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

Axel Simon *

Technische Universität München
Lehrstuhl für Informatik 2
Garching b. München, Germany
Axel.Simon@in.tum.de

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. Instead of type schemes, it uses *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.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; F.3.3 [*Studies of Program Constructs*]: Type structure; D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.3 [*Language Constructs and Features*]: Polymorphism

General Terms Languages, Theory, Verification

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

1. Introduction

New computer languages are continuously designed, often in the form of a 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 type for the same input and thereby accept a different set of programs. 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 an inadequate implementation of the type system can easily be spotted (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]. The first contribution of this paper is to show that a backward complete type inference can be derived by simply abstracting the semantics of the language, thereby giving a constructive way to create type inference that infer best types. However, the presence of any branching construct (an **`if`** or a **`case`** statement) make the derivation of a backward complete type inference impossible, as illustrated by the following example:

EXAMPLE 1. *Consider the semantics of the Haskell expression* **`if`** `(f x)` **`then`** `42` **`else`** `[]` *which is* $\perp$ *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* $[t]$. *In case the first evaluation of* `(f x)` *does not terminate, then* a *is indeed the best type of the expression. Since determining whether* `(f x)` *always terminates is undecidable, a type inference assumes that both branches of the* **`if`***-statement are taken which is, in fact, an abstraction of the denotational semantics.*

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 that encodes the difference between **let** and λ-bound variables that the Hindley-Milner type system prescribes. Hence, the second contribution of this paper is this 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].

* Supported by DFG Emmy Noether programme SI 1579/1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'13, January 21–22, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1842-6/13/01. . . $15.00

What about the special language features of the DSL? For instance, we designed a language to specify instruction decoders [16] which are programs that turn a bit stream into a processor instruction. These programs makes heavy use of bit-vectors whose type is |a| where a is a type variable denoting the size of the vector. Consider the decoder of an AVR microcontroller instruction:

```
###  ORI: perforam a bit-wise OR with an Immediate
decode ['0110 k k k k d d d d k k k k'] =
  binop ORI rd4 ck8
```

This rule matches a 16-bit word starting with `0110`. The letters `k` and `d` specify that the given bit is to be appended to an initially empty bit-vector `ks` and `ds`, respectively. The vector `ds`, for instance, is read by the function `rd4` to chose the right register. Internally, the following calculation is performed (where `''` denotes the empty bit-vector and ^ denotes concatenation of bit vectors):

```
let ds0 = ''; ds1 = ds0 ^ d1; ds2 = ds1 ^ d2;
    ds3 = ds2 ^ d3; ds4 = ds3 ^ d4 in
case ds4 of '0000' -> R0; '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 schemes 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 previously used 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.

Overall, this paper makes the following three 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.
- In order to derive this backward complete type inference, we formulate an abstraction map for the Hindley-Milner type system. This abstraction map is novel and encodes how λ- and **let**-bound variables differ in the way they are abstracted.
- Our derivation uses a domain-agnostic expand operator rather than type schemes. We illustrate the versatility of the expand operator by extending the derived type system with the inference of vector sizes.

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 extends the inference to also infer the size of bit vectors before Sect. 6 discusses our implementation and related work.

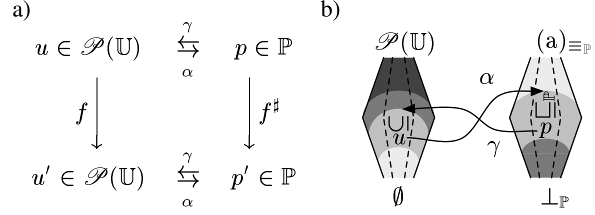

Figure 1. concrete/abstract transformers and antitone abstraction

2. Overview of Abstracting into Types

We review the basics of completeness and how type inference can be described as an abstract interpretation. We 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 $\alpha(u) \sqsubseteq_\mathbb{P} t$ iff $u \subseteq \gamma(t)$. We lift the program semantics that operates on $\mathbb{U}$ 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 computation of the least fixpoint of $f^\sharp$ will infer the best possible 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 $\mathrm{lfp}^{\leq}_{S}$ 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. Given an additional property, backward completeness, namely $\alpha \circ f = f^\sharp \circ \alpha$, follows. Thus, the inferred type for a program variable is the most precise type within the universe of types $\mathbb{P}$. As illustrated by Example 1, the inference is complete with respect to a weaker semantics in which a conditional is abstracted to a non-deterministic choice. However, conditionals are also not considered when discussing the notion of principal typings [21].

2.2 Abstract Interpretation for Types

The abstract interpretation framework is usually used for analyses that state *positive facts* about the behaviors of a program. For instance, the abstract domain of intervals infers facts such as $x \in [2, 5]$ when approximating the possible results of the Haskell expression **if** `(f x)` **then** `2` **else** `5`. In contrast, type inferences gather constraints or *negative facts* 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 $x = 2$ and $x = 5$. However, in the case positive facts are inferred, the states are joined (yielding $[2, 5]$ for the interval domain) whereas in the case where negative 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 illus-

$$
\begin{array}{lcl}
e & \in & \mathbb{E} \\
e & ::= & x \mid \lambda x \,.\, e \mid e_1\, e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \\
 & \mid & C\ e_1 \ldots e_n \mid \mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3
\end{array}
$$

Figure 2. inductive definition of the program syntax

trated in Fig. 1b). The antitone relation between concrete and abstract domain is merely a reflection on the difference between type systems and standard program analysis [1]. It is possible to simply flip the abstract lattice upside down in which case the number of constraints grows with the number of program behaviors [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] whereas the affine domain [10] is used to infer equalities relations between numeric program variables.

2.3 The Hindley-Milner Type System

Let $\mathbb{X}$ denote the set of program variables. For simplicity, we assume that variables at binding sites (λ- and $\mathbf{let}$-bindings) are pairwise 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" variables whereas λ-bound variables are called "monomorphic". However, the latter is a slight misnomer, 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 a *is a variable introduced in* f *but "monomorphic" in* g*.*

Rather than using type schemes, we will define an abstraction map $\alpha^{\mathcal{X}}_{\mathbb{M}}$ for whole environments, rather than for the type of a single variable. All type variables in an environment are then implicitly $\forall$-quantified, resulting in the following types for Ex. 2:

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

In order to clarify how to instantiate the types above, we define a pre-order on $\mathbb{X}$, namely $x \preccurlyeq y$ if x is in scope of y where $x, y \in \mathbb{X}$. In particular, $x \preccurlyeq y$ and $y \preccurlyeq x$ may hold for $x \neq y$ iff x and y occur in the same binding group and are thus visible in each others 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 $x, y \in \mathbb{X}^\lambda$, $f, g \in \mathbb{X} \setminus \mathbb{X}^\lambda$. The type schemes can now be recovered through a traversal starting at the $\prec$-smallest variable and generating type schemes at each variable $f \in \mathbb{X} \setminus \mathbb{X}^\lambda$ for all as-of-yet unquantified variables. Note, though, that the analysis we derive never creates any type schemes and only requires the information in $\preccurlyeq$ and $\mathbb{X}^\lambda$.

This concludes the preliminaries for deriving a complete type inference. We now turn to the definition of the concrete semantics.

