{U}}_{\mathbb{D}}(x)$ is analogous to a Haskell/ML constructor of the algebraic data type $\mathbb{U}$ that wraps $\mathbb{D}$). Conversely, define an extraction function $\downarrow^{\mathbb{U}}_{\mathbb{D}}(\cdot) : \mathbb{U}_\perp \to \mathbb{D}_\perp$ with $\downarrow^{\mathbb{U}}_{\mathbb{D}}(\uparrow^{\mathbb{U}}_{\mathbb{D}}(v)) = v$ for all $v \in \mathbb{D}$ and $\downarrow^{\mathbb{U}}_{\mathbb{D}}(\cdot) = \perp_{\mathbb{U}}$ otherwise. For brevity, let $\Omega = \uparrow^{\mathbb{U}}_{\mathbb{W}}(\omega) \in \mathbb{U}$ denote the error value. We define the following C-like conditional to test if $v \in \mathbb{U}$ originated in $\mathbb{D}$:

$$
(v \trianglelefteq \mathbb{D} \stackrel{?}{:} v_1 \stackrel{:}{:} v_2) = \begin{cases} v_1 & \text{if } v = \uparrow^{\mathbb{U}}_{\mathbb{D}}(d) \text{ for some } d \in \mathbb{D} \\ \perp_{\mathbb{U}} & \text{if } v = \perp_{\mathbb{U}} \\ v_2 & \text{if otherwise} \end{cases}
$$

$$
\begin{array}{rcl}
\mathcal{C}[\![\cdot]\!] & : & \mathbb{E} \to \mathscr{P}((\mathbb{X} \to \mathbb{U}_\bot) \to \mathbb{U}_\bot) \\
\mathcal{C}[\![x]\!] & = & \{\lambda\rho\,.\,\rho(x)\} \\
\mathcal{C}[\![\lambda x\,.\,e]\!] & = & \{\lambda\rho\,.\,\uparrow^{\mathbb{U}}_{\mathbb{F}}(\lambda v\,.\,S(\rho[x \mapsto v])) \mid S \in \mathcal{C}[\![e]\!]\} \\
\mathcal{C}[\![e_1\,e_2]\!] & = & \{\lambda\rho\,.\,(S_1\rho \subseteq \mathbb{F} \stackrel{?}{}\, (S_2\rho \subseteq \mathbb{W} \stackrel{?}{}\, \Omega \stackrel{:}{}\, \downarrow^{\mathbb{U}}_{\mathbb{F}}(S_1\rho)S_2\rho) \stackrel{:}{}\, \Omega) \mid S_i \in \mathcal{C}[\![e_i]\!], i = 1,2\} \\
\mathcal{C}[\![\mathbf{let}\ x = e\ \mathbf{in}\ e']\!] & = & \{\lambda\rho\,.\,(v \subseteq \mathbb{W} \stackrel{?}{}\, \Omega \stackrel{:}{}\, S'(\rho[x \mapsto v])) \mid S' \in \mathcal{C}[\![e']\!] \wedge v \in V\} \\
 & & \text{where } V = \{\mathrm{lfp}^{\preceq}_{\bot_{\mathbb{U}}} \lambda v\,.\,S(\rho[x \mapsto v]) \mid S \in \mathcal{C}[\![e]\!]\} \\
\mathcal{C}[\![C\ e_1 \ldots e_n]\!] & = & \{\lambda\rho\,.\,(S_1\rho \subseteq \mathbb{W} \stackrel{?}{}\, \Omega \stackrel{:}{}\, \ldots (S_n\rho \subseteq \mathbb{W} \stackrel{?}{}\, \Omega \stackrel{:}{}\, \text{if } \delta_{\mathbb{U}}(C\,S_1\rho \ldots S_n\rho) \neq \emptyset \\
 & & \text{then } \uparrow^{\mathbb{U}}_{\mathbb{A}}(C\ S_1\rho \ldots S_n\rho) \text{ else } \Omega) \ldots) \qquad \mid S_i \in \mathcal{C}[\![e_i]\!], i \in [1,n]\} \\
\mathcal{C}[\![\mathbf{case}\,e_s\,\mathbf{of}\,C\ x_1 \ldots x_n : e_t; e_e]\!] & := & \{\lambda\rho\,.\,\text{if } \nexists a\,.\uparrow^{\mathbb{U}}_{\mathbb{A}}(a) = S_s\rho \vee \delta_{\mathbb{U}}(\downarrow^{\mathbb{U}}_{\mathbb{A}}(S_s\rho)) \cap (\cup\{\delta_{\mathbb{U}}(C\,v'_1 \ldots v'_n) | v'_i \in \mathbb{U}\}) = \emptyset \\
 & & \text{then } \Omega \text{ else } S_t\rho[x_1 \mapsto v_1, \ldots x_n \mapsto v_n] \\
 & & \mid \delta_{\mathbb{U}}(\downarrow^{\mathbb{U}}_{\mathbb{A}}(S_s\rho)) \cap \delta_{\mathbb{U}}(C\,v_1 \ldots v_n) \neq \emptyset \wedge v_i \in \mathbb{U} \wedge S_s \in \mathcal{C}[\![e_s]\!] \wedge S_t \in \mathcal{C}[\![e_t]\!]\} \\
 & \cup & \{\lambda\rho\,.\,\text{if } \nexists a\,.\uparrow^{\mathbb{U}}_{\mathbb{A}}(a) = S_s\rho \vee \delta_{\mathbb{U}}(\downarrow^{\mathbb{U}}_{\mathbb{A}}(S_s\rho)) \cap (\cup\{\delta_{\mathbb{U}}(C\,v'_1 \ldots v'_n) | v'_i \in \mathbb{U}\}) = \emptyset \\
 & & \text{then } \Omega \text{ else } S_e\rho \qquad \mid S_s \in \mathcal{C}[\![e_s]\!] \wedge S_e \in \mathcal{C}[\![e_e]\!]\}
\end{array}
$$

Figure 3: collecting semantics for $e \in \mathbb{E}$, with **case**-statements being abstracted

The evaluation of an expression uses a value environment $\rho \in \mathbb{X} \to \mathbb{U}_\bot$ that binds program variables $\mathbb{X}$ to values $\mathbb{U}_\bot = \mathbb{U} \cup \{\bot_{\mathbb{P}}\}$. Choose $x, y \in \mathbf{dom}(\rho)$ such that $x \neq y$. An updated environment $\rho' = \rho[x \mapsto v] \in \mathbb{X} \to \mathbb{U}_\bot$ obeys $\rho'(x) = v$ and $\rho'(y) = \rho(y)$. A variable x is removed from ρ by projection, written $\rho' = \exists_x(\rho)$, where $\rho'(y) = \rho(y)$.

For the sake of defining the semantics of the language in a denotational way, we introduce fixpoint computations and an order on values. Given a complete partial order $\langle L, \sqsubseteq \rangle$ and a monotone function $\psi : L \to L$, define the least-fixpoint $\mathrm{lfp}^{\sqsubseteq}_{\bot}\psi$ to be the $\sqsubseteq$-smallest element $l \in L$ with $\bot \sqsubseteq l$ and $\psi(l) \sqsubseteq l$. Define the partial order $\preceq$ on values $v \in \mathbb{U}_\bot$ as follows: $\bot_{\mathbb{U}} \preceq v$ for all $v \in \mathbb{U}_\bot$. Moreover, $f_1 \preceq f_2$ if $f_i = \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_i) \neq \bot_{\mathbb{U}}$ for $i = 1, 2$ and $f_1(v) \preceq f_2(v)$ for all $v \in \mathbb{U}$. Also, $C\,v^1_1 \ldots v^1_n \preceq C\,v^2_1 \ldots v^2_n$ if $C\,v^i_1 \ldots v^i_n = \downarrow^{\mathbb{U}}_{\mathbb{A}}(v_i)$ for $i = 1, 2$ and $v^1_k \preceq v^2_k$ for all $k \in [1, n]$.

The denotational semantics $\mathcal{S}[\![e]\!] \in (\mathbb{X} \to \mathbb{U}_\bot) \to \mathbb{U}_\bot$ of an expression $e \in \mathbb{E}$ is defined in Fig 2. Here, the conditional statement is used in two ways: Firstly, $(v \subseteq \mathbb{D} \stackrel{?}{}\, \ldots \downarrow^{\mathbb{U}}_{\mathbb{D}}(v) \ldots \stackrel{:}{}\, \Omega)$ is used to test if $v \in \mathbb{U}$ is in fact element of $\mathbb{D}$ and then to extract this element $\downarrow^{\mathbb{U}}_{\mathbb{D}}(v) \in \mathbb{D}$ while returning the type error Ω otherwise. Secondly, if a value v is merely required not to be a type error, the statement $(v \subseteq \mathbb{W} \stackrel{?}{}\, \Omega \stackrel{:}{}\, \ldots)$ is used which returns Ω if v is a type error. Note that both conditionals evaluate to $\bot_{\mathbb{U}}$ if $v = \bot_{\mathbb{U}}$ and, hence, the definition of $e_1\,e_2$ returns $\bot_{\mathbb{U}}$ if e_2 is undefined which gives the language a call-by-value semantics.

User-defined data types are given by the function $\delta_{\mathbb{U}} : \mathbb{C} \times \mathbb{U}^* \to \mathscr{P}(\Delta)$ that maps a constructor and its arguments to a set of data types Δ, to be defined later. For instance, $\delta_{\mathbb{U}}(42) = \{\mathtt{Int}\}$, $\delta_{\mathbb{U}}(\mathbf{Nil}) = \{[\mathtt{Int}], [[\mathtt{Int}]], \ldots\}$ and $\delta_{\mathbb{U}}(\mathbf{Cons}\ \uparrow^{\mathbb{U}}_{\mathbb{A}}(42)\ \uparrow^{\mathbb{U}}_{\mathbb{A}}(\mathbf{Nil})\rangle) = \{[\mathtt{Int}]\}$. Applying a constructor $C \in \mathbb{C}$ to arguments $v_1, \ldots v_n$ is a type error if $\delta_{\mathbb{U}}(C\,v_1 \ldots v_n) = \emptyset$, that is, if no matching user-define data type exists. Analogously, for **case**-statements, the value v_s of the scrutinee e_s must not only be a constructor term $\mathbb{A}$ but also share possible data types $\delta_{\mathbb{U}}(\downarrow^{\mathbb{U}}_{\mathbb{A}}(v_s))$ with the constructor C. Here, $\bigcup\{\delta_{\mathbb{U}}(C\,v'_1 \ldots v'_n) \mid v'_i \in \mathbb{U}\}$ computes the possible data types that can be constructed using C.

3.2. Collecting Semantics

In program analysis, every abstract value (here: a type) represents many concrete values. In order to relate a type with a set of values, we lift the semantics $\mathcal{S}[\![e]\!] : (\mathbb{X} \to \mathbb{U}_\bot) \to \mathbb{U}_\bot$ of an expression $e \in \mathbb{E}$ to a collecting semantics $\mathcal{C}[\![e]\!] : \mathscr{P}((\mathbb{X} \to \mathbb{U}_\bot) \to \mathbb{U}_\bot)$. We lift each rule of the standard semantics in Fig. 3 to work over sets by converting each rule $\mathcal{S}[\![e]\!]\ \rho = \ldots \mathcal{S}[\![e']\!]\ \ldots$ to $\mathcal{C}[\![e]\!] = \{\lambda\rho.\ \ldots S \ldots \mid S \in \mathcal{C}[\![e']\!]\}$, resulting in the rules in Fig. 3. The only exception is the **case**-statement for which the collecting semantics is a proper abstraction of the standard semantics. Here, the collecting semantics returns two function sets, one that always returns e_t and one that always returns e_e. The test is further abstracted to ensure that the denotation does not depend on the actual value of e_s. Section 4.8 will illustrate that this abstraction is required to derive a complete abstraction to types. Note that the evaluation of e_t requires that the variables $x_1, \ldots x_n$ are bound to some value. We chose all values $C\,v_1 \ldots v_n$ that belong to the same datatype as the value of e_s. This concludes the discussion of syntax and semantics.

4. Abstraction to Types

This section defines types, abstractions for (environments of) values, and finally derives the type inference algorithm.

4.1. Monomorphic and Polymorphic Types

The underlying idea of types is to assign a type expression $t \in \mathbb{M}$ (or simply "type") to a value that is calculated in a program. Types in the context of the language $\mathbb{E}$ are defined as follows:

$$\begin{array}{rcll} D\ t_1 \ldots t_m & \in & \Delta & \text{algebraic data types} \\ t & \in & \mathbb{M} & \text{monomorphic type expressions} \\ t & ::= & t_1 \to t_2 \mid D\ t_1 \ldots t_m & \end{array}$$

Here, the types of values that are not functions are given by some user-defined set of algebraic data types $\Delta \subset \mathbb{M}$. For the sake of simplicity, we fix $\Delta = \{\texttt{Int}\} \cup \{[t] \mid t \in \mathbb{M}\}$. The definition of monotypes thus simplifies as follows:

$$\begin{array}{rcll} t & \in & \mathbb{M} & \text{monomorphic type expressions} \\ t & ::= & t_1 \to t_2 \mid \texttt{Int} \mid [t] & \end{array}$$

During inference, and later also for polymorphic types, it is necessary to allow type variables $v \in \mathbb{V}$ to occur in a type expression, giving rise to polymorphic type expressions:

$$\begin{array}{rcll} \mathrm{a}, \mathrm{b}, \ldots & \in & \mathbb{V} & \text{type variables} \\ t & \in & \mathbb{P} & \text{polymorphic type expressions} \\ t & ::= & \mathrm{a} \mid \mathrm{b} \mid \ldots \mid t_1 \to t_2 \mid \texttt{Int} \mid [t] & \end{array}$$

The purpose of type variables is to act as a placeholder for other type expressions. Replacing type variables in $t \in \mathbb{P}$ by another type expression creates an *instance* of t. The set of ground instances are those that contain no more type variables and is given by $\mathbf{ground}$:

$$\begin{array}{rcl} \mathbf{ground} & : & \mathbb{P} \to \mathscr{P}(\mathbb{M}) \\ \mathbf{ground}(t) & = & \{\sigma t \in \mathbb{M} \mid \sigma \in \mathbb{V} \hookrightarrow \mathbb{M}\} \end{array}$$

