Optimal Inference of Fields in Row-Polymorphic Records

Axel Simon

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

Abstract

Flexible records are a powerful concept in type systems that form the basis of, for instance, objects in dynamically typed languages. One caveat of using flexible records is that a program may try to access a record field that does not exist. We present a type inference algorithm that checks for these runtime errors. The novelty of our algorithm is that it satisfies a clear notion of completeness: The inferred types are optimal in the sense that type annotations cannot increase the set of typeable programs. Under certain assumptions, our algorithm guarantees the following stronger property: it rejects a program if and only if it contains a path from an empty record to a field access on which the field has not been added. We derive this optimal algorithm by abstracting a semantics to types. The derived inference rules use a novel combination of type terms and Boolean functions that retains the simplicity of unification-based type inference but adds the ability of Boolean functions to express implications, thereby addressing the challenge of combining implications and types. By following our derivation method, we show how various operations such as record concatenation and branching if a field exists lead to Boolean satisfiability problems of different complexity. Analogously, we show that more expressive type systems give rise to SMT problems. On the practical side, we present an implementation of the select and update operations and give practical evidence that these are sufficient in real-world applications.

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

General Terms Languages, Theory, Verification

Keywords type inference, abstract interpretation, expansion, completeness, row polymorphism, Boolean functions

1. Introduction

One of the productivity advantages of strongly-typed functional languages is their ability to infer types automatically, thereby allowing for shorter programs that can be written and changed more rapidly than programs written in explicitly typed languages.

Type inferences abstract program values with terms consisting of constructors (such as $\cdot\!\rightarrow\!\cdot$ for functions) and constants (such as `Int`). For the purpose of the inference and polymorphism, type terms may also contain type variables a. Type variables are a powerful mechanism to state that two or more (sub-)terms are equal. This power has been used in other contexts, such as Rémy's inference of record fields where they are called row-variables [19]. Here, a record $e : \{\mathrm{X} : t_x, \mathrm{Z} : t_z, \mathrm{a}\}$ contains the set of fields $\{\mathrm{X}, \mathrm{Z}\}$, where t_x and t_z denote the types of the fields. Additionally, the *row variable* a can be replaced by a field and another row variable, thereby extending the records. For instance, applying the substitution $\sigma = [\mathrm{a}/\mathrm{Y} : t_y, \mathrm{b}]$ results in the record type $\{\mathrm{X} : t_x, \mathrm{Y} : t_y, \mathrm{Z} : t_z, \mathrm{b}\}$. Observe that the possible fields are interpreted as set. Moreover, a row variable a occurring in two different record types indicates they contain the same set of fields.

However, the possible set of fields does not convey whether a field is present. Rémy addresses this question by enhancing record types to $\{\mathrm{X}.f_X : t_x, \mathrm{Z}.f_Z : t_z, \mathrm{a}.f_a\}$ where f_X, f_Z, f_a are flag variables. A flag is unified with a constant Pre if the corresponding field must be present and with Abs if the field is definitely absent. Consider the following expression that uses Haskell syntax augmented with constructs for an empty record $\{\} : \{\mathrm{a}.\mathrm{Abs}\}$, a field selector $\#\mathrm{N} : \{\mathrm{N}.\mathrm{Pre} : \mathrm{a}, \mathrm{b}.f_b\} \rightarrow \mathrm{a}$ to extract the field N from a record and a function $@\{\mathrm{N} = e\} : \{\mathrm{N}.f_N : \mathrm{a}, \mathrm{b}.f_b\} \rightarrow \{\mathrm{N}.f'_N : \mathrm{c}, \mathrm{b}.f_b\}$ that adds or replaces the field N in its argument with $e : \mathrm{c}$: (note that f'_N is not Pre so that it can still be unified with Abs):

```
let f s = if some_condition then
    let
        s' = @{ foo=42 } s
        v  = #foo s'
    in
       s'
  else s
in f {}
```

The function mimics a situation where a record `s` is used as a state to which a producer adds a field (here $\mathrm{FOO} : \texttt{Int}$) before a consumer extracts this value (here into $v : \texttt{Int}$). This computation is embedded in the **then**-branch of a conditional while the **else** branch returns the input state unchanged. From the update `@{foo=42}`, Rémy's inference deduces the type $s : \{\mathrm{FOO}.f_N : \mathrm{v}, \mathrm{a}.f_a\}$ and $s' : \{\mathrm{FOO}.f'_N : \texttt{Int}, \mathrm{a}.f_a\}$. Note that the type and the flag of field FOO are different. The selector function refines the type of the state to $s' : \{\mathrm{FOO}.\mathrm{Pre} : \texttt{Int}, \mathrm{a}.f_a\}$ which is also the type of the whole **then**-branch. This type is unified with the type of the **else**-branch which is that of `s`. Thus, the flag f_N is unified with Pre (and v with `Int`), yielding the overall type $f : \{\mathrm{FOO}.\mathrm{Pre} : \texttt{Int}, \mathrm{a}.f_a\} \rightarrow \{\mathrm{FOO}.\mathrm{Pre} : \texttt{Int}, \mathrm{a}.f_a\}$. Note how unification has propagated the fact that FOO must exist in `s'` to the input of the function. This has the effect that the call `f {}` is considered to be type incorrect since, after expanding the row variable r in `{}`: $\{\mathsf{r}.\mathsf{Abs}\}$ to $\{\text{FOO}.\mathsf{Abs} : \mathsf{b}, \mathsf{a}.\mathsf{Abs}\}$, the flag Abs of FOO clashes with the Pre flag of the argument of f, rejecting the program with "`f` expects a field FOO but is called with `{}`".

The presented program is a minimal example that illustrates how more advanced type inferences may yield surprising results once programs become sufficiently complex. It represents powerful programming styles that in Haskell or ML require the use of fixed records whose field carry undefined values when not in use. One scenario is a state monad whose state is a record. We use this scenario in a DSL [25] where different sub-computations are evaluated inside each branch of a **`case`**-statement. Each sub-computation may store some intermediate results in the record state, just like the **`then`**-branch does in the example. Another interesting scenario are compiler passes that compute and store information in the nodes of an abstract syntax tree. Here, checking that fields in flexible records exist ensures that an attribute of an AST node is computed before it is accessed. The example program above illustrates the situation when certain compiler passes, that communicate by annotating the AST, are run conditionally. On a broader scale, our inference can verify that no field in an object is accessed without being set first in featherweight Java [11] or pure subsets of other object-oriented languages like Python or JavaScript that are dynamically typed.

Our inference is able to check the program above by modeling the existence of fields using a Boolean formula and to interpret the flags $f_a, \ldots$ as propositional variables. Thus, the type of `f` consists of the *type term* $\{\text{FOO}.f_N : \texttt{Int}, \mathsf{a}.f_a\} \to \{\text{FOO}.f'_N : \texttt{Int}, \mathsf{a}.f'_a\}$ and the Boolean formula $f'_N \to f_N \wedge f'_a \to f_a$ indicating that the field FOO exists in the result (f_N is true) if it existed in the input (and analogous for any field that the row variable a is instantiated to). A program has a type error when unification of the type term fails or if the Boolean function becomes unsatisfiable. The latter can be checked using a SAT solver. In the example, the function `f` is applied to the empty record of type `{}`: $\{\text{FOO}.f''_N : \mathsf{b}, \mathsf{a}.f''_a\}$ with formula $\neg f''_N \wedge \neg f''_a$ so that the result has type $\{\text{FOO}.f'_N : \texttt{Int}, \mathsf{a}.f'_a\}$ with $\neg f'_N \wedge \neg f'_a$, indicating that `f {}` returns a record containing no accessible field. Specifically, applying the field selector $\#\text{FOO} : \{\text{FOO}.f_M : \mathsf{c}\}$ with formula f_M to the result generates the constraint $f_M \leftrightarrow f'_N$ which, together with f_M and $\neg f'_N$ is unsatisfiable. Thus, the expression `#foo (f {})` is type incorrect.

A novelty of our type system is that row- and type-variables all carry an additional flag that is used to describe the flow of values. For instance, the type `id` : $\mathsf{a}.f_a \to \mathsf{a}.f_b$ with formula $f_b \to f_a$ of the identity function states that values "flow" from the input of `id` to its output (or, rather, that a field is in the output if it was in the input). The application **`id`** `(f {})` instantiates the type term of `id` to $\{\text{FOO}.f_c : \texttt{Int}, \mathsf{b}.f_d\} \to \{\text{FOO}.f'_c : \texttt{Int}, \mathsf{b}.f'_d\}$. Additionally, the flow $f_b \to f_a$ is duplicated to $f'_c \to f_c \wedge f'_d \to f_d$ thus stating that the FOO-field exists in the output if it exists in the input. This duplication of flow information can be done implicitly by piggy-backing onto the operations on type terms: whenever a substitution $[\mathsf{a}/t]$ is applied, we extract all flags that a is annotated with (here: f_a, f_b) and count how many flags the type t contains (here: two). The flow between the flags of a is duplicated this many times, which gives the flow between the types t that a is replaced with. This separation of type terms and flows addresses the open "implication issue" [26] in monolithic solvers, as discussed next.

1.1 Monolithic Subtyping versus Unification + SAT Solving

Type inference of current functional languages like Haskell and OCaml translate each mutually recursive group of functions into a system of constraints [17] using sub-type and/or conditional constraints whose satisfiability is ascertained by a potent solver. In this setting, a type scheme $f : \forall\vec{\alpha}[D] \,.\, \tau$ for a function consists of the type term τ and a set of constraints D that the quantified variables $\vec{\alpha}$ have to satisfy. The constraint language can be arbitrarily powerful. Indeed, Pottier combines row-polymorphism with non-atomic subtyping where τ_i in $\tau_1 \leq \tau_2$ may be a constructor Pre taking the field type as argument [18]. This language allows him to type check the introductory example by inferring the type $f : \forall \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}[D] \,.\, \{\text{FOO} : \mathsf{a}, \mathsf{b}\} \to \{\text{FOO} : \mathsf{c}, \mathsf{d}\}$ with the constraints $D = \exists \mathsf{a}_2 \,.\, \mathsf{a} \leq \mathsf{c} \wedge \texttt{Int} \leq \mathsf{a}_2 \wedge \mathsf{Pre}\ \mathsf{a}_2 \leq \mathsf{c} \wedge \mathsf{b} \leq \mathsf{d}$. By using an extra element Any with $\mathsf{Pre}\ \tau \leq \mathsf{Any}$ and $\mathsf{Abs} \leq \mathsf{Any}$, the subtype constraints can express that `f {}` is typeable (with $\mathsf{a} = \mathsf{Abs}$ and $\mathsf{c} = \mathsf{Any}$), as is `f {foo="bad"}` (with $\mathsf{a} = \mathsf{Pre}\ \texttt{String}$ and $\mathsf{c} = \mathsf{Any}$). Our type inference rejects the latter call since the type of field FOO is not unifiable. So why consider a weaker inference?

In the context of Generalized Algebraic Data Types (GADTs), Pottier has raised an open "implication issue" [26] that relates to implication constraints $D_1 \Rightarrow D_2$. These implications express that the subtyping constraints D_2 hold if the current solution satisfies the subtyping constraints D_1. Implication constraints are used in Pottier's record inference for typing asymmetric record concatenation $r_3 = r_1 @ r_2$ where $\#\text{N}\ r_3$ evaluates to $\#\text{N}\ r_2$ if N exists in r_2 and $\#\text{N}\ r_3$ evaluates to $\#\text{N}\ r_1$ otherwise. Typing the concatenation of records requires yet another new field state, namely $\mathsf{Either}\ \tau$ with $\mathsf{Pre}\ \tau \leq \mathsf{Either}\ \tau \leq \mathsf{Any}$ and $\mathsf{Abs} \leq \mathsf{Either}\ \tau \leq \mathsf{Any}$. The intuition is that a field with type $\mathsf{Either}\ \tau$ may not exist, but if it does, it has type τ. Assuming the record types are $r_i : \{\text{N}.\mathsf{a}_i, \mathsf{b}_i\}$, two conditional constraints describe the behavior of the concatenation:

$$\begin{aligned} D_r = \exists \mathsf{d} \,.\, &(\mathsf{Pre}\ \mathsf{d} \leq \mathsf{a}_2 \wedge \mathsf{a}_2 \leq \mathsf{Either}\ \mathsf{d}) \Rightarrow (\mathsf{Pre}\ \mathsf{d} \leq \mathsf{a}_3) \\ &\wedge (\mathsf{Abs} \leq \mathsf{a}_2) \Rightarrow (\mathsf{a}_1 \leq \mathsf{a}_3) \end{aligned}$$

The two implications reflect the two reduction rules: the first states that the output contains the content of the field in record r_2 if it definitely (Pre) or possibly exists (Either); the second implication states that the output record contains the field from the first record if the field in r_2 may be absent, that is, Abs, Either or Any. The first implication is particularly troublesome for a solver as it is not monotone: the premise holds only as long as the field in record r_2 has a single type d (i.e. $\mathsf{a}_2 \in \{\mathsf{Pre}\ \mathsf{d}, \mathsf{Either}\ \mathsf{d}\}$) but once $\mathsf{a}_2 = \mathsf{Any}$ the conclusion $\mathsf{Pre}\ \mathsf{d} \leq \mathsf{a}_3$ does not hold since it would imply that r_3 definitely contains a field with any type (d is unrestricted once $\mathsf{a}_2 = \mathsf{Any}$). Indeed, due to limitations of his solver, Pottier proposes the following rule that simplifies the first premise [18]:

$$\begin{aligned} D'_r = \exists \mathsf{d} \,.\, &(\mathsf{a}_2 \leq \mathsf{Either}\ \mathsf{d}) \wedge \\ &(\mathsf{Pre}\ \mathsf{d} \leq \mathsf{a}_2) \Rightarrow (\mathsf{Pre}\ \mathsf{d} \leq \mathsf{a}_3) \wedge (\mathsf{Abs} \leq \mathsf{a}_2) \Rightarrow (\mathsf{a}_1 \leq \mathsf{a}_3) \end{aligned}$$

The requirement $\mathsf{a}_2 \leq \mathsf{Either}\ \mathsf{d}$ means that the program is rejected as soon as any of the fields in r_2 has the incompatible type Any. Thus, `{} @ (`**`if`**` c `**`then`**` {f=42} `**`else`**` {f="42"})` is a type error, although the program contains no field selector at all. Note that Pottier only proposes D'_r rules rather than the more precise D_r rules [18]. The "implication issue" [26] for GADTs that highlights the difficulty of finding solvable and complete implication constraints therefore also exists in record inference. Other reasons that make implications in subtype constraints undesirable related to existentially quantified variables such as d in D_r. These variables can usually not be eliminated until enough information is available to determine whether the implication holds. Hence, the size of the constraint system D of a function signature $f : \forall\vec{\alpha}[D] \,.\, \tau$ is not limited by the variables in τ but may encode the whole control flow within f. This unsolved set of constraints has to be reported and understood by the programmer in case of type errors (which has been used as an argument against implications [22, Sect. 5.1]) and duplicated when computing an instance of a function type. Our approach of associating each type variable and record field with a flag allows us to use a Boolean function to express implications. Since Boolean functions are closed under projection onto a subset of variables, the flow information generated while analyzing the body of a function f can be projected onto the flag variables in the type of f without losing precision. For inferences that only require Boolean functions, the obtained type for a function is thus concise. The separation of type term and flow information shows that the various inferences in the literature that improve over Rémy's work [19] give rise to different Boolean satisfiability problems. We found the following classes:

- In a language with empty records `{}`, record selector functions `#f` and updates of fields `@{f=e}`, we observe that all Boolean functions are two-variable Horn clauses. These can be solved in linear time. We observe that these operations can be expressed with atomic subtyping where τ_i in $\tau_1 \leq \tau_2$ may only range over constructors without argument (e.g. Pre and Abs).
- We address further operations on records [2]. We highlight that asymmetric record concatenation yields multi-variable Horn clauses which can be solved in linear time [7]. In contrast, symmetric record concatenation (where a field may not be present in both arguments) requires more complex Boolean formulae.
- We discuss type system refinements à la Pottier where fields only have to have a consistent type if they are accessed. We illustrate how this more generic type system can be expressed as an SMT problem with unification constraints as theory.

As far as we know, we are the first to categorize the aforementioned operations into classes of Boolean satisfiability problems. Correctness of this classification requires that the type inference has a notion of optimality, which we address as follows.