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 in Fig. 2. The shown standard constructs of the λ-calculus are extended by a $\mathbf{let}$-construct that allows x to be **recursively** used in e_1 (x is in scope in e_1). The second line defines values in form of constructors with n arguments and a conditional that evaluates e_2 if e_1 evaluates to non-zero and that evaluates e_3 if e_1 evaluate to zero. For the sake of presentation, we set $\mathbb{C} = \{\mathbf{Cons}, \mathbf{Nil}\} \cup \mathbb{Z}$ allowing us to construct lists containing integers. Here the constructor $\mathbf{Nil}$ takes no arguments and $\mathbf{Cons}$ takes two, a value and a list.

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 the formal semantics to $e \in \mathbb{E}$.

3.1 Semantics

Let $S_\bot := S \cup \{\bot_S\}$ where $\bot_S \notin S$ denotes an undefined value or a non-terminating evaluation. The denotational standard semantics evaluates $e \in \mathbb{E}$ to a program value $\mathbb{U}_\bot$ 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:

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

We construct values of type $\mathbb{U}$ from a value in $\mathbb{D} = \mathbb{W}, \mathbb{A}, \mathbb{F}$ 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 applying a Haskell/ML constructor $\mathbb{D}$ of the algebraic data type $\mathbb{U}$). Conversely, define an extraction function $\downarrow^{\mathbb{U}}_{\mathbb{D}}(\cdot) : \mathbb{U}_\bot \to \mathbb{D}_\bot$ with $\downarrow^{\mathbb{U}}_{\mathbb{D}}(\uparrow^{\mathbb{U}}_{\mathbb{D}}(v)) = v$ for all $v \in \mathbb{D}$ and $\downarrow^{\mathbb{U}}_{\mathbb{D}}(\cdot) = \bot_{\mathbb{U}}$ otherwise. For brevity, let $\Omega = \uparrow^{\mathbb{U}}_{\mathbb{W}}(\omega)$ define the error value, i.e. $\Omega \in \mathbb{U}$. In order to test if $v \in \mathbb{U}$ originated in $\mathbb{D}$ we define the following C-like conditional:

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

In order to evaluate an expression, it is necessary to track the values of variables. We therefore define program environments:

$$\rho \quad \in \quad \mathbb{X} \to \mathbb{U}_\bot$$

An updated value environment $\rho' = \rho[x \mapsto v] \in \mathbb{X} \to \mathbb{U}_\bot$ obeys $\rho'(x) = v$ and $\rho'(y) = \rho(y)$ for $y \neq x$. Note that concrete environments may map variables to $\bot_{\mathbb{U}}$ which simplifies the definition of the fixpoint calculation in the recursive $\mathbf{let}$.

Values $v \in \mathbb{U}_\bot$ are partially ordered by $\preceq$ as follows: $v_1 \preceq v_2$ iff $v_1 = \bot_{\mathbb{U}}$ or if $f_i = \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_i) \neq \bot_{\mathbb{U}}$ for $i = 1, 2$ then $f_1(v) \preceq f_2(v)$ for all $v \in \mathbb{U}$. 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 $\psi(l) \sqsubseteq l$.

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 3. Here, the conditional statement is used in two ways: Firstly, $(v \,\underline{\in}\, \mathbb{D} \;\underline{?}\; \ldots \downarrow^{\mathbb{U}}_{\mathbb{D}}(v) \ldots \;\underline{:}\; \Omega)$ 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 $(v \,\underline{\in}\, \mathbb{W} \;\underline{?}\; \Omega \;\underline{:}\; \ldots)$ 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.

$$
\begin{array}{rcl}
\mathcal{S}[\![\cdot]\!] & : & \mathbb{E} \to (\mathbb{X} \to \mathbb{U}_\bot) \to \mathbb{U}_\bot \\
\mathcal{S}[\![x]\!]\ \rho & = & \rho(x) \\
\mathcal{S}[\![\lambda x\,.\,e]\!]\ \rho & = & \uparrow^{\mathbb{U}}_{\mathbb{F}}(\lambda v\,.\,\mathcal{S}[\![e]\!]\ (\rho[x \mapsto v])) \\
\mathcal{S}[\![e_1\ e_2]\!]\ \rho & = & (v_1 \mathrel{\underline{\in}} \mathbb{F} \mathrel{\underline{?}} (v_2 \mathrel{\underline{\in}} \mathbb{W} \mathrel{\underline{?}} \Omega \mathrel{\underline{:}} \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2) \mathrel{\underline{:}} \Omega) \text{ where } v_i = \mathcal{S}[\![e_i]\!]\ \rho \text{ for } i = 1,2 \\
\mathcal{S}[\![\mathbf{let}\ x = e\ \mathbf{in}\ e']\!]\ \rho & = & (v \mathrel{\underline{\in}} \mathbb{W} \mathrel{\underline{?}} \Omega \mathrel{\underline{:}} \mathcal{S}[\![e']\!]\ (\rho[x \mapsto v])) \text{ where } v = \mathrm{lfp}^{\preceq}_{\bot_{\mathbb{U}}} \lambda v\,.\,\mathcal{S}[\![e]\!]\ (\rho[x \mapsto v]) \\
\mathcal{S}[\![C\ e_1 \ldots e_n]\!]\ \rho & = & (v_1 \mathrel{\underline{\in}} \mathbb{W} \mathrel{\underline{?}} \Omega \mathrel{\underline{:}} \ldots (v_n \mathrel{\underline{\in}} \mathbb{W} \mathrel{\underline{?}} \Omega \mathrel{\underline{:}} \uparrow^{\mathbb{U}}_{\mathbb{A}}(C\ v_1 \ldots v_n))\ldots) \text{ where } v_i = \mathcal{S}[\![e_i]\!]\ \rho \text{ for } i \in [1,n] \\
\mathcal{S}[\![\mathbf{if}\ e_s\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e]\!]\ \rho & = & (v_s \mathrel{\underline{\in}} \mathbb{A} \mathrel{\underline{?}} \text{if } \downarrow^{\mathbb{U}}_{\mathbb{A}}(v_s) \subseteq \mathbb{Z} \text{ then (if } \downarrow^{\mathbb{U}}_{\mathbb{A}}(v_s) \neq 0 \text{ then } \mathcal{S}[\![e_t]\!]\ \rho \text{ else } \mathcal{S}[\![e_e]\!]\ \rho) \text{ else } \Omega \mathrel{\underline{:}} \Omega) \text{ where } v_s = \mathcal{S}[\![e_s]\!]\ \rho
\end{array}
$$

Figure 3. standard denotational semantics for $e \in \mathbb{E}$

3.2 Collecting Semantics

In any type discipline, a program value always has a single (not necessarily best) type, whereas each type can represent many 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 $\mathcal{S}[\![e]\!]$ straightforwardly except for conditionals that we relax to a non-deterministic choice, thus obtaining a flow-insensitive semantics:

$$
\begin{array}{rcl}
\mathcal{C}[\![\cdot]\!] & : & \mathbb{E} \to \mathscr{P}((\mathbb{X} \to \mathbb{U}_\bot) \to \mathbb{U}_\bot) \\
\mathcal{C}[\![\mathbf{if}\ e_s\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e]\!] & := & \mathcal{C}[\![e_t]\!] \cup \mathcal{C}[\![e_e]\!] \cup \\
& & \{\lambda\rho\,.\,\Omega \mid f \in \mathcal{C}[\![e_s]\!] \wedge f\rho \notin \mathbb{Z} \cup \{\bot_{\mathbb{U}}\}\}
\end{array}
$$

This concludes the discussion of syntax and semantics.