Here, $\sigma \in \mathbb{V} \hookrightarrow \mathbb{M}$ is a substitution that replaces type variables by monomorphic types. We define a partial order $\sqsubseteq_{\mathbb{P}}$ on $t_1, t_2 \in \mathbb{P}$ such that $t_1 \sqsubseteq_{\mathbb{P}} t_2$ iff $\mathbf{ground}(t_1) \subseteq \mathbf{ground}(t_2)$. Let $t_1 \equiv_{\mathbb{P}} t_2$ iff $\mathbf{ground}(t_1) = \mathbf{ground}(t_2)$, that is, t_1 is equal to t_2 modulus the renaming of type variables. We use $\mathbb{P}^{\perp}_{\equiv_{\mathbb{P}}}$ to denote the set of $\equiv_{\mathbb{P}}$-equivalence classes of terms augmented with $\perp_{\mathbb{P}}$. Then

$$\langle \mathbb{P}^{\perp}_{\equiv_{\mathbb{P}}}, \sqsubseteq_{\mathbb{P}}, \mathbf{lca}, \mathbf{gci}, (\mathrm{a})_{\equiv_{\mathbb{P}}}, \perp_{\mathbb{P}} \rangle$$

is a complete lattice with join $\mathbf{lca}$, meet $\mathbf{gci}$, top element $(\mathrm{a})_{\equiv_{\mathbb{P}}}$ and a bottom element $\perp_{\mathbb{P}}$ with $\perp_{\mathbb{P}} \sqsubseteq_{\mathbb{P}} t \sqsubseteq_{\mathbb{P}} (\mathrm{a})_{\equiv_{\mathbb{P}}}$ for all elements $t \in \mathbb{P}^{\perp}_{\equiv_{\mathbb{P}}}$. Here, $(t)_{\equiv_{\mathbb{P}}}$ denotes the $\equiv_{\mathbb{P}}$-equivalence class of $t \in \mathbb{P}$. The greatest common instance $t = \mathbf{gci}(t_1, t_2)$ is calculated using unification [11]: Rename the variables in t_i, giving t'_i, such that t'_1, t'_2 do not share any variables and $t'_i \equiv_{\mathbb{P}} t_i$ for $i = 1, 2$. Then $t = (\sigma t'_1)_{\equiv_{\mathbb{P}}}$ if the most general unifier $\sigma = \mathbf{mgu}(t'_1, t'_2)$ exists and $t = \perp_{\mathbb{P}}$ otherwise. For example, $\mathbf{gci}([\mathrm{a}] \to [\texttt{Int}], [\texttt{Int}] \to \mathrm{b}) = [\texttt{Int}] \to [\texttt{Int}] = \sigma([\mathrm{a}] \to [\texttt{Int}])$ with $\sigma = \mathbf{mgu}([\mathrm{a}] \to [\texttt{Int}], [\texttt{Int}] \to \mathrm{b}) = \{\mathrm{a}/\texttt{Int}, \mathrm{b}/[\texttt{Int}]\}$. The dual operation $\mathbf{lca} : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ calculates the least common anti-instance [11]. For instance, $\mathbf{lca}([\texttt{Int}] \to \mathrm{a}, [[\texttt{Int}]] \to \mathrm{a}) = [\mathrm{b}] \to \mathrm{a}$. Note that $\mathbf{lca}$ preserves type variables when they occur in compatible locations. In abuse of syntax, we also use $\mathbf{lca} : \mathscr{P}(\mathbb{M}) \to \mathbb{P} \cup \{\perp_{\mathbb{P}}\}$ on sets of monotypes with $\mathbf{lca}(\emptyset) := \perp_{\mathbb{P}}$. The relationship between sets of monomorphic type expressions and polymorphic type expressions can now be described by the following Galois insertion[1] that uses $\mathbf{lca}$ on sets as α and a slightly extended $\mathbf{ground}$ function of type $\mathbf{ground} : \mathbb{P}^{\perp}_{\equiv_{\mathbb{P}}} \to \mathscr{P}(\mathbb{M})$ as γ with $\mathbf{ground}(\perp_{\mathbb{P}}) := \emptyset$:

$$\langle \mathscr{P}(\mathbb{M}), \subseteq, \cup, \cap, \mathbb{M}, \emptyset \rangle \underset{\mathbf{lca}}{\overset{\mathbf{ground}}{\leftrightarrows}} \langle \mathbb{P}^{\perp}_{\equiv_{\mathbb{P}}}, \sqsubseteq_{\mathbb{P}}, \mathbf{lca}, \mathbf{gci}, (\mathrm{a})_{\equiv_{\mathbb{P}}}, \perp_{\mathbb{P}} \rangle$$

Given the ability to transform a set of types into a polymorphic type, it remains to show how to abstract a value to a set of types.

[1] A Galois insertion is a Galois connection where γ is injective which we ensure by factoring the abstract type lattice into equivalence classes.

$$
\begin{array}{rcl}
\alpha_{\mathbb{M}} & : & \mathbb{U}_\perp \to \mathscr{P}(\mathbb{M}) \\
\alpha_{\mathbb{M}}(\bot_{\mathbb{U}}) & = & \mathbb{M} \\
\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(c)) & = & \{\texttt{Int}\} \qquad \text{if } c \in \mathbb{Z} \\
\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(\mathbf{Cons}\ l\ ll)) & = & \{[t] \mid t \in \alpha_{\mathbb{M}}(l)\} \cap \alpha_{\mathbb{M}}(ll) \\
\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(\mathbf{Nil})) & = & \{[t] \mid t \in \mathbb{M}\} \\
\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{F}}(f)) & = & \{t_1 \to t_2 \mid v_1 \in \mathbb{U} \wedge \\
 & & \qquad v_2 = f(v_1) \wedge \\
 & & \qquad t_i \in \alpha_{\mathbb{M}}(v_i)\} \\
\alpha_{\mathbb{M}}(\Omega) & = & \emptyset
\end{array}
$$

$$
\begin{array}{rcl}
\gamma_{\mathbb{M}} & : & \mathbb{M} \to \mathscr{P}(\mathbb{U}_\perp) \\
\gamma_{\mathbb{M}}(\texttt{Int}) & = & \{\bot_{\mathbb{U}}\} \cup \{\uparrow^{\mathbb{U}}_{\mathbb{Z}}(c) \mid c \in \mathbb{Z}\} \\
\gamma_{\mathbb{M}}([t]) & = & \{\bot_{\mathbb{U}}\} \cup \{\uparrow^{\mathbb{U}}_{\mathbb{A}}(\mathbf{Nil})\} \cup \\
 & & \{\uparrow^{\mathbb{U}}_{\mathbb{A}}(\mathbf{Cons}\ l\ ll) \mid l \in \gamma_{\mathbb{M}}(t) \setminus \bot_{\mathbb{U}} \wedge \\
 & & \qquad ll \in \gamma_{\mathbb{M}}([t]) \setminus \bot_{\mathbb{U}}\} \\
\gamma_{\mathbb{M}}(t_a \to t_r) & = & \{\bot_{\mathbb{U}}\} \cup \{\uparrow^{\mathbb{U}}_{\mathbb{F}}(f) \mid f \in \mathbb{U} \to \mathbb{U}_\perp \wedge \\
 & & \qquad \forall v \in \gamma_{\mathbb{M}}(t_a) \setminus \bot_{\mathbb{U}} . \\
 & & \qquad f(v) \in \gamma_{\mathbb{M}}(t_r)\}
\end{array}
$$

Figure 4: abstracting and concretizing values to types

4.2. Relating Values and Types

The basic ingredients to relating program values and types are the functions $\alpha_{\mathbb{M}} : \mathbb{U}_\perp \to \mathscr{P}(\mathbb{M})$ and $\gamma_{\mathbb{M}} : \mathbb{M} \to \mathscr{P}(\mathbb{U}_\perp)$ in Fig. 4 that map a single value to the corresponding set of types and a single type to the corresponding sets of values, respectively. For instance, $\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(\mathbf{Nil})) = \{[\texttt{Int}], [\texttt{Int} \to \texttt{Int}], \ldots\}$ which can be turned into a polymorphic type $\mathbf{lca}(\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(\mathbf{Nil}))) = [\mathrm{a}]$. In fact, these definitions encapsulate the definition of the user-defined data types and thus allow us to give a formal definition of $\delta_{\mathbb{U}}$ that mapped a constructor expression datatype:

Definition 1. *The map $\delta_{\mathbb{U}} : \mathbb{C} \times \mathbb{U}^* \to \Delta$ of user-defined datatypes is defined $\delta_{\mathbb{U}}(C\ v_1 \ldots v_n) = \alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(C\ v_1 \ldots v_n))$.*

Observe that lifting the functions in Fig. 4 to accept sets, that is, $\lambda \bar{u} \,.\, \bigcap_{u \in \bar{u}} \alpha_{\mathbb{M}}(u)$ and $\lambda \bar{t} \,.\, \bigcup_{t \in \bar{t}} \gamma_{\mathbb{M}}(t)$ yields an (antitone) Galois connection for single variables. The next section addresses how to abstract whole environments.

4.3. Relational Abstraction to Types

The previous section illustrated how values of variables can be abstracted into types. While it is straightforward to lift this abstraction point-wise to an environment, it is also incorrect:

Example 3. *Consider the type environment when evaluating* 42 *in* $(\lambda id \,.\, (\lambda y \,.\, 42)\, id)\, (\lambda x \,.\, x)$*. At this point,* $\lambda x \,.\, x$ *has been passed to* id *and to* y*. Applying* $\alpha_{\mathbb{M}}$ *point-wise gives* $[id \mapsto T, y \mapsto T]$ *where* $T = \alpha_{\mathbb{M}}(\lambda x \,.\, x) = \{\texttt{Int} \to \texttt{Int}, [\texttt{Int}] \to [\texttt{Int}], \ldots\}$*. Thus, under this abstraction,* id *and* y *may have different types, that is,* $[id \mapsto (\mathrm{a} \to \mathrm{a})_{\equiv_{\mathbb{P}}}, x \mapsto (\mathrm{a} \to \mathrm{a})_{\equiv_{\mathbb{P}}}] = [id \mapsto \mathrm{a} \to \mathrm{a}, x \mapsto \mathrm{b} \to \mathrm{b}]$*. This is incorrect, since* y *and* id *always observe the same value.*

In order to state that two variables must have the same type, we represent an environment as a set of vectors of types. For instance, an environment $[x \mapsto a, y \mapsto a]$ is represented by the set of vectors $\{\langle t, t\rangle \mid t \in \mathbb{M}\}$ whereas $[x \mapsto a, y \mapsto b]$ is represented by $\{\langle t_1, t_2\rangle \mid t_1, t_2 \in \mathbb{M}\}$. This abstraction preserves the relation between different variables and is therefore called *relational*.

Since vectors representing types of variables vary in length, depending on the number of variables in scope, we construct them as maps $\mathscr{P}(\mathbb{X} \to \mathbb{M})$ from program variables $\mathbb{X}$ to a type $\mathbb{M}$, just like the maps for concrete environments $\rho : \mathbb{X} \to \mathbb{U}$. For any fixed $\mathbb{X}$, we interpret $\mathscr{P}(\mathbb{X} \to \mathbb{M})$ as $\mathbb{X}$-indexed vectors $\mathscr{P}(\mathbb{M}^{\mathbb{X}})$. As such, they form a complete lattice $\langle \mathscr{P}(\mathbb{X} \to \mathbb{M}), \subseteq, \cup, \cap, [x \mapsto t]_{x \in \mathbb{X}, t \in \mathbb{M}}, \emptyset\rangle$. Vectors of polymorphic types are defined analogously as maps from $\mathbb{X}$ to $\mathbb{P}$ but with the addition of identifying vectors that are equal modulus renaming of variables, written $(\mathbb{X} \to \mathbb{P})_{\equiv_{\mathbb{P}}}$, and the addition of a bottom element $\bot_{\mathbb{P}}$. The resulting set forms the complete lattice

$$\langle (\mathbb{X} \to \mathbb{P})^{\bot}_{\equiv_{\mathbb{P}}}, \overline{\sqsubseteq_{\mathbb{P}}}\ , \overline{\mathbf{lca}}, \overline{\mathbf{gci}}, [x \mapsto (\mathrm{a})_{\equiv_{\mathbb{P}}}]_{x \in \mathbb{X}}, \bot_{\mathbb{P}} \rangle$$

with join $\overline{\mathbf{lca}}$ and meet $\overline{\mathbf{gci}}$ defined below. Here $\overline{\sqsubseteq_{\mathbb{P}}}$ denotes the conjunction of the point-wise lifting of the relation $\sqsubseteq_{\mathbb{P}}$ on polymorphic type expressions with $\bot_{\mathbb{P}} \overline{\sqsubseteq_{\mathbb{P}}}\ t$ for all $t \in (\mathbb{X} \to \mathbb{P})^{\bot}_{\equiv_{\mathbb{P}}}$. Note that the top element $[x \mapsto (\mathrm{a})_{\equiv_{\mathbb{P}}}]_{x \in \mathbb{X}}$ maps each x to a different type variable since two $(\mathrm{a})_{\equiv_{\mathbb{P}}}$ represent different equivalence classes. The join $\overline{\mathbf{gci}}$ and meet $\overline{\mathbf{lca}}$ is calculated by interpreting the vector $\langle t_1, \ldots, t_n\rangle$ as a single type $t_1 \to \ldots \to t_n$, applying $\mathbf{gci}$ (resp. $\mathbf{lca}$), and converting the result back to a vector. Since relations within the type $t_1 \to \ldots \to t_n$ are preserved, the relations between $t_1, \ldots t_n$ are preserved. We consider an example that contrasts vector and point-wise application of $\mathbf{lca}$:

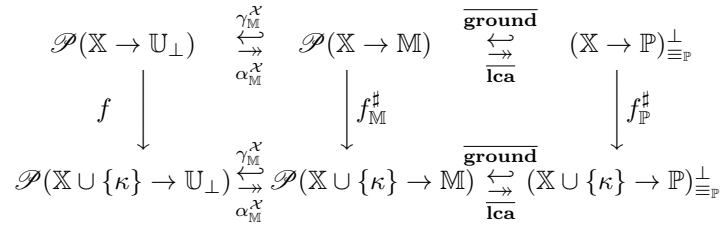

Figure 5: our approach to abstracting denotational semantics

Example 4. *Suppose two identifiers f, g can take on the following two monotypes $\vec{t}_1, \vec{t}_2 \in \mathbb{M}^{\langle f,g\rangle}$:*

$$\langle \mathtt{Int} \to (\mathtt{Int} \to [\mathtt{Int}]) \to \mathtt{Int}, \mathtt{Int} \to (\mathtt{Int} \to [\mathtt{Int}]) \to [\mathtt{Int}] \rangle$$
$$\langle \mathtt{Int} \to ([\mathtt{Int}] \to \mathtt{Int}) \to [\mathtt{Int}], \mathtt{Int} \to ([\mathtt{Int}] \to \mathtt{Int}) \to [[\mathtt{Int}]] \rangle$$

Replacing the comma by a function type symbol $\to$, applying $\mathbf{lca}$ *and replacing $\to$ again by a comma yields:*

$$\langle \mathtt{Int} \to \mathrm{a} \to \mathrm{c}, \mathtt{Int} \to \mathrm{a} \to [\mathrm{c}] \rangle$$

This result is an anti-instance since applying $\sigma_1 = \{a/\mathtt{Int} \to [\mathtt{Int}], c/\mathtt{Int}\}$ yields the first row and $\sigma_2 = \{a/[\mathtt{Int}] \to \mathtt{Int}, c/[\mathtt{Int}]\}$ the second row. By way of contrast, applying $\mathbf{lca}$ *point-wise to $\vec{t}_1$ and $\vec{t}_2$, that is, computing $\langle \mathbf{lca}(\{\vec{t}_1(f), \vec{t}_2(f)\}), \mathbf{lca}(\{\vec{t}_1(g), \vec{t}_2(g)\}\rangle \in \mathbb{P}^{\{f,g\}}$ results in the anti-instance*

$$\langle \mathtt{Int} \to \mathrm{a} \to \mathrm{c}, \mathtt{Int} \to \mathrm{b} \to [\mathrm{d}] \rangle$$

which is not the least-common anti-instance as it can be instantiated to $\langle \mathtt{Int} \to \mathrm{a} \to \mathrm{c}, \mathtt{Int} \to \mathrm{a} \to [\mathrm{c}] \rangle$.

