Data Flow Refinement Type Inference

ZVONIMIR PAVLINOVIC, New York University, USA and Google, USA
YUSEN SU, New York University, USA and University of Waterloo, Canada
THOMAS WIES, New York University, USA

Refinement types enable lightweight verification of functional programs. Algorithms for statically inferring refinement types typically work by reduction to solving systems of constrained Horn clauses extracted from typing derivations. An example is Liquid type inference, which solves the extracted constraints using predicate abstraction. However, the reduction to constraint solving in itself already signifies an abstraction of the program semantics that affects the precision of the overall static analysis. To better understand this issue, we study the type inference problem in its entirety through the lens of abstract interpretation. We propose a new refinement type system that is parametric with the choice of the abstract domain of type refinements as well as the degree to which it tracks context-sensitive control flow information. We then derive an accompanying parametric inference algorithm as an abstract interpretation of a novel data flow semantics of functional programs. We further show that the type system is sound and complete with respect to the constructed abstract semantics. Our theoretical development reveals the key abstraction steps inherent in refinement type inference algorithms. The trade-off between precision and efficiency of these abstraction steps is controlled by the parameters of the type system. Existing refinement type systems and their respective inference algorithms, such as Liquid types, are captured by concrete parameter instantiations. We have implemented our framework in a prototype tool and evaluated it for a range of new parameter instantiations (e.g., using octagons and polyhedra for expressing type refinements). The tool compares favorably against other existing tools. Our evaluation indicates that our approach can be used to systematically construct new refinement type inference algorithms that are both robust and precise.

CCS Concepts: • **Theory of computation → Program analysis**; **Type theory**.

Additional Key Words and Phrases: refinement type inference, abstract interpretation, Liquid types

ACM Reference Format:
Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. 2021. Data Flow Refinement Type Inference. *Proc. ACM Program. Lang.* 5, POPL, Article 19 (January 2021), 31 pages. https://doi.org/10.1145/3434300

1 INTRODUCTION

Refinement types are at the heart of static type systems that can check a range of safety properties of functional programs [Champion et al. 2018a; Chugh et al. 2012; Dunfield and Pfenning 2003, 2004; Freeman and Pfenning 1991; Rondon et al. 2008; Vazou et al. 2014; Vekris et al. 2016; Xi and Pfenning 1999; Zhu and Jagannathan 2013]. Here, basic types are augmented with *refinement predicates* that express relational dependencies between inputs and outputs of functions. For example, the contract

Authors' addresses: Zvonimir Pavlinovic, New York University, USA , Google, USA, zvonimir.pavlinovic@gmail.com; Yusen Su, New York University, USA , University of Waterloo, Canada, ys3547@nyu.edu; Thomas Wies, New York University, USA, wies@cs.nyu.edu.

2475-1421/2021/1-ART19
https://doi.org/10.1145/3434300

of an array read operator can be expressed using the refinement type

$$\mathsf{get} :: (a : \alpha\ \mathsf{array}) \to (i : \{\nu : \mathsf{int} \mid 0 \leq \nu < \mathsf{length}\ a\}) \to \alpha \ .$$

This type indicates that get is a function that takes an array a over some element type α and an index i as input and returns a value of type α. The type $\{\nu : \mathsf{int} \mid 0 \leq \nu < \mathsf{length}\ a\}$ of the parameter i refines the base type int to indicate that i must be an index within the bounds of the array a. This type can then be used to statically check the absence of erroneous array reads in a program. However, such a check will only succeed if the index expressions used in calls to get are also constrained by appropriate refinement types. Therefore, a number of type inference algorithms have been proposed that relieve programmers of the burden to provide such type annotations manually. These algorithms deploy a variety of analysis techniques ranging from predicate abstraction [Rondon et al. 2008; Vazou et al. 2013, 2014] to interpolation [Unno and Kobayashi 2009; Zhu and Jagannathan 2013] and machine learning [Champion et al. 2018a; Zhu et al. 2015, 2016]. A common intermediate step of these algorithms is that they reduce the inference problem to solving a system of constrained Horn clauses that is induced by a typing derivation for the program to be analyzed. However, this reduction already represents an abstraction of the program's higher-order control flow and affects the precision of the overall static analysis.

To better understand the interplay between the various abstraction steps underlying refinement type inference algorithms, this paper forgoes the usual reduction to constraints and instead studies the inference problem in its entirety through the lens of abstract interpretation [Cousot and Cousot 1977, 1979]. We start by introducing a parametric *data flow refinement type system*. The type system generalizes from the specific choice of logical predicates by allowing for the use of arbitrary relational abstract domains as type refinements (including e.g. octagons [Miné 2007], polyhedra [Bagnara et al. 2008; Cousot and Halbwachs 1978; Singh et al. 2017] and automata-based domains [Arceri et al. 2019; Kim and Choe 2011]). Moreover, it is parametric in the degree to which it tracks context-sensitive control flow information. This is achieved through intersection function types, where the granularity at which such intersections are considered is determined by how stacks are abstracted at function call sites. Next, we propose a novel concrete data flow semantics of functional programs that captures the program properties abstracted by refinement type inference algorithms. From this concrete semantics we then construct an abstract semantics through a series of Galois abstractions and show that the type system is sound and complete with respect to this abstract semantics. Finally, we combine the abstract semantics with an appropriate widening strategy to obtain an accompanying parametric refinement type inference algorithm that is sound by construction. The resulting analysis framework enables the exploration of the broader design space of refinement type inference algorithms. Existing algorithms such as Liquid type inference [Rondon et al. 2008] represent specific points in this design space.

To demonstrate the versatility of our framework, we have implemented it in a verification tool targeting a subset of OCaml. We have evaluated this tool for a range of new parameter instantiations and compared it against existing verification tools for functional programs. Our evaluation shows that the tool compares favorably against the state of the art. In particular, for the higher-order programs over integers and lists in our benchmark suite, the tool improves over existing tools both in terms of precision (more benchmarks solved) as well as robustness (no analysis timeouts).

Additional details, including proofs, are available in a companion technical report [Pavlinovic et al. 2020].

2 MOTIVATION

To motivate our work, we provide an overview of common approaches to inferring refinement types and discuss their limitations.

Refinement Type Inference. Consider the Fibonacci function defined in OCaml:

```
let rec fib x = if x >= 2 then fib (x - 1) + fib (x - 2) else 1
```

The typical refinement type inference algorithm works as follows. First, the analysis performs a standard Hindley-Milner type inference to infer the basic shape of the refinement type for every subexpression of the program. For instance, the inferred type for the function `fib` is $\mathsf{int} \to \mathsf{int}$. For every function type $\tau_1 \to \tau_2$, where τ_1 is a base type such as int, the analysis next introduces a fresh dependency variable x which stands for the function parameter, $x : \tau_1 \to \tau_2$. The scope of x is the result type τ_2, i.e., type refinements inferred for τ_2 can express dependencies on the input value by referring to x. Further, every base type τ is replaced by a refinement type, $\{\nu\colon \tau \mid \phi(\nu, \vec{x})\}$, with a placeholder refinement predicate $\phi(\nu, \vec{x})$ that expresses a relation between the members ν of τ and the other variables $\vec{x}$ in scope of the type. For example, the augmented type for function `fib` is

$$x : \{\nu\colon \mathsf{int} \mid \phi_1(\nu)\} \to \{\nu\colon \mathsf{int} \mid \phi_2(\nu, x)\} \ .$$

The algorithm then derives, either explicitly or implicitly, a system of Horn clauses modeling the subtyping constraints imposed on the refinement predicates by the program *data flow*. For example, the body of `fib` induces the following Horn clauses over the refinement predicates in `fib`'s type:

$$\phi_1(x) \land x \geq 2 \land \nu = x - 1 \Rightarrow \phi_1(\nu) \quad (1) \qquad \phi_1(x) \land x \geq 2 \land \nu = x - 2 \Rightarrow \phi_1(\nu) \quad (2)$$

$$\phi_1(x) \land x \geq 2 \land \phi_2(\nu_1, x - 1) \land \phi_2(\nu_2, x - 2) \land \nu = \nu_1 + \nu_2 \Rightarrow \phi_2(\nu, x) \quad (3)$$

$$\phi_1(x) \land x < 2 \land \nu = 1 \Rightarrow \phi_2(\nu, x) \quad (4) \qquad 0 \leq \nu \Rightarrow \phi_1(\nu) \quad (5)$$

Clauses (1) and (2) model the data flow from `x` to the two recursive calls in the *then* branch of the conditional. Clauses (3) and (4) capture the constraints on the result value returned by `fib` in the *then* and *else* branch. Clause (5) captures an assumption that we make about the *external calls* to `fib`, namely, that these calls always pass non negative values. We note that the inference algorithm performs a whole program analysis. Hence, when one analyzes a program fragment or individual function as in this case, one has to specify explicitly any assumptions made about the context.

The analysis then solves the obtained Horn clauses to derive the refinement predicates ϕ_i. For instance, Liquid type inference uses monomial predicate abstraction for this purpose. That is, the analysis assumes a given set of atomic predicates $Q = \{p_1(\nu, \vec{x}_1), \ldots, p_n(\nu, \vec{x}_n)\}$, which are either provided by the programmer or derived from the program using heuristics, and then infers an assignment for each ϕ_i to a conjunction over Q such that all Horn clauses are valid. This can be done effectively and efficiently using the Houdini algorithm [Flanagan and Leino 2001; Lahiri and Qadeer 2009]. For example, if we choose $Q = \{0 \leq \nu, 0 > \nu, \nu < 2, \nu \geq 2\}$, then the final type inferred for function `fib` will be: $x : \{\nu\colon \mathsf{int} \mid 0 \leq \nu\} \to \{\nu\colon \mathsf{int} \mid 0 \leq \nu\}$. The meaning of this type is tied to the current program, or in this case the assumptions made about the context of the program fragment being analyzed. In particular, due to the assumption expressed in Clause (5) the analysis does not infer a more general type that would leave the input parameter `x` of `fib` unconstrained.

Challenges in Inferring Precise Refinement Types. Now, suppose that we want to verify that function `fib` is increasing, which can be done by inferring a refinement predicate $\phi_2(\nu, x)$ for the return type of `fib` that implies $x \leq \nu$. Note that $x \leq \nu$ itself is not inductive for the system of Horn clauses derived above because clause (3) does not hold for $x = 2$, $v_1 = 1$, and $v_2 = 0$. However, if we strengthen $\phi_2(\nu, x)$ to $x \leq \nu \land 1 \leq \nu$, then it is inductive.