1.2 Assessing Completeness by Abstract Interpretation

An optimality of inferred types is often specified by showing that inference rules compute the principal type, that is, the most general type that is expressible in the given universe of types. We derive our type inference by computing an abstract semantics that is backward complete, a property that ensures that both, the type and the environment have the most generic types. This completeness is relative to a semantics that corresponds to the concrete semantics where the **if**-statement is abstracted to a non-deterministic choice. This optimal type inference rejects a program if and only if one of its sub-expressions cannot be given a type from the universe of types (i.e. type term and Boolean function). We illustrate when the orthogonal notion of forward completeness holds, allowing us to define potentially more efficient inference rules. For instance, we observe that function application requires no co- and contra-variant flow but merely equality constraints between flags. Finally, we illustrate the utility of our weaker type inference by presenting an implementation with which we have type checked over 22,000 lines of code.

In summary, our paper presents the following contributions:

- It presents a flow-sensitive inference for flexible records without resorting to subtyping constraints.
- We show how abstract interpretation can be used to guarantee optimality by construction.
- We categorize various language constructs and type universes for records according to their complexity.

The remainder of the paper is organized as follows: We commence by presenting the type inference algorithm for records. Section 3 revisits completeness definitions from the abstract interpretation literature that are used in Sect. 4 to sketch how the optimal inference rules were derived. Section 5 discusses extensions for other record operations and more general universes of types. Section 6 discusses our implementation before Sect. 7 presents related work.

$$
\begin{array}{rcll}
e & \in & \mathbb{E} & \text{programs} \\
\mathrm{N}, \mathrm{N}_1, \mathrm{N}_2, \ldots & \in & \mathbb{L} & \text{universe of field names} \\
e & ::= & x \mid \lambda x \,.\, e \mid e_1\, e_2 \mid \textbf{let } x = e_1 \textbf{ in } e_2 & \\
 & \mid & 0 \mid 1 \mid \ldots \mid \underline{\{\}} \mid @\underline{\{}\mathrm{N} = e\underline{\}} \mid \#\mathrm{N} &
\end{array}
$$

Figure 1. inductive definition of the program syntax

2. Type Inference for Records

After some preliminary definitions, we present a type inference for the Milner-Mycroft type system that is similar to the classic Damas-Milner algorithm but additionally admits polymorphic recursion. We then extend this inference to track flow information and records.

2.1 Preliminaries

Our type inference operates on $\mathcal{L}(\mathbb{E})$, an extended λ-calculus defined in Fig. 1. In terms of constructors, only integers $c \in \mathbb{Z}$ are handled. For the sake of examples, we will also use polymorphic lists, such as `[0,1,2]` : `[Int]` or `[[0]]` : `[[Int]]` but we will not present inference rules for them. The **let**-construct allows x to be **recursively** used in e_1. The last three constructs relate to records: $\mathbb{E}$ may be an empty record, an update/addition of a field and a selector. The latter two are functions that expect a record as argument.

A Hindley-Milner type inference algorithm for $\mathbb{E}$ without flows of record fields assigns, to each program variable, a type in $\mathbb{P}$:

$$
\begin{array}{rcll}
\mathsf{a}, \mathsf{b}, \ldots & \in & \mathbb{V} & \text{type variables} \\
t & \in & \mathbb{P} & \text{polymorphic type expressions} \\
t & ::= & \mathsf{a} \mid \mathsf{b} \mid \ldots \mid t_1 \rightarrow t_2 \mid \mathtt{Int} \mid [t] \mid & \\
 & & \underline{\{}\mathrm{N}_1 : t_1, \ldots \mathrm{N}_n : t_n, \mathsf{a}\underline{\}} &
\end{array}
$$

An environment of polymorphic types $\rho_{\mathbb{P}} \subseteq \mathbb{X} \rightarrow \mathbb{P}$ is a mapping from program variables $\mathbb{X}$ to a type $\mathbb{P}$. For convenience, we write them as $\mathbb{X}$-index vectors such as $[x \mapsto \mathsf{a}, y \mapsto \mathsf{a}]$. The set of variables that $\rho_{\mathbb{P}}$ binds is given by $\mathbf{dom}(\rho_{\mathbb{P}}) \subseteq \mathbb{X}$. Let $\rho_{\mathbb{P}}[x \mapsto t]$ denote the environment $\rho_{\mathbb{P}}$ in which the binding for $x \in \mathbb{X}$ has been added or updated to map to type t. A binding to x is removed by the projection operator $\exists_x$, such that $\exists_x(\rho_{\mathbb{P}}[x \mapsto t]) = \rho_{\mathbb{P}}$ if $x \notin \mathbf{dom}(\rho_{\mathbb{P}})$. Two environments $\rho^1_{\mathbb{P}}, \rho^2_{\mathbb{P}}$ are unified by applying a substitution $\sigma = [\mathsf{a}/t_1, \ldots]$, to one of the environments, written $\sigma(\rho^1_{\mathbb{P}})$, where σ is the most general unifier $\sigma = mgu(\rho^1_{\mathbb{P}}, \rho^2_{\mathbb{P}})$. If this unifier does not exist, the type inference fails.

2.2 Inference of Milner-Mycroft Types

Our inference is optimal in the sense that, for each expression, it computes the most general type expressible in the universe of types, relative to the types inferred for its sub-expression. Intuitively, we call an inference optimal if user-supplied type annotations cannot increase the set of programs that can be type-checked. For instance, the Damas-Milner algorithm is not optimal since it can type check a polymorphic recursive function when given a type annotation, but it cannot infer the type itself. Optimality can be formalized using the concept of backward-completeness [8, 21] in abstract interpretations. For instance, we have shown that the Milner-Mycroft type inference [16] is backward-complete and thus optimal [23].

Figure 2 re-formulates the optimal type inference algorithm [23] using inference rules for $\mathbb{E}$ without records. We use judgements of the form $\rho_{\mathbb{P}} \vdash e : t; \rho'_{\mathbb{P}}$ where $\rho_{\mathbb{P}}$ is the input type environment and $e \in \mathbb{E}$ the expression to be typed. The result is written $t; \rho'_{\mathbb{P}}$ where t is the type of e and $\rho'_{\mathbb{P}}$ is the modified type environment. We use $t; \rho_{\mathbb{P}}$ synonymous with $\rho_{\mathbb{P}}[\kappa \mapsto t]$, that is, we bind the type t to a special variable $\kappa \notin \mathbb{X}$. Hence, $t'; \rho'_{\mathbb{P}} = \sigma(t; \rho_{\mathbb{P}})$ applies the substitution σ to type t and the environment $\rho_{\mathbb{P}}$, yielding t' and $\rho'_{\mathbb{P}}$.

$$\frac{x \in \mathbb{X}^\lambda \qquad t_x = \rho_{\mathbb{P}}(x)}{\rho_{\mathbb{P}} \vdash x : t_x; \rho_{\mathbb{P}}}\ (\text{VAR}) \qquad \frac{\begin{array}{c}\rho_{\mathbb{P}} \vdash e_1 : t_1; \rho^1_{\mathbb{P}} \qquad \rho_{\mathbb{P}} \vdash e_2 : t_2; \rho^2_{\mathbb{P}} \qquad \mathrm{r}\ \text{fresh} \\ \sigma = mgu(t_1; \rho^1_{\mathbb{P}}, t_2 \to \mathrm{r}; \rho^2_{\mathbb{P}}) \qquad t_a \to t_r; \rho^{1\sigma}_{\mathbb{P}} = \sigma(t_1; \rho^1_{\mathbb{P}})\end{array}}{\rho_{\mathbb{P}} \vdash e_1\ e_2 : t_r; \rho^{1\sigma}_{\mathbb{P}}}\ (\text{APP})$$

$$\frac{\forall \mathrm{a}_1, \ldots \mathrm{a}_n \,.\, t_x = \rho_{\mathbb{P}}(x) \qquad \sigma = [\mathrm{a}_1/\mathrm{b}_1, \ldots \mathrm{a}_n/\mathrm{b}_n] \qquad \mathrm{b}_i\ \text{fresh}}{\rho_{\mathbb{P}} \vdash x : \sigma(t_x; \rho_{\mathbb{P}})}\ (\text{VAR-LET}) \qquad \frac{\rho_{\mathbb{P}}[x \mapsto \mathrm{a}] \vdash e : \rho^1_{\mathbb{P}}, t_2 \qquad \mathrm{a}\ \text{fresh}}{\rho_{\mathbb{P}} \vdash \lambda x \,.\, e : \rho^1_{\mathbb{P}}(x) \to t_2; \exists_x(\rho^1_{\mathbb{P}})}\ (\text{LAM})$$

$$\frac{\rho^0_{\mathbb{P}} = \rho_{\mathbb{P}} \qquad t_0 = \mathrm{a}, \mathrm{a}\ \text{fresh} \qquad \rho^i_{\mathbb{P}}[x \mapsto \forall(vars(t_i) \setminus vars(\rho^i_{\mathbb{P}})) \,.\, t_i] \vdash e : t_{i+1}; \rho^{i+1}_{\mathbb{P}} \qquad \exists k \,.\, t_k = t_{k+1} \qquad \rho^k_{\mathbb{P}} \vdash e' : t; \rho'_{\mathbb{P}}}{\rho_{\mathbb{P}} \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : t; \exists_x(\rho'_{\mathbb{P}})}\ (\text{LETREC})$$

$$\frac{\rho_{\mathbb{P}} \vdash e_s : \mathtt{Int}; \rho^c_{\mathbb{P}} \qquad \rho^c_{\mathbb{P}} \vdash e_t : t_t; \rho^t_{\mathbb{P}} \qquad \rho^c_{\mathbb{P}} \vdash e_e : t_e; \rho^e_{\mathbb{P}} \qquad \sigma = mgu(t_t; \rho^t_{\mathbb{P}}, t_e; \rho^e_{\mathbb{P}}) \qquad t^\sigma_t; \rho^{t\sigma}_{\mathbb{P}} = \sigma(t_t; \rho^t_{\mathbb{P}})}{\rho_{\mathbb{P}} \vdash \mathbf{if}\ e_s \mathbf{then}\ e_t\ \mathbf{else}\ e_e : t^\sigma_t; \rho^{t\sigma}_{\mathbb{P}}}\ (\text{COND})$$

Figure 2. type inference algorithm with polymorphic recursion

$$\frac{x \in \mathbb{X}^\lambda \qquad t_x = \Uparrow^{\mathbb{R}}_{\mathbb{P}}(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(\rho_{\mathbb{R}}(x)))}{\rho_{\mathbb{R}}|\beta \vdash x : t_x; \rho_{\mathbb{R}}|\beta \wedge \wr t_x \int \Rightarrow \wr \rho_{\mathbb{R}}(x) \int}\ (\text{VAR}) \qquad \frac{\begin{array}{c}\rho_{\mathbb{R}}|\beta \vdash e_1 : t_1; \rho^1_{\mathbb{R}}|\beta^1 \qquad \rho_{\mathbb{R}}|\beta \vdash e_2 : t_2; \rho^2_{\mathbb{R}}|\beta^2 \qquad \mathrm{r}.f_r\ \text{fresh} \\ \sigma = mgu(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_1; \rho^1_{\mathbb{R}}), \Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_2 \to \mathrm{r}.f_r; \rho^2_{\mathbb{R}})) \qquad t_f; \rho^{1\sigma}_{\mathbb{R}}|\beta^{1\sigma} = applyS(\sigma, t_1; \rho^1_{\mathbb{R}}|\beta^1) \\ t_a \to t_r; \rho^{2\sigma}_{\mathbb{R}}|\beta^{2\sigma} = applyS(\sigma, t_2 \to \mathrm{r}.f_r; \rho^2_{\mathbb{R}}|\beta^2)\end{array}}{\rho_{\mathbb{R}}|\beta \vdash e_1\ e_2 : t_r; \rho^{1\sigma}_{\mathbb{R}}|\beta^{1\sigma} \wedge \beta^{2\sigma} \wedge \wr \rho^{1\sigma}_{\mathbb{R}} \int^{\mathbb{X}} \Leftrightarrow \wr \rho^{2\sigma}_{\mathbb{R}} \int^{\mathbb{X}} \wedge \wr t_a \to t_r \int \Leftrightarrow \wr t_f \int}\ (\text{APP})$$

$$\frac{\forall \mathrm{a}_1, \ldots \mathrm{a}_n \,.\, t_x = \rho_{\mathbb{R}}(x) \qquad \sigma = [\mathrm{a}_1/\mathrm{b}_1, \ldots \mathrm{a}_n/\mathrm{b}_n] \qquad \mathrm{b}_i\ \text{fresh}}{\rho_{\mathbb{R}}|\beta \vdash x : applyS(\sigma, t_x; \rho_{\mathbb{R}}|\beta)}\ (\text{VAR-LET}) \qquad \frac{\rho_{\mathbb{R}}[x \mapsto \mathrm{a}.f_a]|\beta \vdash e : t_2; \rho^1_{\mathbb{R}}|\beta^1 \qquad \mathrm{a}.f_a\ \text{fresh}}{\rho_{\mathbb{R}}|\beta \vdash \lambda x \,.\, e : \rho^1_{\mathbb{R}}(x) \to t_2; \exists_x(\rho^1_{\mathbb{R}})|\beta^1}\ (\text{LAM})$$

$$\frac{\begin{array}{c}\rho^0_{\mathbb{R}}|\beta^0 = \rho_{\mathbb{R}}|\beta \qquad t_0 = \mathrm{a}, \mathrm{a}\ \text{fresh} \qquad \rho^i_{\mathbb{R}}[x \mapsto \forall(vars(t_i) \setminus vars(\rho^i_{\mathbb{P}})) \,.\, t_i]|\beta^i \vdash e : t_{i+1}; \rho^{i+1}_{\mathbb{R}}|\beta^{i+1} \\ \exists k \,.\, \Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_k) = \Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_{k+1}) \qquad \rho^k_{\mathbb{R}}|\beta^k \vdash e' : t; \rho'_{\mathbb{R}}|\beta'\end{array}}{\rho_{\mathbb{R}}|\beta \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : t; \exists_x(\rho'_{\mathbb{R}})|\beta'}\ (\text{LETREC})$$

$$\frac{\begin{array}{c}\rho_{\mathbb{R}}|\beta \vdash e_s : \mathtt{Int}; \rho^c_{\mathbb{R}}|\beta^c \qquad \rho^c_{\mathbb{R}}|\beta^c \vdash e_t : t_t; \rho^t_{\mathbb{R}}|\beta^t \qquad \rho^c_{\mathbb{R}}|\beta^c \vdash e_e : t_e; \rho^e_{\mathbb{R}}|\beta^e \qquad \sigma = mgu(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_t; \rho^t_{\mathbb{R}}), \Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_e; \rho^e_{\mathbb{R}})) \\ t^\sigma_t; \rho^{t\sigma}_{\mathbb{R}}|\beta^{t\sigma} = applyS(\sigma, t_t; \rho^t_{\mathbb{R}}|\beta^t) \qquad t^\sigma_e; \rho^{e\sigma}_{\mathbb{R}}|\beta^{e\sigma} = applyS(\sigma, t_e; \rho^e_{\mathbb{R}}|\beta^e) \qquad t_r = \Uparrow^{\mathbb{R}}_{\mathbb{P}}(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(t^\sigma_t))\end{array}}{\rho_{\mathbb{R}}|\beta \vdash \mathbf{if}\ e_s \mathbf{then}\ e_t\ \mathbf{else}\ e_e : t_r; \rho^{t\sigma}_{\mathbb{R}}|\beta^{t\sigma} \wedge \beta^{e\sigma} \wedge \wr \rho^{t\sigma}_{\mathbb{R}} \int^{\mathbb{X}} \Leftrightarrow \wr \rho^{e\sigma}_{\mathbb{R}} \int^{\mathbb{X}} \wedge \wr t_r \int \Rightarrow \wr t^\sigma_t \int \wedge \wr t_r \int \Rightarrow \wr t^\sigma_e \int}\ (\text{COND})$$

$$\frac{t_r = \underline{\{}\mathrm{a}.f_a\underline{\}} \qquad \mathrm{a}, f_a\ \text{fresh}}{\rho_{\mathbb{R}}|\beta \vdash \underline{\{\}} : t_r; \rho_{\mathbb{R}}|\beta \wedge \neg f_a}\ (\text{REC-EMPTY}) \qquad \frac{t = \underline{\{}\mathrm{N}.f_N : \mathrm{a}.f_a; \mathrm{b}.f_b\underline{\}} \qquad \mathrm{a}, \mathrm{b}, f_N, f_a, f'_a, f_b\ \text{fresh}}{\rho_{\mathbb{R}}|\beta \vdash \#\mathrm{N} : t \to \mathrm{a}.f'_a; \rho_{\mathbb{R}}|\beta \wedge f_N \wedge f_a \leftrightarrow f'_a}\ (\text{REC-SELECT})$$

$$\frac{\rho_{\mathbb{R}}|\beta \vdash e : t; \rho'_{\mathbb{R}}|\beta' \qquad \mathrm{a}, \mathrm{b}, f_N, f'_N, f_a, f_b, f'_b\ \text{fresh}}{\rho_{\mathbb{R}}|\beta \vdash @\underline{\{}\mathrm{N} = e\underline{\}} : \underline{\{}\mathrm{N}.f_N : \mathrm{a}.f_a; \mathrm{b}.f_b\underline{\}} \to \underline{\{}\mathrm{N}.f'_N : t; \mathrm{b}.f'_b\underline{\}}; \rho'_{\mathbb{R}}|\beta' \wedge f_b \leftrightarrow f'_b}\ (\text{REC-UPDATE})$$

Figure 3. type inference algorithm with flow of fields

Given this notation, the inference rules for λ- and **let**-bound variables (VAR), (VAR-LET), function application (APP), abstraction (LAM) and conditionals resemble the standard $\mathcal{W}$-algorithm [6]. The (LETREC)-rule resembles that of Mycroft [16] in that it computes the type of a recursive function using a fixpoint computation: The body of the **let** is checked first with $\rho^0_{\mathbb{P}}$ in which x is bound to the most general type $\forall \mathrm{a} \,.\, \mathrm{a}$ since $vars(t_i) \setminus vars(\rho^i_{\mathbb{P}}) = \{\mathrm{a}\}$. From the resulting type t_{i+1} a more precise type scheme is computed with which the body is examined again until a fixpoint is reached after k iterations. This fixpoint computation does not terminate for definitions like `f x = f 1 x` that result in infinite types $\mathtt{Int} \to \mathtt{Int} \to \ldots$. These simple forms of recursion can be easily detected [16]. Besides these pathological cases, the fixpoint computation seems to always terminate [12, 23] although the problem is, in general, undecidable [13].