In order to account for the bottom element $\bot_{\mathbb{P}}$ in $(\mathbb{X} \to \mathbb{P})^{\bot}_{\equiv_{\mathbb{P}}}$ we define $\overline{\mathbf{lca}}(\bot_{\mathbb{P}}, t) = \overline{\mathbf{lca}}(t, \bot_{\mathbb{P}}) = t$ and $\overline{\mathbf{gci}}(\bot_{\mathbb{P}}, t) = \overline{\mathbf{gci}}(t, \bot_{\mathbb{P}}) = \bot_{\mathbb{P}}$. We lift the $\overline{\mathbf{lca}}$ operation to sets, allowing us to relate polytype environments with their corresponding monotype environments, as given by the following Galois insertion:

$$\langle \mathscr{P}(\mathbb{X} \to \mathbb{M}), \subseteq \rangle \underset{\overline{\mathbf{lca}}}{\overset{\overline{\mathbf{ground}}}{\leftrightarrows}} \langle (\mathbb{X} \to \mathbb{P})^{\bot}_{\equiv_{\mathbb{P}}}, \overline{\sqsubseteq_{\mathbb{P}}} \rangle$$

In this context, we assume that $\overline{\mathbf{lca}}(\emptyset) = \bot_{\mathbb{P}}$. We lift $\mathbf{ground} : \mathbb{P}^{\bot}_{\equiv_{\mathbb{P}}} \to \mathscr{P}(\mathbb{M})$ to vectors $(\mathbb{X} \to \mathbb{P})^{\bot}_{\equiv_{\mathbb{P}}}$ as follows:

$$\begin{array}{rcl} \overline{\mathbf{ground}} & : & (\mathbb{X} \to \mathbb{P})^{\bot}_{\equiv_{\mathbb{P}}} \to \mathscr{P}(\mathbb{X} \to \mathbb{M}) \\ \overline{\mathbf{ground}}(\bot_{\mathbb{P}}) & = & \emptyset \\ \overline{\mathbf{ground}}(\langle t_1, \ldots t_n \rangle) & = & \{\langle \sigma t_1, \ldots \sigma t_n \rangle \in \mathbb{M} \mid \sigma \in \mathbb{V} \hookrightarrow \mathbb{M}\} \end{array}$$

Thus $\overline{\mathbf{ground}}(\langle \mathrm{a}, \mathrm{b}, \mathtt{Int} \rangle) = \{\langle t_1, t_2, \mathtt{Int} \rangle \mid t_1 \in \mathbb{M}, t_2 \in \mathbb{M}\}$ whereas $\overline{\mathbf{ground}}(\langle \mathrm{a} \to \mathtt{Int}, \mathrm{a} \rangle) = \{\langle t \to \mathtt{Int}, t \rangle \mid t \in \mathbb{M}\}$. Given operations to map sets of monotype vectors to a polytype vector, we now consider the abstraction of concrete environments.

4.4. Abstracting Value Environments to Types

This section presents how the denotational semantics is abstracted to sets of type vectors. In order to express the relationship of the environment ρ that is passed to the denotational semantics $\mathcal{S}[\![\cdot]\!]$ with the resulting value $u = \mathcal{S}[\![\cdot]\!]\ \rho$, we simply bind u to a special variable κ in the same environment. Thus, instead of abstracting $\lambda\rho\,.\,S\rho : (\mathbb{X} \to \mathbb{U}_{\bot}) \to \mathbb{U}_{\bot}$ for each $S \in \mathcal{C}[\![e]\!]$, we abstract $\lambda\rho\,.\,\rho[\kappa \mapsto (S\rho)] : (\mathbb{X} \to \mathbb{U}_{\bot}) \to (\mathbb{X} \cup \{\kappa\} \to \mathbb{U}_{\bot})$ which is shown as f in Fig. 5.

Since the previous section already discussed the abstraction between sets of type vectors and polytypes, that is, between $f^{\sharp}_{\mathbb{M}}$ and $f^{\sharp}_{\mathbb{P}}$ in Fig. 5, we now address how to abstract a value environment to a set of type vectors, that is, between f and $f^{\sharp}_{\mathbb{M}}$. Let $\rho_{\mathbb{M}} \in \mathbb{X} \to \mathbb{M}$ denote a single vector of types and $\overline{\rho}_{\mathbb{M}}$ a set of type vectors. Figure 6

$$
\begin{array}{rcl}
\gamma^{\mathcal{X}}_{\mathbb{M}} & : & \mathscr{P}(\mathbb{X} \to \mathbb{M}) \to \mathscr{P}(\mathbb{X} \to \mathbb{U}_\perp) \\
\gamma^{\mathcal{X}}_{\mathbb{M}}(\overline{\rho}_{\mathbb{M}}) & := & \bigcup\{\bar{\rho} \subseteq \mathbb{X} \to \mathbb{U}_\perp \mid \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}}(\bar{\rho})\} = \{\rho \in \mathbb{X} \to \mathbb{U}_\perp \mid \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
\alpha^{\mathcal{X}}_{\mathbb{M}1} & : & (\mathbb{X} \to \mathbb{U}_\perp) \to \mathscr{P}(\mathbb{X} \to \mathbb{M}) \\
\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho) & := & \bigcup\{\overline{\rho}_{\mathbb{M}} \subseteq \mathbb{X} \to \mathbb{M} \mid \forall \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}, x \in \mathbb{X} \,.\, \rho_{\mathbb{M}}(x) \in \alpha_{\mathbb{M}}(\rho(x)) \wedge \textbf{m.t.}\} \text{ where} \\
 & & \textbf{m.t.} \equiv \begin{array}{l} \forall\{x_1, \ldots x_n\} \in \mathit{mono}(P) \,.\, \forall \rho^1_{\mathbb{M}}, \rho^2_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \,.\, \forall i, j \in [1, n] \,. \\ \rho^1_{\mathbb{M}}(x_i) = \rho^2_{\mathbb{M}}(x_i) \Rightarrow \rho^1_{\mathbb{M}}(x_j) = \rho^2_{\mathbb{M}}(x_j) \end{array} \\
\alpha^{\mathcal{X}}_{\mathbb{M}} & : & \mathscr{P}(\mathbb{X} \to \mathbb{U}_\perp) \to \mathscr{P}(\mathbb{X} \to \mathbb{M}) \\
\alpha^{\mathcal{X}}_{\mathbb{M}}(\bar{\rho}) & := & \bigcap_{\rho \in \bar{\rho}} \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho) \\
\mathit{lift}^{\mathcal{X}}_{\mathbb{M}} & : & \mathscr{P}((\mathbb{X} \to \mathbb{U}_\perp) \to \mathbb{U}_\perp) \to \mathscr{P}(\mathbb{X} \to \mathbb{M}) \to \mathscr{P}(\mathbb{X} \cup \{\kappa\} \to \mathbb{M}) \\
\mathit{lift}^{\mathcal{X}}_{\mathbb{M}}(C) & := & \alpha^{\mathcal{X}}_{\mathbb{M}} \circ \lambda\bar{\rho} \,.\, \{\rho[\kappa \mapsto S\rho] \mid S \in C, \rho \in \bar{\rho}\} \circ \gamma^{\mathcal{X}}_{\mathbb{M}} = \\
 & & \lambda\overline{\rho}_{\mathbb{M}} \,.\, \bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\perp} \{\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho[\kappa \mapsto S\rho]) \mid S \in C \wedge \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\}
\end{array}
$$

Figure 6: abstracting the collecting semantics to vectors of types

presents three functions that transform concrete (value) environments to abstract (set of type vector) environments and vice-versa. Specifically, the concretization function $\gamma^{\mathcal{X}}_{\mathbb{M}}$ takes a set of type vectors $\overline{\rho}_{\mathbb{M}}$ and accumulates any matching value environment ρ. The abstraction from concrete environments $\bar{\rho}$ to sets of type vectors $\overline{\rho}_{\mathbb{M}}$ is split into two functions: $\alpha^{\mathcal{X}}_{\mathbb{M}1}$ takes a single environment ρ to matching type environments, while $\alpha^{\mathcal{X}}_{\mathbb{M}}$ lifts this operation to sets of concrete environments $\bar{\rho}$. The function $\alpha^{\mathcal{X}}_{\mathbb{M}1}$ that abstracts a single environment features a restriction **m.t.** which stands for *monotype restriction* that results in λ-bound variables taking on fewer types than $\mathbf{let}$-bound variables. We will present this restriction in Sect. 4.6. The fourth function $\mathit{lift}^{\mathcal{X}}_{\mathbb{M}}$ in Fig. 6 takes the collecting semantics $C = \mathcal{C}[\![e]\!]$ of some expression e and lifts it to the corresponding semantics on sets of type vectors. It is here where we return the original environment in which the result type is bound to the special variable κ, as illustrated in Fig. 5. This simplifies the presentation and implementation since only vectors need to be handled, rather than functions. We now derive a type inference algorithm by applying $\mathit{lift}^{\mathcal{X}}_{\mathbb{M}}$ to each syntactic construct in Fig. 3, commencing with $\mathit{lift}^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![x]\!])$.

4.5. Lifting Variable Accesses to Sets of Types

In this section we apply the lifting function $\mathit{lift}^{\mathcal{X}}_{\mathbb{M}}$ to the collecting semantics of a variable lookup which Fig. 3 defined as $\mathcal{C}[\![x]\!] = \{\lambda\rho \,.\, \rho(x)\}$. This seemingly innocuous construct turns out to be the most intricate: The $\mathit{lift}^{\mathcal{X}}_{\mathbb{M}}$ function simply binds the value of $\rho(x)$ to κ, leading to:

$$\mathit{lift}^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![x]\!]) = \lambda\overline{\rho}_{\mathbb{M}} \,.\, \bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\perp} \{\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho[\kappa \mapsto \rho(x)]) \mid \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\}$$

For the sake of illustration, suppose that x is bound to $\lambda v \,.\, v$. Since the value $\lambda v \,.\, v$ in ρ and the value $\lambda v \,.\, v$ that is bound to κ can be invoked with different inputs, their abstraction to types may not be the same. Consider the evaluation for the set $\{\rho^1_{\mathbb{M}}, \ldots \rho^m_{\mathbb{M}}\} = \overline{\rho}_{\mathbb{M}}$ and a corresponding value environment ρ with $\overline{\rho}_{\mathbb{M}} = \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)$. We evaluate $\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho[\kappa \mapsto \rho(x)])$ by inlining the definition of $\alpha^{\mathcal{X}}_{\mathbb{M}1}$ but ignore the monotype restriction for now:

$$\bigcup\{\overline{\rho}^{\kappa}_{\mathbb{M}} \subseteq \mathbb{X} \to \mathbb{M} \mid \forall \rho_{\mathbb{M}} \in \overline{\rho}^{\kappa}_{\mathbb{M}}, x \in \mathbb{X} \,.\, \rho_{\mathbb{M}}(x) \in \alpha_{\mathbb{M}}(\rho(x))\}$$

We now address the task of finding $\overline{\rho}^{\kappa}_{\mathbb{M}}$ which is a set of vectors over $\mathbf{dom}(\rho) \cup \{\kappa\}$. Let $\exists_x(\overline{\rho}_{\mathbb{M}})$ denote an environment in which the component x is omitted. Then $\exists_\kappa(\overline{\rho}^{\kappa}_{\mathbb{M}}) = \overline{\rho}_{\mathbb{M}}$ where $\overline{\rho}_{\mathbb{M}} := \{\rho^1_{\mathbb{M}}, \ldots \rho^m_{\mathbb{M}}\}$ and $\rho^1_{\mathbb{M}}, \ldots \rho^m_{\mathbb{M}}$ are defined as above. The challenge is to extend each $\rho^i_{\mathbb{M}}$ with a binding for κ. One possibility is to bind κ in each $\rho^i_{\mathbb{M}}$ to $\rho^j_{\mathbb{M}}(x)$, yielding $\{\rho^i_{\mathbb{M}}[\kappa \mapsto \rho^j_{\mathbb{M}}(x)] \mid i, j \in [1, m]\}$. However, this definition ignores the fact that $\rho^j_{\mathbb{M}}(z) = \rho^i_{\mathbb{M}}(z)$ for all $z \prec x$ since otherwise $\rho^j_{\mathbb{M}}$ and $\rho^i_{\mathbb{M}}$ correspond to different evaluations of the program. Thus, we define $\overline{\rho}^{\kappa}_{\mathbb{M}}$ as:

$$\overline{\rho}^{\kappa}_{\mathbb{M}} = \{\rho^{i}_{\mathbb{M}}[\kappa \mapsto \rho^{j}_{\mathbb{M}}(x)] \mid i,j \in [1,m] \wedge \forall z \in \mathbf{dom}(\rho^{j}_{\mathbb{M}}) \,.\, z \prec x \Rightarrow \rho^{j}_{\mathbb{M}}(z) = \rho^{i}_{\mathbb{M}}(z)\}$$

Since this definition is tied to sets of type vectors, we observe that the calculation above can be expressed using a generic *expand* operation from the literature [17] that can be applied to any abstract domain. We define it as follows:

Definition 2. *Let $\mathcal{D}^{\mathbb{X}} = \mathbb{X} \to \mathcal{D}$ be an abstract domain mapping variables $\mathbb{X}$ to an element $d \in \mathcal{D}$ of a lattice $\langle \mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$. Let $add_x : \mathcal{D}^{\mathbb{X}} \to \mathcal{D}^{\mathbb{X} \cup \{x\}}$ augment a domain element with a new dimension set to top, that is, $add_x(\bar{d}) = \{d[x \mapsto \top] \mid d \in \bar{d}\}$. We also use its natural lifting to sets of variables $add_X : \mathcal{D}^{\mathbb{X}} \to \mathcal{D}^{\mathbb{X} \cup X}$. Let $y_1, \ldots y_n, y'_1, \ldots y'_n$ be pair-wise different. Then let $swap_{y_1 \ldots y_n, y'_1 \ldots y'_n} : \mathcal{D}^{\mathbb{X}} \to \mathcal{D}^{\mathbb{X}}$ exchange the value of y_i with that of y'_i for all $i \in [1,n]$. Expansion of variables $y_1 \ldots y_n$ to $y'_1 \ldots y'_n$ can now be defined as follows:*

$$\begin{array}{rcl} expand_{y_1 \ldots y_n, y'_1 \ldots y'_n} & : & \mathcal{D}^{\mathbb{X}} \to \mathcal{D}^{\mathbb{X} \cup \{y'_1 \ldots y'_n\}} \\ expand_{y_1 \ldots y_n, y'_1 \ldots y'_n}(\bar{d}) & = & add_{y'_1 \ldots y'_n}(\bar{d}) \sqcap swap_{y_1 \ldots y_n, y'_1 \ldots y'_n}(add_{y'_1 \ldots y'_n}(\bar{d})) \end{array}$$