One issue with using predicate abstraction for inferring type refinements is that the analysis needs to guess in advance which auxiliary predicates will be needed, here $1 \leq \nu$. Existing tools based on this approach, such as DSolve [Rondon et al. 2008] and Liquid Haskell [Vazou et al.

```
1 let apply f x = f x and g y = 2 * y and h y = -2 * y
2 let main z =
3   let v = if 0 <= z then (apply_i g)_j z else (apply_k h)_ℓ z in
4   assert (0 <= v)
```

Program 1

2018a], use heuristics for this purpose. However, these heuristics tend to be brittle. In fact, both tools fail to verify that `fib` is increasing if the predicate $1 \leq \nu$ is not explicitly provided by the user. Other tools such as R_Type [Champion et al. 2018a] are based on more complex analyses that use counterexamples to inductiveness to automatically infer the necessary auxiliary predicates. However, these tools no longer guarantee that the analysis terminates. Instead, our approach enables the use of expressive numerical abstract domains such as polyhedra to infer sufficiently precise refinement types in practice, without giving up on termination of the analysis or requiring user annotations (see § 8).

If the goal is to improve precision, one may of course ask why it is necessary to develop a new refinement type inference analysis from scratch. Is it not sufficient to improve the deployed Horn clause solvers, e.g. by using better abstract domains? Unfortunately, the answer is "no" [Unno et al. 2013]. The derived Horn clause system already signifies an abstraction of the program's semantics and, in general, entails an inherent loss of precision for the overall analysis.

To motivate this issue, consider Program 1. You may ignore the program location labels i, j, k, ℓ for now. Suppose that the goal is to verify that the `assert` statement in the last line is safe. The templates for the refinement types of the top-level functions are as follows:

$$\texttt{apply} :: (y : \{\nu\colon \mathsf{int} \mid \phi_1(\nu)\} \to \{\nu\colon \mathsf{int} \mid \phi_2(\nu, y)\}) \to x : \{\nu\colon \mathsf{int} \mid \phi_3(\nu)\} \to \{\nu\colon \mathsf{int} \mid \phi_4(\nu, x)\}$$
$$\texttt{g} :: y : \{\nu\colon \mathsf{int} \mid \phi_5(\nu)\} \to \{\nu\colon \mathsf{int} \mid \phi_6(\nu, y)\} \quad \texttt{h} :: y : \{\nu\colon \mathsf{int} \mid \phi_7(\nu)\} \to \{\nu\colon \mathsf{int} \mid \phi_8(\nu, y)\}$$

Moreover, the key Horn clauses are:

$$\begin{array}{lll}
0 \leq z \wedge \nu = z \Rightarrow \phi_3(\nu) & \phi_5(y) \wedge \nu = 2y \Rightarrow \phi_6(\nu, y) & \phi_3(x) \Rightarrow \phi_1(\nu) \\
0 \leq z \wedge \phi_1(\nu) \Rightarrow \phi_5(\nu) & 0 \leq z \wedge \phi_6(\nu, y) \Rightarrow \phi_2(\nu, y) & \phi_2(\nu, x) \Rightarrow \phi_4(\nu, x) \\
0 > z \wedge \nu = z \Rightarrow \phi_3(\nu) & \phi_7(y) \wedge \nu = -(2y) \Rightarrow \phi_8(\nu, y) & \\
0 > z \wedge \phi_1(\nu) \Rightarrow \phi_7(\nu) & 0 > z \wedge \phi_8(\nu, y) \Rightarrow \phi_2(\nu, y) &
\end{array}$$

Note that the least solution of these Horn clauses satisfies $\phi_1(\nu) = \phi_3(\nu) = \phi_5(\nu) = \phi_7(\nu) = \mathsf{true}$ and $\phi_2(\nu, x) = \phi_4(\nu, x) = (\nu = 2x \vee \nu = -2x)$. Specifically, $\phi_4(\nu, x)$ is too weak to conclude that the two calls to `apply` on line 3 always return positive integers. Hence, any analysis based on deriving a solution to this Horn clause system will fail to infer refinement predicates that are sufficiently strong to entail the safety of the assertion in `main`. The problem is that the generated Horn clauses do not distinguish between the two functions `g` and `h` that `apply` is called with. All existing refinement type inference tools that follow this approach of generating a context-insensitive Horn clause abstraction of the program therefore fail to verify Program 1.

To obtain a better understanding where existing refinement type inference algorithms lose precision, we take a fresh look at this problem through the lens of abstract interpretation.

3 PRELIMINARIES

We introduce a few notations and basic definitions that we use throughout the paper.

Notation. We often use meta-level **let** $x = t_1$ **in** t_0 and conditional **if** t_0 **then** t_1 **else** t_2 constructs in mathematical definitions. We compress consecutive let bindings **let** $x_1 = t_1$ **in** $\ldots$ **let** $x_n = t_n$ **in** t_0 as **let** $x_1 = t_1$; $\ldots$; $x_n = t_n$ **in** t_0. We use capital lambda notation ($\Lambda x.\ \ldots$) for defining mathematical functions. For a function $f : X \to Y$, $x \in X$, and $y \in Y$, we write $f[x \mapsto y]$ to denote a function that maps x to y and otherwise agrees with f on every element of $X \setminus \{x\}$. We use the notation $f.x\colon y$ instead of $f[x \mapsto y]$ if f is an environment mapping variables x to their bindings y. For a set X we denote its powerset by $\wp(X)$. For a relation $R \subseteq X \times Y$ over sets X, Y and a natural number $n > 0$, we use $\dot{R}_n$ to refer to the point-wise lifting of R to a relation on n-tuples $X^n \times Y^n$. That is $\langle (x_1, \ldots, x_n), (y_1, \ldots, y_n) \rangle \in \dot{R}_n$ iff $(x_i, y_i) \in R$ for all $1 \leq i \leq n$. Similarly, for any nonempty set Z we denote by $\dot{R}_Z$ the point-wise lifting of R to a relation over $(Z \to X) \times (Z \to Y)$. More precisely, if $f_1 : Z \to X$ and $f_2 : Z \to Y$, then $(f_1, f_2) \in \dot{R}_Z$ iff $\forall z \in Z.\, (f_1(z), f_2(z)) \in R$. Typically, we drop the subscripts from these lifted relations when they are clear from the context.

For sets X, Y and a function $d : X \to \wp(Y)$, we use the notation $\Pi x \in X.d(x)$ to refer to the set $\{\, f : X \to Y \mid \forall x \in X.\, f(x) \in d(x) \,\}$ of all dependent functions with respect to d. Similarly, for given sets X and Y we use the notation $\Sigma x \in X.d(x)$ to refer to the set $\{\, \langle x, y \rangle : X \times Y \mid y \in d(x) \,\}$ of all dependent pairs with respect to d. We use the operators π_1 and π_2 to project to the first, resp., second component of a pair.

Abstract interpretation. A partially ordered set (poset) is a pair $(L, \sqsubseteq)$ consisting of a set L and a binary relation $\sqsubseteq$ on L that is reflexive, transitive, and antisymmetric. Let $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$ be two posets. We say that two functions $\alpha \in L_1 \to L_2$ and $\gamma \in L_2 \to L_1$ form a *Galois connection* iff

$$\forall x \in L_1, \forall y \in L_2.\ \alpha(x) \sqsubseteq_2 y \iff x \sqsubseteq_1 \gamma(y)\ .$$

We call L_1 the *concrete* domain and L_2 the *abstract* domain of the Galois connection. Similarly, α is called *abstraction* function (or left adjoint) and γ *concretization* function (or right adjoint). Intuitively, $\alpha(x)$ is the most precise approximation of $x \in L_1$ in L_2 while $\gamma(y)$ is the least precise element of L_1 that can be approximated by $y \in L_2$.

A *complete lattice* is a tuple $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ where $(L, \sqsubseteq)$ is a poset such that for any $X \subseteq L$, the least upper bound $\sqcup X$ (join) and greatest lower bound $\sqcap X$ (meet) with respect to $\sqsubseteq$ exist. In particular, we have $\bot = \sqcap L$ and $\top = \sqcup L$. We often identify a complete lattice with its carrier set L.

Let $(L_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1)$ and $(L_2, \sqsubseteq_2, \bot_2, \top_2, \sqcup_2, \sqcap_2)$ be two complete lattices and let (α, γ) be a Galois connection between L_1 and L_2. Each of these functions uniquely determines the other:

$$\alpha(x) = \sqcap_2\{y \in L_2 \mid x \sqsubseteq_1 \gamma(y)\} \qquad\qquad \gamma(y) = \sqcup_1\{x \in L_1 \mid \alpha(x) \sqsubseteq_2 y\}$$

Also, α is a *complete join-morphism* $\forall S \subseteq L_1.\ \alpha(\sqcup_1 S) = \sqcup_2\{\alpha(x) \mid x \in S\}, \alpha(\bot_1) = \bot_2$ and γ is a *complete meet-morphism* $\forall S \subseteq L_2.\ \gamma(\sqcap_2 S) = \sqcap_1\{\gamma(x) \mid x \in S\}, \gamma(\top_2) = \top_1$. A similar result holds in the other direction: if α is a complete join-morphism and γ is defined as above, then (α, γ) is a Galois connection between L_1 and L_2. Likewise, if γ is a complete meet-morphism and α is defined as above, then (α, γ) is a Galois connection [Cousot and Cousot 1979].

4 PARAMETRIC DATA FLOW REFINEMENT TYPES

We now introduce our parametric data flow refinement type system. The purpose of this section is primarily to build intuition. The remainder of the paper will then formally construct the type system as an abstract interpretation of our new data flow semantics. As a reward, we will also obtain a parametric algorithm for inferring data flow refinement types that is sound by construction.

Language. Our formal presentation considers a simple untyped lambda calculus:

$$e \in \mathit{Exp} ::= c \mid x \mid e_1\, e_2 \mid \lambda x.\, e$$

The language supports constants $c \in \mathit{Cons}$ (e.g. integers and Booleans), variables $x \in \mathit{Var}$, lambda abstractions, and function applications. An expression e is *closed* if all using occurrences of variables within e are bound in lambda abstractions $\lambda x.\, e'$. A *program* is a closed expression.

Let e be an expression. Each subexpression of e is uniquely annotated with a location drawn from the set Loc. We denote locations by ℓ, i, j and use subscript notation to indicate the location identifying a (sub)expression as in $(e_i\, e_j)_\ell$ and $(\lambda x.e_i)_\ell$. The location annotations are omitted whenever possible to avoid notational clutter. Variables are also locations, i.e. $\mathit{Var} \subseteq \mathit{Loc}$.

In our example programs we often use let constructs. Note that these can be expressed using lambda abstraction and function application as usual:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \stackrel{\text{def}}{=} (\lambda x.\, e_2)\, e_1 \ .$$

Types. Our data flow refinement type system takes two parameters: (1) a finite set of *abstract stacks* $\hat{\mathcal{S}}$, and (2) a (possibly infinite) complete lattice of *basic refinement types* $\langle \mathcal{R}^{\mathsf{t}}, \sqsubseteq^b, \bot^b, \top^b, \sqcup^b, \sqcap^b \rangle$. We will discuss the purpose of abstract stacks in a moment. A basic refinement type $b \in \mathcal{R}^{\mathsf{t}}$ comes equipped with an implicit scope $X \subseteq \mathit{Var}$. Intuitively, b represents a relation between primitive constant values (e.g. integers) and other values bound to the variables in X. The partial order $\sqsubseteq^b$ is an abstraction of subset inclusion on such relations. We will make this intuition formally precise later. For $X \subseteq \mathit{Var}$, we denote by $\mathcal{R}^{\mathsf{t}}_X$ the set of all basic refinement types with scope X.