Note that simpler inference rules à la Damas-Milner [6] that do not allow for polymorphic recursion may be used for the upcoming inference of record fields. However, such simpler rules are not backward-complete and the inferred type terms are thus not optimal (nor is the Boolean function describing the existence of fields).

2.3 Inference of Record Fields

This section presents the backward-complete inference algorithm for expressions $e \in \mathbb{E}$ including the flow of record fields.

We associate a Boolean flag with each record field that indicates if that field exist. We also associate a flag with each type variable indicating if record fields exist in types that the variable can be instantiated to. Thus, define record polymorphic types $\mathbb{P}^{\mathbb{R}}$ as follows:

$$\begin{array}{rcll} \mathrm{a}, \mathrm{b}, \ldots & \in & \mathbb{V} & \text{type variables} \\ f_a, f_b, \ldots & \in & \mathbb{N} & \text{Boolean flags / propositional variables} \\ t & \in & \mathbb{P}^{\mathbb{R}} & \text{record polymorphic types} \\ t & ::= & \multicolumn{2}{l}{\mathrm{a}.f_a \mid \mathrm{b}.f_b \mid \ldots \mid t_1 \to t_2 \mid \mathtt{Int} \mid \underline{[}t\underline{]} \mid} \\ & & \multicolumn{2}{l}{\underline{\{}\mathrm{N}_1.f_1 : t_1, \ldots \mathrm{N}_n.f_n : t_n, \mathrm{a}.f_a\underline{\}}} \end{array}$$

In order to re-use the operations on $\mathbb{P}$ for $\mathbb{P}^{\mathbb{R}}$, we define a projection $\Downarrow^{\mathbb{R}}_{\mathbb{P}}(\cdot) : \mathbb{P}^{\mathbb{R}} \to \mathbb{P}$ that strips Boolean variables from a type so that $\Downarrow^{\mathbb{R}}_{\mathbb{P}}(\underline{\{}\mathrm{N}.f_N : \underline{[}\mathrm{a}.f_a\underline{]}; \mathrm{b}.f_b\underline{\}}) = \underline{\{}\mathrm{N} : \underline{[}\mathrm{a}\underline{]}; \mathrm{b}\underline{\}}$. Symmetrically, $\Uparrow^{\mathbb{R}}_{\mathbb{P}}(\cdot) : \mathbb{P} \to \mathbb{P}^{\mathbb{R}}$ decorates plain polymorphic types with fresh propositional variables. For example, the type $\Uparrow^{\mathbb{R}}_{\mathbb{P}}(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(t))$

has all propositional variables of t renamed to fresh variables. Both functions $\Downarrow^{\mathbb{R}}_{\mathbb{P}}(\cdot)$ and $\Uparrow^{\mathbb{R}}_{\mathbb{P}}(\cdot)$ lift naturally to whole environments.

The type inference rules that include the tracking of flows are presented in Fig. 3. A judgement $\rho_{\mathbb{R}}|\beta \vdash e : t; \rho'_{\mathbb{R}}|\beta'$ specifies how an environment $\rho_{\mathbb{R}}$ and a Boolean function β are transformed to $\rho'_{\mathbb{R}}$ and β', respectively. In order to explain the computation of the flow information, we define some basic notation for Boolean functions.

A Boolean function β is represented in conjunctive normal form $c_1 \wedge \ldots \wedge c_n$ where each clause c_i is a disjunction of atoms. An atom is a propositional variable $f_a, f_b, \ldots$ or its negation $\neg f_a, \neg f_b, \ldots$. We write $f_a \rightarrow f_b$ as shorthand for $\neg f_a \vee f_b$ and $f_a \leftrightarrow f_b$ for $f_a \rightarrow f_b \wedge f_b \rightarrow f_a$. Let $\langle f_1, \ldots f_n \rangle$ denote a sequence of propositional variables. Variable sequences of equal lengths are related using the lifted implication $\langle f_1, \ldots f_n \rangle \Rightarrow \langle f'_1, \ldots f'_n \rangle \equiv f_1 \rightarrow f'_1 \wedge \ldots \wedge f_n \rightarrow f'_n$. Moreover, define $s_1 \Leftrightarrow s_2 \equiv (s_1 \Rightarrow s_2) \wedge (s_2 \Rightarrow s_1)$.

We now define a function $\lbag t \rbag$ that extracts a sequence of all Boolean variables occurring in a type term $t \in \mathbb{P}^{\mathbb{R}}$. The purpose of this function is to create (bi-)implications between the flags of two types $t_1, t_2 \in \mathbb{P}^{\mathbb{R}}$ using $\lbag t_1 \rbag \Leftrightarrow \lbag t_2 \rbag$ or $\lbag t_1 \rbag \Rightarrow \lbag t_2 \rbag$. Throughout the inference, it is guaranteed that the two sequences are of the same length by ensuring that both types contain the same type term, that is, $\Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_1) = \Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_2)$. The function is defined as follows:

DEFINITION 1. *Let $\cdot$ denote the concatenation of sequences. Define the function $\lbag \cdot \rbag : \mathbb{P}^{\mathbb{R}} \rightarrow (\mathbb{N} \cup \{\neg f \mid f \in \mathbb{N}\})^*$ as follows:*

$$\begin{aligned}
\lbag \mathrm{a}.f_a \rbag &= \langle f_a \rangle \\
\lbag t_1 \rightarrow t_2 \rbag &= \langle \neg f_1, \ldots \neg f_n \rangle \cdot \lbag t_2 \rbag \quad \textit{where } \langle f_1, \ldots f_n \rangle = \lbag t_1 \rbag \\
\lbag \mathtt{Int} \rbag &= \langle\rangle \\
\lbag [t] \rbag &= \lbag t \rbag \\
\lbag \underline{\{}\mathrm{N}_1.f_1 : t_1, \ldots \mathrm{N}_n.f_n : t_n, \mathrm{a}.f_a \underline{\}} \rbag &= \langle f_1, \ldots f_a \rangle \cdot \lbag t_1 \rbag \cdots \lbag t_n \rbag
\end{aligned}$$

Moreover, define $\lbag \rho_{\mathbb{R}} \rbag^{\mathbb{X}}$ to extract the variables of $\rho_{\mathbb{R}}$ in sequence.

Note that $\lbag \cdot \rbag$ returns atoms, that is, propositional variables and their negation. The reason for negating the arguments of a function is to account for the contra-variant position of a function argument. Co- and contra-variant flow will be illustrated in Sect. 2.4.

Figure 3 shows the inference rules extended with the tracking of flows. We illustrate the (VAR) and (LAM) rules with an example:

EXAMPLE 1. *We infer the type of $\lambda x \,.\, x$ in the empty environment $\emptyset|true$. The (LAM)-rule requires the type of $[x \mapsto \mathrm{a}.f_1]|true \vdash x$ which is computed by the (VAR)-rule. The latter decorates the type of x with a new flow $t_x = \Uparrow^{\mathbb{R}}_{\mathbb{P}}(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(\rho_{\mathbb{R}}(x)))$, say $t_x = \mathrm{a}.f_2$. The result $t_x; \rho_{\mathbb{P}}$ is returned together with the flow $true \wedge \lbag t_x \rbag \Rightarrow \lbag \rho_{\mathbb{R}}(x) \rbag$ where $\lbag t \rbag$ returns a sequence of all flag in t, here $\lbag t_x \rbag = \langle f_2 \rangle$ and $\lbag \rho_{\mathbb{R}}(x) \rbag = \langle f_1 \rangle$. Thus, the returned flow is $f_2 \rightarrow f_1$, indicating that a record field is in the output of the function if it was in the input. This flow becomes β_1 in the premise of the (LAM)-rule. The result is $\mathrm{a}.f_1 \rightarrow \mathrm{a}.f_2; \emptyset|f_2 \rightarrow f_1$, i.e., $\lambda x \,.\, x : \mathrm{a}.f_1 \rightarrow \mathrm{a}.f_2$ with flow $f_2 \rightarrow f_1$.*

The individual inference rules in Fig. 3 over $\mathbb{P}^{\mathbb{R}}$ differ from the simpler rules over $\mathbb{P}$ as follows: The (VAR)-rule decorates the type of x with fresh variables, yielding t_x. This type is returned with β being extended by implications from the flags of t_x to those in $\rho_{\mathbb{R}}(x)$, indicating that a field is in t_x if it is in $\rho_{\mathbb{R}}(x)$. The (VAR-LET)-rule strips the flags from the types in order to obtain the set of variables that need to be instantiated. Since σ replaces type variables a with types $t \in \mathbb{P}$ and not with types in $\mathbb{P}^{\mathbb{R}}$, a special function *applyS*$(\sigma, \rho_{\mathbb{R}}|\beta)$ is called that, for each occurrence of a, decorates t with fresh flag variables and adjusts β accordingly. This function will be detailed in the next section. The rules (LAM) and (LETREC) resemble their $\mathbb{P}$ counterparts. The more complex rules are those that unify types. We first consider the (APP)-rule. Here, the types of e_1 and e_2 are inferred and the results are unified appropriately after stripping off the flags using $\Downarrow^{\mathbb{R}}_{\mathbb{P}}(\cdot)$. In order to compute a new Boolean function from β^1 and β^2, the the substitution σ is applied twice, yielding $\beta^{1\sigma}$ and $\beta^{2\sigma}$. Since $\beta^{2\sigma}$ may contain flow information over variables that are only mentioned in $\rho^{2\sigma}_{\mathbb{R}}$ but not in the returned type environment $\rho^{1\sigma}_{\mathbb{R}}$, the flow over the flags in $\rho^{2\sigma}_{\mathbb{R}}$ is propagated to the flags in $\rho^{1\sigma}_{\mathbb{R}}$ using the sequence implication $\lbag \rho^{1\sigma}_{\mathbb{R}} \rbag^{\mathbb{X}} \Leftrightarrow \lbag \rho^{2\sigma}_{\mathbb{R}} \rbag^{\mathbb{X}}$.

Finally, the formula $\lbag t_a \rightarrow t_r \rbag \Leftrightarrow \lbag t_f \rbag$ equates the flow between the context and the function. Interestingly, we do not need a weaker sequence implication to obtain an optimal inference: This is surprising because subtyping-based inferences [26] generally assume that the formal and actual argument of a function are in a subtype relationship rather than being equal (analogous for the return value). However, our derivation shows that they can be equated. We illustrate the use of bi-implications using an example:

EXAMPLE 2. *We show that passing the identity function to itself returns the identity function. We assume that the type terms are already unified. Thus, suppose $t_o = (\mathrm{a}.f_1 \rightarrow \mathrm{a}.f_2) \rightarrow (\mathrm{a}.f_3 \rightarrow \mathrm{a}.f_4)$ with flow $\beta_o = f_1 \rightarrow f_3 \wedge f_4 \rightarrow f_2$ denotes the type of the outer identity function (which will be derived in Ex. 3) and let $t_i = \mathrm{a}.f_5 \rightarrow \mathrm{a}.f_6$ with flow $\beta_i = f_6 \rightarrow f_5$ denote that of the argument. We compute the flow of the return type $t_r = \mathrm{a}.f_7 \rightarrow \mathrm{a}.f_8$ using $\lbag t_i \rightarrow t_r \rbag \Leftrightarrow \lbag t_o \rbag$:*

$$\begin{array}{llcccccccc}
\textit{function} & t_o \equiv & (\mathrm{a}. & f_1 & \rightarrow \mathrm{a}. & f_2) & \rightarrow (\mathrm{a}. & f_3 & \rightarrow \mathrm{a}. & f_4) \\
 & \lbag t_o \rbag & & \neg\neg f_1 & & \neg f_2 & & \neg f_3 & & f_4 \\
\textit{flow} & \Updownarrow \; = & & \updownarrow & & \updownarrow & & \updownarrow & & \updownarrow \\
 & \lbag t_i \rightarrow t_r \rbag & & \neg\neg f_5 & & \neg f_6 & & \neg f_7 & & f_8 \\
\textit{context} & t_i \rightarrow t_r \equiv & (\mathrm{a}. & f_5 & \rightarrow \mathrm{a}. & f_6) & \rightarrow (\mathrm{a}. & f_7 & \rightarrow \mathrm{a}. & f_8)
\end{array}$$

The combined flow $\beta_o \wedge \beta_i \wedge \lbag t_i \rightarrow t_r \rbag \Leftrightarrow \lbag t_o \rbag$ implies $f_8 \rightarrow f_7$ since $f_8 \rightarrow f_4 \rightarrow f_2 \rightarrow f_6 \rightarrow f_5 \rightarrow f_1 \rightarrow f_3 \rightarrow f_7$.

The other rule that unifies two environments is the (COND)-rule. Here, the same principle is followed. The only difference is the generation of the flow of the result type. We take the type t^{σ}_t of the **then**-branch and decorate it with fresh flags, yielding t_r. The flow information $\lbag t_r \rbag \Rightarrow \lbag t^{\sigma}_t \rbag \wedge \lbag t_r \rbag \Rightarrow \lbag t^{\sigma}_t \rbag$ states that t_r contains a field if one of the branches contained the field. The remaining rules are those for record operations: (REC-EMPTY) returns an empty record and states that no field exists in any instance of the row variable a by adding $\neg f_a$ to the flow information. The (REC-SELECT)-rule returns the field of type a and asserts that it contains the same fields by adding $f_a \leftrightarrow f'_a$ to the flow information. Accessing the field N is only allowed if it exists, so f_N is asserted in the flow β. Finally, (REC-UPDATE) returns a function type where the input field N may be of any type while the output field may or may not be required (f_N is unrestricted) and is of the type of e. The type and flow of other fields is not affected, hence the flow $f_b \leftrightarrow f'_b$ is added. Note that the bi-implications $f_a \leftrightarrow f'_a$ and $f_b \leftrightarrow f'_b$ in the last two rules ensure that all flags in an environment are different. This ensures that $\lbag \cdot \rbag$ returns sequences without duplicates.