An alternative definition of $\overline{\rho}^{\kappa}_{\mathbb{M}}$ using *expand* can now be given as follows. Let $\exists_{y_1,\ldots y_n}(\bar{d}) = \exists_{y_1}(\ldots \exists_{y_n}(\bar{d}) \ldots)$. Let $\{y_1, \ldots y_n\} = \{y \in \mathbf{dom}(\overline{\rho}_{\mathbb{M}}) \mid x \prec y\}$ denote the set of arguments and local definitions within x. Then

$$\overline{\rho}^{\kappa'}_{\mathbb{M}} = \exists_{y'_1,\ldots y'_n}(expand_{x y_1 \ldots y_n, \kappa y'_1 \ldots y'_n}(\overline{\rho}_{\mathbb{M}}))$$

is equivalent to $\overline{\rho}^{\kappa}_{\mathbb{M}}$, that is, κ is bound to an instance of the type of x. The following proposition ascertains this:

Proposition 1. *Let $\{y_1, \ldots y_n\} = \{y \in \mathbf{dom}(\overline{\rho}_{\mathbb{M}}) \mid x \prec y\}$. Creating an instance κ of x by computing $\overline{\rho}^{\kappa}_{\mathbb{M}}$ is equivalent to expanding $x, y_1, \ldots y_n$ to $\kappa, y'_1, \ldots y'_n$ in $\overline{\rho}_{\mathbb{M}}$ and projecting out $y'_1, \ldots y'_n$.*

Proof. Show $\overline{\rho}^{\kappa'}_{\mathbb{M}} = \overline{\rho}^{\kappa}_{\mathbb{M}}$. Set $\bar{v} = add_{\kappa y'_1 \ldots y'_n}(\overline{\rho}_{\mathbb{M}}) = \{\rho_{\mathbb{M}}[\kappa \mapsto t_0, y'_1 \mapsto t_1, \ldots y'_n \mapsto t_n] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}, t_0, \ldots t_n \in \mathbb{M}\}$. Then $\overline{\rho}^{\kappa'}_{\mathbb{M}} = \exists_{y'_1,\ldots y'_n}(\bar{v} \cap \bar{v}_s)$ where $\bar{v}_s = swap_{x y_1 \ldots y_n, \kappa y'_1 \ldots y'_n}(\bar{v})$. Since y'_i are arbitrary in $\bar{v}$, $\overline{\rho}^{\kappa'}_{\mathbb{M}} = \exists_{y'_1,\ldots y'_n} \bar{v} \cap \exists_{y'_1,\ldots y'_n} \bar{v}_s$ follows. Observe that $\exists_{y'_1,\ldots y'_n} \bar{v}_s = \{\rho_{\mathbb{M}}[x \mapsto t_0, y_1 \mapsto t_1, \ldots y_n \mapsto t_n, \kappa \mapsto \rho_{\mathbb{M}}(x)] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \wedge t_0, \ldots t_n \in \mathbb{M}\}$. Note that $\{z \in \mathbf{dom}(\overline{\rho}_{\mathbb{M}}) \mid z \prec x\} = \mathbf{dom}(\overline{\rho}_{\mathbb{M}}) \setminus \{x, y_1, \ldots y_n\}$. Thus, $\overline{\rho}^{\kappa'}_{\mathbb{M}} = \overline{\rho}^{\kappa}_{\mathbb{M}}$ follows. ■

The advantage of instantiating types using *expand* is that it is agnostic to the underlying domain and, hence, can also be applied to polymorphic types. Consider the following example:

Example 5. *Consider instantiating the definition of g when evaluating one of its two uses in $f = \lambda x \,.\, \mathbf{let}\ g = \lambda y \,.\, x\ \mathbf{in}\ g\, g$. At this point the return type f is still unrestricted, hence let $\rho_{\mathbb{P}} = [f \mapsto \mathrm{a} \to \mathrm{r}, x \mapsto \mathrm{a}, g \mapsto \mathrm{b} \to \mathrm{a}]$. As $\rho_{\mathbb{P}}$ maps no variables y with $g \prec y$, the monotype semantics $lift^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![g]\!]) = \lambda \overline{\rho}_{\mathbb{M}} \,.\, \exists_{y'_1,\ldots y'_n}(expand_{g y_1 \ldots y_n, \kappa y'_1 \ldots y'_n}(\overline{\rho}_{\mathbb{M}}))$ simplifies to $\lambda \overline{\rho}_{\mathbb{M}} \,.\, expand_{g,\kappa}(\overline{\rho}_{\mathbb{M}}))$ and lifts to polymorphic types as $\lambda \rho_{\mathbb{P}} \,.\, expand_{g,\kappa}(\rho_{\mathbb{P}}))$. Thus, we add a binding to a $\top$ variable, yielding $\rho'_{\mathbb{P}} = \rho_{\mathbb{P}}[\kappa \mapsto \mathrm{c}]$, and calculate the meet $\overline{\mathbf{gci}}$ of $\rho'_{\mathbb{P}}$ and $\rho'_{\mathbb{P}}$ where g and κ are swapped:*

$$\begin{array}{rcl} & & \begin{array}{cccc} f & x & g & \kappa \end{array} \\ expand_{g,\kappa}(\rho_{\mathbb{P}}) & = \overline{\mathbf{gci}} & \left(\begin{array}{l} \langle\, \mathrm{a} \to \mathrm{r}\ ,\ \mathrm{a}\ ,\ \mathrm{b} \to \mathrm{a}\ ,\ \mathrm{c} \quad\ \rangle_{\equiv_{\mathbb{P}}}, \\ \langle\, \mathrm{a} \to \mathrm{r}\ ,\ \mathrm{a}\ ,\ \mathrm{c} \quad\ ,\ \mathrm{b} \to \mathrm{a} \,\rangle_{\equiv_{\mathbb{P}}} \end{array} \right) \\ & = \overline{\mathbf{gci}} & \left(\begin{array}{l} \langle\, \mathrm{a} \to \mathrm{r}\ ,\ \mathrm{a}\ ,\ \mathrm{b} \to \mathrm{a}\ ,\ \mathrm{c} \quad\ \rangle, \\ \langle\, \mathrm{d} \to \mathrm{s}\ ,\ \mathrm{d}\ ,\ \mathrm{f} \quad\ ,\ \mathrm{e} \to \mathrm{d} \,\rangle \end{array} \right) \\ & = & \langle\, \mathrm{a} \to \mathrm{r}\ ,\ \mathrm{a}\ ,\ \mathrm{b} \to \mathrm{a}\ ,\ \mathrm{e} \to \mathrm{a} \,\rangle_{\equiv_{\mathbb{P}}} \end{array}$$

Note that arguments to $\overline{\mathbf{gci}}$ are mere representatives of their equivalence classes and recall that **gci** *was defined to rename variables before applying unification, as done in the second equation. The result contains a fresh instance $\rho_{\mathbb{P}}(\kappa) = \mathrm{e} \to \mathrm{a}$ of g whose return type is the same type* a *as the argument x of f.*

Note that the definition of *expand* represents a universal concept that can be found in many contexts. Any actual implementation is likely to use a domain-specific, more efficient way of calculating *expand*. For instance, $expand_{x y_1 \ldots y_m, x' y'_1 \ldots y'_m}$ on a polytype environment $[z_1 \mapsto t_1, \ldots z_n \mapsto t_n, x \mapsto t_x, y_1 \mapsto u_1, \ldots y_m \mapsto u_m]$ can be calculated by renaming all type variables in $t_x, u_1, \ldots u_m$ that do not occur in $t_1, \ldots t_n$. When $m = 0$ then this is exactly what generalizing the type t_x to a type scheme $\forall(vars(t_x) \setminus vars(t_1, \ldots t_n)) \,.\, t_x$ calculates in the **let**-rule of the $\mathcal{W}$-algorithm [3]. Our inference performs this computation for each *use* of x. Given $\mathbf{let}\ x\, y_1 \ldots y_n = \ldots x \ldots$, expansion instantiates t_x by replacing the variables $vars(t_x, u_1, \ldots u_m) \setminus vars(t_1, \ldots t_n)$ in t_x with fresh ones.

$$
\begin{array}{rcl}
own & : & \mathbb{X} \times \mathbb{E} \to \mathscr{P}(\mathbb{X} \backslash \mathbb{X}^\lambda \times \mathbb{X}^\lambda) \\
own(f, x) & = & \emptyset \\
own(f, \lambda x \,.\, e) & = & \{\langle f, x\rangle\} \cup own(f, e) \\
own(f, e_1\, e_2) & = & own(f, e_1) \cup own(f, e_2) \\
own(f, \mathbf{let}\ x = e\ \mathbf{in}\ e') & = & own(x, e) \cup own(f, e') \\
own(f, C\, e_1\, \ldots e_n) & = & \cup_{i=1,\ldots n} own(f, e_i) \\
own(f, \mathbf{case}\, e_s\, \mathbf{of}\, C\ x_1 \ldots x_n : e_t; e_e) & = & \{\langle f, x_1\rangle, \ldots \langle f, x_n\rangle\} \cup own(f, e_s) \cup own(f, e_t) \cup own(f, e_e)
\end{array}
$$

Figure 7: calculating the owner of a variable

4.6. Monomorphic Groups

Section 2.3 stipulated that it is important to define what an argument of a function is. To this end, Fig. 7 defines the owner of a variable $x \in \mathbb{X}^\lambda$ to be the function $f \in \mathbb{X} \backslash \mathbb{X}^\lambda$ in which x is bound.

Let $\xi \in \mathbb{X} \setminus \mathbb{X}^\lambda$ be a function symbol that represents the owner of outermost λ-variables. Each function symbol and its owned λ-bound variables form a so-called *monomorphic group*, a notion we use to abstract $\mathbf{let}$- and λ-bound variables differently.

Definition 3. *A variable set $V \subseteq \mathbb{X}$ forms a monomorphic group in program P iff $V \in mono(P)$ where $mono(P) := \{V \subseteq \mathbb{X} \mid \exists f \,.\, \{f\} = (V \setminus \mathbb{X}^\lambda) \wedge \forall x \in (V \cap \mathbb{X}^\lambda) \,.\, \langle f, x\rangle \in own(\xi, P)\}$.*

The remainder of the paper will only refer to the $mono : \mathbb{E} \to \mathscr{P}(\mathscr{P}(\mathbb{X}))$ function. In particular, we will write $mono(P)$ to denote the set of all monomorphic groups of the current program $P \in \mathbb{E}$. Observe that $mono(P)$ partitions λ-bound variables:

Proposition 2. *Let $\{V_1, \ldots V_n\} = mono(P)$ and $\mathbb{X}$ the be the set of variables in P. Then $V_i \cap V_j = \emptyset$ for all $i \neq j$ and $\mathbb{X}^\lambda \subseteq \bigcup_{i=1}^n V_i$.*

Proof. Let e_x denote the expression $e_x \equiv \lambda x \,.\, e'$ that binds x. Show $i \neq j$ implies $V_i \cap V_j = \emptyset$. By construction, exactly one function symbol $f \in \mathbb{X} \setminus \mathbb{X}^\lambda$ exists in V_i. Let $x \in V_i \cap V_j \cap \mathbb{X}^\lambda$ and thus $\langle f, x\rangle \in own(\xi, P)$. Then there exists a function symbol $g \in V_j$ with $\langle g, x\rangle \in own(\xi, P)$. Since all variable bindings feature different variables, the same expression e_x must have been visited by own, once with $own(f, e_x)$ and once with $own(g, e_x)$ which is a contradiction since own visits every sub-expression of P exactly once. Now show $\mathbb{X}^\lambda \subseteq \bigcup_{i=1}^n V_i$. By definition of own, for each $x \in \mathbb{X}^\lambda$, e_x is visited exactly once, returning $\langle f, x\rangle$. Thus there exists V_i with $f, x \in V_i$ as required. ■

Intuitively, each monomorphic group V_i contains one $\mathbf{let}$-bound function f and its arguments $V_i \setminus \{f\}$. Since $mono(P)$ partitions $\mathbb{X}^\lambda$, each λ-bound variable is argument to only one function. We now formalize the difference between λ- and $\mathbf{let}$-bound variables by restricting the polymorphism between the variables in each V_i.

4.7. Monotype Restriction

Up to now, we have ignored the monotype restriction in $\alpha^{\mathcal{X}}_{\mathbb{M}1}$. In this section we detail its effect on calculating variable lookup.

Let $V \in mono(P)$ be the set that contains the variable x when calculating $lift^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![x]\!])$. Suppose that $x \in V$ is the $\preccurlyeq$-smallest variable in V. Then all other variables $\{x_1, \ldots x_n\} \in V \cap \mathbf{dom}(\overline{\rho}_{\mathbb{M}})$ become arguments to $expand$. By the proof of Prop. 1, these variables have no effect on the generated types. Thus, the semantics for variable lookup presented in the last section is valid whenever the $\preccurlyeq$-smallest variable is looked up which, by definition of $mono$, is always a $\mathbf{let}$-bound variable.

Suppose now that there exists $y \in V \cap \mathbf{dom}(\overline{\rho}_{\mathbb{M}})$ with $y \prec x$. The definition of $\overline{\rho}^{\kappa}_{\mathbb{M}}$ in Sect. 4.5 required that $\rho^j_{\mathbb{M}}(y) = \rho^i_{\mathbb{M}}(y)$. Since x and y are both in V, the monotype restriction requires that $\rho^j_{\mathbb{M}}(x) = \rho^i_{\mathbb{M}}(x)$. Thus, the type of x and κ is identical and, hence, for all variables x that are not the smallest in V the abstract semantics of $\mathcal{S}[\![x]\!]$ is simply $lift^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![x]\!]) = \lambda\overline{\rho}_{\mathbb{M}} \,.\, \{\rho_{\mathbb{M}}[\kappa \mapsto \rho_{\mathbb{M}}(x)] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}\}$. Note that all variables in each $V \in mono(P)$ that are not $\preccurlyeq$-smallest are, by definition, λ-bound.

4.8. Deriving the Complete Abstract Type Semantics

In this section we detail how an analysis can be derived that delivers principal typings, that is, the best type for an expression. Here, the notion of best type is synonymous with a $\sqsubseteq_{\mathbb{P}}$-largest type.