Example 4.1. Let $\phi(X)$ stand for a convex linear constraint over (integer) variables in $X \cup \{\nu\}$. Then define the set of basic refinement types

$$\mathcal{R}^{\mathsf{lia}}_X ::= \bot^b \mid \top^b \mid \{\nu : \mathsf{int} \mid \phi(X)\}$$

An example of a basic type in $\mathcal{R}^{\mathsf{lia}}$ with scope $\{x\}$ is $\{\nu : \mathsf{int} \mid x \le \nu \wedge 1 \le \nu\}$. The order $\sqsubseteq^b$ on $\mathcal{R}^{\mathsf{lia}}$ is obtained by lifting the entailment ordering on linear constraints to $\mathcal{R}^{\mathsf{lia}}$ in the expected way. If we identify linear constraints up to equivalence, then $\sqsubseteq^b$ induces a complete lattice on $\mathcal{R}^{\mathsf{lia}}$.

The basic refinement types are extended to data flow refinement types as follows:

$$t \in \mathcal{V}^{\mathsf{t}}_X ::= \bot^{\mathsf{t}} \mid \top^{\mathsf{t}} \mid b \mid x : \boldsymbol{t} \qquad b \in \mathcal{R}^{\mathsf{t}}_X \qquad x : \boldsymbol{t} \in \mathcal{T}^{\mathsf{t}}_X \stackrel{\text{def}}{=} \Sigma x \in \mathit{Var} \setminus X.\, \hat{\mathcal{S}} \to \mathcal{V}^{\mathsf{t}}_X \times \mathcal{V}^{\mathsf{t}}_{X \cup \{x\}}$$

A data flow refinement type $t \in \mathcal{V}^{\mathsf{t}}_X$ also has an implicit scope X. We denote by $\mathcal{V}^{\mathsf{t}} = \bigcup_{X \subseteq \mathit{Var}} \mathcal{V}^{\mathsf{t}}_X$ the set of all such types for all scopes. There are four kinds of types. First, the type $\bot^{\mathsf{t}}$ should be interpreted as *unreachable* or *nontermination* and the type $\top^{\mathsf{t}}$ stands for a *type error*. We introduce these types explicitly so that we can later endow $\mathcal{V}^{\mathsf{t}}$ with a partial order to form a complete lattice. In addition to $\bot^{\mathsf{t}}$ and $\top^{\mathsf{t}}$, we have basic refinement types b and (dependent) function types $x : \boldsymbol{t}$. The latter are pairs consisting of a dependency variable $x \in \mathit{Var} \setminus X$ and a *type table* $\boldsymbol{t}$ that maps abstract stack $\hat{S} \in \hat{\mathcal{S}}$ to pairs of types $\boldsymbol{t}(\hat{S}) = \langle t_i, t_o \rangle$. That is, $x : \boldsymbol{t}$ can be understood as capturing a separate dependent function type $x : t_i \to t_o$ per abstract stack $\hat{S}$. Abstract stacks represent abstractions of concrete call stacks, enabling function types to case split on different calling contexts of the represented functions. In this sense, function types resemble intersection types with *ad hoc* polymorphism on contexts. Note that the scope of the output type t_o includes the dependency variable x, thus enabling the output type to capture input/output dependencies.

Let $t = x : \boldsymbol{t}$ be a function type and $\hat{S} \in \hat{\mathcal{S}}$ such that $\boldsymbol{t}(\hat{S}) = \langle t_i, t_o \rangle$ for some t_i and t_o. We denote t_i by $\boldsymbol{t}(\hat{S})_{\mathsf{in}}$ and t_o by $\boldsymbol{t}(\hat{S})_{\mathsf{out}}$. We say that t has been called at $\hat{S}$, denoted $\hat{S} \in t$, if $\boldsymbol{t}(\hat{S})_{\mathsf{in}}$ is not $\bot^{\mathsf{t}}$. We denote by $\boldsymbol{t}_\bot$ the *empty type table* that maps every abstract stack to the pair $\langle \bot^{\mathsf{t}}, \bot^{\mathsf{t}} \rangle$ and write $[\hat{S} \triangleleft t_i \to t_o]$ as a short hand for the *singleton type table* $\boldsymbol{t}_\bot[\hat{S} \mapsto \langle t_i, t_o \rangle]$. We extend this notation to tables obtained by explicit enumeration of table entries. Finally, we denote by $t|_{\hat{S}}$ the function type $x : [\hat{S} \triangleleft t_i \to t_o]$ obtained from t by restricting it to the singleton table for $\hat{S}$.

Example 4.2. Consider again the set $\mathcal{R}^{\mathsf{lia}}$ from Example 4.1. In what follows, we write int for $\{\nu : \mathsf{int} \mid \mathit{true}\}$ and $\{\nu \mid \phi(X)\}$ for $\{\nu : \mathsf{int} \mid \phi(X)\}$. Let further $\hat{\mathcal{S}}^0 = \{\epsilon\}$ be a trivial set of abstract stacks. We then instantiate $\mathcal{V}^{\mathsf{t}}$ with $\mathcal{R}^{\mathsf{lia}}$ and $\hat{\mathcal{S}}^0$. Since tables only have a single entry, we use the more familiar notation $x : t_i \to t_o$ for function types, instead of $x : [\epsilon \triangleleft t_i \to t_o]$. We can then represent the type of `fib` in § 2 by the data flow refinement type

$$x : \{\nu \mid 0 \leq \nu\} \to \{\nu \mid x \leq \nu \wedge 1 \leq \nu\} \ .$$

The most precise type that we can infer for function `apply` in Program 1 is

$$f : (y : \mathsf{int} \to \mathsf{int}) \to x : \mathsf{int} \to \mathsf{int} \ .$$

Here, the variables f, x, and y refer to the corresponding parameters of the function `apply`, respectively, the functions `g` and `h` that are passed to parameter f of `apply`. Now, define the set of abstract stacks $\hat{\mathcal{S}}^1 = \mathit{Loc} \cup \{\epsilon\}$. Intuitively, we can use the elements of $\hat{\mathcal{S}}^1$ to distinguish table entries in function types based on the program locations where the represented functions are called. Instantiating our set of types $\mathcal{V}^{\mathsf{t}}$ with $\mathcal{R}^{\mathsf{lia}}$ and $\hat{\mathcal{S}}^1$, we can infer a more precise type for function `apply` in Program 1:

$$\begin{aligned} f : [i \triangleleft y : [j \triangleleft \{\nu \mid 0 \leq \nu\} \to \{\nu \mid \nu = 2y\}] \to x : [j \triangleleft \{\nu \mid 0 \leq \nu\} \to \{\nu \mid \nu = 2x\}], \\ k \triangleleft y : [\ell \triangleleft \{\nu \mid 0 > \nu\} \to \{\nu \mid \nu = -2y\}] \to x : [\ell \triangleleft \{\nu \mid 0 > \nu\} \to \{\nu \mid \nu = -2x\}] \ . \end{aligned}$$

Note that the type provides sufficient information to distinguish between the calls to `apply` with functions `g` and `h` at call sites i and k, respectively. This information is sufficient to guarantee the correctness of the assertion on line 4.

Typing environments and operations on types. Before we can define the typing rules, we first need to introduce a few operations for constructing and manipulating types. We here only provide the intuition for these operations through examples. In § 7 we will then explain how to define these operations in terms of a few simple primitive operations provided by the domain $\mathcal{R}^{\mathsf{t}}$.

First, the (basic) refinement type abstracting a single constant $c \in \mathit{Cons}$ is denoted by $[\nu{=}c]^{\mathsf{t}}$. For instance, for our basic type domain $\mathcal{R}^{\mathsf{lia}}$ from Example 4.1 and an integer constant $c \in \mathbb{Z}$, we define $[\nu{=}c]^{\mathsf{t}} = \{\nu \mid \nu = c\}$. Next, given a type t over scope X and a type t' over scope $X' \subseteq X \setminus \{x\}$, we denote by $t[x \leftarrow t']$ the type obtained from t by strengthening the relation to x with the information provided by type t'. Returning to Example 4.1, for two basic types $b = \{\nu \mid \phi(X)\}$ and $b' = \{\nu \mid \phi'(X')\}$, we have $b[x \leftarrow b'] = \{\nu \mid \phi(X) \wedge \phi'(X')[x/\nu]\}$. Finally, for a variable $x \in X$ and $t \in \mathcal{V}^{\mathsf{t}}_X$, we assume an operation $t[\nu{=}x]^{\mathsf{t}}$ that strengthens t by enforcing equality between the value bound to x and the value represented by t. For instance, for a base type $b = \{\nu \mid \phi(X)\}$ from Example 4.1 we have $b[\nu{=}x]^{\mathsf{t}} = \{\nu \mid \phi(X) \wedge \nu = x\}$.

A *typing environment* for scope X is a function $\Gamma^{\mathsf{t}} \in (\Pi x \in X.\, \mathcal{V}^{\mathsf{t}}_{X \setminus \{x\}})$. We lift $t[x \leftarrow t']$ to an operation $t[\Gamma^{\mathsf{t}}]$ that strengthens t with the constraints on variables in X imposed by the types bound to these variables in Γ^{t}.

We additionally assume that abstract stacks $\hat{\mathcal{S}}$ come equipped with an *abstract concatenation operation* $\hat{:} : \mathit{Loc} \times \hat{\mathcal{S}} \to \hat{\mathcal{S}}$ that prepends a call site location i onto an abstract stack $\hat{S}$, denoted $i \hat{:} \hat{S}$. For instance, consider again the sets of abstract stacks $\hat{\mathcal{S}}^0$ and $\hat{\mathcal{S}}^1$ introduced in Example 4.2. We define $\hat{:}$ on $\hat{S} \in \hat{\mathcal{S}}^0$ as $i \hat{:} \hat{S} = \epsilon$ and on $\hat{S} \in \hat{\mathcal{S}}^1$ we define it as $i \hat{:} \hat{S} = i$. The general specification of $\hat{:}$ is given in § 6.2.