2.4 Applying Substitutions

This section details how $\rho'_{\mathbb{R}}|\beta' = applyS(\sigma, \rho_{\mathbb{R}}|\beta)$ is computed, that is, how a substitution is applied to types with flow information. The challenge is that a substitution $\sigma \in \mathbb{V} \rightarrow \mathbb{P}$ takes type variables to terms without flow information. We consider the function `cond` that has the $\mathbb{P}^{\mathbb{R}}$-type $t \equiv \mathrm{a}.f_1 \rightarrow \mathrm{a}.f_2 \rightarrow \mathrm{a}.f_3$ with the flow $\beta_t = f_3 \rightarrow f_1 \wedge f_3 \rightarrow f_2$. The function is defined as follows:

```
let cond = λx . λy . if 0 then x else y in cond
```

Now consider applying a substitution $\sigma = [\mathrm{a}/\underline{\{}\mathrm{FOO} : \mathrm{b}, \mathrm{c}\underline{\}}]$ to the type of `cond` so that $t' = \underline{\{}\mathrm{FOO} : \mathrm{b}, \mathrm{c}\underline{\}}$ replaces a three times. Since each occurrence of t' may have a different flow information, we decorate t' with fresh flags, once for each a it replaces. To this

```
$applyS([a/\tau], \rho_{\mathbb{P}}|\beta) =$
    let $\langle f_1, \ldots f_n \rangle = flags(a, \rho_{\mathbb{P}})$
        $\tau_1 = \Uparrow_{\mathbb{P}}^{\mathbb{R}}(\Downarrow_{\mathbb{P}}^{\mathbb{R}}(\tau)) \quad \ldots \quad \tau_n = \Uparrow_{\mathbb{P}}^{\mathbb{R}}(\Downarrow_{\mathbb{P}}^{\mathbb{R}}(\tau))$
        $dup(f_1' \cdot \vec{f_1}, \ldots f_n' \cdot \vec{f_n}, \beta) =$
            $dup(\vec{f_1}, \ldots \vec{f_n}, expand_{f_1 \ldots f_n, f_1' \ldots f_n'}(\beta))$
        $dup(\langle\rangle, \ldots \langle\rangle, \beta) = \exists_{f_1, \ldots f_n}(\beta)$
    in $\rho_{\mathbb{P}}[a.f_1/\tau_1, \ldots a.f_n/\tau_n]|dup(\wr\tau_1\int, \ldots, \wr\tau_n\int, \beta)$
```

Figure 4. Pseudo-code for applying a substitution.

end, we lift $t' \in \mathbb{P}$ to $\mathbb{P}^{\mathbb{R}}$ three times by computing $t_i = \Uparrow_{\mathbb{P}}^{\mathbb{R}}(t) = \underline{\{}\text{FOO}.f_f^i : b.f_b^i, c.f_c^i\underline{\}}$ for $i = 1, 2, 3$. The flow between the three flags of the type variable a must be replicated to f_f^i, f_b^i, f_c^i. Let us illustrate the flow $\beta_t = f_3 \rightarrow f_1 \wedge f_3 \rightarrow f_2$ at a as follows:

$$\begin{array}{ccc}
\text{occurrences of a} & & \text{type } t_i \text{ to substitute} \\
a.f_1 & & \underline{\{}\text{FOO}.f_f^1 : b.f_b^1, c.f_c^1\underline{\}} \\
\uparrow & & \uparrow \quad \uparrow \quad \uparrow \\
a.f_3 & & \underline{\{}\text{FOO}.f_f^3 : b.f_b^3, c.f_c^3\underline{\}} \\
\downarrow & & \downarrow \quad \downarrow \quad \downarrow \\
a.f_2 & & \underline{\{}\text{FOO}.f_f^2 : b.f_b^2, c.f_c^2\underline{\}}
\end{array}$$

The diagram shows the flow of a, namely β_t, as vertical arrows. Three instances of this flow β_t are needed, one instance for each flow variable in the t_i as indicated on the right. Thus, the result is $applyS(\sigma, [\texttt{cond} \mapsto t']|\beta_t) = [\texttt{cond} \mapsto t_1 \rightarrow t_2 \rightarrow t_3]|\beta_t'$ where $\beta_t' = f_f^3 \rightarrow f_f^1 \wedge f_f^3 \rightarrow f_f^2 \wedge f_b^3 \rightarrow f_b^1 \wedge f_b^3 \rightarrow f_b^2 \wedge f_c^3 \rightarrow f_c^1 \wedge f_c^3 \rightarrow f_c^2$.

In general, the flow information is replicated as follows:

DEFINITION 2 (Expansion). *We replicate the flow of $f_1, \ldots f_n$ to $f_1', \ldots f_n'$ in the Boolean function $\beta = c_1 \wedge \ldots \wedge c_k \wedge c_{k+1} \wedge \ldots \wedge c_l$. Without loss of generality, assume that $c_{k+1}, \ldots c_l$ do not contain any variables $f_1, \ldots f_n$. Then define the replicated flow to be $expand_{f_1 \ldots f_n, f_1' \ldots f_n'}(\beta) = c_1 \wedge \ldots \wedge c_k \wedge \sigma(c_1) \wedge \ldots \wedge \sigma(c_k) \wedge c_{k+1} \wedge \ldots \wedge c_l$ where $\sigma = [f_1/f_1', \ldots f_n/f_n']$ is a substitution.*

Thus, the replication of flows in the current example is given by

$$\begin{aligned}
&expand_{f_1 f_2 f_3, f_f^1 f_f^2 f_f^3}(\\
&\quad expand_{f_1 f_2 f_3, f_b^1 f_b^2 f_b^3}(\\
&\quad\quad expand_{f_1 f_2 f_3, f_c^1 f_c^2 f_c^3}(\beta_t))) = \beta_t \wedge \beta_t'
\end{aligned}$$

An algorithm for *applyS* is given in Fig. 4. Here, *flags* extracts the flags of the n occurrences of a in the input environment $\rho_{\mathbb{P}}$. The n freshly decorated types $\tau_1, \ldots \tau_n$ replace one of each occurrences of a in the result. The flow is duplicated by passing a vector $\vec{f_i}$ to *dup* for each τ_i. The *dup* function strips the first flag f_i' off each vector and expands the flow of a to them. Once the flow has been duplicated for all τ_i, the original flow is removed from β using a projection operation $\exists_{f_1, \ldots f_n}(\beta)$ on Boolean functions.

Observe that a flag $f \in \wr t_i \int$ to which the flag f_i of the type variable is expanded to can be a negated flag, namely if it occurs in an argument in contra-variant position. In these cases, *expand* must replace f_i with a negated flag, thereby replicating the contra-variant behavior. This is illustrated in the following example:

EXAMPLE 3. *Consider applying $\sigma = [a/b \rightarrow b]$ to the type of the identity function $t_{id} = a.f_i \rightarrow a.f_o$ with flow $\beta_{id} = f_o \rightarrow f_i$, that is, compute $applyS(\sigma, [id \mapsto t_{id}]|\beta_{id}) = [id \mapsto t_{id}']|\beta_{id}'$. For the two occurrences of a we use $t_i = \Uparrow_{\mathbb{P}}^{\mathbb{R}}(b \rightarrow b) = b.f_1 \rightarrow b.f_2$ and $t_o = \Uparrow_{\mathbb{P}}^{\mathbb{R}}(b \rightarrow b) = b.f_3 \rightarrow b.f_4$ so that $t_{id}' = t_i \rightarrow t_o$. Note that $\wr t_i \int = \langle \neg f_1, f_2 \rangle$ and $\wr t_o \int = \langle \neg f_3, f_4 \rangle$ so that the new flow is $\beta_{id}' = expand_{f_i f_o, \neg f_1 \neg f_3}(expand_{f_i f_o, f_2 f_4}(\beta_{id})) = \beta_{id} \wedge f_4 \rightarrow f_2 \wedge f_1 \rightarrow f_3$. Note that this is the type used in Ex. 2.*

We now address how to formally derive the presented inference.

a)

$$\begin{tikzcd}
u \in \mathscr{P}(\mathbb{U}) \arrow[r, shift left, leftarrow, "\gamma"] \arrow[r, shift right, "\alpha"'] \arrow[d, "f"'] & p \in \mathbb{P} \arrow[d, "f^\sharp"] \\
u' \in \mathscr{P}(\mathbb{U}) \arrow[r, shift left, leftarrow, "\gamma"] \arrow[r, shift right, "\alpha"'] & p' \in \mathbb{P}
\end{tikzcd}$$

b)

$$\begin{tikzcd}
\mathscr{P}(\mathbb{X} \rightarrow \mathbb{U}) \arrow[r, shift left, leftarrow, "\gamma"] \arrow[r, shift right, "\alpha"'] \arrow[d, "f"] & (\mathbb{X} \rightarrow \mathbb{P})^\perp \arrow[d, "f^\sharp"] \\
\mathscr{P}(\mathbb{X} \cup \{\kappa\} \rightarrow \mathbb{U}) \arrow[r, shift left, leftarrow, "\gamma"] \arrow[r, shift right, "\alpha"'] & (\mathbb{X} \cup \{\kappa\} \rightarrow \mathbb{P})^\perp
\end{tikzcd}$$

Figure 5. Abstract interpretation a) in general and b) for types.

3. Abstract Interpretation for Type Inference

This section revisits some of the more relevant concepts of abstract interpretation with an emphasis on completeness properties.

We illustrate the underlying idea of abstract interpretation [4] using Fig. 5a). The diagram shows how some abstract property $p \in \mathbb{P}$ represents several concrete properties $u \subseteq \mathbb{U}$ using an abstraction function $\alpha : \mathscr{P}(\mathbb{U}) \rightarrow \mathbb{P}$ and/or a concretization function $\gamma : \mathbb{P} \rightarrow \mathscr{P}(\mathbb{U})$. For an imperative program, an abstract interpretation usually tracks one property $p \in \mathscr{P}(\mathbb{U})$ per program point. Suppose that p represents the values of a variable. Since p is a summary, the concrete transformer f operates on a collection of values and the various concrete transformers are therefore called the collecting semantics. Each f is approximated by an abstract transformer $f^\sharp : \mathbb{P} \rightarrow \mathbb{P}$. As an example, suppose that the concrete values $\{5, 7, 8\} \subseteq \mathscr{P}(\mathbb{U})$ denote the possible content of an integer variable x and that f doubles x. Now assume that $\mathbb{P}$ is the set of intervals, so that the value of x is approximated by a single interval. The abstract transformer that approximates f is thus $f^\sharp([l, u]) = [2l, 2u]$. With these assumptions, the concrete result is $f(\{5, 7, 8\}) = \{10, 14, 16\}$ whereas the abstract result is $f([5, 8]) = [10, 16]$ since $\alpha(\{5, 7, 8\}) = [5, 8]$. Note that the abstract doubling function $f^\sharp$ obeys $\alpha \circ f = f^\sharp \circ \alpha$ where $(f \circ g) = \lambda x \,.\, f(g(x))$. We say that $f^\sharp$ is backward-complete [8, 21]. Intuitively, $f^\sharp$ computes the best abstract value, that is, the smallest interval. On the contrary, $f \circ \gamma = \gamma \circ f^\sharp$, which is called forward-completeness [21], does not hold for $f^\sharp$. For example, $f(\gamma([5, 8])) = \{10, 12, 14, 16\}$ since $\gamma([5, 8]) = \{5, 6, 7, 8\}$ but $\gamma(f^\sharp([5, 8])) = \{10, 11, 12, 13, 14, 15, 16\}$. Intuitively, $f^\sharp$ is not exact; the abstraction to intervals forces $f^\sharp$ to approximate the result of f. It seems that in an ideal analysis, all transfer functions are forward- and backward-complete. However, suppose we had an analysis where this is the case and assume, without loss of generality, that $\mathbb{P}$ is factored into equivalence classes so that $\alpha \circ \gamma = id$. Then $\alpha \circ f = id \circ f^\sharp \circ \alpha = \alpha \circ \gamma \circ f^\sharp \circ \alpha$. With $\gamma \circ f^\sharp = f \circ \gamma$, the equation changes to $\alpha \circ f = \alpha \circ f \circ \gamma \circ \alpha$. Hence, $\gamma \circ \alpha$ must be the identity which shows that the analysis does not approximate. Hence, the abstract semantics is undecidable if the concrete semantics is undecidable. An effective analysis must therefore contain at least one abstract transfer function $f^\sharp$ that is not forward- and backward-complete at the same time. We derive our type inference from a concrete semantics in which the **`if`**-statement is abstracted to a non-deterministic choice. We thus seek abstract transformers that are complete with respect to this abstracted semantics.

We now illustrate how these observations relate to type inference. Let $\rho_{\mathbb{P}} \in (\mathbb{X} \rightarrow \mathbb{P})^\perp = (\mathbb{X} \rightarrow \mathbb{P}) \cup \{\perp\}$ be an environment of polytypes where $\perp$ denotes a type error. Given an environment $\rho_{\mathbb{P}} \neq \perp$, the notation $\rho_{\mathbb{P}} \vdash e : \tau$ is used to indicate that the expression e has type τ. Consider a typing rule for function application $e_1\, e_2$ which can be written as follows:

$$\frac{\rho_{\mathbb{P}} \vdash e_1 : t_1 \qquad \rho_{\mathbb{P}} \vdash e_2 : t_2 \qquad t_2 \rightarrow t_r \doteq t_1 \qquad t_r \text{ fresh}}{\rho_{\mathbb{P}} \vdash e_1\, e_2 : t_r} \;(\text{APP})$$

In order to use this typing rule for type inference, the constraint $t_2 \rightarrow t_r \doteq t_1$ needs to be solved by unification, leading to a substitution that is applied to $\rho_{\mathbb{P}}$ and the type t_r. This leads to the inference rule (APP) in Fig. 2 whose conclusion has the form $\rho_{\mathbb{P}} \vdash e : \tau | \rho'_{\mathbb{P}}$. Here, the result is the inferred type τ and a modified environment $\rho'_{\mathbb{P}}$. Rather than handling the type τ explicitly, we simply bind it to the special symbol κ in the returned environment $\rho'_{\mathbb{P}}$. Now each inference rule can be written as an abstract transfer function $[\![e_1\, e_2]\!]^\sharp : (\mathbb{X} \rightarrow \mathbb{P})^\perp \rightarrow (\mathbb{X} \cup \{\kappa\} \rightarrow \mathbb{P})^\perp$, that is, a function from an environment $\rho_{\mathbb{P}}$ to a new environment where the type of the expression is bound to κ. The abstraction relation for these rules is shown in Fig. 5b). Here, the collecting semantics $f : \mathscr{P}(\mathbb{X} \rightarrow \mathbb{U}) \rightarrow \mathscr{P}(\mathbb{X} \cup \{\kappa\} \rightarrow \mathbb{U})$ of each language construct $e \in \mathbb{E}$, that binds the result of evaluating e to κ, is approximated by a type inference rule $f^\sharp : (\mathbb{X} \rightarrow \mathbb{P})^\perp \rightarrow (\mathbb{X} \cup \{\kappa\} \rightarrow \mathbb{P})^\perp$.

Note that an abstract transfer function such as $[\![e_1\, e_2]\!]^\sharp$ computes the type of the expression by recursively calling the abstract transfer functions for its sub-expressions, here e_1 and e_2. Each transfer function has a straight-forward translation to a type inference rule which we illustrate with the following polytype semantics (instead of $[\![e_1\, e_2]\!]^\sharp$ we use $\mathcal{H}[\![e_1\ e_2]\!]$ for "Herbrand semantics"):

$$\mathcal{H}[\![e_1\ e_2]\!] \quad = \quad (\mathbb{X} \rightarrow \mathbb{P})^\perp \rightarrow (\mathbb{X} \cup \{\kappa\} \rightarrow \mathbb{P})^\perp$$

$$\mathcal{H}[\![e_1\ e_2]\!]\rho_{\mathbb{P}} \quad = \quad \begin{cases} \rho_{\mathbb{P}}^{1\sigma}[\kappa \mapsto t_r] & \text{if } \perp_{\mathbb{P}} \neq \rho^i_{\mathbb{P}} = \mathcal{H}[\![e_i]\!]\rho_{\mathbb{P}}, i = 1, 2 \\ & \text{and } t_2 = \rho^2_{\mathbb{P}}(\kappa) \\ & \text{and } \rho^{2'}_{\mathbb{P}} = \rho^2_{\mathbb{P}}[\kappa \mapsto \underline{t_2 \rightarrow (\mathrm{r})_{\equiv_{\mathbb{P}}}}] \\ & \text{and } \perp_{\mathbb{P}} \neq \rho^{1\sigma}_{\mathbb{P}} = \overline{\mathbf{gci}}(\rho^1_{\mathbb{P}}, \rho^{2'}_{\mathbb{P}}) \\ & \text{and } t_a \rightarrow t_r = \rho^{1\sigma}_{\mathbb{P}}(\kappa) \\ \perp_{\mathbb{P}} & \text{otherwise} \end{cases}$$

We ignoring type errors $\perp_{\mathbb{P}}$. The computations in the if-guard of $\mathcal{H}[\![e_1\ e_2]\!]$ are rephrased to the premise of the $\underline{(\text{APP})}$-rule in Fig. 2. To this end, note that the next section defines $\overline{\mathbf{gci}}$ in terms of mgu.