4. Abstraction to Types

This section introduces types, the abstractions for values and for an environment of values, and finally presents the derivation of 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 $\Delta \subset \mathbb{M}$. The declaration of data types gives rise to a map $\delta : \Delta \to (\mathbb{C} \to \mathbb{M}^*)$ from algebraic data types $D\ t_1 \ldots t_m \in \Delta$ to the a map from constructors $C \in \mathbb{C}$ to their argument types where $\mathbb{M}^*$ denotes the set of vectors over $\mathbb{M}$. For the sake of clarity, we fix $\Delta = \{\mathtt{Int}\} \cup \{[t] \mid t \in \mathbb{M}\}$. Constructors are thus $c \in \mathbb{Z}$, i.e. $\delta(\mathtt{Int}, c) = \emptyset$ and for lists $\delta([t], \mathbf{Nil}) = \emptyset$ and $\delta([t], \mathbf{Cons}) = \langle t, [t]\rangle$ for each $t \in \mathbb{M}$. The definition of monotypes therefore simplifies as follows:

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

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:

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

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. Define a partial order 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 $\mathbb{P}^{\bot}_{\equiv_{\mathbb{P}}}$ to denote the set of $\equiv_{\mathbb{P}}$-equivalence classes of terms augmented with $\bot_{\mathbb{P}}$. Then

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

is a complete lattice with join $\mathbf{lca}$, meet $\mathbf{gci}$, top element $(\mathrm{a})_{\equiv_{\mathbb{P}}}$ and a bottom element $\bot_{\mathbb{P}}$ with $\bot_{\mathbb{P}} \sqsubseteq_{\mathbb{P}} t \sqsubseteq_{\mathbb{P}} (\mathrm{a})_{\equiv_{\mathbb{P}}}$ for all elements $t \in \mathbb{P}^{\bot}_{\equiv_{\mathbb{P}}}$. 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 $t = \bot_{\mathbb{P}}$ otherwise. For example, $\mathbf{gci}([\mathrm{a}] \to [\mathtt{Int}], [\mathtt{Int}] \to \mathrm{b}) = [\mathtt{Int}] \to [\mathtt{Int}] = \sigma([\mathrm{a}] \to [\mathtt{Int}])$ with $\sigma = \mathbf{mgu}([\mathrm{a}] \to [\mathtt{Int}], [\mathtt{Int}] \to \mathrm{b}) = \{\mathrm{a}/\mathtt{Int}, \mathrm{b}/[\mathtt{Int}]\}$. The dual operation $\mathbf{lca} : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ calculates the least common anti-instance [11]. For instance, $\mathbf{lca}([\mathtt{Int}] \to \mathrm{a}, [[\mathtt{Int}]] \to \mathrm{a}) = [\mathrm{b}] \to \mathrm{a}$. Note that $\mathbf{lca}$ preserves type variables when they occur in compatible locations. In abuse of syntax, we also use $\mathbf{lca} : \mathscr{P}(\mathbb{M}) \to \mathbb{P} \cup \{\bot_{\mathbb{P}}\}$ on sets of monotypes with $\mathbf{lca}(\emptyset) := \bot_{\mathbb{P}}$. 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 $\mathbf{ground} : \mathbb{P}^{\bot}_{\equiv_{\mathbb{P}}} \to \mathscr{P}(\mathbb{M})$ as γ with $\mathbf{ground}(\bot_{\mathbb{P}}) := \emptyset$:

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

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.

4.2 Relating Values and Types

The basic ingredients to relating program values and types are the functions $\alpha_{\mathbb{M}} : \mathbb{U}_\bot \to \mathscr{P}(\mathbb{M})$ and $\gamma_{\mathbb{M}} : \mathbb{M} \to \mathscr{P}(\mathbb{U}_\bot)$ 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, $\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(\mathbf{Nil})) = \{[\mathtt{Int}], [\mathtt{Int} \to \mathtt{Int}], \ldots\}$ which can be turned into a polymorphic type $\mathbf{lca}(\alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{A}}(\mathbf{Nil}))) = [\mathrm{a}]$. The next section addresses how to abstract whole environments.

[1] A Galois insertion is a Galois connection where γ is injective which, in this context, follows from the use of equivalence classes for the type lattice.

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

Figure 4. abstracting and concretizing values to types

4.3 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,* $[id \mapsto (\mathrm{a} \to \mathrm{a})_{\equiv_{\mathbb{P}}}, x \mapsto (\mathrm{a} \to \mathrm{a})_{\equiv_{\mathbb{P}}}] = [id \mapsto \mathrm{a} \to \mathrm{a}, x \mapsto \mathrm{b} \to \mathrm{b}]$.

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 $\langle (\mathbb{X} \to \mathbb{P})^{\bot}_{\equiv_{\mathbb{P}}}, \overline{\sqsubseteq_{\mathbb{P}}}, \overline{\mathbf{lca}}, \overline{\mathbf{gci}}, [x \mapsto (\mathrm{a})_{\equiv_{\mathbb{P}}}]_{x \in \mathbb{X}}, \bot_{\mathbb{P}}\rangle$. 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 function type $t_1 \to \ldots \to t_n$, applying $\mathbf{gci}$ (resp. $\mathbf{lca}$), and converting the result back to a vector. We consider an example that contrasts vector and point-wise application of $\mathbf{lca}$:

$$
\begin{array}{ccccc}
\mathscr{P}(\mathbb{X} \to \mathbb{U}_\bot) & \overset{\gamma^{\mathcal{X}}_{\mathbb{M}}}{\underset{\alpha^{\mathcal{X}}_{\mathbb{M}}}{\leftrightarrows}} & \mathscr{P}(\mathbb{X} \to \mathbb{M}) & \overset{\overline{\mathbf{ground}}}{\underset{\overline{\mathbf{lca}}}{\leftrightarrows}} & (\mathbb{X} \to \mathbb{P})^{\bot}_{\equiv_{\mathbb{P}}} \\
\downarrow f & & \downarrow f^{\sharp}_{\mathbb{M}} & & \downarrow f^{\sharp}_{\mathbb{P}} \\
\mathscr{P}(\mathbb{X} \cup \{\kappa\} \to \mathbb{U}_\bot) & \overset{\gamma^{\mathcal{X}}_{\mathbb{M}}}{\underset{\alpha^{\mathcal{X}}_{\mathbb{M}}}{\leftrightarrows}} & \mathscr{P}(\mathbb{X} \cup \{\kappa\} \to \mathbb{M}) & \overset{\overline{\mathbf{ground}}}{\underset{\overline{\mathbf{lca}}}{\leftrightarrows}} & (\mathbb{X} \cup \{\kappa\} \to \mathbb{P})^{\bot}_{\equiv_{\mathbb{P}}}
\end{array}
$$

Figure 5. our approach to abstracting denotational semantics

EXAMPLE 4. *Suppose two identifiers* f, g *can take on the following two monotypes* $\vec{t_1}, \vec{t_2} \in \mathbb{M}^{\langle f,g\rangle}$:

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

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

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

This result is an anti-instance since applying $\sigma_1 = \{a/\texttt{Int} \to [\texttt{Int}], c/\texttt{Int}\}$ *yields the first and* $\sigma_2 = \{a/[\texttt{Int}] \to \texttt{Int}, c/[\texttt{Int}]\}$ *the second row. By way of contrast, applying* $\mathbf{lca}$ *point-wise, that is,* $\langle \mathbf{lca}(\{\vec{t_1}(f), \vec{t_2}(f)\}), \mathbf{lca}(\{\vec{t_1}(g), \vec{t_2}(g)\})\rangle \in \mathbb{P}^{\{f,g\}}$ *results in*

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

which is an anti-instance but not the least-common one as it represents more monotypes than $\langle \texttt{Int} \to \mathrm{a} \to \mathrm{c}, \texttt{Int} \to \mathrm{a} \to [\mathrm{c}]\rangle$.