Typing rules. Typing judgements take the form $\Gamma^{\mathsf{t}}, \hat{S} \vdash e : t$ and rely on the *subtype relation* $t <: t'$ defined in Fig.1. The rule S-BOT states that $\bot^{\mathsf{t}}$ is a subtype of all other types except $\top^{\mathsf{t}}$. Since $\top^{\mathsf{t}}$ denotes a type error, the rules must ensure that we do not have $t <: \top^{\mathsf{t}}$ for any type t. The rule

$$\textsc{s-bot}\ \frac{t \neq \top^{\mathrm{t}}}{\bot^{\mathrm{t}} <: t} \qquad \textsc{s-base}\ \frac{b_1 \sqsubseteq^b b_2}{b_1 <: b_2} \qquad \textsc{s-fun}\ \frac{\boldsymbol{t}_1(\hat{S}) = \langle t_{i1}, t_{o1}\rangle \qquad \boldsymbol{t}_2(\hat{S}) = \langle t_{i2}, t_{o2}\rangle \qquad t_{i2} <: t_{i1} \qquad t_{o1}[x \leftarrow t_{i2}] <: t_{o2}[x \leftarrow t_{i2}]}{x : \boldsymbol{t}_1 <: x : \boldsymbol{t}_2}\ \forall \hat{S} \in \hat{\mathcal{S}}$$

Fig. 1. Data flow refinement subtyping rules

$$\textsc{t-var}\ \frac{\Gamma^{\mathrm{t}}(x)[v{=}x]^{\mathrm{t}}[\Gamma^{\mathrm{t}}] <: t[v{=}x]^{\mathrm{t}}[\Gamma^{\mathrm{t}}]}{\Gamma^{\mathrm{t}}, \hat{S} \vdash x : t} \qquad \textsc{t-app}\ \frac{\Gamma^{\mathrm{t}}, \hat{S} \vdash e_i : t_i \qquad \Gamma^{\mathrm{t}}, \hat{S} \vdash e_j : t_j \qquad t_i <: x : [i \mathbin{\hat{\cdot}} \hat{S} \triangleleft t_j \rightarrow t]}{\Gamma^{\mathrm{t}}, \hat{S} \vdash e_i\, e_j : t}$$

$$\textsc{t-const}\ \frac{[v{=}c]^{\mathrm{t}}[\Gamma^{\mathrm{t}}] <: t}{\Gamma^{\mathrm{t}}, \hat{S} \vdash c : t} \qquad \textsc{t-abs}\ \frac{\Gamma^{\mathrm{t}}_i = \Gamma^{\mathrm{t}}.x : t_x \qquad \Gamma^{\mathrm{t}}_i, \hat{S}' \vdash e_i : t_i \qquad x : [\hat{S}' \triangleleft t_x \rightarrow t_i] <: t|_{\hat{S}'}}{\Gamma^{\mathrm{t}}, \hat{S} \vdash \lambda x.\, e_i : t}\ \forall \hat{S}' \in t$$

Fig. 2. Data flow refinement typing rules

S-BASE defines subtyping on basic refinement types, which simply defers to the partial order $\sqsubseteq^b$ on $\mathcal{R}^{\mathrm{t}}$. Finally, the rule S-FUN is reminiscent of the familiar contravariant subtyping rule for dependent function types, except that it quantifies over all entries in the type tables. In § 7.2, we will establish a formal correspondence between subtyping and the propagation of values along data flow paths.

The rules defining the typing relation $\Gamma^{\mathrm{t}}, \hat{S} \vdash e : t$ are shown in Fig. 2. We implicitly require that all free variables of e are in the scope of Γ^{t}. The rule T-CONST for typing constants requires that $[v{=}c]^{\mathrm{t}}$ is a subtype of the type t, after strengthening it with all constraints on the variables in scope obtained from Γ^{t}. That is, we *push* all environmental assumptions into the types. This way, subtyping can be defined without tracking explicit typing environments. We note that this formalization does not preclude an implementation of basic refinement types that tracks typing environments explicitly. The rule T-VAR for typing variable expressions x is similar to the rule T-CONST. That is, we require that the type $\Gamma^{\mathrm{t}}(x)$ bound to x is a subtype of t, modulo strengthening with the equality constraint $v = x$ and all environmental constraints. Note that we here strengthen both sides of the subtype relation, which is necessary for completeness of the rules, due to the bidirectional nature of subtyping for function types.

The rule T-APP for typing function applications $e_i\, e_j$ requires that the type t_i of e_i must be a subtype of the function type $x : [i \mathbin{\hat{\cdot}} \hat{S} : t_j \rightarrow t]$ where t_j is the type of the argument expression e_j and t is the result type of the function application. Note that the rule extends the abstract stack $\hat{S}$ with the call site location i identifying e_i. The subtype relation then forces t_i to have an appropriate entry for the abstract call stack $i \mathbin{\hat{\cdot}} \hat{S}$. The rule T-ABS for typing lambda abstraction is as usual, except that it universally quantifies over all abstract stacks $\hat{S}'$ at which t has been called. The side condition $\forall \hat{S}' \in t$ implicitly constrains t to be a function type.

5 DATA FLOW SEMANTICS

Our goal is to construct our type system from a concrete semantics of functional programs, following the usual calculational approach taken in abstract interpretation. This imposes some constraints on our development. In particular, given that the typing rules are defined by structural recursion over the syntax of the evaluated expression, the same should be true for the concrete semantics that

we take as the starting point of our construction. This requirement rules out standard operational semantics for higher-order programs (e.g. based on function closures) because they evaluate function bodies at call sites rather than definition sites, making these semantics nonstructural. A more natural choice is a standard denotational semantics (e.g. one that interprets lambda terms by mathematical functions). However, the problem with denotational semantics is that it is inherently compositional; functions are given meaning irrespective of the context in which they appear. However, as we have discussed in § 2, the function types inferred by algorithms such as Liquid types only consider the inputs to functions that are observed in the program under consideration. Denotational semantics are therefore ill-suited for capturing the program properties abstracted by these type systems.

Hence, we introduce a new *data flow refinement type semantics*. Like standard denotational semantics, it is fully structural in the program syntax but without being compositional. That is, it captures the intuition behind refinement type inference algorithms that view a function value as a *table* that records all inputs the function will receive *from this point onwards* as it continues to flow through the program.

5.1 Semantic Domains

We start with the semantic domains used for giving meaning to expressions $e \in Exp$:

$$
\begin{array}{lll}
n \in \mathcal{N} \stackrel{\text{def}}{=} \mathcal{N}_e \cup \mathcal{N}_x & \mathcal{N}_e \stackrel{\text{def}}{=} Loc \times \mathcal{E} & \mathcal{N}_x \stackrel{\text{def}}{=} Var \times \mathcal{E} \times \mathcal{S} \\
S \in \mathcal{S} \stackrel{\text{def}}{=} Loc^* & E \in \mathcal{E} \stackrel{\text{def}}{=} Var \rightharpoonup_{fin} \mathcal{N}_x & M \in \mathcal{M} \stackrel{\text{def}}{=} \mathcal{N} \to \mathcal{V} \\
v \in \mathcal{V} ::= \bot \mid \top \mid c \mid \boldsymbol{v} & \boldsymbol{v} \in \mathcal{T} \stackrel{\text{def}}{=} \mathcal{S} \to \mathcal{V} \times \mathcal{V} &
\end{array}
$$

Nodes, stacks, and environments. Every intermediate point of a program's execution is uniquely identified by an *(execution) node*, $n \in \mathcal{N}$, a concept which we adapt from [Jagannathan and Weeks 1995]. We distinguish *expression nodes* $\mathcal{N}_e$ and *variable nodes* $\mathcal{N}_x$. An expression node $\langle \ell, E \rangle$, denoted $\ell \diamond E$, captures the execution point where a subexpression e_ℓ is evaluated in the environment E. An environment E is a (finite) partial map binding variables to variable nodes. A variable node $\langle x, E, S \rangle$, denoted $x \diamond E \diamond S$, is created at each execution point where an argument value is bound to a formal parameter x of a function at a call site. Here, E is the environment at the point where the function is defined and S is the *call site stack*[1] of the variable node. Note that we define nodes and environments using mutual recursion where the base cases are defined using the empty environment ϵ. The call site stack captures the sequence of program locations of all pending function calls before this variable node was created. That is, intuitively, S can be thought of as recording the return addresses of these pending calls. We write $\ell \cdot S$ to denote the stack obtained from S by prepending ℓ. Call site stacks are used to uniquely identify each variable binding.

We explain the role of expression and variable nodes in more detail later. For any node n, we denote by $loc(n)$ the location of n and by $env(n)$ its environment. If n is a variable node, we denote its stack component by $stack(n)$. A pair $\langle e, E \rangle$ is called well-formed if $E(x)$ is defined for every variable x that occurs free in e.

Values and execution maps. Similar to our data flow refinement types, there are four kinds of (data flow) values $v \in \mathcal{V}$. First, every constant c is also a value. The value $\bot$ stands for non-termination or unreachability of a node, and the value $\top$ models execution errors. Functions are represented by *tables*. A table $\boldsymbol{v}$ maintains an input/output value pair $T(S) = \langle v_i, v_o \rangle$ for each call site stack S. We adapt similar notation for concrete function tables as we introduced for type tables in § 4. In particular, we denote by $\boldsymbol{v}_\bot$ the table that maps every call site stack to the pair $\langle \bot, \bot \rangle$.

[1] referred to as *contour* in [Jagannathan and Weeks 1995]

We say that a value v is safe, denoted $safe(v)$, if $\top$ does not occur anywhere in v, i.e., $v \neq \top$ and if $v \in \mathcal{T}$ then for all S, both $v(S)_{\mathsf{in}}$ and $v(S)_{\mathsf{out}}$ are safe.

The data flow semantics computes *execution maps* $M \in \mathcal{M}$, which map nodes to values. We write $M_\bot$ ($M_\top$) for the execution map that assigns $\bot$ ($\top$) to every node.

As a precursor to defining the data flow semantics, we define a *computational order* $\sqsubseteq$ on values. In this order, $\bot$ is the smallest element, $\top$ is the largest element, and tables are ordered recursively on the pairs of values for each call site:

$$v_1 \sqsubseteq v_2 \overset{\text{def}}{\iff} v_1 = \bot \ \vee \ v_2 = \top \ \vee \ (v_1, v_2 \in \mathit{Cons} \wedge v_1 = v_2) \ \vee \ (v_1, v_2 \in \mathcal{T} \wedge \forall S.\, v_1(S) \dot{\sqsubseteq} v_2(S))$$

Defining the error value as the largest element of the order is nonessential but simplifies the presentation. This definition ensures that the least upper bounds (lub) of arbitrary sets of values exist, denoted by the join operator $\sqcup$. In fact, $\sqsubseteq$ is a partial order that induces a complete lattice on values which can be lifted point-wise to execution maps.

Example 5.1. Let us provide intuition for the execution maps through an example. To this end, consider Program 2. The middle shows the corresponding expression in our simple language with each subexpression annotated with its unique location. E.g., the using occurrence of `x` on line 1 is annotated with location k.