The main challenge is to find abstract transformers that are backward complete. Schmidt observes [21] that backward completeness, i.e. $\alpha \circ f = f^\sharp \circ \alpha$, is equivalent to the following two properties:

1. $\forall u_1, u_2 \subseteq \mathbb{U} \,.\, \alpha(u_1) = \alpha(u_2) \implies \alpha(f(u_1)) = \alpha(f(u_2))$
2. $f^\sharp = \alpha \circ f \circ \gamma$

With respect to type inference, rule (1.) states that α does not define a dependent type system: the type $\alpha(f(u_1))$ that results when executing f does not change if the value u_1 is replaced by a different value u_2 that has the same type as u_1. Once (1.) is shown, rule (2.) is a recipe for computing the inference rules. We apply this idea in the next section to derive backward complete transformers for programs without records (called $\mathcal{H}[\![\cdot]\!]$) and with records (called $\mathcal{F}[\![\cdot]\!]$), yielding the inference rules in Fig. 2 and Fig. 3, respectively.

4. Deriving the Type Inference

This section describes the derivation of the inference rules. The derivation first abstracts a standard, denotational semantics $\mathcal{S}[\![\cdot]\!]$ to a collecting semantics $\mathcal{C}_1[\![\cdot]\!]$ and from there to a semantics over sets of monotypes $\mathcal{T}[\![\cdot]\!]$. The latter semantics is then abstracted twice: once for polytypes without records $\mathcal{H}[\![\cdot]\!]$, and once for polytypes with flow of record fields $\mathcal{F}[\![\cdot]\!]$. For brevity, we only sketch the derivation up to the monotype semantics.

4.1 Abstracting Values to Types

Let $\mathbb{U}$ denote a universe of program values containing integers, functions, records and the special error value Ω that represents a run-time type error. Using this universe, we define a denotational semantics $\mathcal{S}[\![e]\!] : (\mathbb{X} \rightarrow \mathbb{U}_\perp) \rightarrow \mathbb{U}_\perp$ for each language construct $e \in \mathcal{L}(\mathbb{E})$ in Fig. 1. This denotational semantics is a straightforward extension of Milner's semantics [15]. We then abstract $\mathcal{S}[\![\cdot]\!]$ by replacing conditionals with a non-deterministic choice [23], yielding a collecting semantics $\mathcal{C}_1[\![\cdot]\!] : \mathscr{P}((\mathbb{X} \rightarrow \mathbb{U}_\perp) \rightarrow \mathbb{U}_\perp)$.

While it is possible to abstract the collecting semantics directly to polytypes $\mathbb{P}$ [23], we first abstract it to sets of monotypes in order to have a common basis for a further abstraction to the two sets of typing rules in Fig. 2 and Fig. 3. Monotypes are defined as follows:

$$\begin{array}{lcll} t & \in & \mathbb{M} & \text{monomorphic type expressions} \\ t & ::= & t_1 \rightarrow t_2 \mid \texttt{Int} \mid [t] \mid \underline{\{}\mathrm{N}_1 : t_1, \ldots \mathrm{N}_n : t_n\underline{\}} & \end{array}$$

We re-use an abstraction $\alpha^{\mathcal{X}}_{\mathbb{M}} : \mathscr{P}(\mathbb{X} \rightarrow \mathbb{U}_\perp) \rightarrow \mathscr{P}(\mathbb{X} \rightarrow \mathbb{M})$ and a concretization $\gamma^{\mathcal{X}}_{\mathbb{M}} : \mathscr{P}(\mathbb{X} \rightarrow \mathbb{M}) \rightarrow \mathscr{P}(\mathbb{X} \rightarrow \mathbb{U}_\perp)$ from value- to sets of monotype environments as defined in [23]. The monotype semantics $\mathcal{T}[\![e]\!] : \mathscr{P}(\mathbb{X} \rightarrow \mathbb{M}) \rightarrow \mathscr{P}(\mathbb{X} \cup \{\kappa\} \rightarrow \mathbb{M})$ follows the convention of binding the result type to κ which is essential to retain the relation between environment and returned type. For instance, the monotype semantics of x in the context of **let** $id = \lambda x \,.\, x$ **in** $\ldots$ is computed as follows:

$$\mathcal{T}[\![x]\!] \left(\begin{array}{l} \{[id \mapsto \texttt{Int}\rightarrow\texttt{Int}, x \mapsto \texttt{Int}], \\ \ [id \mapsto [\texttt{Int}]\rightarrow[\texttt{Int}], x \mapsto [\texttt{Int}], \ldots\} \end{array} \right) = $$
$$\begin{array}{l} \{[id \mapsto \texttt{Int}\rightarrow\texttt{Int}, x \mapsto \texttt{Int}], \kappa \mapsto \texttt{Int}], \\ \ [id \mapsto [\texttt{Int}]\rightarrow[\texttt{Int}], x \mapsto [\texttt{Int}], \kappa \mapsto [\texttt{Int}]], \ldots\} \end{array}$$

In order to abstract the collecting semantics, we also need to bind its result to κ. Thus, instead of $\mathcal{C}_1[\![\cdot]\!]$, we abstract the function $\mathcal{C}[\![e]\!] = \lambda \bar{\rho} \,.\, \{\rho[\kappa \mapsto S\rho] \mid S \in \mathcal{C}_1[\![e]\!] \wedge \rho \in \bar{\rho}\}$ that has the type $\mathcal{C}[\![e]\!] : \mathscr{P}(\mathbb{X} \rightarrow \mathbb{U}_\perp) \rightarrow \mathscr{P}(\mathbb{X} \cup \{\kappa\} \rightarrow \mathbb{U}_\perp)$. Before detailing the resulting rules in Fig. 6, we observe that they satisfy $f^\sharp = \alpha \circ f \circ \gamma$:

LEMMA 1. *The rules shown in Fig. 6 satisfy* $\mathcal{T}[\![e]\!] = \alpha^{\mathcal{X}}_{\mathbb{M}} \circ \mathcal{C}[\![e]\!] \circ \gamma^{\mathcal{X}}_{\mathbb{M}}$*, for all* $e \in \mathbb{E}$.

Proof. By computing $\mathcal{T}[\![e]\!]$ for all $e \in \mathbb{E}$. In [24]. ■

By additionally proving that $\forall u_1, u_2 \subseteq \mathbb{U} \,.\, \alpha(u_1) = \alpha(u_2) \implies \alpha(f(u_1)) = \alpha(f(u_2))$, backward-completeness of $\mathcal{T}[\![\cdot]\!]$ follows:

LEMMA 2. *For all* $\bar{\rho}_1, \bar{\rho}_2 \subseteq \mathbb{X} \rightarrow \mathbb{U}_\perp$ *with* $\alpha^{\mathcal{X}}_{\mathbb{M}}(\bar{\rho}_1) = \alpha^{\mathcal{X}}_{\mathbb{M}}(\bar{\rho}_2)$ *and all* $e \in \mathbb{E}$ *it follows that* $\alpha^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![e]\!]\bar{\rho}_1) = \alpha^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![e]\!]\bar{\rho}_2)$.

Proof. By structural induction over $e \in \mathbb{E}$. In [24]. ■

The next two sections address the translation of the monotype semantics to polytypes and to polytypes with flows.

4.2 Abstraction to Polytypes

This section shows that the inference rules in Fig. 2 are backward-complete abstractions of the monotypes semantics in Fig. 6 by deriving them. It also discusses which rules are forward-complete. Let $\mathbf{ground} : \mathbb{P} \rightarrow \mathscr{P}(\mathbb{M})$ replace all variables in a polymorphic type such that the result is a well-formed monotype, that is, row-variables are replaced by a (possibly empty) set of fields and other type variables are replaced by monotypes. For $t_1, t_2 \in \mathbb{P}$ define $t_1 \sqsubseteq_{\mathbb{P}} t_2$ 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. Let $(t)_{\equiv_{\mathbb{P}}}$ denotes the $\equiv_{\mathbb{P}}$-equivalence class of $t \in \mathbb{P}$ that shares no variables with other equivalence class.

Let $t = \mathbf{gci}(t_1, t_2)$ denote the greatest common instance which is computed using unification [14] as follows: 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 $\sigma = mgu(t'_1, t'_2)$ exists and $t = \perp_{\mathbb{P}}$ otherwise. For example, $\mathbf{gci}(([\mathrm{a}] \rightarrow [\texttt{Int}])_{\equiv_{\mathbb{P}}}, ([\texttt{Int}] \rightarrow \mathrm{a})_{\equiv_{\mathbb{P}}}) = [\texttt{Int}] \rightarrow [\texttt{Int}] = \sigma([\mathrm{a}] \rightarrow [\texttt{Int}])$ with $\sigma = mgu([\mathrm{a}] \rightarrow [\texttt{Int}], [\texttt{Int}] \rightarrow \mathrm{b}) = \{\mathrm{a}/\texttt{Int}, \mathrm{b}/[\texttt{Int}]\}$. The join $\mathbf{lca}$ is not used but would compute

$$
\begin{array}{l}
\mathcal{T}[\![\cdot]\!] \quad : \ \mathbb{E} \to \mathscr{P}(\mathbb{X} \to \mathbb{M}) \to \mathscr{P}(\mathbb{X} \cup \{\kappa\} \to \mathbb{M}) \\
\mathcal{T}[\![x]\!]\,\overline{\rho}_{\mathbb{M}} = \begin{cases} \{\rho_{\mathbb{M}}[\kappa \mapsto t] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \wedge t \in \mathbb{M}\} \cap \{\rho_{\mathbb{M}}[x \mapsto t_0, y_1 \mapsto t_1, \ldots y_n \mapsto t_n, \kappa \mapsto \rho_{\mathbb{M}}(x)] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}} \wedge t_i \in \mathbb{M}\} & \text{if } x \notin \mathbb{X}^{\lambda} \\ \{\rho_{\mathbb{M}}[\kappa \mapsto \rho_{\mathbb{M}}(x)] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}\} & \text{if } x \in \mathbb{X}^{\lambda} \end{cases} \\
\mathcal{T}[\![\lambda x \,.\, e]\!]\,\overline{\rho}_{\mathbb{M}} = \{\exists_x(\rho'_{\mathbb{M}}[\kappa \mapsto t_1 {\to} t_2]) \mid t_1 \in \rho'_{\mathbb{M}}(x) \wedge t_2 \in \rho'_{\mathbb{M}}(\kappa) \wedge \rho'_{\mathbb{M}} \in \overline{\rho}'_{\mathbb{M}}\} \text{ with } \overline{\rho}'_{\mathbb{M}} = \mathcal{T}[\![e]\!]\{\rho_{\mathbb{M}}[x \mapsto t_1] \mid t_1 \in \mathbb{M} \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}\} \\
\mathcal{T}[\![e_1\ e_2]\!]\,\overline{\rho}_{\mathbb{M}} = \{\rho_{\mathbb{M}}[\kappa \mapsto t_r] \mid t_a {\to} t_r = \rho_{\mathbb{M}}(\kappa) \wedge \rho_{\mathbb{M}} \in (\mathcal{T}[\![e_1]\!]\overline{\rho}_{\mathbb{M}}) \cap \{\rho_{\mathbb{M}}[\kappa \mapsto t_a {\to} t_r] \mid t_r \in \mathbb{M} \wedge t_a \in \rho_{\mathbb{M}}(\kappa) \wedge \rho_{\mathbb{M}} \in \mathcal{T}[\![e_2]\!]\overline{\rho}_{\mathbb{M}}\}\} \\
\mathcal{T}[\![\textbf{let } x = e \textbf{ in } e']\!]\,\overline{\rho}_{\mathbb{M}} = \exists_x(\mathcal{T}[\![e']\!]\overline{\rho}'_{\mathbb{M}}) \text{ where } \overline{\rho}'_{\mathbb{M}} = \mathrm{gfp}^{\subseteq}_{\{\rho_{\mathbb{M}}[x \mapsto t] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}, t \in \mathbb{M}\}} \lambda \overline{\rho}_{\mathbb{M}} \,.\, \{\rho_{\mathbb{M}}[x \mapsto \rho_{\mathbb{M}}(\kappa)] \mid \rho_{\mathbb{M}} \in \mathcal{T}[\![e]\!]\overline{\rho}_{\mathbb{M}}\} \\
\mathcal{T}[\![c]\!]\,\overline{\rho}_{\mathbb{M}} \quad = \{\rho_{\mathbb{M}}[\kappa \to \texttt{Int}] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}\} \\
\mathcal{T}[\![\underline{\{\}}]\!]\,\overline{\rho}_{\mathbb{M}} \quad = \{\rho_{\mathbb{M}}[\kappa \to \underline{\{\}}] \mid \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}\} \\
\mathcal{T}[\![@\underline{\{}\mathrm{N} = e\underline{\}}]\!]\,\overline{\rho}_{\mathbb{M}} = \{\rho_{\mathbb{M}}[\kappa \mapsto t_r {\to} t_u] \mid t_r = \underline{\{}\mathrm{N}_1 : t_1; \ldots \mathrm{N}_n : t_n\underline{\}} \wedge t_i \in \mathbb{M} \wedge t_u = t_r[\mathrm{N} \mapsto \rho_{\mathbb{M}}(\kappa)] \wedge \rho_{\mathbb{M}} \in \mathcal{T}[\![e]\!]\,\overline{\rho}_{\mathbb{M}}\} \\
\mathcal{T}[\![\#\overline{\mathrm{N}}]\!]\,\overline{\rho}_{\mathbb{M}} \quad = \{\rho_{\mathbb{M}}[\kappa \mapsto t_r {\to} t_i] \mid t_r = \underline{\{}\mathrm{N}_1 : t_1; \ldots \mathrm{N}_n : t_n\underline{\}} \wedge \exists \overline{i} \,.\, \mathrm{N}_i = \mathrm{N} \wedge \rho_{\mathbb{M}} \in \overline{\rho}_{\mathbb{M}}\} \\
\mathcal{T}[\![\textbf{if } e_s \textbf{ then } e_t \textbf{ else } e_e]\!]\,\overline{\rho}_{\mathbb{M}} = \{\rho_{\mathbb{M}} \mid \texttt{Int} = \rho'_{\mathbb{M}}(\kappa) \wedge \rho'_{\mathbb{M}} \in \mathcal{T}[\![e_s]\!]\overline{\rho}_{\mathbb{M}} \wedge \rho_{\mathbb{M}} \in \mathcal{T}[\![e_t]\!]\overline{\rho}_{\mathbb{M}}\} \\
\qquad\qquad\qquad\qquad\qquad \cap\ \{\rho_{\mathbb{M}} \mid \texttt{Int} = \rho'_{\mathbb{M}}(\kappa) \wedge \rho'_{\mathbb{M}} \in \mathcal{T}[\![e_s]\!]\overline{\rho}_{\mathbb{M}} \wedge \rho_{\mathbb{M}} \in \mathcal{T}[\![e_e]\!]\overline{\rho}_{\mathbb{M}}\}
\end{array}
$$

Figure 6. abstract transfer functions on set of monotype environments

the least common anti-instance using anti-unification. We lift **gci**, **lca**, and $\sqsubseteq_{\mathbb{P}}$ to environments of polytypes ($\mathbb{X} \to \mathbb{P}$) as $\overline{\textbf{gci}}$, $\overline{\textbf{lca}}$, and $\overline{\sqsubseteq_{\mathbb{P}}}$. We identify vectors that are equal modulus renaming of variables, written $(\mathbb{X} \to \mathbb{P})_{\equiv_{\mathbb{P}}}$ and add a bottom element $\perp_{\mathbb{P}}$. The resulting set $(\mathbb{X} \to \mathbb{P})^{\perp}_{\equiv_{\mathbb{P}}}$ forms the following complete lattice:

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

Note that the top element $[x \mapsto (\mathrm{a})_{\equiv_{\mathbb{P}}}]_{x \in \mathbb{X}}$ maps each x to a different type. In order to relate sets of monotype environments to a polytype environment, we use $\overline{\textbf{lca}} : \mathscr{P}(\mathbb{X} \to \mathbb{M}) \to (\mathbb{X} \to \mathbb{P})^{\perp}_{\equiv_{\mathbb{P}}}$ with $\overline{\textbf{lca}}(\emptyset) = \perp_{\mathbb{P}}$. Also, use $\overline{\textbf{ground}} : (\mathbb{X} \to \mathbb{P})^{\perp}_{\equiv_{\mathbb{P}}} \to \mathscr{P}(\mathbb{X} \to \mathbb{M})$ with $\overline{\textbf{ground}}(\perp_{\mathbb{P}}) = \emptyset$. The following is a Galois connection:

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

We compute the polytype semantics $\mathcal{H}[\![e]\!] := \overline{\textbf{lca}} \circ \mathcal{T}[\![e]\!] \circ \overline{\textbf{ground}}$ for each $e \in \mathbb{E}$, yielding the rules in Fig. 2 as follows.