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:

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

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 $\overline{\mathbf{ground}}(\langle \mathrm{a}, \mathrm{b}, \texttt{Int}\rangle) = \{\langle t_1, t_2, \texttt{Int}\rangle \mid t_1 \in \mathbb{M}, t_2 \in \mathbb{M}\}$ whereas $\overline{\mathbf{ground}}(\langle \mathrm{a} \to \texttt{Int}, \mathrm{a}\rangle) = \{\langle t \to \texttt{Int}, t\rangle \mid t \in \mathbb{M}\}$. 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 $\lambda\rho\,.\,\mathcal{S}[\![\cdot]\!]\ \rho : (\mathbb{X} \to \mathbb{U}_\bot) \to \mathbb{U}_\bot$ we abstract $\lambda\rho\,.\,\rho[\kappa \mapsto (\mathcal{S}[\![\cdot]\!]\ \rho)] : (\mathbb{X} \to \mathbb{U}_\bot) \to (\mathbb{X} \cup \{\kappa\} \to \mathbb{U}_\bot)$ 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 presents three functions that transform concrete (value) environments to abstract (set of type vector) environments and vice-versa.

$$
\begin{array}{rcl}
\gamma_{\mathbb{M}}^{\mathcal{X}} & : & \mathscr{P}(\mathbb{X} \to \mathbb{M}) \to \mathscr{P}(\mathbb{X} \to \mathbb{U}_\perp) \\
\gamma_{\mathbb{M}}^{\mathcal{X}}(\overline{\rho}_{\mathbb{M}}) & := & \bigcup\{\bar{\rho} \subseteq \mathbb{X} \to \mathbb{U}_\perp \mid \overline{\rho}_{\mathbb{M}} \subseteq \alpha_{\mathbb{M}}^{\mathcal{X}}(\bar{\rho})\} = \{\rho \in \mathbb{X} \to \mathbb{U}_\perp \mid \overline{\rho}_{\mathbb{M}} \subseteq \alpha_{\mathbb{M}1}^{\mathcal{X}}(\rho)\} \\
\alpha_{\mathbb{M}1}^{\mathcal{X}} & : & (\mathbb{X} \to \mathbb{U}_\perp) \to \mathscr{P}(\mathbb{X} \to \mathbb{M}) \\
\alpha_{\mathbb{M}1}^{\mathcal{X}}(\rho) & := & \bigcup\left\{\overline{\rho}_{\mathbb{M}} \subseteq \mathbb{X} \to \mathbb{M} \,\middle|\, \begin{pmatrix} \forall \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}, x \in \mathbb{X}\,. \\ \rho_{\mathbb{M}}(x) \in \alpha_{\mathbb{M}}(\rho(x)) \end{pmatrix} \wedge \overbrace{\begin{array}{l} \forall\{x_1, \ldots x_n\} \in \mathit{mono}(P)\,.\,\forall \rho_{\mathbb{M}}^1, \rho_{\mathbb{M}}^2 \in \overline{\rho}_{\mathbb{M}}\,.\,\forall i,j \in [1,n]\,. \\ \rho_{\mathbb{M}}^1(x_i) = \rho_{\mathbb{M}}^2(x_i) \Rightarrow \rho_{\mathbb{M}}^1(x_j) = \rho_{\mathbb{M}}^2(x_j) \end{array}}^{\text{monotype restriction}} \right\} \\
\alpha_{\mathbb{M}}^{\mathcal{X}} & : & \mathscr{P}(\mathbb{X} \to \mathbb{U}_\perp) \to \mathscr{P}(\mathbb{X} \to \mathbb{M}) \\
\alpha_{\mathbb{M}}^{\mathcal{X}}(\bar{\rho}) & := & \bigcap_{\rho \in \bar{\rho}} \alpha_{\mathbb{M}1}^{\mathcal{X}}(\rho) \\
\mathit{lift}_{\mathbb{M}}^{\mathcal{X}} & : & \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}_{\mathbb{M}}^{\mathcal{X}}(C) & := & \alpha_{\mathbb{M}}^{\mathcal{X}} \circ \lambda\bar{\rho}\,.\,\{\rho[\kappa \mapsto S\rho] \mid S \in C, \rho \in \bar{\rho}\} \circ \gamma_{\mathbb{M}}^{\mathcal{X}} = \lambda\overline{\rho}_{\mathbb{M}}\,.\,\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_\perp} \{\alpha_{\mathbb{M}1}^{\mathcal{X}}(\rho[\kappa \mapsto S\rho]) \mid S \in C \wedge \overline{\rho}_{\mathbb{M}} \subseteq \alpha_{\mathbb{M}1}^{\mathcal{X}}(\rho)\}
\end{array}
$$

Figure 6. abstracting the collecting semantics to vectors of types

Specifically, the concretization function $\gamma_{\mathbb{M}}^{\mathcal{X}}$ 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: $\alpha_{\mathbb{M}1}^{\mathcal{X}}$ takes a single environment ρ to matching type environments, while $\alpha_{\mathbb{M}}^{\mathcal{X}}$ lifts this operation to sets of concrete environments $\bar{\rho}$. The function abstracting a single environment, $\alpha_{\mathbb{M}1}^{\mathcal{X}}$ features a restriction dubbed *monotype restriction* that enforces that λ-bound variables take on fewer types than $\mathbf{let}$-bound variables. We will present this restriction in Sect. 4.6. The fourth function $\mathit{lift}_{\mathbb{M}}^{\mathcal{X}}$ 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 $\mathit{lift}_{\mathbb{M}}^{\mathcal{X}}$ to each syntactic construct in Fig. 2, commencing with $\mathit{lift}_{\mathbb{M}}^{\mathcal{X}}(\{\mathcal{S}[\![x]\!]\})$.