The program's execution map is given on the right. We abbreviate call stacks occurring in table entries by the last location pushed onto the stack (e.g. writing just a instead of adh). This simplification preserves the uniqueness of call stacks for this specific program. We similarly denote a node just by its location if this already uniquely identifies the node. During the execution of Program 2, the lambda abstraction at location o is called twice whereas all other functions are called once. Due to the two calls to the function at o (which is later bound to `id`), the variable `x` is bound twice and the subexpression at location k is also evaluated two times. This is reflected in the execution map by entries for two distinct variable nodes associated with `x` and two distinct expression nodes associated with k. We use superscript notation to indicate the environment associated with these nodes. For instance, k^q is the expression node that records the value obtained from the subexpression at location k when it is evaluated in the environment binding `x` to the variable node x^q. In turn, x^q records the binding of `x` for the function call at location q.

The cumulative entry $\mathsf{u}, f, g, \mathsf{x}^q, k^q \mapsto 1$ in the execution map indicates that the result computed at nodes u, f, g, and k^q is 1, respectively, that x is bound to 1 for the call at location q. Some of the nodes are mapped to function values. The function `id` is represented by the table $[q \triangleleft 1 \rightarrow 1,\ a \triangleleft 2 \rightarrow 2]$ that stores input-output values for the two calls to `id` at q and a. The nodes corresponding to the two usages of `id` are also mapped to tables. However, these tables only have a single entry each. Intuitively, `id` takes two separate data flow paths in the program starting from its definition at o. For each node on these two paths, the associated table captures how `id` will be used at the nodes on any data flow path that continues from that point onward. The tables stored at nodes q and a thus only contain information about the input and output for the respective call site whereas the table stored at `id` captures both call sites because the node for the definition of `id` occurs on both paths.

Some additional examples involving higher-order functions and recursion can be found in the companion report [Pavlinovic et al. 2020, § A] .

5.2 Concrete Semantics

We define the data flow semantics of a higher-order program e formally as the least fixpoint of a concrete transformer, step, on execution maps.

Concrete Transformer. The idea is that we start with the map $M_\bot$ and then use step to consecutively update the map with new values as more and more nodes are reached during the execution

```
1  let id x = x_k in
2  let u = (id_q 1_f)_g in
3  (id_a 2_b)_c
```

$((\lambda \mathsf{id}.\,(\lambda \mathsf{u}.\,(\mathsf{id}_a\ 2_b)_c)_d\ (\mathsf{id}_q\ 1_f)_g)_h\ (\lambda x.\,x_k)_o)_p$

$h \mapsto [h \triangleleft [q \triangleleft 1 \to 1,\ a \triangleleft 2 \to 2] \to 2]$
$\mathsf{id}, o \mapsto [q \triangleleft 1 \to 1,\ a \triangleleft 2 \to 2] \quad d \mapsto [d \triangleleft 1 \to 2]$
$q \mapsto [q \triangleleft 1 \to 1] \quad a \mapsto [a \triangleleft 2 \to 2]$
$\mathsf{u}, f, g, \mathsf{x}^q, k^q \mapsto 1 \quad b, c, d, h, \mathsf{x}^a, k^a, p \mapsto 2$

Program 2. The right side shows the program's execution map

(0) $\_ \mapsto \bot$ (1) $h \mapsto [h \triangleleft v_\bot \to \bot]$, $o \mapsto v_\bot$ (2) $\mathsf{id} \mapsto v_\bot$, $d \mapsto v_\bot$ (3) $q \mapsto [q \triangleleft 1 \to \bot]$, $f \mapsto 1$

(4) $\mathsf{id} \mapsto [q \triangleleft 1 \to \bot]$, $h \mapsto [h \triangleleft [q \triangleleft 1 \to \bot] \to \bot]$, $o \mapsto [q \triangleleft 1 \to \bot]$ (5) $\mathsf{x}^q \mapsto 1$ (6) $k^q \mapsto 1$, $o \mapsto [q \triangleleft 1 \to 1]$, $h \mapsto [h \triangleleft [q \triangleleft 1 \to 1] \to \bot]$ (7) $\mathsf{id} \mapsto [q \triangleleft 1 \to 1]$

(8) $q \mapsto [q \triangleleft 1 \to 1]$, $g \mapsto 1$, $d \mapsto [d \triangleleft 1 \to \bot]$ (9) $\mathsf{u} \mapsto 1$, $b \mapsto 2$, $a \mapsto [a \triangleleft 2 \to \bot]$ (10) $\mathsf{id} \mapsto [q \triangleleft 1 \to 1, a \triangleleft 2 \to \bot]$, $h \mapsto [h \triangleleft [q \triangleleft 1 \to 1, a \triangleleft 2 \to \bot] \to \bot]$, $o \mapsto [q \triangleleft 1 \to 1, a \triangleleft 2 \to \bot]$ (11) $\mathsf{x}^a \mapsto 2$

(12) $k^a \mapsto 2$, $o \mapsto [q \triangleleft 1 \to 1, a \triangleleft 2 \to 2]$, $h \mapsto [h \triangleleft [\ldots, a \triangleleft 2 \to 2] \to \bot]$ (13) $\mathsf{id} \mapsto [q \triangleleft 1 \to 1, a \triangleleft 2 \to 2]$ (14) $c, g, p \mapsto 2$, $a \mapsto [a \triangleleft 2 \to 2]$, $d \mapsto [d \triangleleft 1 \to 2]$, $h \mapsto [h \triangleleft [\ldots] \to 2]$

Program 3. Fixpoint iterates of the data flow semantics for Program 2

of e. The signature of step is as follows:

$$\mathsf{step} : \mathit{Exp} \to \mathcal{E} \times \mathcal{S} \to \mathcal{M} \to \mathcal{V} \times \mathcal{M}$$

It takes an expression e_ℓ, an environment E, and a stack S, and returns a transformer $\mathsf{step}[\![e_\ell]\!](E,S) : \mathcal{M} \to \mathcal{V} \times \mathcal{M}$ on execution maps. Given an input map M, the transformer $\mathsf{step}[\![e_\ell]\!](E,S)$ returns the new value v' computed at node $\ell \diamond E$ together with the updated execution map M'. That is, we always have $M'(\ell \diamond E) = v'$. We could have defined the transformer so that it returns only the updated map M', but returning v' in addition yields a more concise definition: observe that $\mathcal{M} \to \mathcal{V} \times \mathcal{M}$ is the type of the computation of a state monad [Wadler 1990]. We exploit this observation and define step using monadic composition of primitive state transformers. This allows us to hide the stateful nature of the definition and make it easier to see the connection to the type system later.

The primitive state transformers and composition operations are defined in Fig. 3. For instance, the transformer $!n$ reads the value at node n in the current execution map M and returns that value together with the unchanged map. Similarly, the transformer $n := v$ updates the entry at node n in the current map M by taking the join of the current value at n with v, returning the obtained new value v' and the updated map. We compress a sequence of update operations $n_1 := v_1, \ldots, n_n := v_n$, by using the shorter notation $n_1, \ldots, n_n := v_1, \ldots, v_n$ to reduce clutter. We point out that the result of this sequenced update operation is the result of the last update $n_n := v_n$.

The operation $\mathbf{bind}(F, G)$ defines the usual composition of stateful computations F and G in the state monad, where $F \in \mathcal{M} \to \alpha \times \mathcal{M}$ and $G \in \alpha \to \mathcal{M} \to \beta \times \mathcal{M}$ for some α and β. Note that the composition is short-circuiting in the case where the intermediate value u produced by F is $\top$ (i.e. an error occurred). We use Haskell-style monad comprehension syntax for applications of **bind** at the meta-level. That is, we write $\mathbf{do}\ x \leftarrow F; G$ for $\mathbf{bind}(F, \Lambda x.\, G)$. We similarly write $\mathbf{do}\ x \leftarrow F\ \mathbf{if}\ P; G$ for $\mathbf{bind}(F, \Lambda x.\, \mathbf{if}\ P\ \mathbf{then}\ G\ \mathbf{else\ return}\ \bot)$ and we shorten $\mathbf{do}\ x \leftarrow F; G$ to just $\mathbf{do}\ F; G$ in cases

where x does not occur free in G. Moreover, we write $\mathbf{do}\, x_1 \leftarrow F_1;\ \ldots;\ x_n \leftarrow F_n;\ G$ for the comprehension sequence $\mathbf{do}\, x_1 \leftarrow F_1; (\ldots; (\mathbf{do}\, x_n \leftarrow F_n; G)\ldots)$. We also freely mix the monad comprehension syntax with standard let binding syntax and omit the semicolons whenever this causes no confusion.

The definition of $\mathsf{step}[\![e]\!](E, S)$ is given in Fig. 4 using induction over the structure of e. As we discussed earlier, the structural definition of the transformer enables an easier formal connection to the data flow refinement typing rules. Note that in the definition we implicitly assume that $\langle e, E\rangle$ is well-formed. We discuss the cases of the definition one at a time using Program 2 as our running example. Figure 3 shows the fixpoint iterates of $\mathsf{step}[\![e]\!](\epsilon, \epsilon)$ starting from the execution map $M_\perp$ where e is Program 2 and ϵ refers to both the empty environment and empty stack. For each iterate (i), we only show the entries in the execution map that change in that iteration. We will refer to this figure throughout our discussion below.

Constant $e = c^\ell$. Here, we simply set the current node n to the join of its current value $M(n)$ and the value c. For example, in Fig. 3, when execution reaches the subexpression at location f in iteration (2), the corresponding entry for the node n identified by f is updated to $M(n) \sqcup 1 = \perp \sqcup 1 = 1$.

Variable $e = x_\ell$. This case implements the data flow propagation between the variable node n_x binding x and the current expression node n where x is used. This is realized using the *propagation function* $\ltimes$. Let $v_x = \Gamma(x) = M(n_x)$ and $v = M(n)$ be the current values stored at the two nodes in M. The function $\ltimes$ takes these values as input and propagates information between them, returning two new values v_x' and v' which are then stored back into M. The propagation function is defined in Fig. 5 and works as follows. If v_x is a constant or the error value and v is still $\perp$, then we simply propagate v_x forward, replacing v and leaving v_x unchanged. The interesting cases are when we propagate information between tables. The idea is that inputs in a table v flow backward to v_x whereas outputs for these inputs flow forward from v_x to v. For example, consider the evaluation of the occurrence of variable `id` at location a in step (10) of Fig. 3. Here, the expression node n is identified by a and the variable node n_x by `id`. Moreover, we have $v = [a \triangleleft 2 \to \perp]$ from step (9) and $v_x = [q \triangleleft 1 \to 1]$ from step (7). We then obtain

$$v_x \ltimes^{\mathsf{t}} v = [q \triangleleft 1 \to 1] \ltimes^{\mathsf{t}} [a \triangleleft 2 \to \perp] = \langle [q \triangleleft 1 \to 1, a \triangleleft 2 \to \perp], [a \triangleleft 2 \to \perp]\rangle$$

That is, the propagation causes the entry in the execution map for the node identified by `id` to be updated to $[q \triangleleft 1 \to 1, a \triangleleft 2 \to \perp]$.