In the context of abstract interpretation, an abstract transformer $f^\sharp$ is called *backward complete* [5] iff $\alpha \circ f = f^\sharp \circ \alpha$. An analysis in which each abstract transformer is backward complete with respect to a monotone α will infer the $\sqsubseteq$-smallest fixpoint, i.e. $\alpha(\mathrm{lfp}^{\subseteq}_{\emptyset}(f)) = \mathrm{lfp}^{\sqsubseteq}_{\bot}(f^\sharp)$. In the context of an antitone α, the abstract lattice is upside-down, thus, $\alpha(\mathrm{lfp}^{\subseteq}_{\emptyset}(f)) = \mathrm{gfp}^{\sqsubseteq}_{\top}(f^\sharp)$ where $\mathrm{gfp}^{\sqsubseteq}_{S}$ is the $\sqsubseteq$-greatest fixpoint starting with S. Rather than showing $\alpha \circ f = f^\sharp \circ \alpha$ for our transformers directly, we use an alternative formulation of backward completeness [15]: if for all concrete values c_1, c_2, if $\alpha(c_1) = \alpha(c_2)$ then $\alpha(f(c_1)) = \alpha(f(c_2))$, it follows that $f^\sharp_{\mathbb{P}} = \alpha \circ f \circ \gamma$ is the backward complete abstract transformer. We therefore first establish that $\alpha(c_1) = \alpha(c_2) \Rightarrow \alpha(f(c_1)) = \alpha(f(c_2))$ holds in our context. Intuitively, this condition states that the result type of a calculation does not change if the input types stay the same. (This condition states that our type system is not a dependent type system where the output type of a function may change with the concrete value of its input.) We assert this first for abstracting values to sets of monotypes:

Lemma 1. *Let $\rho_1, \rho_2 \in \mathbb{X} \to \mathbb{U}_\bot$ with $\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_1) = \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_2)$. Then $\alpha_{\mathbb{M}}(S\rho_1) = \alpha_{\mathbb{M}}(S\rho_2)$ for all $S \in \mathcal{C}[\![e]\!]$ and $e \in \mathbb{E}$.*

Proof. By structural induction over $e \in \mathbb{E}$. We use "i.h." (induction hypothesis) for $\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_1) = \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_2)$.

x**:** $S = \lambda\rho\,.\,\rho(x)$. Thus, $\alpha_{\mathbb{M}}(S\rho_1) = \alpha_{\mathbb{M}}(\rho_1(x)) = \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_1)(x) \overset{\text{i.h.}}{=} \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_2)(x) = \alpha_{\mathbb{M}}(\rho_2(x)) = \alpha_{\mathbb{M}}(S\rho_2)$.

$\lambda x\,.\,e$**:** Pick $S \in \mathcal{C}[\![e]\!]$. Let $u_i = S(\rho_i[x \mapsto v])$ for some $v \in \mathbb{U}_\bot$. Thus $\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{F}}(\lambda v\,.\,S(\rho_i[x \mapsto v]))) = \{t_v \to t_u \mid t_v \in \alpha_{\mathbb{M}}(v) \wedge t_u \in \alpha_{\mathbb{M}}(u_i)\}$ for $i = 1, 2$. By induction hypothesis, it follows that $\alpha_{\mathbb{M}}(u_1) = \alpha_{\mathbb{M}}(u_2)$.

$e^1\,e^2$**:** Pick $S^j \in \mathcal{C}[\![e^j]\!]$. The bodies of $\mathcal{C}[\![e^1\,e^2]\!]$ have type $t_i = \alpha_{\mathbb{M}}((S^1\rho_i \sqsubseteq \mathbb{F} \mathbin{\underline{?}} (S^2\rho_i \sqsubseteq \mathbb{W} \mathbin{\underline{?}} \Omega \mathbin{\underline{:}} \downarrow^{\mathbb{U}}_{\mathbb{F}}(S^1\rho_i)S^2\rho_i) \mathbin{\underline{:}} \Omega))$ for $i = 1, 2$. Assume $S^2\rho_i \neq \Omega$ and $S^1\rho_i \sqsubseteq \mathbb{F}$ since otherwise the claim holds trivially by i.h. Thus, $t_i = \alpha_{\mathbb{M}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(S^1\rho_i)S^2\rho_i) = \{t_r \mid t_a \to t_r \in \alpha_{\mathbb{M}}(S^1\rho_i) \wedge t_a \in \alpha_{\mathbb{M}}(S^2\rho_i)\}$. By induction hypothesis $\alpha_{\mathbb{M}}(S^j\rho_1) = \alpha_{\mathbb{M}}(S^j\rho_2)$ for $j = 1, 2$ and thus $t_1 = t_2$.

let $x = e$ **in** e'**:** Pick $S \in \mathcal{C}[\![e]\!]$. Let $v'_i = S\rho_i[x \mapsto v])$ for all $v \in \mathbb{U}_\bot$, then $\alpha_{\mathbb{M}}(v'_1) = \alpha_{\mathbb{M}}(v'_2)$ by i.h., and, by extension, it follows that for $\hat{v}_i = \mathrm{lfp}^{\preceq}_{\bot_{\mathbb{U}}}\lambda v\,.\,\mathcal{S}[\![e]\!]\ (\rho_i[x \mapsto v])$ it holds that $\alpha_{\mathbb{M}}(\hat{v}_1) = \alpha_{\mathbb{M}}(\hat{v}_2)$. Now let $t_i = \alpha_{\mathbb{M}}((v \sqsubseteq \mathbb{W} \mathbin{\underline{?}} \Omega \mathbin{\underline{:}} S'(\rho[x \mapsto v])))$. Pick $S' \in \mathcal{C}[\![e']\!]$. Then $t_1 = t_2$ follows.

$C\ e_1 \ldots e_n$**:** Pick $S_j \in \mathcal{C}[\![e_j]\!]$, $j = [1, n]$. If there exists $S_j\rho_1 = \Omega$ then Ω is returned and the claim holds by i.h. Otherwise, $\delta_{\mathbb{U}}(C\,S_1\rho_i \ldots S_n\rho_i) \neq \emptyset$ is tested which by Def. 1 is equivalent to $\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(C\,S_1\rho_i \ldots S_n\rho_i)) = \emptyset$. By i.h., $\alpha_{\mathbb{M}}(S_j\rho_1) = \alpha_{\mathbb{M}}(S_j\rho_2)$ for $j = [1, n]$, thus, $\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(C\,S_1\rho_1 \ldots S_n\rho_1)) = \alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(C\,S_1\rho_2 \ldots S_n\rho_2))$. Hence the outcome of the test is the same for $i = 1, 2$ as is the result in case $\uparrow^{\mathbb{U}}_{\mathbb{A}}(C\,S_1\rho_i \ldots S_n\rho_i)$ is returned.

case e_s **of** $C\ x_1 \ldots x_n : e_t; e_e$**:** Pick $S_s \in \mathcal{C}[\![e_s]\!]$, $S_t \in \mathcal{C}[\![e_t]\!]$, and $S_e \in \mathcal{C}[\![e_e]\!]$. For either function, the condition $c_i \equiv \nexists a\,.\,\uparrow^{\mathbb{U}}_{\mathbb{A}}(a) = S_s\rho_i \vee \delta_{\mathbb{U}}(\downarrow^{\mathbb{U}}_{\mathbb{A}}(S_s\rho_i)) \cap (\cup\{\delta_{\mathbb{U}}(C\,v'_1 \ldots v'_n) \mid v'_i \in \mathbb{U}\}) = \emptyset$ is tested. The first disjunct $\nexists a\,.\,\uparrow^{\mathbb{U}}_{\mathbb{A}}(a) = S_s\rho_i$ holds iff $\alpha_{\mathbb{M}}(S_s\rho_i) \in \Delta$. Thus, by i.h. this test holds in $S_s\rho_1$ iff it holds in $S_s\rho_2$. With respect to the set intersection, note that $\delta_{\mathbb{U}}(\downarrow^{\mathbb{U}}_{\mathbb{A}}(S_s\rho_i)) = \alpha_{\mathbb{M}}(S_s\rho_i)$ by Def. 1. Thus, by i.h. it is empty for either both $S_s\rho_1, S_s\rho_2$ or non-empty for both. We now consider the evaluation of the two expressions e_t and e_e.

- e_t**:** The condition $\delta_{\mathbb{U}}(\downarrow^{\mathbb{U}}_{\mathbb{A}}(S_s\rho_i)) \cap \delta_{\mathbb{U}}(C\,v_1 \ldots v_n) \neq \emptyset \wedge v_i \in \mathbb{U}$ is by Def. 1 equivalent to $\bar{t}_i = \alpha_{\mathbb{M}}(S_s\rho_i) \cap \alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(C\,v_1 \ldots v_n)) \neq \emptyset$. By induction hypothesis, $\alpha_{\mathbb{M}}(S_s\rho_1) = \alpha_{\mathbb{M}}(S_s\rho_2)$ and therefore $\bar{t}_1 = \bar{t}_2$. Hence, for all $v_1, \ldots v_n$ for which $t_1 = t_2 = \alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(C\,v_1 \ldots v_n))$ holds, the result of the case statement is $\alpha_{\mathbb{M}}(S_t\rho_1[x_1 \mapsto v_1, \ldots x_n \mapsto v_n]) = \alpha_{\mathbb{M}}(S_t\rho_2[x_1 \mapsto v_1, \ldots x_n \mapsto v_n])$ as required.
- e_e**:** The environments $S_e\rho_1 = S_e\rho_2$ by induction hypothesis. ■

Adjust Lemma 1 so that the results of the transfer function is stored in the environment, that is, show $\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_1[\kappa \mapsto S\rho_1]) = \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_2[\kappa \mapsto S\rho_2])$ instead of $\alpha_{\mathbb{M}}(S\rho_1) = \alpha_{\mathbb{M}}(S\rho_2)$ for $S \in \mathcal{C}[\![e]\!]$. Lifting this result to sets yields:

Corollary 1. *For all $\bar{\rho}_1, \bar{\rho}_2 \subseteq \mathbb{X} \to \mathbb{U}_\bot$ with $\alpha^{\mathcal{X}}_{\mathbb{M}}(\bar{\rho}_1) = \alpha^{\mathcal{X}}_{\mathbb{M}}(\bar{\rho}_2)$ and all expressions $e \in \mathbb{E}$ it follows that $\alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho_1[\kappa \mapsto S\rho_1] \mid S \in \mathcal{C}[\![e]\!] \wedge \rho_1 \in \bar{\rho}_1\}) = \alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho_2[\kappa \mapsto S\rho_2] \mid S \in \mathcal{C}[\![e]\!] \wedge \rho_2 \in \bar{\rho}_2\})$.*

By applying $\overline{\mathbf{lca}}$ on each side of the equations we obtain Lemma 2:

Lemma 2. *For all $\bar{\rho}_1, \bar{\rho}_2 \subseteq \mathbb{X} \to \mathbb{U}_\perp$ with $\overline{\mathbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\bar{\rho}_1)) = \overline{\mathbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\bar{\rho}_2))$ and all expressions $e \in \mathbb{E}$, it follows that $\overline{\mathbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho_1[\kappa \mapsto S\rho_1] \mid S \in \mathcal{C}[\![e]\!] \wedge \rho_1 \in \bar{\rho}_1\})) = \overline{\mathbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho_2[\kappa \mapsto S\rho_2] \mid S \in \mathcal{C}[\![e]\!] \wedge \rho_2 \in \bar{\rho}_2\}))$.*

Proof. For the sake of a contradiction, let $t_i = \overline{\mathbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho_i[\kappa \mapsto S\rho_i] \mid S \in \mathcal{C}[\![e]\!] \wedge \rho_i \in \bar{\rho}_i\}))\kappa$ and $t_1 \neq t_2$. Then there exists $S \in \mathcal{C}[\![e]\!]$ and $\rho^i_{\mathbb{M}} \in \mathscr{P}(\mathbb{X} \cup \{\kappa\} \to \mathbb{M})$ such that $\rho^i_{\mathbb{M}} \in \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_i[\kappa \mapsto S\rho_i])$ and $\rho^1_{\mathbb{M}}(\kappa) \neq \rho^2_{\mathbb{M}}(\kappa)$. However, this is a contradiction to Lem. 1. ■

Lemma 2 allows us to derive backward complete abstract transfer functions as $f^\sharp_{\mathbb{P}} = \alpha \circ f \circ \gamma$ for each concrete f.

4.9. Calculating the Best Abstract Transformers

In this section we derive abstract transformers for the semantic rules (aka concrete transformer) in Fig. 3. The abstraction map $\mathit{lift}^{\mathcal{X}}_{\mathbb{M}}$ in Fig. 6 takes a collecting semantics to a semantics over sets of type vectors and binds the result to the special variable κ. Thus, the semantic equation over polymorphic types for an expression $e \in \mathbb{E}$ is given by $\mathcal{T}[\![e]\!] := \overline{\mathbf{lca}} \circ \mathit{lift}^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![e]\!]) \circ \overline{\mathbf{ground}}$ resulting in the polymorphic type $(\mathcal{T}[\![e]\!]\rho)\kappa$. Consider each rule in turn.

Variable. Equation (1) in Fig. 8 expands $\mathcal{C}[\![x]\!]$ and pushes $\overline{\mathbf{ground}}$ and $\overline{\mathbf{lca}}$ into the set expression. Inlining the first $\alpha^{\mathcal{X}}_{\mathbb{M}1}$ results in Eq. (2) that returns type sets whose projection on $\mathbb{X} \setminus \{\kappa\}$ is bounded by $\overline{\mathbf{ground}}(\rho_{\mathbb{P}})$ and $\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)$. We chose ρ such that both bounds coincide which eliminates that intersection over all ρ. As discussed in Sect. 4.5, the monotype restriction in $\alpha^{\mathcal{X}}_{\mathbb{M}1}$ creates the case distinction between **let**-bound variables and λ-bound variables. The former translates to an *expand* construct in Eq. (3) that Eq. (4) lifts to polytypes. Section 4.7 detailed that λ-bound variables translate to a simple variable lookup, as implemented by Eq. (5).

Abstraction. Eqn. (6) expands $\mathit{lift}^{\mathcal{X}}_{\mathbb{M}}$ before (7) pushes $\overline{\mathbf{ground}}$ inside and moves $\overline{\mathbf{lca}}$ past the intersection over ρ. Equation (8) expands the definition of $\mathcal{C}[\![\lambda x\,.\,e]\!]$ and Eqn. (9) expands the application of $\alpha_{\mathbb{M}}$ to the function type. Here, $t_1 \to t_2$ replaces t. The lifting (10) follows from Lemma 1, rendering ρ unused, leading to Eqn. (11). Equation (12) follows by Lemma 2 with $\mathbf{lca}(\mathbb{M}) = (\mathrm{a})_{\equiv_{\mathbb{P}}}$.

Application. We expand the standard semantics, giving Eqn. (13), and then inline $\alpha_{\mathbb{M}}$ in Eqn. (14). Observe that $\uparrow^{\mathbb{U}}_{\mathbb{F}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)) = v_1$ since $v_1 \in \mathbb{F}$. Now choose v_2 for e which fixes t_a, yielding Eqn. (15). Note here that no types are generated when $\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2$ is $\perp_{\mathbb{U}}$. Equation (16) follows due to the identity $\alpha_{\mathbb{M}} \circ \gamma_{\mathbb{M}}$, leading to the typing constraint in Eqn. (17). The latter translates to a meet expression $\overline{\mathbf{gci}}$, i.e., unification (18). Note here that the first case only applies if unification succeeded, that is, if $\rho'_{\mathbb{P}} \neq \perp_{\mathbb{P}}$.