4.5 Lifting Variable Accesses to Sets of Types

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

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

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 types may not be the same. Consider the evaluation for the set $\{\rho_{\mathbb{M}}^1, \ldots \rho_{\mathbb{M}}^m\} = \overline{\rho}_{\mathbb{M}}$ and a corresponding value environment ρ with $\overline{\rho}_{\mathbb{M}} = \alpha_{\mathbb{M}1}^{\mathcal{X}}(\rho)$. We expand $\alpha_{\mathbb{M}1}^{\mathcal{X}}(\rho[\kappa \mapsto \rho(x)])$, ignoring the monotype restriction for now:

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

We now address the task of finding $\overline{\rho}_{\mathbb{M}}^\kappa$ 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 $\exists_\kappa(\overline{\rho}_{\mathbb{M}}^\kappa) = \overline{\rho}_{\mathbb{M}}$ where $\overline{\rho}_{\mathbb{M}} := \{\rho_{\mathbb{M}}^1, \ldots \rho_{\mathbb{M}}^m\}$ and $\rho_{\mathbb{M}}^1, \ldots \rho_{\mathbb{M}}^m$ are defined as above. The challenge is to extend each $\rho_{\mathbb{M}}^i$ with a binding for κ. One possibility is to bind κ in each $\rho_{\mathbb{M}}^i$ to $\rho_{\mathbb{M}}^j(x)$, yielding $\{\rho_{\mathbb{M}}^i[\kappa \mapsto \rho_{\mathbb{M}}^j(x)] \mid i,j \in [1,m]\}$. However, this definition ignores the fact that $\rho_{\mathbb{M}}^j(y) = \rho_{\mathbb{M}}^i(y)$ for all $y \prec x$ since otherwise $\rho_{\mathbb{M}}^j$ and $\rho_{\mathbb{M}}^i$ correspond to different evaluations of the program. Thus, we could define $\overline{\rho}_{\mathbb{M}}^\kappa$ as:

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

Since the definition above 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 1. *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 $\mathit{add}_x : \mathcal{D}^{\mathbb{X}} \to \mathcal{D}^{\mathbb{X} \cup \{x\}}$ augment a domain element with a new dimension set to top, that is, $\mathit{add}_x(\bar{d}) = \{d[x \mapsto \top] \mid d \in \bar{d}\}$. We also use its natural lifting to sets of variables $\mathit{add}_X : \mathcal{D}^{\mathbb{X}} \to \mathcal{D}^{\mathbb{X} \cup X}$. Let $y_1, \ldots y_n, y_1', \ldots y_n'$ be pair-wise different. Then let $\mathit{swap}_{y_1 \ldots y_n, y_1' \ldots y_n'} : \mathcal{D}^{\mathbb{X}} \to \mathcal{D}^{\mathbb{X}}$ 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:*

$$
\begin{array}{rcl}
\mathit{expand}_{y_1 \ldots y_n, y_1' \ldots y_n'} & : & \mathcal{D}^{\mathbb{X}} \to \mathcal{D}^{\mathbb{X} \cup \{y_1' \ldots y_n'\}} \\
\mathit{expand}_{y_1 \ldots y_n, y_1' \ldots y_n'}(\bar{d}) & = & \mathit{add}_{y_1' \ldots y_n'}(\bar{d}) \\
& \sqcap & \mathit{swap}_{y_1 \ldots y_n, y_1' \ldots y_n'}(\mathit{add}_{y_1' \ldots y_n'}(\bar{d}))
\end{array}
$$

Let $\exists_{y_1,\ldots y_n}(\bar{d}) = \exists_{y_1}(\ldots \exists_{y_n}(\bar{d}) \ldots)$. Then the set $\overline{\rho}_{\mathbb{M}}^\kappa$ can now alternatively be defined using expand as follows:

$$\overline{\rho}_{\mathbb{M}}^{\kappa'} = \exists_{y_1',\ldots y_n'}(\mathit{expand}_{xy_1 \ldots y_n, \kappa y_1' \ldots y_n'}(\overline{\rho}_{\mathbb{M}}))$$

Here, $x \prec y_i$ holds for all $i \in [1,n]$. The following proposition ascertains the equivalence:

PROPOSITION 1. *Instantiation of x is equivalent to expanding $x, y_1, \ldots y_n$ and projecting out $y_1, \ldots y_n$ where $\{y_1, \ldots y_n\} = \{y \in \mathbf{dom}(\overline{\rho}_{\mathbb{M}}) \mid x \prec y\}$.*

Proof. We show $\overline{\rho}_{\mathbb{M}}^{\kappa'} = \overline{\rho}_{\mathbb{M}}^\kappa$. Set $\bar{v} = \mathit{add}_{\kappa y_1' \ldots y_n'}(\overline{\rho}_{\mathbb{M}}) = \{\rho_{\mathbb{M}}[\kappa \mapsto t_0, y_1' \mapsto t_1, \ldots y_n' \mapsto t_n] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}, t_0, \ldots t_n \in \mathbb{M}\}$. Then $\overline{\rho}_{\mathbb{M}}^{\kappa'} = \exists_{y_1',\ldots y_n'}(\bar{v}) \cap \exists_{y_1',\ldots y_n'}(\mathit{swap}_{xy_1 \ldots y_n, \kappa y_1' \ldots y_n'}(\bar{v})) = \mathit{add}_\kappa(\overline{\rho}_{\mathbb{M}}) \cap \mathit{add}_{y_1,\ldots y_n}(\mathit{swap}_{x,\kappa}(\exists_{y_1,\ldots y_n}(\overline{\rho}_{\mathbb{M}})) = \mathit{add}_\kappa(\overline{\rho}_{\mathbb{M}}) \cap \mathit{add}_{y_1,\ldots y_n}(\exists_{y_1,\ldots y_n}(\{\rho_{\mathbb{M}}[\kappa \mapsto \rho_{\mathbb{M}}(x)] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}\})) = \overline{\rho}_{\mathbb{M}}^\kappa$. ■

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 $f = \lambda x\,.\,\mathbf{let}\ g = \lambda y\,.\,x\ \mathbf{in}\ g\,g$. In*

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

Figure 7. calculating the owner of a variable

order to simplify the presentation, we assume that the environment $\rho_{\mathbb{P}} = [x \mapsto \mathrm{a}, g \mapsto \mathrm{b} \rightarrow \mathrm{a}]$ does not hold a type for f. Since $\rho_{\mathbb{P}}$ contains no variables y with $g \prec y$, the abstract semantics of $\rho(g)$ merely expands g to κ by adding a binding to a $\top$ variable, $\kappa \mapsto \mathrm{c}$, and calculating the meet of this new environment with the same environment where g and κ are swapped:

$$
\begin{array}{rll}
& & \quad\;\; x \quad\;\; g \qquad\;\; \kappa \\
expand_{g,\kappa}(\rho_{\mathbb{P}}) & = \overline{\mathbf{gci}} & \begin{pmatrix} \langle \mathrm{a}\;,\; \mathrm{b} \!\rightarrow\! \mathrm{a}\;,\; \mathrm{c} \quad\;\; \rangle_{\equiv_{\mathbb{P}}}, \\ \langle \mathrm{a}\;,\; \mathrm{c} \quad\;\;,\; \mathrm{b} \!\rightarrow\! \mathrm{a} \rangle_{\equiv_{\mathbb{P}}} \end{pmatrix} \\
& = \overline{\mathbf{gci}} & \begin{pmatrix} \langle \mathrm{a}\;,\; \mathrm{b} \!\rightarrow\! \mathrm{a}\;,\; \mathrm{c} \quad\;\; \rangle, \\ \langle \mathrm{d}\;,\; \mathrm{f} \quad\;\;,\; \mathrm{e} \!\rightarrow\! \mathrm{d} \rangle \end{pmatrix} \\
& = & \;\; \langle \mathrm{a}\;,\; \mathrm{b} \!\rightarrow\! \mathrm{a}\;,\; \mathrm{e} \!\rightarrow\! \mathrm{a} \rangle_{\equiv_{\mathbb{P}}}
\end{array}
$$

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 line. The result contains a fresh instance $\rho_{\mathbb{P}}(\kappa) = \mathrm{e} \rightarrow \mathrm{a}$ of g whose return type has the same type 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, the previous example hints at the fact that $expand_{xy_1\ldots y_m, x'y'_1\ldots y'_m}$ 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 $\mathbf{let}$-rule of the $\mathcal{W}$-algorithm [3].