In general, if v_x is a table but v is still $\perp$, we initialize v to the empty table $\boldsymbol{v}_\perp$ and leave v_x unchanged (because we have not yet accumulated any inputs in v). If both v_x and v are tables, $v_x \ltimes^{\mathsf{t}} v$ propagates inputs and outputs as described above by calling $\ltimes$ recursively for every call site $S \in v$. Note how the recursive call for the propagation of the inputs $v_{2i} \ltimes v_{1i}$ inverts the direction of the propagation. This has the affect that information about argument values propagate from the call site to the definition site of the function being called, as is the case for the input value 2 at call site a in our example above. Conversely, output values are propagated in the other direction from function definition sites to call sites. For example, in step (14) of Fig. 3, the occurrence of `id` at location a is evaluated again, but now we have $v_x = [q \triangleleft 1 \to 1, a \triangleleft 2 \to 2]$ from step (13) whereas v is as before. In this case, the propagation yields

$$v_x \ltimes^{\mathsf{t}} v = [q \triangleleft 1 \to 1, a \triangleleft 2 \to 2] \ltimes^{\mathsf{t}} [a \triangleleft 2 \to \perp] = \langle [q \triangleleft 1 \to 1, a \triangleleft 2 \to 2], [a \triangleleft 2 \to 2]\rangle$$

That is, the information about the output value 2 for the input value 2 has finally arrived at the call site a. As we shall see, the dataflow propagation between tables closely relates to contravariant subtyping of function types (cf. Lemma 7.5).

$$
\begin{aligned}
!n &\stackrel{\text{def}}{=} \Lambda M.\, \langle M(n), M\rangle \\
n := v &\stackrel{\text{def}}{=} \Lambda M.\, \textbf{let}\, v' = M(n) \sqcup v\, \textbf{in if}\, \mathit{safe}(v')\, \textbf{then}\, \langle v', M[n \mapsto v']\rangle\, \textbf{else}\, \langle \top, M_\top\rangle \\
\textbf{env}(E) &\stackrel{\text{def}}{=} \Lambda M.\, \langle M \circ E, M\rangle \\
\textbf{assert}(P) &\stackrel{\text{def}}{=} \Lambda M.\, \textbf{if}\, P\, \textbf{then}\, \langle \bot, M\rangle\, \textbf{else}\, \langle \top, M_\top\rangle \\
\textbf{for}\, v\, \textbf{do}\, F &\stackrel{\text{def}}{=} \Lambda M.\, \dot{\bigsqcup}_{S \in \boldsymbol{v}} F(S)(M) \\
\textbf{return}\, v &\stackrel{\text{def}}{=} \Lambda M.\, \langle v, M\rangle \\
\textbf{bind}(F, G) &\stackrel{\text{def}}{=} \Lambda M.\, \textbf{let}\, \langle u, M'\rangle = F(M)\, \textbf{in}\, \textbf{if}\, u = \top\, \textbf{then}\, \langle \top, M_\top\rangle\, \textbf{else}\, G(u)(M')
\end{aligned}
$$

Fig. 3. Primitive transformers on state monad for computations over execution maps

$$
\begin{aligned}
&\mathsf{step}[\![c_\ell]\!](E,S) \stackrel{\text{def}}{=} \\
&\quad \textbf{do}\, n = \ell\diamond E\,;\ v' \leftarrow n := c\,;\ \textbf{return}\, v' \\
&\mathsf{step}[\![x_\ell]\!](E,S) \stackrel{\text{def}}{=} \\
&\quad \textbf{do}\, n = \ell\diamond E\,;\, v \leftarrow !n\,;\ n_x = E(x)\,;\ \Gamma \leftarrow \textbf{env}(E) \\
&\qquad v' \leftarrow n_x,\ n := \Gamma(x) \ltimes v \\
&\quad \textbf{return}\, v' \\
&\mathsf{step}[\![(e_i\, e_j)_\ell]\!](E,S) \stackrel{\text{def}}{=} \\
&\quad \textbf{do}\, n = \ell\diamond E\,;\ n_i = i\diamond E\,;\ n_j = j\diamond E\,;\ v \leftarrow !n \\
&\qquad v_i \leftarrow \mathsf{step}[\![e_i]\!](E,S)\, \textbf{if}\, v_i \neq \bot \\
&\qquad \textbf{assert}(v_i \in \mathcal{T}) \\
&\qquad v_j \leftarrow \mathsf{step}[\![e_j]\!](E,S) \\
&\qquad v_i',\ [i \cdot S \triangleleft v_j' \to v'] = v_i \ltimes [i \cdot S \triangleleft v_j \to v] \\
&\qquad v'' \leftarrow n_i,\ n_j,\ n := v_i',\ v_j',\ v' \\
&\quad \textbf{return}\, v''
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{step}[\![(\lambda x.e_i)_\ell]\!](E,S) \stackrel{\text{def}}{=} \\
&\quad \textbf{do}\, n = \ell\diamond E\,;\ \boldsymbol{v} \leftarrow n := \boldsymbol{v}_\bot \\
&\qquad \boldsymbol{v}' \leftarrow \textbf{for}\, \boldsymbol{v}\, \textbf{do}\, \mathsf{body}(x, e_i, E, \boldsymbol{v}) \\
&\qquad \boldsymbol{v}'' \leftarrow n := \boldsymbol{v}' \\
&\quad \textbf{return}\, \boldsymbol{v}'' \\
&\mathsf{body}(x, e_i, E, \boldsymbol{v})(S') \stackrel{\text{def}}{=} \\
&\quad \textbf{do}\, n_x = x\diamond E\diamond S'\,;\ E_i = E.x : n_x\,;\ n_i = i\diamond E_i \\
&\qquad v_x \leftarrow !n_x \\
&\qquad v_i \leftarrow \mathsf{step}[\![e_i]\!](E_i, S') \\
&\qquad [S' \triangleleft v_x' \to v_i'], \boldsymbol{v}' = [S' \triangleleft v_x \to v_i] \ltimes \boldsymbol{v}|_{S'} \\
&\qquad n_x,\ n_i := v_x',\ v_i' \\
&\quad \textbf{return}\, \boldsymbol{v}'
\end{aligned}
$$

Fig. 4. Transformer for the concrete data flow semantics

$$
\begin{aligned}
&\boldsymbol{v}_1 \ltimes \boldsymbol{v}_2 \stackrel{\text{def}}{=} \\
&\quad \textbf{let}\, \boldsymbol{v}' = \Lambda S. \\
&\qquad \textbf{if}\, S \notin \boldsymbol{v}_2\, \textbf{then}\, \langle \boldsymbol{v}_1(S), \boldsymbol{v}_2(S)\rangle\, \textbf{else} \\
&\qquad \textbf{let}\, \langle v_{1i}, v_{1o}\rangle = \boldsymbol{v}_1(S);\, \langle v_{2i}, v_{2o}\rangle = \boldsymbol{v}_2(S) \\
&\qquad\quad \langle v_{2i}', v_{1i}'\rangle = v_{2i} \ltimes v_{1i};\, \langle v_{1o}', v_{2o}'\rangle = v_{1o} \ltimes v_{2o} \\
&\qquad \textbf{in}\, (\langle v_{1i}', v_{1o}'\rangle, \langle v_{2i}', v_{2o}'\rangle) \\
&\quad \textbf{in}\, \langle \Lambda S.\, \pi_1(\boldsymbol{v}'(S)), \Lambda S.\, \pi_2(\boldsymbol{v}'(S))\rangle
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{v} \ltimes \bot &\stackrel{\text{def}}{=} \langle \boldsymbol{v}, \boldsymbol{v}_\bot\rangle \\
\boldsymbol{v} \ltimes \top &\stackrel{\text{def}}{=} \langle \top, \top\rangle \\
v_1 \ltimes v_2 &\stackrel{\text{def}}{=} \langle v_1, v_1 \sqcup v_2\rangle \quad \textbf{(otherwise)}
\end{aligned}
$$

Fig. 5. Value propagation in the concrete data flow semantics

Function application $e = (e_i\, e_j)_\ell$. We first evaluate e_i to obtain the updated map and extract the new value v_i stored at the corresponding expression node n_i. If v_i is not a table, then we must be attempting an unsafe call, in which case the monadic operations return the error map $M_\top$. If v_i is a table, we continue evaluation of e_j obtaining the new value v_j at the associated expression node

n_j. We next need to propagate the information between this call site n_i and v_i. To this end, we use the return value v for the node n of e computed thus far and create a singleton table $[S' \triangleleft v_j \to v]$ where $S' = i \cdot S$ is the extended call stack that will be used for the evaluation of this function call. We then propagate between v_i and this table to obtain the new value v_i' and table $\boldsymbol{v}'$. Note that this propagation boils down to (1) the propagation between v_j and the input of v_i at S' and (2) the propagation between the output of v_i at S' and the return value v. The updated table $\boldsymbol{v}'$ hence contains the updated input and output values v_j' and v' at S'. All of these values are stored back into the execution map. Intuitively, v_i' contains the information that the corresponding function received an input coming from the call site identified by S'. This information is ultimately propagated back to the function definition where the call is actually evaluated.

As an example, consider the evaluation of the function application at location g in step (3) of Fig. 3. That is, node n is identified by g and we have $i = q$, $j = f$, $S = h$, and $v = \bot$. Moreover, we initially have $M(n_i) = \bot$ and $M(n_j) = \bot$. The recursive evaluation of e_i will propagate the information that `id` is a table to location q, i.e., the recursive call to step returns $v_i = \boldsymbol{v}_\bot$. Since v_i is a table, we proceed with the recursive evaluation of 1_f, after which we obtain $M(n_f) = v_j = 1$. Next we compute

$$v_i \ltimes [i \cdot S \triangleleft v_j \to v] = \boldsymbol{v}_\bot \ltimes [qh \triangleleft 1 \to \bot] = \langle [qh \triangleleft 1 \to \bot], [qh \triangleleft 1 \to \bot] \rangle$$

That is, $M(n_i)$ is updated to $v_i' = [qh \triangleleft 1 \to \bot]$[2]. Note that we still have $v' = \bot$ at this point. The final value 1 at location g is obtained later when this subexpression is reevaluated in step (8).

Lambda abstraction $e = (\lambda x.e_i)_\ell$. We first extract the table $\boldsymbol{v}$ computed for the function thus far. Then, for every call stack S' for which an input has already been back-propagated to $\boldsymbol{v}$, we analyze the body by evaluating $\mathsf{body}(x, e_i, E, \boldsymbol{v})(S')$, as follows. First, we create a variable node n_x that will store the input value that was recorded for the call site stack S'. Note that if this value is a table, indicating that the function being evaluated was called with another function as input, then any inputs to the argument function that will be seen while evaluating the body e_i will be back-propagated to the table stored at n_x and then further to the node generating the call stack S'. By incorporating S' into variable nodes, we guarantee that there is a unique node for each call.