Consider computing $\mathcal{H}[\![x]\!]$ from Fig. 6. The simple rule for λ-bound variables can be translated to polytypes by computing $\mathcal{H}[\![x]\!]\rho_{\mathbb{P}} = \overline{\textbf{lca}}(\{\rho_{\mathbb{M}}[\kappa \mapsto \rho_{\mathbb{M}}(x)] \mid \rho_{\mathbb{M}} \in \overline{\textbf{ground}}(\rho_{\mathbb{P}})\}) = \rho_{\mathbb{P}}[\kappa \mapsto \rho_{\mathbb{P}}(x)]$. Recall that $t; \rho_{\mathbb{P}}$ is synonymous with $\rho_{\mathbb{P}}[\kappa \mapsto t]$ and, hence, the (VAR)-rule in Fig. 2 follows. In the case for **let**-bound variables, the set intersection computes an instance of x. Here, $y_1, \ldots y_n$ are variables that have been added to the environment after x (x is in scope of y_i). The corresponding polytype rule is $\mathcal{H}[\![x]\!]\rho_{\mathbb{P}} = \textbf{gci}(\rho_{\mathbb{P}}[\kappa \mapsto \mathrm{a}], \rho_{\mathbb{P}}[x \mapsto \mathrm{a}_0, y_1 \mapsto \mathrm{a}_1, \ldots y_n \mapsto \mathrm{a}_n, \kappa \mapsto \rho_{\mathbb{P}}(x)])$ since an intersection $\cap$ on monotypes translates to unification **gci** on polytypes. Consider an example:

EXAMPLE 4. *When inferring the recursive call to g in the program* `f x = `***`let`***` g y = `***`if null`***` [x,y] `***`then`***` g 7 `***`else`***... *the test* ***`null`*** `[x,y]` *makes the types of* `x` *and* `y` *equal, so that the environment is* $\rho_{\mathbb{P}} = [f \mapsto \mathrm{b}{\to}\mathrm{c}, x \mapsto \mathrm{b}, g \mapsto \mathrm{b}{\to}\mathrm{d}, y \mapsto \mathrm{b}]$. *Now* $\mathcal{H}[\![g]\!]\rho_{\mathbb{P}} = \overline{\textbf{gci}}(\rho_{\mathbb{P}}[\kappa \mapsto \mathrm{a}], \rho_{\mathbb{P}}[g \mapsto \mathrm{a}_0, y \mapsto \mathrm{a}_1, \kappa \mapsto \rho_{\mathbb{P}}(g)])$ *is computed as follows (note that the definition of* $\overline{\textbf{gci}}$ *requires the type variables of one of its arguments to be renamed):*

$$
\begin{array}{lll}
 & & \quad\ \ f \qquad\ \ x \quad\ \ g \qquad\ \ y \quad\ \ \kappa \\
\mathcal{H}[\![g]\!]\rho_{\mathbb{P}} & = \overline{\textbf{gci}} & \left(\begin{array}{l} \langle\, \mathrm{b}{\to}\mathrm{c} \ , \mathrm{b} \ , \mathrm{b}{\to}\mathrm{d} \ , \mathrm{b} \ , \mathrm{a} \ \rangle_{\equiv_{\mathbb{P}}}, \\ \langle\, \mathrm{b}{\to}\mathrm{c} \ , \mathrm{b} \ , \mathrm{a}_0 \quad\ \ , \mathrm{a}_1, \mathrm{b}{\to}\mathrm{d} \ \rangle_{\equiv_{\mathbb{P}}} \end{array} \right) \\
 & = \overline{\textbf{gci}} & \left(\begin{array}{l} \langle\, \mathrm{b}{\to}\mathrm{c} \ , \mathrm{b} \ , \mathrm{b}{\to}\mathrm{d} \ , \mathrm{b} \ , \mathrm{a} \ \rangle, \\ \langle\, \mathrm{e}{\to}\mathrm{f} \ , \mathrm{e} \ , \mathrm{g} \quad\ \ , \mathrm{h} \ , \mathrm{e}{\to}\mathrm{j} \ \rangle \end{array} \right) \\
 & = & \quad \langle\, \mathrm{b}{\to}\mathrm{c} \ , \mathrm{b} \ , \mathrm{b}{\to}\mathrm{d} \ , \mathrm{b} \ , \mathrm{b}{\to}\mathrm{j} \ \rangle_{\equiv_{\mathbb{P}}}
\end{array}
$$

The returned type $\mathrm{b}{\to}\mathrm{j}$ *is an instance of g. Note that the computation above is equivalent to computing free variables when constructing a type scheme: in both cases a set of type variables in the environment at g (namely* $X = vars(\rho_{\mathbb{P}}(f)) \cup vars(\rho_{\mathbb{P}}(x))$*) is gathered and the variables* $vars(\rho_{\mathbb{P}}(g)) \setminus X$ *in* $\rho_{\mathbb{P}}(g)$ *are those variables that can be instantiated.*

$$
\begin{array}{lcl}
model & : & \mathbb{P}^{\mathbb{R}} \times \mathbb{M} \to \mathscr{P}(\mathbb{N}) \\
model(\mathrm{a}.f_a, t) & = & \{f_a \mid t \cap \mathbb{M}^{\underline{\{\}}} = \emptyset\} \\
model(t_1 {\to} t_2, t'_1 {\to} t'_2) & = & \cup_{i=1,2}\ model(t_i, t'_i) \\
model(\texttt{Int}, \texttt{Int}) & = & \emptyset \\
model([\mathrm{a}.f_a], \sigma, \bar{p}) & = & \{[t] \mid t \in model(\mathrm{a}.f_a, t, \bar{p})\} \\
\multicolumn{3}{l}{model(\underline{\{}\mathrm{N}_1.f_1 : t_1, \ldots \mathrm{N}_n.f_n : t_n, \mathrm{a}.f_a\underline{\}}, \bar{f}) =} \\
\multicolumn{3}{l}{\quad \bigcup_{i=1,\ldots n}\{\{f_i\} \cup model(t_i, t'_i) \mid \exists t'_i \in \mathbb{M} \,.\, \mathrm{N}_i : t'_i \in \bar{f}\}} \\
\multicolumn{3}{l}{\quad \cup\{f_a \mid \forall \mathrm{N} \notin \{\mathrm{N}_1, \ldots \mathrm{N}_n\} \,.\exists t' \in \mathbb{M} \,.\, \mathrm{N} : t' \in \bar{f}\}}
\end{array}
$$

Figure 7. extraction of flags that hold for the given monotype

Thus, the derived rules compute the set of type variables X that are to be instantiated at the usage site rather than the definition site. In order to keep with the tradition of using type schemes, the (LET)-rule in Fig. 2 computes the set X and stores a type scheme $\forall X \,.\, t$ in the environment which the (VAR-LET)-rule uses to create a fresh instance. Although the set X will be different in the two approaches, the set of instantiated variables will eventually be equal once a fixpoint is reached. Hence, the two approaches are equivalent in term of the computed types.

The translation of the abstraction rule $\lambda x \,.\, e$ requires the type of e which is expressed as premise $\rho_{\mathbb{P}}[x \mapsto \mathrm{a}] \vdash e$ in rule (LAM) of Fig. 2. The type of an application $e_1\ \ e_2$ is an intersection of two environments which translates to a meet **gci** of two polytype environments which, in turn, is made explicit in the (APP)-rule by computing the most general unifier. The rule for **let** computes the greatest fixpoint for x starting with the most general type a. When using type schemes, this translates to binding x to $\forall \mathrm{a} \,.\, \mathrm{a}$ as done in the (LET)-rule. Finally, the conditional rule is an intersection of two symmetric types, one for each branch. Note that the condition is restricted to be `Int` which is only shown once in the (COND)-rule of Fig. 2. The meet is again expressed using mgu. This completes the translation of the derived monotype semantics to polytypes.

By construction, the obtained inference rules are optimal:

LEMMA 3. *(Optimality)* $\forall e \in \mathbb{E}$, $\mathcal{H}[\![e]\!] = \overline{\textbf{lca}} \circ \mathcal{T}[\![e]\!] \circ \overline{\textbf{ground}}$.

Together with Lemma 1 it follows that type annotations cannot improve the inferred result. Moreover, the abstract transformers soundly approximate the collecting semantics:

LEMMA 4 (Soundness). *For all* $\rho \in \mathbb{X} \to \mathbb{U}_{\perp}$ *and* $e \in \mathbb{E}$ *without record constructs,* $\mathcal{H}[\![e]\!](\overline{\textbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho\}))) \sqsubseteq_{\mathbb{P}} \overline{\textbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![e]\!]\{\rho\}))$.

Note that the approximation is $\sqsubseteq_{\mathbb{P}}$-lower since the $\mathbb{P}$-lattice is constructed so that the type error is the smallest element $\perp_{\mathbb{P}}$. In particular, well-typed programs do not go wrong in the sense that whenever $\Omega \in \mathcal{C}_1[\![e]\!]\{\rho\}$ then $\mathcal{H}[\![e]\!](\overline{\mathbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho\}))) = \perp_{\mathbb{P}}$.

4.3 Abstraction to Polytypes with Flow

Let $[\![\cdot]\!] : \mathbb{B} \to \mathscr{P}(\mathscr{P}(\mathbb{N}))$ denote the set of all propositional variable sets that satisfy a Boolean formula and define $\beta_1 \models \beta_2$ if $[\![\beta_1]\!] \subseteq [\![\beta_2]\!]$. Then $\langle \mathbb{B}_\perp, \models, \vee, \wedge, \mathit{true}, \perp_{\mathbb{B}}\rangle$ is a complete lattice where true is an empty formula and $\perp_{\mathbb{B}}$ a formula with no models.

We combine Boolean functions and record polymorphic types $\mathbb{P}^{\mathbb{R}}$ using a reduced cardinal power construction $\mathcal{D}_2^{\mathcal{D}_1}$ [5, Sect. 10.2] where $\mathcal{D}_1 = \mathbb{B}$ and $\mathcal{D}_2 = \mathbb{P}^{\mathbb{R}}$. Here, a domain element $f \in \mathcal{D}_2^{\mathcal{D}_1}$ is a monotone function: if $\beta_1 \models \beta_2$ then $f(\beta_1) \sqsubseteq_{\mathbb{P}} f(\beta_2)$. Intuitively, the Boolean function β refines the environment $\rho_{\mathbb{R}} = f(\beta)$ by stating which record field exists. For presentational reasons, we use a tuple of domains $\mathbb{P}^{\mathbb{R}} \rhd \mathbb{B}$ that is isomorphic to $\mathcal{D}_2^{\mathcal{D}_1}$.

For reasons of legibility, we write $\langle t_{\mathbb{R}}, \beta\rangle \in \mathbb{P}^{\mathbb{R}} \rhd \mathbb{B}$ instead of $t_{\mathbb{R}}|\beta$. We relate types $\langle t_{\mathbb{R}}, \beta\rangle$ with a set of monotypes using *model* in Fig. 7 that takes a record type $t_{\mathbb{R}} \in \mathbb{P}^{\mathbb{R}}$ and one monotype $t' \in \mathbf{ground}(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_{\mathbb{R}}))$ and returns those flags in $t_{\mathbb{R}}$ for which t' contains the corresponding record field. Here, $\mathbb{M}^{\underline{\{\}}} \subset \mathbb{M}$ denotes monotypes containing only empty records. Abstracting from a set of monotypes $\bar{t} \subseteq \mathscr{P}(\mathbb{M})$ is then defined by $\alpha_{\mathbb{R}} : \mathscr{P}(\mathbb{M}) \to \mathbb{P}^{\mathbb{R}}$ as $\alpha_{\mathbb{R}}(\bar{t}) = \langle t_{\mathbb{R}}, \beta\rangle$ where $t_{\mathbb{R}} = \Uparrow^{\mathbb{R}}_{\mathbb{P}}(\mathbf{lca}(\bar{t}))$ and $[\![\beta]\!] = \{\mathit{model}(t_{\mathbb{R}}, t) \mid t \in \bar{t}\}$. Symmetrically, the concretization $\gamma_{\mathbb{R}} : \mathbb{P}^{\mathbb{R}} \to \mathscr{P}(\mathbb{M})$ is defined by $\gamma_{\mathbb{R}}(\langle t_{\mathbb{R}}, \beta\rangle) = \{t \in \mathbf{ground}(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_{\mathbb{R}})) \mid \mathit{model}(t_{\mathbb{R}}, t) \in [\![\beta]\!]\}$. For example, $\gamma_{\mathbb{R}}(\langle \mathrm{a}.f_a, \neg f_a\rangle) = \{t \in \mathbb{M} \mid t \in \mathbb{M}^{\underline{\{\}}}\}$ and $\gamma_{\mathbb{R}}(\langle \underline{\{}\mathrm{N}.f_a : \mathrm{b}.f_b, \mathrm{c}.f_c\underline{\}}, f_a \wedge \neg f_c\rangle) = \{\underline{\{}\mathrm{N} : t\underline{\}} \mid t \in \mathbb{M}\}$.

We lift $\alpha_{\mathbb{R}}$ and $\gamma_{\mathbb{R}}$ to environments, giving $\alpha^{\mathcal{X}}_{\mathbb{R}}, \gamma^{\mathcal{X}}_{\mathbb{R}}$. Let $\rho^1_{\mathbb{R}} \sqsubseteq_{\mathbb{R}} \rho^2_{\mathbb{R}}$ if $\gamma^{\mathcal{X}}_{\mathbb{R}}(\rho^1_{\mathbb{R}}) \subseteq \gamma^{\mathcal{X}}_{\mathbb{R}}(\rho^2_{\mathbb{R}})$ and $\rho^1_{\mathbb{R}} \equiv_{\mathbb{R}} \rho^2_{\mathbb{R}}$ if $\rho^1_{\mathbb{R}} \sqsubseteq_{\mathbb{R}} \rho^2_{\mathbb{R}}$ and $\rho^2_{\mathbb{R}} \sqsubseteq_{\mathbb{R}} \rho^1_{\mathbb{R}}$. The following Galois connection

$$\langle \mathscr{P}(\mathbb{X} \to \mathbb{M}), \subseteq\rangle \underset{\alpha^{\mathcal{X}}_{\mathbb{R}}}{\overset{\gamma^{\mathcal{X}}_{\mathbb{R}}}{\leftrightarrows}} \langle ((\mathbb{X} \to \mathbb{P}^{\mathbb{R}}) \rhd \mathbb{B})^{\perp}_{\equiv_{\mathbb{R}}}, \sqsubseteq_{\mathbb{R}}\rangle$$

replaces the simpler abstraction to polytypes in Sect. 4.2 that was based on $\mathbf{ground}$ and $\overline{\mathbf{lca}}$ only. We now use the following lattice:

$$\langle ((\mathbb{X} \to \mathbb{P}^{\mathbb{R}}) \rhd \mathbb{B})^{\perp}_{\equiv_{\mathbb{R}}}, \sqsubseteq_{\mathbb{R}}, \sqcup_{\mathbb{R}}, \sqcap_{\mathbb{R}}, [x \mapsto \langle \mathrm{a}.f_a, \mathit{true}\rangle_{\equiv_{\mathbb{R}}}]_{x \in \mathbb{X}}, \perp_{\mathbb{R}}\rangle$$

Here, $\sqcup_{\mathbb{R}}$ and $\sqcap_{\mathbb{R}}$ operate on both, type terms and Boolean functions. The meet $\langle \rho_{\mathbb{R}}, \beta\rangle := \langle \rho^x_{\mathbb{R}}, \beta^x\rangle \sqcap_{\mathbb{R}} \langle \rho^y_{\mathbb{R}}, \beta^y\rangle$ is computed by first computing a unifier $\sigma = \mathit{mgu}(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(\rho^x_{\mathbb{R}}), \Downarrow^{\mathbb{R}}_{\mathbb{P}}(\rho^y_{\mathbb{R}}))$ for types and adjusting the flows to $\mathit{applyS}(\sigma, \rho^x_{\mathbb{R}}|\beta^x) = \rho^{x\sigma}_{\mathbb{R}}|\beta^{x\sigma}$ and to $\mathit{applyS}(\sigma, \rho^y_{\mathbb{R}}|\beta^y) = \rho^{y\sigma}_{\mathbb{R}}|\beta^{y\sigma}$. Since the two environments $\rho^{x\sigma}_{\mathbb{R}}$ and $\rho^{y\sigma}_{\mathbb{R}}$ may contain different Boolean flags, we equate them and conjoin both flows, so that $\beta = \beta^{x\sigma} \wedge \beta^{y\sigma} \wedge \wr\rho^{x\sigma}_{\mathbb{R}}\wr \Leftrightarrow \wr\rho^{y\sigma}_{\mathbb{R}}\wr$. Since we do not require the join $\sqcup_{\mathbb{R}}$, we will not define it here.