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 2. *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 $mono : \mathbb{E} \rightarrow \mathscr{P}(\mathscr{P}(\mathbb{X}))$ 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 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. See App. [18].

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 $lift^{\mathcal{X}}_{\mathbb{M}}(\{\mathcal{S}[\![x]\!]\})$. 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 $lift^{\mathcal{X}}_{\mathbb{M}}(\{\mathcal{S}[\![x]\!]\}) = \lambda\overline{\rho}_{\mathbb{M}} \, . \, \{\rho_{\mathbb{M}}[\kappa \mapsto \rho_{\mathbb{M}}(x)] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}\}$. 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* iff $\alpha \circ f = f^\sharp \circ \alpha$ [5, 15]. 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.) The following assert this for mono- and polytypes (Proof in [18]):

LEMMA 1. *For all $\rho_1, \rho_2 \in \mathbb{X} \rightarrow \mathbb{U}_\bot$ with $\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_1) = \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho_2)$ it follows that $\alpha_{\mathbb{M}}(\mathcal{S}[\![e]\!] \, \rho_1) = \alpha_{\mathbb{M}}(\mathcal{S}[\![e]\!] \, \rho_2)$ for all $e \in \mathbb{E}$.*

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

Lemma 2 now allows us to derive backward complete abstract transformers as $f^\sharp_{\mathbb{P}} = \alpha \circ f \circ \gamma$ for each concrete transformer 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. For brevity, we omit the development of the last two rules. The abstraction map $lift^{\mathcal{X}}_{\mathbb{M}}$ in Fig. 8 takes a collecting semantics to a semantics over sets of

$$\mathcal{T}[\![\cdot]\!] \quad : \quad \mathbb{E} \to (\mathbb{X} \to \mathbb{P})^{\perp}_{\equiv_{\mathbb{P}}} \to (\mathbb{X} \cup \{\kappa\} \to \mathbb{P})^{\perp}_{\equiv_{\mathbb{P}}}$$

$$\mathcal{T}[\![x]\!] \quad = \quad \overline{\mathbf{lca}} \circ \lambda \overline{\rho}_{\mathbb{M}} \,.\, \bigcap \{\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho[\kappa \mapsto \mathcal{S}[\![x]\!]\ \rho]) \mid \rho \in \gamma^{\mathcal{X}}_{\mathbb{M}}(\overline{\rho}_{\mathbb{M}})\} \circ \overline{\mathbf{ground}}$$

$$\stackrel{(1)}{=} \lambda \rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}(\textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \{\alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho[\kappa \mapsto \rho(x)]) \mid \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\})$$

$$\stackrel{(2)}{=} \lambda \rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}\Big(\textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \Big\{ \bigcup \Big\{ \overline{\rho}_{\mathbb{M}} \subseteq \mathbb{X} \cup \{\kappa\} \to \mathbb{M} \,\Big|\, \begin{array}{l} \forall \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}, y \in \mathbb{X} \cup \{\kappa\} \,.\, \rho_{\mathbb{M}}(y) \in \alpha_{\mathbb{M}}(\rho[\kappa \mapsto \rho(x)](y)) \wedge \\ \text{monotype restr. } \wedge\, \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \exists_{\kappa}(\overline{\rho}_{\mathbb{M}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho) \end{array} \Big\} \Big\} \Big)$$

$$\text{if } x \notin \mathbb{X}^{\lambda} \stackrel{(3)}{=} \lambda \rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}(\exists_{y'_1,\ldots y'_n}(\mathit{expand}_{xy_1\ldots y_n,\kappa y'_1\ldots y'_n}(\overline{\mathbf{ground}}(\rho_{\mathbb{P}})))) \text{ where } \{y_1,\ldots y_n\} = \{y \in \mathbf{dom}(\rho_{\mathbb{P}}) \mid x \prec y\}$$

$$\text{if } x \notin \mathbb{X}^{\lambda} \stackrel{(4)}{=} \lambda \rho_{\mathbb{P}} \,.\, \exists_{y'_1,\ldots y'_n}(\mathit{expand}_{xy_1\ldots y_n,\kappa y'_1\ldots y'_n}(\rho_{\mathbb{P}})) \text{ where } \{y_1,\ldots y_n\} = \{y \in \mathbf{dom}(\rho_{\mathbb{P}}) \mid x \prec y\}$$

$$\text{if } x \in \mathbb{X}^{\lambda} \stackrel{(5)}{=} \lambda \rho_{\mathbb{P}} \,.\, \rho_{\mathbb{P}}[\kappa \mapsto \rho_{\mathbb{P}}(x)]$$

$$\mathcal{T}[\![\lambda x \,.\, e]\!] \stackrel{(6)}{=} \overline{\mathbf{lca}} \circ \lambda \overline{\rho}_{\mathbb{M}} \,.\, (\textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} (\bigcup \{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(\mathcal{S}[\![\lambda x \,.\, e]\!]\ \rho) \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\})) \circ \overline{\mathbf{ground}}$$

$$\stackrel{(7)}{=} \lambda \rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(\mathcal{S}[\![\lambda x \,.\, e]\!]\ \rho) \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\}$$

$$\stackrel{(8)}{=} \lambda \rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(\uparrow^{\mathbb{U}}_{\mathbb{F}}(\lambda v \,.\, \mathcal{S}[\![e]\!]\ (\rho[x \mapsto v]))) \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\}$$

$$\stackrel{(9)}{=} \lambda \rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t_1 \to t_2] \mid v_2 = \mathcal{S}[\![e]\!]\ (\rho[x \mapsto v_1]) \wedge v_1 \in \mathbb{U} \wedge t_i \in \alpha_{\mathbb{M}}(v_i) \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \ldots\}$$

$$\stackrel{(10)}{=} \lambda \rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t_1 \to t_2] \mid t_2 = \mathcal{T}[\![e]\!](\rho_{\mathbb{M}}[x \mapsto t_1])\kappa \wedge t_1 \in \mathbb{M} \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}}) \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\}$$

$$\stackrel{(11)}{=} \lambda \rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t_1 \to t_2] \mid t_2 = \mathcal{T}[\![e]\!](\rho_{\mathbb{M}}[x \mapsto t_1])\kappa \wedge t_1 \in \mathbb{M} \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}})\}$$

$$\stackrel{(12)}{=} \lambda \rho_{\mathbb{P}} \,.\, \begin{cases} \rho'_{\mathbb{P}}[\kappa \mapsto t_1 \to t_2] & \text{if } \perp_{\mathbb{P}} \neq \rho'_{\mathbb{P}} = \mathcal{T}[\![e]\!](\rho_{\mathbb{P}}[x \mapsto (\mathrm{a})_{\equiv_{\mathbb{P}}}]) \text{ and } t_1 = \rho'_{\mathbb{P}}(x) \text{ and } t_2 = \rho'_{\mathbb{P}}(\kappa) \\ \perp_{\mathbb{P}} & \text{otherwise} \end{cases}$$