We next evaluate the function body e_i with the input associated with stack S'. To this end, we extend the environment E to E_i by binding x to the variable node n_x. We then propagate the information between the values stored at the node n_i, i.e., the result of evaluating the body e_i, the nodes n_x for the bound variable, and the table $\boldsymbol{v}$. That is, we propagate information from (1) the input of $\boldsymbol{v}$ at S' and the value at node n_x, and (2) the value assigned to the function body under the updated environment E_i and $\boldsymbol{v}(S')_{\mathsf{out}}$, the output value at $\boldsymbol{v}(S')$. Finally, the updated tables $\boldsymbol{v}'$ for all call site stacks S' are joined together and stored back at node n.

As an example, consider the evaluation of the lambda abstraction at location o in step (5) of Fig. 3. Here, n is identified by o and we initially have $M(n) = \boldsymbol{v} = [qh \triangleleft 1 \to \bot]$. Thus, we evaluate a single call to body for $e_i = \mathsf{x}_k$ and $S' = qh$. In this call we initially have $M(n_x) = M(\mathsf{x}^q) = \bot$. Hence, the recursive evaluation of e_i does not yet have any effect and we still obtain $v_i = \bot$ at this point. However, the final propagation step in body yields:

$$[S' \triangleleft v_x \to v_i] \ltimes \boldsymbol{v}(S') = [qh \triangleleft \bot \to \bot] \ltimes [qh \triangleleft 1 \to \bot] = \langle [qh \triangleleft 1 \to \bot], [qh \triangleleft 1 \to \bot] \rangle$$

and we then update $M(n_x)$ to $v_x' = 1$. In step (6), when the lambda abstraction at o is once more evaluated, we now have initially $M(n_x) = M(\mathsf{x}^q) = 1$ and the recursive evaluation of $e_i = \mathsf{x}_k$ will update the entry for k^q in the execution map to 1. Thus, we also obtain $v_i = 1$. The final propagation

[2]Recall that in Fig. 3 we abbreviate the call stack qh by just q.

step in body now yields:

$$[S' \triangleleft v_x \longrightarrow v_i] \ltimes \boldsymbol{v}(S') = [qh \triangleleft 1 \longrightarrow 1] \ltimes [qh \triangleleft 1 \longrightarrow \bot] = \langle [qh \triangleleft 1 \longrightarrow 1], [qh \triangleleft 1 \longrightarrow 1] \rangle$$

which will cause the execution map entry for n (identified by o) to be updated to $[qh \triangleleft 1 \longrightarrow 1]$.

Observe that the evaluation of a lambda abstraction for a new input value always takes at least two iterations of step. This can be optimized by performing the propagation in body both before and after the recursive evaluation of e_i. However, we omit this optimization here for the sake of maintaining a closer resemblance to the typing rule for lambda abstractions.

Lemma 5.2. *The function $\ltimes$ is monotone and increasing.*

Lemma 5.3. *For every $e \in Exp$, $E \in \mathcal{E}$, and $S \in \mathcal{S}$ such that $\langle e, E \rangle$ is well-formed, $\mathsf{step}[\![e]\!](E, S)$ is monotone and increasing.*

We define the semantics $\mathrm{S}[\![e]\!]$ of a program e as the least fixpoint of step over the complete lattice of execution maps:

$$\mathrm{S}[\![e]\!] \stackrel{\text{def}}{=} \mathrm{lfp}^{\dot{\sqsubseteq}}_{M_\bot} \Lambda M.\, \textbf{let}\ \langle \_, M' \rangle = \mathsf{step}[\![e]\!](\epsilon, \epsilon)(M)\ \textbf{in}\ M'$$

Lemma 5.3 guarantees that $\mathrm{S}[\![e]\!]$ is well-defined by Knaster-Tarski. We note that the above semantics does not precisely model certain non-terminating programs where tables grow infinitely deep. The semantics of such programs is simply $M_\top$. A more precise semantics can be defined using step-indexing [Appel and McAllester 2001]. However, we omit this for ease of presentation and note that our semantics is adequate for capturing refinement type systems and inference algorithms *à la* Liquid types, which do not support infinitely nested function types.

Properties and collecting semantics. As we shall see, data flow refinement types abstract programs by *properties* $P \in \mathcal{P}$, which are sets of execution maps: $\mathcal{P} \stackrel{\text{def}}{=} \wp(\mathcal{M})$. Properties form the concrete lattice of our abstract interpretation and are ordered by subset inclusion. That is, the concrete semantics of our abstract interpretation is the *collecting semantics* $\mathrm{C} : Exp \longrightarrow \mathcal{P}$ that maps programs to properties: $\mathrm{C}[\![e]\!] \stackrel{\text{def}}{=} \{\mathrm{S}[\![e]\!]\}$. An example of a property is *safety*: let P_{safe} consist of all execution maps that map all nodes to safe values. Then a program e is *safe* if $\mathrm{C}[\![e]\!] \subseteq P_{\mathsf{safe}}$.

6 INTERMEDIATE ABSTRACT SEMANTICS

We next present two abstract semantics that represent crucial abstraction steps when calculating our data flow refinement type system from the concrete data flow semantics. Our formal exposition focuses mostly on the aspects of these semantics that are instrumental in understanding the loss of precision introduced by these intermediate abstractions. Other technical details can be found in the designated appendices of the companion report [Pavlinovic et al. 2020] .

6.1 Relational Semantics

A critical abstraction step performed by the type system is to conflate the information in tables that are propagated back from different call sites to function definitions. That is, a pair of input/output values that has been collected from one call site will also be considered as a possible input/output pair at other call sites to which that function flows. To circumvent a catastrophic loss of precision caused by this abstraction step, we first introduce an intermediate semantics that explicitly captures the relational dependencies between input and output values of functions. We refer to this semantics as the *relational data flow semantics*.

Abstract domains. The definitions of nodes, stacks, and environments in the relational semantics are the same as in the concrete semantics. Similar to data flow refinement types, the relational abstractions of values, constants, and tables are defined with respect to *scopes* $X \subseteq_{fin} Var$. The

variables in scopes are used to track how a value computed at a specific node in the execution map relates to the other values bound in the current environment. We also use scopes to capture how function output values depend on the input values, similar to the way input-output relations are captured in dependent function types. The scope of a node n, denoted X_n, is the domain of n's environment: $X_n \stackrel{\text{def}}{=} \mathsf{dom}(env(n))$. The new semantic domains are defined as follows:

$$u \in \mathcal{V}^{\mathsf{r}}_X ::= \ \bot^{\mathsf{r}} \mid \top^{\mathsf{r}} \mid r \mid x : \boldsymbol{u} \qquad d \in \mathcal{D}^{\mathsf{r}}_X \stackrel{\text{def}}{=} X \cup \{v\} \to \mathit{Cons} \cup \{\mathsf{F}\}$$
$$r \in \mathcal{R}^{\mathsf{r}}_X \stackrel{\text{def}}{=} \wp(\mathcal{D}^{\mathsf{r}}_X) \qquad x : \boldsymbol{u} \in \mathcal{T}^{\mathsf{r}}_X \stackrel{\text{def}}{=} \Sigma x \in (\mathit{Var} \setminus X).\, \mathcal{S} \to \mathcal{V}^{\mathsf{r}}_X \times \mathcal{V}^{\mathsf{r}}_{X \cup \{x\}}$$
$$M^{\mathsf{r}} \in \mathcal{M}^{\mathsf{r}} \stackrel{\text{def}}{=} \Pi n \in \mathcal{N}.\, \mathcal{V}^{\mathsf{r}}_{X_n}$$

Relational values $u \in \mathcal{V}^{\mathsf{r}}_X$ model how concrete values, stored at some node, depend on the concrete values of nodes in the current scope X. The relational value $\bot^{\mathsf{r}}$ again models nontermination or unreachability and imposes no constraints on the values in its scope. The relational value $\top^{\mathsf{r}}$ models every possible concrete value, including $\top$. Concrete constant values are abstracted by relations r, which are sets of *dependency vectors* d. A dependency vector tracks the dependency between a constant value associated with the special symbol v, and the values bound to the variables in scope X. Here, we assume that v is never contained in X.

We only track dependencies between constant values precisely: if a node in the scope stores a table, we abstract it by the symbol F which stands for an arbitrary concrete table. We assume F to be distinct from all other constants Cons. We also require that for all $d \in r$, $d(v) \in \mathit{Cons}$. Relational tables $\langle x, \boldsymbol{u} \rangle$, denoted $x : \boldsymbol{u}$, are defined analogously to concrete tables except that $\boldsymbol{u}$ now maps call site stacks S to pairs of relational values $\langle u_i, u_o \rangle$. As for dependent function types, we add a dependency variable x to every relational table to track input-output dependencies. Note that we consider relational tables to be equal up to α-renaming of dependency variables. Relational execution maps M^{r} assign each node n a relational value with scope X_n.

The relational semantics of a program is the relational execution map obtained as the least fixpoint of a Galois abstraction of the concrete transformer step. The formal definition is mostly straightforward, so we delegate it to the companion report [Pavlinovic et al. 2020, § B] .

Example 6.1. The relational execution map obtained for Program 2 is as follows (we only show the entries for the nodes `id`, q, and a):

$$\texttt{id} \mapsto x : [q \triangleleft \{(v\colon 1)\} \to \{(x\colon 1, v\colon 1)\},\ a \triangleleft \{(v\colon 2)\} \to \{(x\colon 2, v\colon 2)\}]$$
$$q \mapsto x : [q \triangleleft \{(\texttt{id}\colon \mathsf{F}, v\colon 1)\} \to \{(\texttt{id}\colon \mathsf{F}, x\colon 1, v\colon 1)\}]$$
$$a \mapsto x : [a \triangleleft \{(\texttt{id}\colon \mathsf{F}, \texttt{u}\colon 1, v\colon 2)\} \to \{(\texttt{id}\colon \mathsf{F}, \texttt{u}\colon 1, x\colon 2, v\colon 2)\}]$$

Each concrete value v in the concrete execution map shown to the right of Program 2 is abstracted by a relational value that relates v with the values bound to the variables that are in the scope of the node where v was observed. Consider the entry for node `id`. As expected, this entry is a table that has seen inputs at call site stacks identified with q and a. The actual input stored for call site stack q is now a relation consisting of the single row $(v : 1)$, and similarly for call site stack a. As the node `id` has no other variables in its scope, these input relations are simply representing the original concrete input values 1 and 2. We associate these original values with the symbol v. The output relation for q consists of the single row $(x : 1, v : 1)$, stating that for the input value 1 (associated with x), the output value is also 1 (associated with v). Observe how we use x to capture explicitly the dependencies between the input and output values.