Binding. After inlining $\mathit{lift}^{\mathcal{X}}_{\mathbb{M}}$ in Fig. 9, the expanded semantic in Eqn. (19) is lifted to types in Eqn. (20) and the lfp is replaced with a greatest-fixpoint computation. Equation (21) lifts this fixpoint computation to one on polymorphic types, thereby making the possibility that an empty environment is computed explicit by returning $\perp_{\mathbb{P}}$.

Constructors. After pushing $\overline{\mathbf{lca}}$ and $\overline{\mathbf{ground}}$ inside, the collecting semantics $\mathcal{C}[\![C\ e_1 \ldots e_n]\!]$ needs expanding in Eqn. (22). We distinguish the case where $S_i\rho \subseteq \mathbb{W}$ and the remaining if-statement in Fig. 3. With $\alpha_{\mathbb{M}}(\Omega) = \emptyset$, the translation of the first case yields Eqn. (23). The second case is represented by $t_c(\rho_{\mathbb{P}})$ and is computed in Eqn. (25). Mapping the constructor and its argument to a datatype is performed by $\delta_{\mathbb{M}}$ and $\delta_{\mathbb{P}}$ which are here defined as follows:

Definition 4. *The datatype of a constructor is given by $\delta_{\mathbb{P}} : \mathbb{C} \times \mathbb{P}^* \to \mathbb{P}_\perp$, namely: $\delta_{\mathbb{P}}(\mathbf{Nil}) = [(\mathrm{a})_{\equiv_{\mathbb{P}}}]$, $\delta_{\mathbb{P}}(\mathbf{Cons}\ \mathrm{h}\ \mathrm{t}) = \mathbf{gci}([\mathrm{h}], \mathrm{t})$ and $\delta_{\mathbb{P}}(c) = \mathtt{Int}$ for all $c \in \mathbb{Z}$. Define $\delta_{\mathbb{M}} : \mathbb{C} \times \mathbb{M}^* \to \mathscr{P}(\mathbb{P})$ as $\delta_{\mathbb{M}}(C\ t_1 \ldots t_n) = \mathbf{ground}(\delta_{\mathbb{P}}(C\ t_1 \ldots t_n))$.*

Thus, Eqn. (26) computes the datatype without resorting the concrete values. The lifting to polytypes in Eqn. (27) separates the empty set of types into an error case. This error case is merged with that of Eqn. (23), yielding Eqn. (24).

$$
\begin{array}{lcl}
\mathcal{T}[\![\cdot]\!] & : & \mathbb{E} \to (\mathbb{X} \to \mathbb{P})^{\perp}_{\equiv_{\mathbb{P}}} \to (\mathbb{X} \cup \{\kappa\} \to \mathbb{P})^{\perp}_{\equiv_{\mathbb{P}}} \\
\mathcal{T}[\![x]\!] & = & \overline{\mathbf{lca}} \circ \lambda\overline{\rho}_{\mathbb{M}} \,.\, \bigcap\{\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho[\kappa \mapsto S\rho]) \mid S \in \mathcal{C}[\![x]\!] \wedge \rho \in \gamma^{\mathcal{X}}_{\mathbb{M}}(\overline{\rho}_{\mathbb{M}})\} \circ \overline{\mathbf{ground}} \\
 & \overset{(1)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}(\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \{\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho[\kappa \mapsto \rho(x)]) \mid \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\}) \\
 & \overset{(2)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}\big(\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \left\{ \bigcup \left\{ \overline{\rho}_{\mathbb{M}} \subseteq \mathbb{X} \cup \{\kappa\} \to \mathbb{M} \,\middle|\, \begin{array}{l} \forall \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}, y \in \mathbb{X} \cup \{\kappa\} \,.\, \rho_{\mathbb{M}}(y) \in \alpha_{\mathbb{M}}(\rho[\kappa \mapsto \rho(x)](y)) \wedge \\ \mathbf{m.r.} \wedge \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \exists_{\kappa}(\overline{\rho}_{\mathbb{M}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho) \end{array} \right\} \right\}\big) \\
\text{if } x \notin \mathbb{X}^{\lambda} & \overset{(3)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}(\exists_{y'_1,\ldots y'_n}(\mathit{expand}_{xy_1\ldots y_n,\kappa y'_1\ldots y'_n}(\overline{\mathbf{ground}}(\rho_{\mathbb{P}})) \text{ where } \{y_1,\ldots y_n\} = \{y \in \mathbf{dom}(\rho_{\mathbb{P}}) \mid x \prec y\} \\
\text{if } x \notin \mathbb{X}^{\lambda} & \overset{(4)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \exists_{y'_1,\ldots y'_n}(\mathit{expand}_{xy_1\ldots y_n,\kappa y'_1\ldots y'_n}(\rho_{\mathbb{P}})) \text{ where } \{y_1,\ldots y_n\} = \{y \in \mathbf{dom}(\rho_{\mathbb{P}}) \mid x \prec y\} \\
\text{if } x \in \mathbb{X}^{\lambda} & \overset{(5)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \rho_{\mathbb{P}}[\kappa \mapsto \rho_{\mathbb{P}}(x)] \\
\mathcal{T}[\![\lambda x \,.\, e]\!] & \overset{(6)}{=} & \overline{\mathbf{lca}} \circ \lambda\overline{\rho}_{\mathbb{M}} \,.\, (\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} (\bigcup\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(S\rho) \wedge S \in \mathcal{C}[\![\lambda x \,.\, e]\!] \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\})) \circ \overline{\mathbf{ground}} \\
 & \overset{(7)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(S\rho) \wedge S \in \mathcal{C}[\![\lambda x \,.\, e]\!] \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
 & \overset{(8)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{F}}(\lambda v \,.\, S(\rho[x \mapsto v]))) \wedge S \in \mathcal{C}[\![e]\!] \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
 & \overset{(9)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t_1 \to t_2] \mid v_2 = S(\rho[x \mapsto v_1]) \wedge S \in \mathcal{C}[\![e]\!] \wedge v_1 \in \mathbb{U} \wedge t_i \in \alpha_{\mathbb{M}}(v_i) \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
 & \overset{(10)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t_1 \to t_2] \mid t_2 = \mathcal{T}[\![e]\!](\rho_{\mathbb{M}}[x \mapsto t_1])\kappa \wedge t_1 \in \mathbb{M} \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
 & \overset{(11)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t_1 \to t_2] \mid t_2 = \mathcal{T}[\![e]\!](\rho_{\mathbb{M}}[x \mapsto t_1])\kappa \wedge t_1 \in \mathbb{M} \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}})\} \\
 & \overset{(12)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \begin{cases} \rho'_{\mathbb{P}}[\kappa \mapsto t_1 \to t_2] & \text{if } \perp_{\mathbb{P}} \neq \rho'_{\mathbb{P}} = \mathcal{T}[\![e]\!](\rho_{\mathbb{P}}[x \mapsto (\mathrm{a})_{\equiv_{\mathbb{P}}}]) \text{ and } t_1 = \rho'_{\mathbb{P}}(x) \text{ and } t_2 = \rho'_{\mathbb{P}}(\kappa) \\ \perp_{\mathbb{P}} & \text{otherwise} \end{cases} \\
\mathcal{T}[\![e_1\ e_2]\!] & = & \overline{\mathbf{lca}} \circ \lambda\overline{\rho}_{\mathbb{M}} \,.\, (\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} (\bigcup\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(S\rho) \wedge S \in \mathcal{C}[\![e_1\ e_2]\!] \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\}) \circ \overline{\mathbf{ground}} \\
 & \overset{(13)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1) \in \mathbb{F} \wedge v_2 \neq \Omega \wedge t \in \alpha_{\mathbb{M}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2) \wedge \mathbf{env}\} \text{ where} \\
 & & \qquad\qquad \mathbf{env} \equiv v_i = S_i\rho \wedge S_i \in \mathcal{S}[\![e_i]\!] \ \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho) \\
 & \overset{(14)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid \exists t_a, t_r \in \mathbb{M} \,.\, \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1) \in \mathbb{F} \wedge v_2 \neq \Omega \wedge \uparrow^{\mathbb{U}}_{\mathbb{F}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)) \in \gamma_{\mathbb{M}}(t_a \to t_r) \wedge \\
 & & \qquad\qquad \forall v \in \gamma_{\mathbb{M}}(t_a) \,.\, \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v \in \gamma_{\mathbb{M}}(t_r) \setminus \perp_{\mathbb{P}} \wedge t \in \alpha_{\mathbb{M}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2) \wedge \mathbf{env}\} \\
 & \overset{(15)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid \exists t_r \in \mathbb{M} \,.\, v_1 \in \gamma_{\mathbb{M}}(t_a \to t_r) \wedge v_2 \in \gamma_{\mathbb{M}}(t_a) \wedge \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2 \in \gamma_{\mathbb{M}}(t_r) \wedge t \in \alpha_{\mathbb{M}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2) \wedge \mathbf{env}\} \\
 & \overset{(16)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t_r] \mid \exists t_r \in \mathbb{M} \,.\, v_1 \in \gamma_{\mathbb{M}}(t_a \to t_r) \wedge v_2 \in \gamma_{\mathbb{M}}(t_a) \wedge \mathbf{env}\} \\
 & \overset{(17)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \begin{cases} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t_r] \mid \exists t_r \in \mathbb{M} \,.\, t_1 = t_a \to t_r \wedge t_2 = t_a \wedge t_i = \mathcal{T}[\![e_i]\!]\{\rho_{\mathbb{M}}\}\,\kappa \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}})\} & \text{if } \mathcal{T}[\![e_i]\!]\rho_{\mathbb{M}} \neq \emptyset \\ \perp_{\mathbb{P}} & \text{otherwise} \end{cases} \\
 & \overset{(18)}{=} & \lambda\rho_{\mathbb{P}} \,.\, \begin{cases} \rho'_{\mathbb{P}}[\kappa \mapsto t_r] & \text{if } \perp_{\mathbb{P}} \neq \rho'_{\mathbb{P}} = \overline{\mathbf{gci}}(\mathcal{T}[\![e_1]\!]\rho_{\mathbb{P}}, \rho_{\mathbb{P}}[\kappa \mapsto (\mathcal{T}[\![e_2]\!]\rho_{\mathbb{P}}\kappa) \to (\mathrm{a})_{\equiv_{\mathbb{P}}}]) \text{ and } \mathcal{T}[\![e_i]\!]\rho_{\mathbb{P}} \neq \perp_{\mathbb{P}} \text{ and let } t_a \to t_r = \rho'_{\mathbb{P}}(\kappa) \\ \perp_{\mathbb{P}} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 8: deriving the type inference algorithm by abstracting the collecting semantics, part one