We now consider abstracting the monotype semantics in Fig. 6 into a flow semantics $\mathcal{F}$ by computing $\mathcal{F}[\![e]\!] = \gamma^{\mathcal{X}}_{\mathbb{R}} \circ \mathcal{T}[\![e]\!] \circ \alpha^{\mathcal{X}}_{\mathbb{R}}$.

We first discuss the derivation of the new record operations. In the record update and selector all fields $\mathrm{N}_1, \ldots \mathrm{N}_n$ except field N_i are abstracted by $\mathbf{lca}$ to a single row variable. For the empty record, all fields are summarized. These rules add $\neg f_a, f_N$ and f'_N as flow information, depending the field being present or absent.

We now address the computation of flows for fields that are not explicitly set or extracted. Observe that each type equation in Fig. 6 extends the vectors $\bar{\rho}_{\mathbb{M}}$ with a binding to κ that represents some or all of the types that the result of the concrete semantics may take on. In case all types are computed, then all fields that are not explicitly mentioned remain the same and bi-implications can be used to describe their flow. This is the case if the type equation $f^\sharp$ is forward-complete, that is, if $f \circ \gamma = \gamma \circ f^\sharp$ [21]. Here, $\gamma = \gamma^{\mathcal{X}}_{\mathbb{M}} \circ \gamma^{\mathcal{X}}_{\mathbb{R}}$ and the concrete semantics is the single denotation $\mathcal{S}[\![e]\!]$ which, when lifted to sets, is $f = \lambda\bar{\rho}\,.\,\{\mathcal{S}[\![e]\!]\ \rho \mid \rho \in \bar{\rho}\}$. By Lemma 1, the abstract semantics $f^\sharp = \mathcal{T}[\![e]\!]$ is equivalent to $f^\sharp = \alpha \circ \mathcal{C}[\![e]\!]\bar{\rho} \circ \gamma =: \alpha \circ f_{col} \circ \gamma$. Hence, the inference is forward-complete if $f \circ \gamma = \gamma \circ \alpha \circ f_{col} \circ \gamma$. Now consider the outermost language construct of an expression e under the assumption that sub-computations in e return forward-complete results. The construct is not forward-complete if either $f \neq f_{col}$ or if $\gamma \circ \alpha$ is applied to value for which $\gamma \circ \alpha$ is not the identity. By construction, $f = f_{col}$ holds for all language constructs except for conditionals since they are abstracted to a non-deterministic choice. This motivates the two sequence implications between output and the two branches in the (COND)-rule of Fig. 3. By definition of α [23], the κ-bound value is abstracted to the best type which, by construction, is computed by the equations in Fig. 6. However, α uses a restriction on λ-bound variables so that the type semantics $\mathcal{T}[\![x]\!]\bar{\rho}_{\mathbb{M}} = \{\rho_{\mathbb{M}}[\kappa \mapsto \rho_{\mathbb{M}}(x)] \mid \rho_{\mathbb{M}} \in \bar{\rho}_{\mathbb{M}}\}$ that accesses a λ-bound variable is a proper approximation of the concrete semantics $\mathcal{S}[\![x]\!]\ \rho = \rho(x)$, hence the monotype semantics $\mathcal{T}[\![x]\!]$ is not forward complete. Due to this approximation, the set of monotypes bound to κ is smaller than the optimal solution. Thus, if a record field exists in the result bound to κ it must exists in x. This is expressed in the (VAR)-rule of Fig. 3 by the implication from the result t_x to $\rho_{\mathbb{P}}(x)$. While the (VAR)-rule for λ-bound variables is not forward complete, it is still backward complete, thus, the type inferred for a λ-bound variable cannot be improved by adding a type signature. We therefore have:

LEMMA 5 (Optimality). $\mathcal{F}[\![e]\!] = \gamma^{\mathcal{X}}_{\mathbb{R}} \circ \mathcal{T}[\![e]\!] \circ \alpha^{\mathcal{X}}_{\mathbb{R}}$ *for all* $e \in \mathbb{E}$.

LEMMA 6 (Soundness). *For all* $e \in \mathbb{E}$ *and* $\rho \in \mathbb{X} \to \mathbb{U}_\perp$, $\mathcal{F}[\![e]\!](\alpha^{\mathcal{X}}_{\mathbb{R}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho\}))) \sqsubseteq_{\mathbb{P}} \alpha^{\mathcal{X}}_{\mathbb{R}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}_1[\![e]\!]\{\rho\}))$.

4.4 Lack of Forward-Completeness

Each abstract transformer on sets of monotype environments computes the best type expressible in $\mathbb{X} \to \mathbb{M}$, that is, $\mathcal{T}[\![e]\!] = \alpha^{\mathcal{X}}_{\mathbb{M}} \circ \mathcal{C}[\![e]\!] \circ \gamma^{\mathcal{X}}_{\mathbb{M}}$. Applying $\gamma^{\mathcal{X}}_{\mathbb{M}}$ to each side of the equation yields $\gamma^{\mathcal{X}}_{\mathbb{M}} \circ \mathcal{T}[\![e]\!] = \gamma^{\mathcal{X}}_{\mathbb{M}} \circ \alpha^{\mathcal{X}}_{\mathbb{M}} \circ \mathcal{C}[\![e]\!] \circ \gamma^{\mathcal{X}}_{\mathbb{M}}$. Thus, $\mathcal{T}[\![e]\!]$ is forward complete ($\gamma \circ f^\sharp = f \circ \gamma$) if $\gamma^{\mathcal{X}}_{\mathbb{M}} \circ \alpha^{\mathcal{X}}_{\mathbb{M}}$ does not impose an approximation of the result. Alas, the $\alpha^{\mathcal{X}}_{\mathbb{M}}$ that we used to derive $\mathcal{T}[\![\cdot]\!]$ restricts the type of λ-bound variables to be monomorphic [23] so that $\gamma^{\mathcal{X}}_{\mathbb{M}} \circ \alpha^{\mathcal{X}}_{\mathbb{M}}$ imposes an approximation for $\mathcal{T}[\![x]\!]$ when x is λ-bound. For instance, consider the following program that we will call p:

```
let g proj xs ys = proj xs && proj ys in g null
```

Here, the infix function `&&` is of type `Bool→Bool→Bool` and **null** : `[a]→Bool`. The collecting semantics $\mathcal{C}[\![p]\!]\{\rho_\emptyset\}$ (where $\rho_\emptyset \in \mathbb{X}$ is an empty concrete environment) computes a function in κ that can be applied to two lists of different types. Hence, computing the type of the collecting semantics of p yields $\overline{\mathbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![p]\!]\{\rho_\emptyset\})) = [\kappa \mapsto \texttt{[a]→[b]→Bool}]$. However, the type is computed as $\mathcal{H}[\![p]\!](\overline{\mathbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho_\emptyset\}))) = [\kappa \mapsto \texttt{[a]→[a]→Bool}]$ because the incompleteness of the abstract transformer for λ-bound variables approximates the collecting semantics by forcing `proj` to have the same type in all uses in `g`. Interestingly, this incompleteness does not surface if each function argument is only used once. Thus, suppose that $\mathbb{E}' \subset \mathbb{E}$ is the set of all programs where each λ-bound variable is use at most once, then both type inferences are complete:

LEMMA 7 (Completeness). *For all* $e \in \mathbb{E}'$ *and* $\rho \in \mathbb{X} \to \mathbb{U}_\perp$, $\mathcal{F}[\![e]\!](\alpha^{\mathcal{X}}_{\mathbb{R}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho\}))) = \alpha^{\mathcal{X}}_{\mathbb{R}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![e]\!]\{\rho\}))$. *For* $e \in \mathbb{E}'$ *without records,* $\mathcal{H}[\![e]\!](\overline{\mathbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\{\rho\}))) = \overline{\mathbf{lca}}(\alpha^{\mathcal{X}}_{\mathbb{M}}(\mathcal{C}[\![e]\!]\{\rho\}))$.

This incompleteness also affects the flow inference for record fields. Consider the following variant p' of the program above where the identity function **id** has the type derived in Ex. 1:

```
let g proj xs ys = #foo (proj xs) &&
                   #bar (proj ys)     in g id
```

The type of the collecting semantics is $\alpha_{\mathbb{R}}^{\mathcal{X}}(\alpha_{\mathbb{M}}^{\mathcal{X}}(\mathcal{C}[\![p']\!]\{\rho_\emptyset\})) = \langle[\kappa \mapsto \underline{\{}\text{FOO}.f_1 : \texttt{Bool}, \text{a}.f_a\underline{\}}\to\underline{\{}\text{BAR}.f_2 : \texttt{Bool}, \text{b}.f_b\underline{\}}\to\texttt{Bool}], \beta\rangle$ where $\beta \equiv f_1 \wedge f_2$. The flow semantics $\mathcal{F}[\![p']\!]\alpha_{\mathbb{R}}^{\mathcal{X}}(\alpha_{\mathbb{M}}^{\mathcal{X}}(\{\rho_\emptyset\})) = \langle[\kappa \mapsto \underline{\{}\text{BAR}.f_1 : \texttt{Bool}, \text{FOO}.f_2 : \texttt{Bool}, \text{a}.f_a\underline{\}}\to\underline{\{}\text{BAR}.f_3 : \texttt{Bool}, \text{FOO}.f_4 : \texttt{Bool}, \text{a}.f'_a\underline{\}}\to\texttt{Bool}], \beta\rangle$ where $\beta \equiv f_1 \wedge f_2 \wedge f_3 \wedge f_4$. Thus, the function can only be applied to records that contain both fields. The transformer is approximate in that it adds implications between the uses of `proj` and the formal argument, thereby creating spurious flow information, namely from the result of the first use of `proj` to the argument of the second use of `proj`, thereby propagating that the FOO-field is required by `#foo` to the argument `ys` (and vice-versa for `#bar` and `xs`). No spurious flow is inferred when flow information only travels from the formal arguments to the usage site, that is, when no function is passed as argument that is applied to a record. Hence, we conclude:

OBSERVATION 1. *Under the assumption that conditionals are abstracted to non-deterministic choices and that no argument is a function expecting a record or that such functions are only used once, our inference rejects a program if and only if it contains a path from an empty record to a field access on which the field has not been added.*

5. Extensions

The inference rules in Fig. 3 only create atoms f_a, $\neg f_a$ and two variable Horn clauses $f_a \to f_b$. Moreover, expansion on Boolean functions only duplicates clauses (after renaming them), thereby also only creating two-variable Horn clauses. Satisfiability can therefore be checked in linear time using a 2-SAT solver. Indeed, it is folklore that solvers with atomic subtyping can express this type system by using constraints of the form $\mathit{false} \subseteq \mathit{true}$, $\text{a} \subseteq \text{b}$ for each clause $f_b \to f_a$, $\text{c} \doteq \mathit{true}$ for each clause f_c, and $\text{d} \doteq \mathit{false}$ for each clause $\neg f_d$. Our approach of separating the flow information from type terms has the advantage that no complex subtyping constraints are required to model the passing of arguments. This makes our approach attractive to solvers that rely on cheap unification for type terms, such as the one in Haskell [22].

What about other language constructs for records, such as those proposed by Cardelli and Mitchell [2]? By deriving backward-complete inference rules from the concrete semantics, the constraint language necessary for implementing various operations becomes evident. We require backward-complete rules since incomplete rules could suggest that a weaker constraint language suffices.

For instance, the removal of a record field or the renaming of a field can be implemented with 2-variable Horn clauses. Typing the (so-called asymmetric) concatenation of two records e_1 @ e_2 with type $\underline{\{}\text{N}.\text{a}.f_a^1, \text{b}.f_b^1\underline{\}}$ and $\underline{\{}\text{N}.\text{a}.f_a^2, \text{b}.f_b^2\underline{\}}$ requires that the output record $\underline{\{}\text{N}.\text{a}.f_a, \text{b}.f_b\underline{\}}$ contains a field if one of the input records contains a field. Thus, the flow information is $f_a \to (f_a^1 \vee f_a^2) \wedge f_b \to (f_b^1 \vee f_b^2)$ which cannot be simplified to clauses with two variables. However, consider inverting the meaning of a flag so that $\neg f_a$ states that a field exists and f_a states that it does not. Then each atom in the clauses above is inverted and the constraint for record concatenation is $(f_a^1 \wedge f_a^2 \to f_a) \wedge (f_b^1 \wedge f_b^2) \to f_b$. These multi-variable Horn clauses can also be solved in linear time [7], thereby showing that record concatenation can be implemented efficiently. Now consider symmetric record concatenation e_1 @@ e_2 where it is a type error if a field exists in both records. With the types for e_i as given above and using the original interpretation of flags, an optimal transformer must assert $\neg(f_a^1 \wedge f_a^2) \equiv \neg f_a^1 \vee \neg f_a^2$ (analogous for f_b^i). In the inverted setting, this constraint becomes $f_a^1 \vee f_a^2$ which is not in Horn form. Although satisfiability and existential projection can be computed using a SAT solver [1], it is questionable if this operation justifies this increase in complexity.

Since GADTs were proposed, there has been an increased interest in inferring each branches of a conditional under a different typing assumption. A very simple instance of this idea is given by the construct $e \equiv$ **when** N **in** x **then** e_1 **else** e_2 that evaluates e_1 if the record x contains the field N. In order to honor the information gained from the test, the type of x must be different in e_1 than in e_2 since x always contains N in e_1 but it never contains N in e_2. As with the normal conditional, we abstract the concrete semantics to a non-deterministic choice except that the information whether a field is present is retained. The result is the first type inference rule in Fig. 8 that resembles the (COND)-rule except that flag f_f which indicates if the field existed in x is used to generate flow information that depends on f_f: The returned flow consists of the required flow to make the two environments compatible and the flow $f_f \to (\wr t_r \int \Rightarrow \wr t_t^\sigma \int) \wedge \neg f_f \to (\wr t_r \int \Rightarrow \wr t_e^\sigma \int)$ qualifying the set of records in the output, depending on whether N was in x. Note that this complex construct can implement simpler ones such as field renaming or concatenation (for a fixed number of fields). Satisfiability of the generated flow requires a generic SAT solver.