$$\mathcal{T}[\![e_1\ e_2]\!] \quad = \quad \overline{\mathbf{lca}} \circ \lambda \overline{\rho}_{\mathbb{M}} \,.\, (\textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} (\bigcup \{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(\mathcal{S}[\![e_1\ e_2]\!]\ \rho) \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\})) \circ \overline{\mathbf{ground}}$$

$$\stackrel{(13)}{=} \lambda \rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1) \in \mathbb{F} \wedge v_2 \neq \Omega \wedge t \in \alpha_{\mathbb{M}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2) \wedge \mathbf{env}\} \text{ where } \mathbf{env} \equiv v_i = \mathcal{S}[\![e_i]\!]\ \rho \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)$$

$$\stackrel{(14)}{=} \lambda \rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid \exists t_a, t_r \in \mathbb{M} \,.\, \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1) \in \mathbb{F} \wedge v_2 \neq \Omega \wedge \uparrow^{\mathbb{U}}_{\mathbb{F}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)) \in \gamma_{\mathbb{M}}(t_a \to t_r) \wedge \forall v \in \gamma_{\mathbb{M}}(t_a) \,.\, \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v \in \gamma_{\mathbb{M}}(t_r) \setminus \perp_{\mathbb{P}} \wedge t \in \alpha_{\mathbb{M}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2) \wedge \mathbf{env}\}$$

$$\stackrel{(15)}{=} \lambda \rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid \exists t_r \in \mathbb{M} \,.\, v_1 \in \gamma_{\mathbb{M}}(t_a \to t_r) \wedge v_2 \in \gamma_{\mathbb{M}}(t_a) \wedge \downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2 \in \gamma_{\mathbb{M}}(t_r) \wedge t \in \alpha_{\mathbb{M}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2) \wedge \mathbf{env}\}$$

$$\stackrel{(16)}{=} \lambda \rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t_r] \mid \exists t_r \in \mathbb{M} \,.\, v_1 \in \gamma_{\mathbb{M}}(t_a \to t_r) \wedge v_2 \in \gamma_{\mathbb{M}}(t_a) \wedge \mathbf{env}\}$$

$$\stackrel{(17)}{=} \lambda \rho_{\mathbb{P}} \,.\, \begin{cases} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t_r] \mid \exists t_r \in \mathbb{M} \,.\, t_1 = t_a \to t_r \wedge t_2 = t_a \wedge t_i = \mathcal{T}[\![e_i]\!]\rho_{\mathbb{M}}\,\kappa \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}})\} & \text{if } \mathcal{T}[\![e_i]\!]\rho_{\mathbb{M}} \neq \perp_{\mathbb{P}} \\ \perp_{\mathbb{P}} & \text{otherwise} \end{cases}$$

$$\stackrel{(18)}{=} \lambda \rho_{\mathbb{P}} \,.\, \begin{cases} \rho'_{\mathbb{P}}[\kappa \mapsto t_r] & \text{if } \perp_{\mathbb{P}} \neq \rho'_{\mathbb{P}} = \overline{\mathbf{gci}}(\mathcal{T}[\![e_1]\!]\rho_{\mathbb{P}}, \rho_{\mathbb{P}}[\kappa \mapsto (\mathcal{T}[\![e_2]\!]\rho_{\mathbb{P}}\kappa) \to (\mathrm{a})_{\equiv_{\mathbb{P}}}]) \text{ and } \mathcal{T}[\![e_i]\!]\rho_{\mathbb{P}} \neq \perp_{\mathbb{P}} \text{ and let } t_a \to t_r = \rho'_{\mathbb{P}}(\kappa) \\ \perp_{\mathbb{P}} & \text{otherwise} \end{cases}$$

$$\mathcal{T}[\![\mathbf{let}\ x = e\ \mathbf{in}\ e']\!] = \overline{\mathbf{lca}} \circ \lambda \overline{\rho}_{\mathbb{M}} \,.\, (\textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} (\bigcup \{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid t \in \alpha_{\mathbb{M}}(\mathcal{S}[\![\mathbf{let}\ x = e\ \mathbf{in}\ e']\!]\ \rho) \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \subseteq \alpha^{\mathcal{X}}_{\mathbb{M}1}(\rho)\})) \circ \overline{\mathbf{ground}}$$

$$\stackrel{(19)}{=} \lambda \rho_{\mathbb{P}} \,.\, \textstyle\bigcap_{\rho \in \mathbb{X} \to \mathbb{U}_{\perp}} \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t'] \mid v \neq \Omega \wedge t' \in \alpha_{\mathbb{M}}(\mathcal{S}[\![e']\!]\ \rho[x \mapsto v]) \wedge v = \mathrm{lfp}^{\preceq}_{\perp_{\mathbb{U}}} \lambda v \,.\, \mathcal{S}[\![e]\!]\ (\rho[x \mapsto v]) \wedge \rho_{\mathbb{M}} \in \ldots\}$$

$$\stackrel{(20)}{=} \lambda \rho_{\mathbb{P}} \,.\, \overline{\mathbf{lca}}\{\rho_{\mathbb{M}}[\kappa \mapsto t'] \mid t' \in \mathcal{T}[\![e']\!](\rho_{\mathbb{M}}[x \mapsto t])\kappa \wedge t \in \mathrm{gfp}^{\subseteq}_{\mathbb{M}} \lambda T \,.\, \textstyle\bigcap_{t \in T}(\mathcal{T}[\![e]\!]\rho_{\mathbb{M}}[x \mapsto t])\kappa \wedge \rho_{\mathbb{M}} \in \overline{\mathbf{ground}}(\rho_{\mathbb{P}})\}$$

$$\stackrel{(21)}{=} \lambda \rho_{\mathbb{P}} \,.\, \begin{cases} \mathcal{T}[\![e']\!]\rho_{\mathbb{P}}[x \mapsto t] & \text{if } \perp_{\mathbb{P}} \neq t = \mathrm{gfp}^{\sqsubseteq_{\mathbb{P}}}_{(\mathrm{a})_{\equiv_{\mathbb{P}}}} \lambda t \,.\, \mathcal{T}[\![e]\!](\rho_{\mathbb{P}}[x \mapsto t])\kappa \\ \perp_{\mathbb{P}} & \text{otherwise} \end{cases}$$

$$\mathcal{T}[\![\mathbf{Cons}\ l\ ll]\!] \stackrel{(22)}{=} \lambda \rho_{\mathbb{P}} \,.\, \begin{cases} \overline{\mathbf{gci}}(\rho'_{\mathbb{P}}[\kappa \mapsto [t]], \mathcal{T}[\![ll]\!]\rho_{\mathbb{P}}) & \text{if } \perp_{\mathbb{P}} \neq \rho'_{\mathbb{P}} = \mathcal{T}[\![l]\!]\rho_{\mathbb{P}} \text{ and } t = \rho'_{\mathbb{P}}\kappa \\ \perp_{\mathbb{P}} & \text{otherwise} \end{cases} \qquad \begin{array}{l} \mathcal{T}[\![\mathbf{Nil}]\!] = \lambda \rho_{\mathbb{P}} \,.\, \rho_{\mathbb{P}}[\kappa \mapsto [(\mathrm{a})_{\equiv_{\mathbb{P}}}]] \\ \mathcal{T}[\![c]\!] = \lambda \rho_{\mathbb{P}} \,.\, \rho_{\mathbb{P}}[\kappa \mapsto \mathtt{Int}] \text{ for } c \in \mathbb{Z} \end{array}$$

$$\mathcal{T}[\![\mathbf{if}\ e_s\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e]\!] \stackrel{(23)}{=} \lambda \rho_{\mathbb{P}} \,.\, \mathbf{if}\ \mathcal{T}[\![e_s]\!]\rho_{\mathbb{P}} = \perp_{\mathbb{P}} \vee \mathtt{Int} \not\sqsubseteq_{\mathbb{P}} (\mathcal{T}[\![e_s]\!]\rho_{\mathbb{P}})\kappa\ \mathbf{then}\ \perp_{\mathbb{P}}\ \mathbf{else}\ \overline{\mathbf{gci}}(\mathcal{T}[\![e_t]\!]\rho_{\mathbb{P}}, \mathcal{T}[\![e_e]\!]\rho_{\mathbb{P}})$$

Figure 8. deriving the type inference algorithm by abstracting the collecting semantics

type vectors that 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 $(\mathcal{T}[\![e]\!]\rho)\kappa$. We consider each rule in Fig 8 in turn.