$$
\begin{aligned}
&\mathcal{T}[\![\textbf{let } x = e \textbf{ in } e']\!] = \overline{\textbf{lca}} \circ \lambda\overline{\rho}_{\mathbb{M}} \,.\, (\textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\bot} (\bigcup \{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(S\rho) \wedge S \in \mathcal{C}[\![\textbf{let } x = e \textbf{ in } e']\!] \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\})) \circ \overline{\textbf{ground}} \\
&\overset{(19)}{=} \lambda\rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\bot} \overline{\textbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t'] \mid v \neq \Omega \wedge t' \in \alpha_{\mathbb{M}}(S'\rho[x \mapsto v]) \wedge S' \in \mathcal{S}[\![e']\!] \ \wedge v \in \{\mathrm{lfp}^{\preceq}_{\bot_{\mathbb{U}}} \lambda v \,.\, S(\rho[x \mapsto v]) \mid S \in \mathcal{C}[\![e]\!]\} \wedge \\
&\qquad\qquad \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
&\overset{(20)}{=} \lambda\rho_{\mathbb{P}} \,.\, \overline{\textbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t'] \mid t' \in \mathcal{T}[\![e']\!](\rho_{\mathbb{M}}[x \mapsto t])\kappa \wedge t \in \mathrm{gfp}^{\subseteq}_{\overline{\mathbb{M}}} \lambda T \,.\, \textstyle\bigcap_{t \in T} (\mathcal{T}[\![e]\!]\rho_{\mathbb{M}}[x \mapsto t])\kappa \wedge \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}})\} \\
&\overset{(21)}{=} \lambda\rho_{\mathbb{P}} \,.\, \begin{cases} \mathcal{T}[\![e']\!]\rho_{\mathbb{P}}[x \mapsto t] & \text{if } \bot_{\mathbb{P}} \neq t = \mathrm{gfp}^{\sqsubseteq_{\mathbb{P}}}_{(\mathrm{a})_{\equiv_{\mathbb{P}}}} \lambda t \,.\, \mathcal{T}[\![e]\!](\rho_{\mathbb{P}}[x \mapsto t])\kappa \\ \bot_{\mathbb{P}} & \text{otherwise} \end{cases} \\
&\mathcal{T}[\![C\ e_1 \dots e_n]\!] = \overline{\textbf{lca}} \circ \lambda\overline{\rho}_{\mathbb{M}} \,.\, (\textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\bot} (\bigcup \{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(S\rho) \wedge S \in \mathcal{C}[\![C\ e_1 \dots e_n]\!] \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\})) \circ \overline{\textbf{ground}} \\
&\overset{(22)}{=} \lambda\rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\bot} \overline{\textbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(S\rho) \wedge S \in \mathcal{C}[\![C\ e_1 \dots e_n]\!] \wedge \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
&\overset{(23)}{=} \lambda\rho_{\mathbb{P}} \,.\, \begin{cases} \overline{\textbf{lca}}(\emptyset) & (\alpha_{\mathbb{M}}(S_1\rho) = \emptyset \vee \dots \vee \alpha_{\mathbb{M}}(S_n\rho) = \emptyset) \wedge S_i \in \mathcal{C}[\![e_i]\!] \wedge \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho) \\ t_c(\rho_{\mathbb{P}}) & \text{otherwise} \end{cases} \\
&\overset{(24)}{=} \lambda\rho_{\mathbb{P}} \,.\, \begin{cases} \bot_{\mathbb{P}} & \text{if } \mathcal{T}[\![e_1]\!] = \bot_{\mathbb{P}} \vee \dots \vee \mathcal{T}[\![e_n]\!] = \bot_{\mathbb{P}} \vee \delta_{\mathbb{P}}(C\,(\mathcal{T}[\![e_1]\!]\rho_{\mathbb{P}}) \dots (\mathcal{T}[\![e_n]\!]\rho_{\mathbb{P}})) = \bot_{\mathbb{P}} \\ \rho_{\mathbb{P}}[\kappa \mapsto \delta_{\mathbb{P}}(C\,(\mathcal{T}[\![e_1]\!]\rho_{\mathbb{P}}) \dots (\mathcal{T}[\![e_n]\!]\rho_{\mathbb{P}}))] & \text{otherwise} \end{cases} \\
\text{where} \quad t_c &\overset{(25)}{=} \lambda\rho_{\mathbb{P}} \,.\, \overline{\textbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(\delta_{\mathbb{U}}(C\,S_1\rho \dots S_n\rho)) \wedge S_i \in \mathcal{C}[\![e_i]\!] \wedge \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
&\overset{(26)}{=} \lambda\rho_{\mathbb{P}} \,.\, \overline{\textbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \delta_{\mathbb{M}}(C\,t_1 \dots t_n)) \wedge t_i \in \alpha_{\mathbb{M}}(S_i\rho) \wedge S_i \in \mathcal{C}[\![e_i]\!] \wedge \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
&\overset{(27)}{=} \lambda\rho_{\mathbb{P}} \,.\, \begin{cases} \bot_{\mathbb{P}} & \text{if } \delta_{\mathbb{P}}(C\,(\mathcal{T}[\![e_1]\!]\rho_{\mathbb{P}}) \dots (\mathcal{T}[\![e_n]\!]\rho_{\mathbb{P}})) = \bot_{\mathbb{P}} \\ \rho_{\mathbb{P}}[\kappa \mapsto \delta_{\mathbb{P}}(C\,(\mathcal{T}[\![e_1]\!]\rho_{\mathbb{P}}) \dots (\mathcal{T}[\![e_n]\!]\rho_{\mathbb{P}}))] & \text{otherwise} \end{cases} \\
&\mathcal{T}[\![\textbf{case } e_s \textbf{ of } C\ x_1 \dots x_n : e_t; e_e]\!] \\
&= \overline{\textbf{lca}} \circ \lambda\overline{\rho}_{\mathbb{M}} \,.\, (\textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\bot} (\bigcup \{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(S\rho) \wedge S \in \mathcal{C}[\![\textbf{case } e_s \textbf{ of } C\ x_1 \dots x_n : e_t; e_e]\!] \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\})) \circ \overline{\textbf{ground}} \\
&= \lambda\rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\bot} \overline{\textbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(S\rho) \wedge S \in \mathcal{C}[\![\textbf{case } e_s \textbf{ of } C\ x_1 \dots x_n : e_t; e_e]\!] \wedge \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
&\overset{(28)}{=} \lambda\rho_{\mathbb{P}} \,.\, \begin{cases} \overline{\textbf{lca}}(\emptyset) & \forall \rho \in \mathbb{X} \to \mathbb{U}_\bot \,.\, \nexists a \,.\, \uparrow^{\mathbb{U}}_{\mathbb{A}}(a) = S_s\rho \vee \delta_{\mathbb{U}}(\downarrow^{\mathbb{U}}_{\mathbb{A}}(S_s\rho)) \cap (\cup\{\delta_{\mathbb{U}}(C\,v'_1 \dots v'_n) \mid v'_i \in \mathbb{U}\}) = \emptyset \wedge S_s \in \mathcal{C}[\![e_s]\!] \wedge \\ & \qquad\qquad \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho) \\ \overline{\textbf{gci}}(t_t(\rho_{\mathbb{P}}), t_e(\rho_{\mathbb{P}})) & \text{otherwise} \end{cases} \\
&\overset{(29)}{=} \lambda\rho_{\mathbb{P}} \,.\, \begin{cases} \overline{\textbf{lca}}(\emptyset) & \forall \rho \in \mathbb{X} \to \mathbb{U}_\bot \,.\, \alpha_{\mathbb{M}}(S_s\rho) \cap (\cup\{\delta_{\mathbb{M}}(C\,t'_1 \dots t'_n) \mid t'_i \in \mathbb{M}\}) = \emptyset \wedge S_s \in \mathcal{C}[\![e_s]\!] \wedge \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho) \\ \overline{\textbf{gci}}(t_t(\rho_{\mathbb{P}}), t_e(\rho_{\mathbb{P}})) & \text{otherwise} \end{cases} \\
&\overset{(30)}{=} \lambda\rho_{\mathbb{P}} \,.\, \begin{cases} \bot_{\mathbb{P}} & \textbf{gci}(\mathcal{T}[\![e_s]\!]\rho_{\mathbb{P}}, \delta_{\mathbb{P}}(C\,(\mathrm{a})_{\equiv_{\mathbb{P}}} \dots (\mathrm{a})_{\equiv_{\mathbb{P}}})) = \bot_{\mathbb{P}} \\ \overline{\textbf{gci}}(t_t(\rho_{\mathbb{P}}), \mathcal{T}[\![e_e]\!]\rho_{\mathbb{P}}) & \text{otherwise} \end{cases} \\
\text{where} \quad t_t &\overset{(31)}{=} \lambda\rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\bot} \overline{\textbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(S\rho[x_1 \mapsto v_1, \dots x_n \mapsto v_n]) \wedge \delta_{\mathbb{U}}(\downarrow^{\mathbb{U}}_{\mathbb{A}}(S_s\rho)) \cap \delta_{\mathbb{U}}(C\,v_1 \dots v_n) \neq \emptyset \wedge v_i \in \mathbb{U} \wedge \\
&\qquad\qquad S_t \in \mathcal{C}[\![e_t]\!] \wedge \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
&\overset{(32)}{=} \lambda\rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\bot} \overline{\textbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(S\rho[x_1 \mapsto v_1, \dots x_n \mapsto v_n]) \wedge \alpha_{\mathbb{M}}(S_s\rho) \cap \delta_{\mathbb{M}}(C\,t_1 \dots t_n) \neq \emptyset \wedge v_i \in \mathbb{U} \wedge t_i \in \gamma_{\mathbb{M}}(v_i) \wedge \\
&\qquad\qquad S_t \in \mathcal{C}[\![e_t]\!] \wedge \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\} \\
&\overset{(33)}{=} \lambda\rho_{\mathbb{P}} \,.\, \mathcal{T}[\![e_t]\!](\exists_\kappa(\overline{\textbf{gci}}(\rho'_{\mathbb{P}}[\kappa \mapsto \mathcal{T}[\![e_s]\!]\rho_{\mathbb{P}}], \rho'_{\mathbb{P}}[\kappa \mapsto \delta_{\mathbb{P}}(C\ t_1 \dots t_n)]))) \text{ where } \rho'_{\mathbb{P}} = \rho_{\mathbb{P}}[x_1 \mapsto t_1, \dots x_n \mapsto t_n] \text{ and } t_1 = (\mathrm{a})_{\equiv_{\mathbb{P}}}, \dots t_n = (\mathrm{a})_{\equiv_{\mathbb{P}}}
\end{aligned}
$$

Figure 9: deriving the type inference algorithm by abstracting the collecting semantics, part two

$$\frac{x \in \mathbb{X}^\lambda}{\rho_\mathbb{P} \vdash x : \rho_\mathbb{P}(x)} \quad \text{(VAR-LAM)} \qquad \frac{\rho_\mathbb{P} \vdash e_1 : t_1 \qquad \rho_\mathbb{P} \vdash e_2 : t_2 \qquad \sigma = \mathbf{mgu}(t_2 \to t_r, t_1) \qquad t_r \text{ fresh}}{\sigma\rho_\mathbb{P} \vdash e_1\, e_2 : \sigma t_r} \quad \text{(APP)}$$

$$\frac{x \in \mathbb{X} \setminus \mathbb{X}^\lambda \qquad \{x, y_1, \ldots y_n\} = mono(P)}{\rho_\mathbb{P} \vdash x : (expand_{xy_1 \ldots y_n, \kappa y'_1 \ldots y'_n}(\rho_\mathbb{P}))(\kappa)} \quad \text{(VAR)}$$

$$\frac{\rho_\mathbb{P}[x \mapsto t_1] \vdash e : t_2 \qquad t_1 \text{ fresh}}{\rho_\mathbb{P} \vdash \lambda x\,.\,e : t_1 \to t_2} \quad \text{(ABS)} \qquad \frac{\rho_\mathbb{P}[x \mapsto t] \vdash e : t \qquad t \sqsubseteq_\mathbb{P} (\mathrm{a})_{\equiv_\mathbb{P}} \text{ greatest} \qquad \rho_\mathbb{P}[x \mapsto t] \vdash e' : t'}{\rho_\mathbb{P} \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : t'} \quad \text{(LET)}$$

Figure 10: inference rules deduced from the semantic equations

Conditional. We distinguish the common error case when expanding the semantics in Eqn. (28). The error condition is lifted to monotypes in Eqn. (29) and to polytypes in Eqn. (30). The non-error cases translate to $\mathbf{gci}(t_t(\rho_\mathbb{P}), t_e(\rho_\mathbb{P}))$ since both function sets in Fig. 3 can be executed. The latter trivially reduces to $\mathcal{T}[\![e_e]\!]\rho_\mathbb{P}$ as shown in Eqn. (30). The non-trival case is developed in Eqn. (31). The test $\delta_\mathbb{U}(\downarrow^\mathbb{U}_\mathbb{A}(S_s\rho)) \cap \delta_\mathbb{U}(C\, v_1 \ldots v_n) \neq \emptyset$ over values is lifted to sets of types, yielding Eqn. (32). Since the values $v_1, \ldots v_n$ that pass this test are bound to $x_1, \ldots x_n$, this relation needs to be made explicit in the lifting to polytypes. To this end, Eqn. (33) binds the t_i to a different type variable each and adds bindings to x_i, yielding $\rho'_\mathbb{P}$. In order to cater for the test, we bind the type of e_s to the temporary variable κ in $\rho'_\mathbb{P}$ and unify this environment with one where κ is bound to the datatype that $C\ t_1 \ldots t_n$ may belong to, thereby restricting the t_i. The temporary variable κ is then removed using $\exists_\kappa$ before the environment is used to evaluate e_t.

4.10. Translation from Equations to Inference Rules

In order to illustrate the difference between the inference algorithm in Figs. 8, 9 and typing rules, we translate them by writing each rule $\lambda\rho_\mathbb{P}\,.\,body(\rho_\mathbb{P})$ if *cond* as an inference rule with *cond* as antecedent and $\rho_\mathbb{P} \vdash body(\rho_\mathbb{P})\kappa$ as consequence. A first concession is to drop all handling of type errors $\bot_\mathbb{P}$. While the first three rules translate straightforwardly, the application rule $e_1\, e_2$ uses $\overline{\mathbf{gci}}$ which can be simplified: Since both arguments share $\rho_\mathbb{P}$, no substitutions will be calculated for them. Hence, it suffices to calculate the $\mathbf{mgu}$ for the κ-bound type and apply the substitution to the whole environment, the latter being indicated by writing $\sigma\rho_\mathbb{P}$ in the consequence. Finally, the calculation of the greatest fixpoint is colloquially stated by requiring that t is the $\sqsubseteq_\mathbb{P}$-greatest solution for which $\rho_\mathbb{P}[x \mapsto t] \vdash e : t$ holds. Overall, while the resulting typing rules are equivalent to the inference algorithm in Figs. 8 and 9 (except for error checking), it is hard to see that they present an algorithm with a straightforward implementation.

5. Inference of Vector Sizes

In this section, we sketch how we augmented the type inference with bit vectors that are polymorphic in their length. To this end, let $\mathbb{S} = \mathscr{P}(\{\sum_i c_i x_i = c \mid c_i, c \in \mathbb{Z}, x_i \in \mathbb{X}\})$ be the universe of size constraint systems. The solution of $s \in \mathbb{S}$ is given by $\gamma_\mathbb{S} : \mathbb{S} \to \mathscr{P}(\mathbb{Z}^{|\mathbb{X}|})$ with $\gamma_\mathbb{S}(s) = \{\langle v_1, \ldots v_{|\mathbb{X}|}\rangle \in \mathbb{Z}^{|\mathbb{X}|} \mid s[x_1/v_1, \ldots x_{|\mathbb{X}|}/v_{|\mathbb{X}|}] \text{ holds}\}$. For each $s \in \mathbb{S}$ let $s_{\equiv_\mathbb{S}}$ denote some normalized constraint system in row-echelon form with $\gamma_\mathbb{S}(s) = \gamma_\mathbb{S}(s_{\equiv_\mathbb{S}})$. Let $\mathbb{S}_{\equiv_\mathbb{S}} = \{s \in \mathbb{S} \mid s_{\equiv_\mathbb{S}} = s\}$ denote the set of normalized size constraint systems. Define the meet $s_1 \sqcap_\mathbb{S} s_2 = (s_1 \cup s_2)_{\equiv_\mathbb{S}}$ and $s_1 \sqsubseteq_\mathbb{S} s_2$ iff $\gamma_\mathbb{S}(s_1) \subseteq \gamma_\mathbb{S}(s_2)$. Then $\langle \mathbb{S}_{\equiv_\mathbb{S}}, \sqsubseteq_\mathbb{S}, \sqcap_\mathbb{S}\rangle$ is a complete semi-lattice.

5.1. Interaction between Type- and Size Domain

We extend the Herbrand abstractions that represent types with $|a| \in \mathbb{P}$, the type of bit vectors of size $a \in \mathbb{V}$, and also allow Herbrand constants $\mathbb{N} \supset \mathbb{P}$ that may instantiate type variables representing bit-vector sizes. The type inference now operates on tuples $\rho_\mathbb{P} \rhd s \in \mathscr{P}((\mathbb{X} \to \mathbb{P})_{\equiv_\mathbb{P}} \times \mathbb{S}_{\equiv_\mathbb{S}})^\bot$. The two domains are co-fibered [20], that is, a domain operation on the master domain $\rho_\mathbb{P}$ can invoke one or more domain operations on the slave domain s. There are three ways how the size domain is manipulated:

- During instantiation of a $\mathbf{let}$-bound variable x from t to a new instance t', we compute the substitutions $[\mathrm{a}_1/\mathrm{b}_1, \ldots \mathrm{a}_n/\mathrm{b}_n] = \mathbf{mgu}(t, t')$ and update the size domain s to $expand_{\mathrm{a}_1 \ldots \mathrm{a}_n, \mathrm{b}_1 \ldots \mathrm{b}_n}(s)$.

- During the calculation of the meet $\rho^1_{\mathbb{P}} \rhd s^1 \sqcup \rho^2_{\mathbb{P}} \rhd s^2$, we first calculate $\rho'_{\mathbb{P}} = \overline{\mathbf{gci}}(\rho^1_{\mathbb{P}}, \rho^2_{\mathbb{P}})$. All resulting substitutions of the form a/b or a/c where $c \in \mathbb{N}$ are also applied to $s' = s^1 \sqcap_{\mathbb{S}} s^2$. Any equation of the form $\mathrm{a} = \mathrm{b}$ or $\mathrm{a} = c$, $c \in \mathbb{N}$ in s' is applied as substitution $[\mathrm{a}/\mathrm{b}]$, respectively $[\mathrm{a}/c]$, to $\rho'_{\mathbb{P}}$. The result is bottom if either domain is bottom.
- At the end of the **let**- and λ-rule, type variables may exist in s that no longer exist in $\rho_{\mathbb{P}}$. Remove these variables from s using Gaussian elimination.