In the context of GADTs and in many proposed record operations [2, 18, 26], the type term should not be constant but change depending on the current branch of a conditional or the existence of a record field. For instance, suppose that the semantics of the **when**-statement should be not be abstracted to a non-deterministic choice but that the result type may change, depending on whether N is in the record or not. To this end, we require a third domain of conditional unification constraints of the form $t_a =_\beta t_b$ where $t_a, t_b \in \mathbb{P}$ and $\beta \in \mathbb{B}$ is a Boolean function. In the example of the **when**-statement, a transformer of the semantics that is not abstracted to a non-deterministic choice (i.e. one that is forward-complete) is shown as the second rule in Fig. 8. In contrast to the first rule, the types of the two branches are not unified but kept as conditional unification constraints $t_r =_{f_f} t_t \wedge t_r =_{\neg f_f} t_e$. A program is type correct if the there is a truth assignment for the Boolean formulae so that the type terms, including the conditional constraints whose Boolean formula is true, are unifiable. In other words, the shown problem can be solved by an SMT solver with a theory of unification constraints. However, as far as we know, no state-of-the-art SMT solver features a theory of unification constraints. The only implementation we are aware of uses the backtracking capabilities of Prolog [20]. Future work has to determine how efficiently a theory of unification constraints can be implemented, possibly based on similar backtracking techniques.

The type system by Pottier [18] in which a record field becomes unaccessible when its type is inconsistent (not unifiable) can be obtained by altering the rule for record updates that determines the type of a field. To this end, note that the flag f_N in the (REC-UPDATE)-rule in Fig 3 will be true if the record field is eventually accessed. Thus, in order to express that a field must have a unifiable type only if it is accessed, we set the field to a fresh type variable c and track the conditional equation $\text{c} =_{f_N} t$. Together with the rules for record concatenation, the resulting type system repairs the incompleteness in Pottier's type inference illustrated in Sect. 1.

Besides the completeness issues, we observed in Sect. 1 that the use of implication constraints may lead to large types that make the instantiation of a function type costly and error messages hard to understand. A desirable property of an abstract domain is therefore that it is closed under eliminating existentially quantified variables which allows that the information inferred for a function f can be

$$\frac{\begin{array}{c}\rho_{\mathbb{R}}|\beta \vdash x : \underline{\{}\textsc{N}.f_f : t_f, \mathrm{a}.f_a\underline{\}}; \rho^s_{\mathbb{R}}|\beta^s \qquad \rho^s_{\mathbb{R}}|\beta^s \wedge f_f \vdash e_1 : t_t; \rho^{st}_{\mathbb{R}}|\beta^{st} \qquad \rho^s_{\mathbb{R}}|\beta^s \wedge f_f \vdash e_2 : t_e; \rho^{se}_{\mathbb{R}}|\beta^{se} \qquad t_r = \Uparrow^{\mathbb{R}}_{\mathbb{P}}(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(t^\sigma_t)) \\ \sigma = mgu(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_t; \rho^{st}_{\mathbb{R}}), \Downarrow^{\mathbb{R}}_{\mathbb{P}}(t_e; \rho^{se}_{\mathbb{R}})) \qquad t^\sigma_t; \rho^{t\sigma}_{\mathbb{R}}|\beta^{t\sigma} = applyS(\sigma, t_t; \rho^t_{\mathbb{R}}|\beta^t) \qquad t^\sigma_e; \rho^{e\sigma}_{\mathbb{R}}|\beta^{e\sigma} = applyS(\sigma, t_e; \rho^e_{\mathbb{R}}|\beta^e)\end{array}}{\rho_{\mathbb{R}}|\beta \vdash \textbf{when } \textsc{N} \textbf{ in } x \textbf{ then } e_1 \textbf{ else } e_2 : t_r; \rho^{e\sigma}_{\mathbb{R}}|\beta^{t\sigma} \wedge \beta^{e\sigma} \wedge \wr\rho^{t\sigma}_{\mathbb{R}}\int^{\mathbb{X}} \Leftrightarrow \wr\rho^{e\sigma}_{\mathbb{R}}\int^{\mathbb{X}} \wedge f_f \rightarrow (\wr t_r\int \Rightarrow \wr t^\sigma_t\int) \wedge \neg f_f \rightarrow (\wr t_r\int \Rightarrow \wr t^\sigma_e\int)} \text{(FCOND)}$$

$$\frac{\begin{array}{c}\rho_{\mathbb{R}}|\beta \vdash x : \underline{\{}\textsc{N}.f_f : t_f, \mathrm{a}.f_a\underline{\}}; \rho^s_{\mathbb{R}}|\beta^s \qquad \rho^s_{\mathbb{R}}|\beta^s \wedge f_f \vdash e_1 : t_t; \rho^{st}_{\mathbb{R}}|\beta^{st} \qquad \rho^s_{\mathbb{R}}|\beta^s \wedge f_f \vdash e_2 : t_e; \rho^{se}_{\mathbb{R}}|\beta^{se} \qquad t_r = \mathrm{a}.f_a \\ \sigma = mgu(\Downarrow^{\mathbb{R}}_{\mathbb{P}}(\rho^{st}_{\mathbb{R}}), \Downarrow^{\mathbb{R}}_{\mathbb{P}}(\rho^{se}_{\mathbb{R}})) \qquad \rho^{t\sigma}_{\mathbb{R}}|\beta^{t\sigma} = applyS(\sigma, \rho^t_{\mathbb{R}}|\beta^t) \qquad \rho^{e\sigma}_{\mathbb{R}}|\beta^{e\sigma} = applyS(\sigma, \rho^e_{\mathbb{R}}|\beta^e)\end{array}}{\rho_{\mathbb{R}}|\beta \vdash \textbf{when } \textsc{N} \textbf{ in } x \textbf{ then } e_1 \textbf{ else } e_2 : t_r; \rho^{e\sigma}_{\mathbb{R}}|\beta^{t\sigma} \wedge \beta^{e\sigma} \wedge \wr\rho^{t\sigma}_{\mathbb{R}}\int^{\mathbb{X}} \Leftrightarrow \wr\rho^{e\sigma}_{\mathbb{R}}\int^{\mathbb{X}} \wedge f_f \rightarrow \wr t^\sigma_t\int \wedge \neg f_f \rightarrow \wr t^\sigma_e\int \mid t_r =_{f_f} t_t \wedge t_r =_{\neg f_f} t_e}$$

Figure 8. type inference rules for **when** N **in** x **then** e_1 **else** $e_2 \in \mathbb{E}$ and concatenation

decoder	lines	time w/o fields	time w. fields
Atmel AVR	1468	0.18s	0.32s
Atmel AVR + Sem	5166	1.55s	3.01s
Intel x86	9315	6.11s	15.65s
Intel x86 + Sem	18124	15.42s	27.38s

Figure 9. inference times on Ubuntu Linux, 3.4 GHz Intel Core i7

represented without resorting to information from local variables or the body of f. Type terms are trivially closed under projection since computing the **gci** implicitly propagates all information to the type of the function f. With respect to Boolean functions, projection can be computed using resolution or more dedicated algorithms [1]. More challenging are conditional constraints $t_1 =_\beta t_2$ where t_1, t_2 may contain type variables that are not mentioned in the type of the function f. These cannot, in general, be eliminated. However, the type of f from Sect. 1 can be compactly represented with the type term $\underline{\{}\textsc{FOO}.f_N : \mathrm{b}, \mathrm{a}.f_a\underline{\}} \rightarrow \underline{\{}\textsc{FOO}.f'_N : \mathrm{b}, \mathrm{a}.f'_a\underline{\}}$, the flow $f'_N \rightarrow f_N \wedge f'_a \rightarrow f_a$, and the conditional constraint $\mathrm{b} =_{f'_N} \mathtt{Int}$.

6. Implementation

We implemented the type inference with record flows in an open-source DSL used to decode assembler instructions and to translate these instructions into an intermediate language, thus giving them a semantics [25]. The motivation for using an optimal type inference was to avoid the complexity of type annotations, both, for the implementor and the user. The compiler is written in Standard ML and was evaluated when translated with MLton. We benchmarked the type inference with five specifications shown as "decoder" in Fig. 9. The last row shows the time for the Intel decoder and a semantic specification for 440 of the 896 instructions. By commenting out the functions that add clauses to a Boolean function, we obtain the running times without record fields (3^{rd} column) instead of with (4^{th} column). It shows that the 2-SAT solver is not the biggest bottleneck but that applying substitutions is equally expensive. Note that our solver is resolution based and needs quadratic time in the worst case, even though 2-SAT problems are solvable in linear time. Flexible records are used inside a built-in state monad. As such, we never had the need for record concatenation or the need that fields become unaccessible when their type is incompatible, as done by Pottier [18]. One problem we came across was that we needed to store a monadic action inside the state of the monad itself. However, extracting this monad and running it will unify the type of the field holding the monad with the monad type itself. This leads to an occurs check since both monad states share at least the same row variable. Reporting this field as unaccessible, as in Pottier's system, would have turned the type term error into an access to a non-existent field which would be even harder to understand. Our solution was to define an operator to remove a record field.

The implementation mostly follows the rules in Fig. 3. The resulting inference is therefore quite slow as calculating **gci** on two environments must descend simultaneously into the two environments. There is one optimization: each time we add an entry to an environment, we tag the environment with a fresh version. If **gci** is called on two environment with the same version number, it returns one of the identical environments without descending further.

Computing $expand_{f_1 \cdots f_n, f'_1 \cdots f'_n}(\beta)$ is implemented as per Def. 2. Interestingly, *expand* on Boolean functions is sensitive to stale variables: Suppose $\beta \equiv f_a \rightarrow f_b$ then $expand_{f_a f_b, f_a{}' f_b{}'}(\beta) = \beta \wedge f_a{}' \rightarrow f_b{}'$. However, suppose β also contains $f_c \leftrightarrow f_a$ where f_c is associated with a dead type variable. In this case, it will not be found during substitution and we accidentally compute $expand_{f_a f_b, f_a{}' f_b{}'}(\beta) = \beta \wedge f_a{}' \rightarrow f_b{}' \wedge f_c \leftrightarrow f_a{}'$ thereby making f_a and $f_a{}'$ equal. Since this phenomenon only manifests itself in reasonably complex programs, it was difficult to debug. However, the test program satisfied Observation 1 and, hence, we were clearly observing a bug and not an incompleteness of the inference.

7. Discussion and Related Work

Cousot [3] proposed to use his abstract interpretation framework [5] in order to systematically construct type inferences (and thus type systems) by abstracting the language semantics. We build on this work. However, Cousot uses a concretization γ to define the Hindley-Milner system. This suffices to show soundness but is insufficient for discussing completeness. In earlier work, we defines an abstraction α that makes the difference between λ- and **let**-bound variables explicit, thereby showing that the Milner-Mycroft inference [16] is backward complete [23]. Backward-completeness means that each rule computes the best type environment and type. For rules that do not modify the environment, this property corresponds to computing the principal type. In general, the best environment and type is obtained when allowing polymorphic recursion [16]. A stronger property that has been called "principal typing" (in contrast to principal types) [12, 28] stipulates that the type and environment inferred for a whole program are best. Our inference has this property when none of the inference rules approximate their result which - as discussed in Sect. 4.4 – is the case when each λ-bound function is used as most once.

By specifying a type inference of a language by the universe of types and requiring a backward-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 type terms $\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}} \rangle$ has infinite descending chains, the seemingly only work on using abstract interpretation for type inference addresses how to apply widening to ensure termination of the fixpoint computation [3, 10]. Even then, neither Gori et al. [10] nor Jim [12] found any type correct program that required many iterations to type check [10] which coincides with our experience.

Our work proposed to separate type terms and the tracking of constraints into two domains rather than using a unified constraint language that allows various analyses X to be plugged into a generic solver HM(X) [22]. An advantage of choosing separate domains before addressing which language constructs to support allows us to pick only domains that are closed under projection and that are condensing [8], thereby avoiding difficulties such as the "implication issue" highlighted by Pottier [26]. Our construction could also be relevant when certain domains must be excluded for performance reasons (e.g. general subtyping). For instance, the inference for GADTs has been addressed many times before a solution was found that could use normal unification for most parts of the program [22] rather than sub-typing. This solution is incomplete and even rejects programs that do not use GADTs [27]. Interestingly, backward-completeness which requires polymorphic recursion becomes more important in the presence of advanced type systems [22, 26]. For instance, a function that that uses a default value if none is supplied requires polymorphic recursion as follows: $\mathbf{let}\ f = \lambda x\,.\,\mathbf{when}\ \mathrm{N}\ \mathbf{in}\ x\ \mathbf{then}\ \ldots\ \mathbf{else}\ f(@\{\mathrm{N} = 42\}\ s)$.

While combining several abstract domains for type inference is rare [23], Boolean functions have been used to implement program analyses that piggy-back on already inferred type terms: Glynn et al. use a so-called shape mapping to map type terms to a constraint language that is then interpreted as a Boolean function [9]. Instantiation in their work performs the equivalent of expansion on Boolean formulae. By encoding this complex operation into our *applyS* function and using it throughout the type inference, our approach transparently transforms the Boolean formula during the inference of the type terms. Since our rules modify type terms and Boolean formulae at once, debugging and assessing completeness of the inference can be done on a rule-by-rule basis. For instance, we observed that stale variables in the Boolean formula must be removed for the correctness of expansion (see Sect. 6), whereas Glynn et al. are seemingly not aware of this as they merely state that the removal of stale variables "is applied aggressively" [9, Sect. 5].

7.1 Conclusion

We extended a type inference for the Hindley-Milner type system to one that infers row-polymorphic records as well as the information if a record field exists or not. The algorithm is based on a separation of type terms and Boolean functions. This separation of concerns not only provides the basis for an efficient implementation but also illustrates the cost of record operations addressed in the literature.

Acknowledgments

I would like to thank David Walker, Roberto Giacobazzi, Janis Voigtlaender, Rémy Didier, and the anonymous reviewers. This work was supported by DFG SI 1579/1 and MS SEIF 2013 grant.

References

[1] J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, LNCS, page 16. Springer, July 2011.

[2] L. Cardelli and J. C. Mitchell. Operations on Records. In *Mathematical Structures in Computer Science*, volume 1, pages 3–48, 1991.

[3] P. Cousot. Types as Abstract Interpretations. In *Principles of Programming Languages*, pages 316–331, Paris, France, Jan. 1997. ACM. invited paper.

[4] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In B. Robinet, editor, *International Symposium on Programming*, pages 106–130, Paris, France, Apr. 1976.

[5] 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.

[6] 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.

[7] W. F. Dowling and J. H. Gallier. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *J. Log. Program.*, 1 (3):267–284, 1984.

[8] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[9] K. Glynn, P. J. Stuckey, and M. Sulzmann. Boolean Constraints for Binding-time Analysis. In *Programs as Data Objects*, number 2053 in LNCS, pages 39–62. Springer, 2001.

[10] 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.

[11] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[12] 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.

[13] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The Undecidability of the Semi-unification Problem. *Information and Computation*, 102(1):83 – 101, 1993.

[14] 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.

[15] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[16] 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.

[17] M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, Jan. 1999. ISSN 1074-3227.

[18] F. Pottier. A Versatile Constraint-Based Type Inference System. *Nordic Journal of Computing*, 7(4):312–347, Nov. 2000.

[19] D. Rémy. Type Inference for Records in a Natural Extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[20] E. Robbins, A. King, and H. Jacob. Theory Propagation and Rational-Trees. In T. Schrijvers, editor, *Principles and Practice of Declarative Programming*, pages 193–204, Madrid, Spain, 2013. ACM.

[21] 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.

[22] T. Schrijvers, S. Peyton-Jones, M. Sulzmann, and D. Vytiniotis. Complete and Decidable Type Inference for GADTs. In *International Conference on Functional programming*, pages 341–352, Edinburgh, UK, Sept. 2009. ACM.

[23] A. Simon. Deriving a Complete Type Inference for Hindley-Milner and Vector Sizes using Expansion. *Science of Computer Programming*, 2014.

[24] A. Simon. Optimal Inference of Fields in Row-Polymorphic Records (proof appendix). Technical report, TU Munich, June 2014.

[25] A. Simon and J. Kranz. The GDSL toolkit: Generating Frontends for the Analysis of Machine Code. In *Program Protection and Reverse Engineering Workshop*, San Diego, California, USA, Jan. 2014. ACM.

[26] V. Simonet and F. Pottier. A Constraint-Based Approach to Guarded Algebraic Data Types. *Transactions on Programming Languages and Systems*, 29(1), Jan. 2007.

[27] D. Vytiniotis, S. L. Peyton Jones, and T. Schrijvers. Let Should not be Generalized. In A. Kennedy and N. Benton, editors, *Types in Languages Design and Implementation*, pages 39–50, Madrid, Spain, Jan. 2010. ACM.

[28] 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.