Variable. Equation (1) expands $\mathcal{S}[\![x]\!]$ 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 $\mathbf{let}$-bound variables and λ-bound variables. The former translates to an expand construct in Eq. (3) that Eq. (4) lifts to polytypes. Sect. 4.7 stipulated Eq. (5) for λ-bound variables.

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

Application. Expand the standard semantics (13) and $\alpha_{\mathbb{M}}$ (14). Reduce $\uparrow^{\mathbb{U}}_{\mathbb{F}}(\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)) = v_1$ since $v_1 \in \mathbb{F}$, choose v_2 for e which fixes t_a (15). Note here that no types are generated when $\downarrow^{\mathbb{U}}_{\mathbb{F}}(v_1)v_2$ is $\perp_{\mathbb{U}}$. Equation (16) by identity $\alpha_{\mathbb{M}} \circ \gamma_{\mathbb{M}}$, leading to filter expression (17). The latter translates to a meet expression $\overline{\mathbf{gci}}$, i.e., unification (18).

Binding. The expanded semantic (19) is lifted to types (20) and the lfp is replaced with a greatest-fixpoint computation. Equation (21) lifts this fixpoint computation to one on polymorphic types.

$$\frac{x \in \mathbb{X}^\lambda}{\rho_\mathbb{P} \vdash x : \rho_\mathbb{P}(x)} \quad \text{(VAR-LAM)}$$

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

$$\frac{\rho_\mathbb{P}[x \mapsto t_1] \vdash e : t_2 \qquad t_1 \text{ fresh}}{\rho_\mathbb{P} \vdash \lambda x \,.\, e : t_1 \to t_2} \quad \text{(ABS)}$$

$$\frac{\rho_\mathbb{P} \vdash e_1 : t_1 \qquad \rho_\mathbb{P} \vdash e_2 : t_2 \qquad \sigma = \mathbf{mgu}(t_2 \to t_r, t_1) \qquad t_r \text{ fresh}}{\sigma\rho_\mathbb{P} \vdash e_1\, e_2 : \sigma t_r} \quad \text{(APP)}$$

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

Figure 9. inference rules deduced from the semantic equations

Constructors. The meet $\overline{\mathbf{gci}}$ in Eq. (22) is due to the intersection in the $\alpha_\mathbb{M}(\uparrow^\mathbb{U}_\mathbb{A}(\mathbf{Cons}\ l\ ll))$ rule of Fig. 4. Other constructors analog.

Conditional. Follows the collecting semantics in Sect. 3.2, thus returning $\bot_\mathbb{P}$ if the condition does not evaluate to a value in $\mathbb{Z}$.

4.10 Translation from Equations to Inference Rules

In order to illustrate the difference between the inference algorithm in Fig. 8 and typing rules, we translate them by writing each rule $\lambda\rho_\mathbb{P} \,.\, t$ if $cond$ in Fig. 8 as an inference rule with $cond$ as antecedent and $\rho_\mathbb{P} \vdash t$ 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, the resulting typing rules seem to reduce the inference algorithm in Fig. 8 to a mere specification.

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 $\gamma_\mathbb{S}(s) = \{\langle v_1, \dots v_n\rangle \in \mathbb{Z}^{|\mathbb{X}|} \mid s \text{ holds}\}$. For each $s \in \mathbb{S}$ let $s_{\equiv_\mathbb{S}}$ denote a 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.

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', calculate $\sigma = [\mathrm{a}_1/\mathrm{b}_1, \dots \mathrm{a}_n/\mathrm{b}_n] = \mathbf{mgu}(t, t')$ and update the size domain s to $expand_{\mathrm{a}_1 \dots \mathrm{a}_n, \mathrm{b}_1 \dots \mathrm{b}_n}(s)$.

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 10. performance of type checking various decoders

- 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 $\mathrm{a} = c$, $c \in \mathbb{N}$ is applied to $\rho'_\mathbb{P}$. The result is bottom if either domain is bottom.
- At the end of the $\mathbf{let}$- 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. *Consider* `f x = let g y = x^y in g (g '11')`. *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: eliminating y in $x = 4y$ removes all constraints and thereby the information that x is a multiple of four. This loss of information can be preempted by checking that λ-bound variables only mention type variables also occurring in the owning function. These type variables would indicate an ambiguous vector size which might be impossible to translate and, hence, the program needs to be rejected.

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. 10. 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.

6.2 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 rules 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, Gori et al. did not find any type correct program that required three iterations to type check [7] which coincides with our experience. In particular, only programs that trigger an infinite refinement such as `f x = f (x 1)` hit the iteration limit. Thus, it might be worth ignoring the problem of infinite descending chains by limiting the number of iterations and to concentrate on 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 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 $\alpha^{\mathcal{X}}_{\mathbb{M}}$ correctly is non-trival.

6.3 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.

Acknowledgments

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

References

[1] P. Cousot. Types as Abstract Interpretations. In *Principles of Programming Languages*, pages 316–331, Paris, France, Jan. 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, Jan. 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, Jan. 2002. Springer.

[8] F. Henglein. Type Inference with Polymorphic Recursion. *Transactions on Programming Languages and Systems*, 15:253–289, 1991.

[9] T. 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. C. Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming*, volume 306 of *LNCS*, pages 67–113, Trento, Italy, Dec. 1986. Springer.

[12] S. Lenglet and J. Wells. Expansion for Universal Quantifiers. In H. Seidl, editor, *European Symposium on Programming Languages and Systems*, volume 7211 of *LNCS*, pages 456–475, Tallinn, Estonia, Mar. 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, Apr. 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, Nov. 2006. Springer.

[16] A. Sepp, J. Kranz, and A. Simon. GDSL: A Generic Decoder Specication Language for Interpreting Machine Language. In *Tools for Automatic Program Analysis*, ENTCS, Deauville, France, Sept. 2012. Springer. URL `https://bitbucket.org/mb0/gdsl`.

[17] H. Siegel and A. Simon. Summarized Dimensions Revisited. In L. Mauborgne, editor, *Workshop on Numeric and Symbolic Abstract Domains*, ENTCS, Venice, Italy, Sept. 2011. Springer.

[18] A. Simon. Deriving a Complete Type Inference for Hindley-Milner and Vector Sizes using Expansion: Proof Appendix. Technical report, 2012.

[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.