We consider an example in which all three rules are needed:

Example 6. *Using a Haskell-like syntax for legibility, we define* `f x = let g y = x^y in g (g '11')` *and address the task of inferring the type of* f*. Let* $f : |a| \to |b|$ *and* $g : |c| \to |d|$*. Before calculating the instantiations of* g*, we have* $s = \{\mathrm{d} = \mathrm{a} + \mathrm{c}\}$*. Let the outermost instantiation be* $g : |e| \to |f|$*, yielding* $expand_{cd,ef}(s) = \{\mathrm{d} = \mathrm{a} + \mathrm{c}, \mathrm{f} = \mathrm{a} + \mathrm{e}\}$ *and, for the inner instance,* $g : |g| \to |h|$ *and* $s' = \{\mathrm{d} = \mathrm{a} + \mathrm{c}, \mathrm{f} = \mathrm{a} + \mathrm{e}, \mathrm{h} = \mathrm{a} + \mathrm{g}\}$*. Unification of* g *with* 2 *and unifying the result of the inner* g *with the argument of the outer adds* $\mathrm{g} = 2, \mathrm{e} = \mathrm{h}$*, yielding* $s'' = \{\mathrm{d} = \mathrm{a} + \mathrm{c}, \mathrm{f} = 2\mathrm{a} + 2\}$*. The result of the outer* g *is that of* f*, adding* $\mathrm{f} = \mathrm{b}$*. Once* g *goes out of scope, all variables pertaining to* g *and its instantiations are removed, leaving* $b = 2a + 2$ *as size constraint for* f*.*

Interestingly, the type inference may no longer be complete since $\mathbb{S}_{\equiv_{\mathbb{S}}}$ is not closed under projection $\exists$: eliminating y in $x = 4y$ removes all constraints and thereby the information that x is a multiple of four. Thus, when projecting out a variable, it should be checked if type variables relating to vectors sizes are removed that do not occur anywhere else in the environment. Indeed, such type variables would indicate that a vector has an ambiguous size (since it is not defined by any binding in the environment) which is an error in itself and, hence, the program needs to be rejected.

5.2. Application of Polymorphic Recursion

In order to dissect a bit vector, we extend the **case**-statement in Sect. 3 by allowing $\mathbf{case}\ e_s\ \mathbf{of}\ '\mathit{pat}' :\ e_t;\ e_e \in \mathbb{E}$ where $\mathit{pat} ::= 0\,\mathit{pat} \mid 1\,\mathit{pat} \mid x$ is a bit pattern. For instance, $\mathbf{case}\ v\ \mathbf{of}\ '0011x' :\ e_t;\ e_e$ evaluates e_t if v is a bit vector of at least 4 bits that commences with 0011 and otherwise evaluates e_e. Here, the types are $v : |\mathrm{a}|$ and $x : |\mathrm{b}|$ with $\mathrm{b} + 4 = \mathrm{a}$. A function *conv* that converts a bit vector of any size into an integer can now be defined as follows:

$$
\begin{array}{lll}
\mathbf{let}\ \mathit{conv}\ v = & \mathbf{let}\ f\ \mathit{acc}\ v = & \mathbf{case}\ v\ \mathbf{of}\ '0\,x' :\ f\ (2 \cdot \mathit{acc})\ x; \\
& & \quad\ \mathbf{case}\ v\ \mathbf{of}\ '1\,x' :\ f\ (2 \cdot \mathit{acc} + 1)\ x; \\
& & \quad\ \mathit{acc} \\
& \mathbf{in}\ f\ 0\ v &
\end{array}
$$

The function *conv* calls f with an accumulator of zero. The local function f checks if the vector v commences with a zero or a one and computes a new accumulator. If neither case matches, the vector must be empty and the accumulator is returned. Suppose that $f : \mathtt{Int} \to |\mathrm{a}| \to \mathtt{Int}$. Inside f, the type of the remaining vector is $x : |\mathrm{b}|$ where $\mathrm{b} + 1 = \mathrm{a}$ is tracked by the size domain. Thus, inside the body of f, the function is called recursively as $f : \mathtt{Int} \to |\mathrm{b}| \to \mathtt{Int}$. Since unifying a and b would yield a type error, this example requires and demonstrates a practical example of polymorphic recursion. In general, there are few interesting examples of polymorphic recursion. Mycroft who proposed polymorphic recursion motivates it using a universal list traversal within a compiler [14].

6. Discussion

We discuss our implementation, the related work and conclude.

6.1. Implementation

We implemented the type inference with bit vectors in our GDSL [16] compiler which is written in SML/NJ. It is publicly available and ships with three specifications shown as "decoder" in Fig. 11. By commenting out functions that add size constraints to the affine domain, we obtain the running times without size inference (third column) over those with (last column). The inference is comparatively slow as we iterate through the environment each time we need to apply substitutions σ, although we stop early when the environment contains none of the variables that σ replaces. The penalty for size inference is minor albeit noticeable in the AVR decoder as it makes heavy use of the concatenation operator. Interestingly, the inference of the larger Intel decoder is marginally faster with size inference, probably since more type variables are instantiated.

decoder	lines	time w/o sizes	time w. sizes
Texas MSP430	217	0.1s	0.2s
Atmel AVR	941	1.6s	1.9s
Intel x86	5610	3.7s	3.6s

Figure 11: performance of type checking various decoders

6.2. Observing Non-Termination through an Extended Occurs Check

The lattice of Herbrand abstractions has infinite descending chains [11], that is, during a fixpoint computation the type of a function may arbitrarily often be instantiated, thereby creating at least as many type variables than unification eliminates. Thus, the fixpoint computation may not terminate. The possibility of non-termination follows from the undecidability of inferring a polymorphic recursive type for a function [8]. However, all non-terminating programs we know are type incorrect. As an example, consider the function `f x = f (x 1)`. Suppose we compute a fixpoint for f starting with $f : \mathrm{a} \to \mathrm{b}$, thus $x : \mathrm{a}$. When evaluating the body of the function, we instantiate f to $f : \mathrm{c} \to \mathrm{d}$ and observe that $x : \mathtt{Int} \to \mathrm{c}$ thereby applying the substitution $a/\mathtt{Int} \to \mathrm{c}$. In the next iteration, we have $f : (\mathtt{Int} \to \mathrm{c}) \to \mathrm{b}$. Checking the body creates another instance $f : (\mathtt{Int} \to \mathrm{e}) \to \mathrm{g}$ and we observe that $x : \mathtt{Int} \to (\mathtt{Int} \to \mathrm{e})$, thereby applying the substitution $c/(\mathtt{Int} \to \mathrm{e})$. Now the function type is $f : (\mathtt{Int} \to (\mathtt{Int} \to \mathrm{e})) \to \mathrm{b}$. We can check for these re-occurring patterns before each fixpoint iteration by simply unifying the uninstantiated type of f with each of its usage sites. For instance, the unification of the current type of f with the current type of its usage gives $\mathbf{gci}((\mathtt{Int} \to (\mathtt{Int} \to \mathrm{e})) \to \mathrm{b}, (\mathtt{Int} \to \mathrm{e}) \to \mathrm{g}) = \bot_{\mathbb{P}}$. In particular, the unification fails since the substitution $e/\mathtt{Int} \to \mathrm{e}$ triggers an occurs check. In this case, we abort the fixpoint computation since the refinement of f will go on indefinitely. If the unification succeeds then the function is not polymorphic recursive, that is, unifying the function with all its call sites is exactly what Damas' $\mathcal{W}$ algorithm does. Thus, if the unification fails for other reasons (that is, due to a constructor clash), we ignore this and continue with the fixpoint computation to find a polymorphic recursive type. Although this *extended occurs check* is very simple, it catches all non-terminating behaviors we have observed so far, thereby giving a practical inference for polymorphic recursive types.

6.3. Related Work

Tofte [19] observes the following about devising new type systems:

> Guessing and verifying are inseparable parts of developing a new theory. None is prior to the other neither in time nor in importance. I believe the reason why the guessing has been so hard is precisely that the verifying has been hard.

Instead of guessing, Cousot [1] proposes to use his abstract interpretation framework [2] to systematically construct type inferences (and thus type systems) by abstracting the language semantics, thereby replacing the *hard task of verification*. We build on this work. However, Cousot partitions type variables into those bound in a type scheme and those that are free which, as shown by Ex. 2, is not generally possible. Moreover, his derived inference for Milner-Mycroft types [14] eliminates and re-introduces type schemes in every rule which instantiates λ-bound variables just like **let**-bound variables, leading to types that are too general.

Type schemes seem to be a brittle, since syntactic, concept. Henglein's inference for polymorphic recursion [8] gathers type variables of certain λ-bound variables in order to prevent them from being generalized. However, his algorithm collects the wrong variables [4, p. 164]. In fact, Henglein's reduction from semi-unification to polymorphic recursion is complicated by the incorrect inference and a corrected reduction should be much simpler.

The use of expansion [17] instead of type schemes can avoid the complex proofs involving arguments about free and bound type variables [4, 12] and immediately lends itself to new applications as in the instantiation of vector size information as discussed in Sect. 5. Other results in the abstract interpretation literature, namely on completeness [2, 5, 15] and modularity [6], deserve highlighting in the context of type systems: By specifying a type inference of a language by the universe of types and requiring a modular and complete type inference, there is neither ambiguity of what types must be inferable nor a prescription of the employed algorithm. However, given that the lattice of Herbrand abstractions $\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}} \rangle$ has infinite descending chains, the only work on using abstract interpretation for type inference proposes to apply widening to ensure termination of the fixpoint computation [1, 7] rather than addressing the bigger

challenge [5] of restricting the lattice to avoid infinite descending chains. Even then, neither Gori and Levi [7] nor Jim [9] found any type correct program that required many iterations to type check which coincides with our experience. In particular, only incorrect programs in which instantiation replaces a type variable a by an expression involving a seem to trigger an infinite refinement. Since these trivial cases are found by our extended occurs check, we are as of yet not aware of any program for which our inference does not terminate. Thus, it might be worth ignoring the problem of infinite descending chains and to further investigate into complete extensions such as our size inference.

Since Wells states that the Hindley-Milner type system has no "principal typings" [21], the chasm to our work needs to be explained: Wells' notion demands three properties of a type system to have principal typings: translated to abstract interpretation nomenclature, these are completeness of transfer functions, condensing domains [6] (which are both fair) and that types must be inferable in a bottom-up manner (which is unfair to Hindley-Milner since **let**- and λ-bound variables are abstracted differently and in a bottom-up analysis where variables are encountered before their definition sites). In fact, the existence of $\alpha^{\mathcal{X}}_{\mathbb{M}}$ in Fig. 6 alone guarantees that a best type exists, although it might not be possible to infer it. Note that Cousot only defined $\gamma^{\mathcal{X}}_{\mathbb{M}}$ explicitly and defined $\alpha^{\mathcal{X}}_{\mathbb{M}}$ in terms of $\gamma^{\mathcal{X}}_{\mathbb{M}}$ [1], which seems to suggest that defining an $\alpha^{\mathcal{X}}_{\mathbb{M}}$ as we have done in this work is non-trival.

6.4. Conclusion

We derived a type inference algorithm for the Hindley-Milner type system using the abstract interpretation framework. In contrast to previous work, our derived algorithm is complete by construction by being built on a novel $\alpha^{\mathcal{X}}_{\mathbb{M}}$ featuring a monotype restriction. We proposed expansion over type schemes to handle instantiation. The latter allowed us to construct a complete inference for vector sizes.

Acknowledgements. I thank Jan Midtgaard and the anonymous reviewers for improving the presentation and Roberto Giacobazzi and Andy King for useful discussions.

[1] P. Cousot. Types as Abstract Interpretations. In *Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM. invited paper.
[2] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Principles of Programming Languages*, pages 269–282, San Antonio, Texas, USA, January 1979. ACM.
[3] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *Principles of Programming Languages*, pages 207–212, Albuquerque, New Mexico, USA, 1982. ACM.
[4] M. Emms and H. Leiß. Extending the type checker of Standard ML by polymorphic recursion. *Theoretical Computer Science*, 212(1-2):157 – 181, 1999.
[5] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.
[6] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract domains condensing. *Trans. Comput. Log.*, 6(1):33–60, 2005.
[7] R. Gori and G. Levi. An Experiment in Type Inference and Verification by Abstract Interpretation. In A. Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 2294 of *LNCS*, pages 237–241, Venice, Italy, January 2002. Springer.
[8] F. Henglein. Type Inference with Polymorphic Recursion. *Transactions on Programming Languages and Systems*, 15:253–289, 1991.
[9] Trevor Jim. What are principal typings and what are they good for? In *Principles of Programming Languages*, pages 42–53, St. Petersburg Beach, Florida, USA, 1996. ACM.
[10] M. Karr. On affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.
[11] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In M. Boscarol, L. Carlucci Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming*, volume 306 of *LNCS*, pages 67–113, Trento, Italy, December 1986. Springer.
[12] S. Lenglet and J. Wells. Expansion for Universal Quantifiers. In H. Seidl, editor, *European Symp. on Prog. Lang. and Systems*, volume 7211 of *LNCS*, pages 456–475, Tallinn, Estonia, March 2012. Springer.
[13] R. Milner. A Theory of Type Polymorphism in Programming. *J. Comput. Sys. Sci.*, 17(3):348–375, 1978.
[14] A. Mycroft. Polymorphic Type Schemes and Recursive Definitions. In M. Paul and B. Robinet, editors, *Symposium on Programming*, volume 167 of *LNCS*, pages 217–228, Toulouse, France, April 1984. Springer.
[15] D. A. Schmidt. Comparing Completeness Properties of Static Analyses and Their Logics. In N. Kobayashi, editor, *Asian Symposium on Programming Languages and Systems*, volume 4279 of *LNCS*, pages 183–199, Sydney, Australia, November 2006. Springer.
[16] A. Sepp, J. Kranz, and A. Simon. GDSL: A Generic Decoder Specification Language for Interpreting Machine Language. In *Tools for Automatic Program Analysis*, ENTCS, Deauville, France, September 2012. Springer.
[17] H. Siegel and A. Simon. Summarized Dimensions Revisited. In L. Mauborgne, editor, *Workshop on Numeric and Symbolic Abstract Domains*, ENTCS, Venice, Italy, September 2011. Springer.
[18] A. Simon. Deriving a Complete Type Inference for Hindley-Milner and Vector Sizes using Expansion. In *Partial Evaluation and Program Manipulation*, SIGPLAN, Rome, Italy, January 2013. ACM.
[19] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, UK, 1988.
[20] A. Venet. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In *Static Analysis Symposium*, LNCS, pages 366–382, London, UK, 1996. Springer.
[21] J. B. Wells. The Essence of Principal Typings. In P. Widmayer, F. T. Ruiz, R. M. Bueno, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *Colloquium on Automata, Languages and Programming*, volume 2380 of *LNCS*, pages 913–925, Malaga, Spain, July 2002. Springer.