The entry in the relational execution map for the node identified by q is similar to the one for `id`, except that the relation also has an additional entry $\texttt{id} : \mathsf{F}$. This is because `id` is in the scope of q. The value of `id` in the execution map is a table, which the relational values abstract by the symbolic value F. That is, the relational semantics only tracks relational dependencies between primitive

values precisely whereas function values are abstracted by F. The relational table stored at node a is similar, except that we now also have the variable u which is bound to value 1. As in the concrete execution map, the relational tables for q and a contain fewer entries than the table stored at id.

Abstraction. We formalize the meaning of relational execution maps in terms of a Galois connection between $\mathcal{M}^{\mathsf{r}}$ and the complete lattice of sets of concrete execution maps $\wp(\mathcal{M})$. The details of this construction and the resulting abstract transformer of the relational semantics can be found in [Pavlinovic et al. 2020, § B] . We here focus on the key idea of the abstraction by formalizing the intuitive meaning of relational values given above. Our formalization uses a family of concretization functions $\gamma^{\mathsf{r}}_X : \mathcal{V}^{\mathsf{r}}_X \to \wp((X \to \mathcal{V}) \times \mathcal{V})$, parameterized by scopes X, that map relational values to sets of pairs $\langle \Gamma, v\rangle$ where Γ maps the variables in scope X to values in $\mathcal{V}$:

$$\gamma^{\mathsf{r}}_X(\bot^{\mathsf{r}}) \stackrel{\text{def}}{=} (X \to \mathcal{V}) \times \{\bot\} \qquad \gamma^{\mathsf{r}}_X(\top^{\mathsf{r}}) \stackrel{\text{def}}{=} (X \to \mathcal{V}) \times \mathcal{V}$$
$$\gamma^{\mathsf{r}}_X(r) \stackrel{\text{def}}{=} \{\langle \Gamma, c\rangle \mid d{\in}r \wedge d(v) = c \wedge \forall x{\in}X.\ \Gamma(x) \in \gamma^{\mathsf{d}}(d(x))\} \cup \gamma^{\mathsf{r}}_X(\bot^{\mathsf{r}})$$
$$\gamma^{\mathsf{r}}_X(x : \boldsymbol{u}) \stackrel{\text{def}}{=} \{\langle \Gamma, \boldsymbol{v}\rangle \mid \forall S.\ \boldsymbol{v}(S) = \langle v_i, v_o\rangle \wedge \boldsymbol{u}(S) = \langle u_i, u_o\rangle \wedge \langle \Gamma, v_i\rangle \in \gamma^{\mathsf{r}}_X(u_i) \wedge \langle \Gamma[x \mapsto v_i], v_o\rangle \in \gamma^{\mathsf{r}}_{X\cup\{x\}}(u_o)\} \cup \gamma^{\mathsf{r}}_X(\bot^{\mathsf{r}})$$

Here, the function γ^{d}, which we use to give meaning to dependency relations, is defined by $\gamma^{\mathsf{d}}(c) = \{c\}$ and $\gamma^{\mathsf{d}}(\mathsf{F}) = \mathcal{T}$. The meaning of relational execution maps is then given by the function

$$\gamma^{\mathsf{r}}(M^{\mathsf{r}}) \stackrel{\text{def}}{=} \{M \mid \forall n \in \mathcal{N}.\ \langle \Gamma, v\rangle \in \gamma^{\mathsf{r}}_{X_n}(M^{\mathsf{r}}(n)) \wedge \Gamma = M \circ env(n) \wedge v = M(n)\}\ .$$

6.2 Collapsed Semantics

We now describe a key abstraction step in the construction of our data flow refinement type semantics. We formalize this step in terms of a *collapsed (relational data flow) semantics*, which collapses function tables to a bounded number of entries while controlling how much stack information is being lost, thereby allowing for different notions of call-site context sensitivity.

Abstract domains. The collapsed semantics is parameterized by a finite set of *abstract stacks* $\hat{S} \in \hat{\mathcal{S}}$, a *stack abstraction* function $\rho : \mathcal{S} \to \hat{\mathcal{S}}$, and an *abstract stack concatenation* operation $\hat{:} : Loc \times \hat{\mathcal{S}} \to \hat{\mathcal{S}}$. Stack abstraction must be homomorphic with respect to concatenation: for all $\ell \in Loc$ and $S \in \mathcal{S}$, $\rho(\ell \cdot S) = \ell \mathbin{\hat{:}} \rho(S)$.

Abstract stacks induce sets of abstract nodes and abstract environments following the same structure as in the concrete semantics

$$\hat{n} \in \hat{\mathcal{N}} \stackrel{\text{def}}{=} \hat{\mathcal{N}}_e \cup \hat{\mathcal{N}}_x \qquad \hat{\mathcal{N}}_e \stackrel{\text{def}}{=} Loc \times \hat{\mathcal{E}} \qquad \hat{\mathcal{N}}_x \stackrel{\text{def}}{=} Var \times \hat{\mathcal{E}} \times \hat{\mathcal{S}} \qquad \hat{E} \in \hat{\mathcal{E}} \stackrel{\text{def}}{=} Var \rightharpoonup_{fin} \hat{\mathcal{N}}_x$$

We lift ρ from stacks to nodes and environments in the expected way. In particular, for variable nodes $n_x = \ell{\diamond}E{\diamond}S$, we recursively define $\rho(n_x) \stackrel{\text{def}}{=} \ell{\diamond}(\rho \circ E){\diamond}\rho(S)$. Analogous to concrete nodes, we define the scope of an abstract node as $X_{\hat{n}} = \mathsf{dom}(env(\hat{n}))$.

The definition of values and execution maps remains largely unchanged. In particular, dependency vectors and relational values are inherited from the relational semantics. Only the definition of tables changes, which now range over abstract stacks rather than concrete stacks:

$$\hat{u} \in \mathcal{V}^{\hat{\mathsf{r}}}_X ::= \bot^{\hat{\mathsf{r}}} \mid \top^{\hat{\mathsf{r}}} \mid r \mid x : \hat{\boldsymbol{u}} \qquad x : \hat{\boldsymbol{u}} \in \mathcal{T}^{\hat{\mathsf{r}}}_X \stackrel{\text{def}}{=} \Sigma x \in (Var \setminus X).\, \hat{\mathcal{S}} \to \mathcal{V}^{\hat{\mathsf{r}}}_X \times \mathcal{V}^{\hat{\mathsf{r}}}_{X\cup\{x\}}$$
$$M^{\hat{\mathsf{r}}} \in \mathcal{M}^{\hat{\mathsf{r}}} \stackrel{\text{def}}{=} \Pi \hat{n} \in \hat{\mathcal{N}}.\, \mathcal{V}^{\hat{\mathsf{r}}}_{X_{\hat{n}}}$$

Example 6.2. We use Program 2 again to provide intuition for the new semantics. To this end, we first define a family of sets of abstract stacks which we can use to instantiate the collapsed semantics. Let $k \in \mathbb{N}$ and define $\hat{\mathcal{S}}^k = Loc^k$ where Loc^k is the set of all sequences of locations of length at

most k. Moreover, for $\hat{S} \in \hat{\mathcal{S}}^k$ define $\ell \mathbin{\hat{:}} \hat{S} = (\ell \cdot \hat{S})[0,k]$ where $(\ell_1 \dots \ell_n)[0,k]$ is ϵ if $k = 0$, $\ell_1 \dots \ell_k$ if $0 < k < n$, and $\ell_1 \dots \ell_n$ otherwise. Note that this definition generalizes the definitions of $\hat{\mathcal{S}}^0$ and $\hat{\mathcal{S}}^1$ from Example 4.2. An abstract stack in $\hat{\mathcal{S}}^k$ only maintains the return locations of the k most recent pending calls, thus yielding a k-context-sensitive analysis. In particular, instantiating our collapsed semantics with $\hat{\mathcal{S}}^0$ yields a context-insensitive analysis. Applying this analysis to Program 2, we obtain the following *collapsed execution map*, which abstracts the relational execution map for this program shown in Example 6.1:

$$\begin{aligned}
\texttt{id} &\mapsto x\colon \{(v\colon 1), (v\colon 2)\} \to \{(x\colon 1, v\colon 1), (x\colon 2, v\colon 2)\}\\
q &\mapsto x\colon \{(\texttt{id}\colon \mathsf{F}, v\colon 1)\} \to \{(\texttt{id}\colon \mathsf{F}, x\colon 1, v\colon 1)\}\\
a &\mapsto x\colon \{(\texttt{id}\colon \mathsf{F}, \mathsf{u}\colon 1, v\colon 2)\} \to \{(\texttt{id}\colon \mathsf{F}, \mathsf{u}\colon 1, x\colon 2, v\colon 2)\}
\end{aligned}$$

Again, we only show some of the entries and for economy of notation, we omit the abstract stack ϵ in the singleton tables. Since the semantics does not maintain any stack information, the *collapsed* tables no longer track where functions are being called in the program. For instance, the entry for `id` indicates that `id` is a function called at some concrete call sites with inputs 1 and 2. While the precise call site stack information of `id` is no longer maintained, the symbolic variable x still captures the relational dependency between the input and output values for all the calls to `id`.

If we chose to maintain more information in $\hat{\mathcal{S}}$, the collapsed semantics is also more precise. For instance, a 1-context-sensitive analysis is obtained by instantiating the collapsed semantics with $\hat{\mathcal{S}}^1$. Analyzing Program 2 using this instantiation of the collapsed semantics yields a collapsed execution map that is isomorphic to the relational execution map shown in Example 6.1 (i.e., the analysis does not lose precision in this case).

Abstraction. Similar to the relational semantics, we formalize the meaning of the collapsed semantics in terms of a Galois connection between the complete lattices of relational execution maps $\mathcal{M}^{\mathsf{r}}$ and collapsed execution maps $\mathcal{M}^{\hat{\mathsf{r}}}$. Again, we only provide the definition of the right adjoint $\gamma^{\hat{\mathsf{r}}} : \mathcal{M}^{\hat{\mathsf{r}}} \to \mathcal{M}^{\mathsf{r}}$ here. Similar to the relational semantics, $\gamma^{\hat{\mathsf{r}}}$ is defined in terms of a family of concretization functions $\gamma^{\hat{\mathsf{r}}}_X : \mathcal{V}^{\hat{\mathsf{r}}}_X \to \mathcal{V}^{\mathsf{r}}_X$ mapping collapsed values to relational values:

$$\dot{\gamma}^{\hat{\mathsf{r}}}(M^{\hat{\mathsf{r}}}) \stackrel{\text{def}}{=} \Lambda n \in \mathcal{N}.\, (\gamma^{\hat{\mathsf{r}}}_{X_{\rho(n)}} \circ M^{\hat{\mathsf{r}}} \circ \rho)(n) \qquad \gamma^{\hat{\mathsf{r}}}_X(\bot^{\hat{\mathsf{r}}}) \stackrel{\text{def}}{=} \bot^{\mathsf{r}} \qquad \gamma^{\hat{\mathsf{r}}}_X(\top^{\hat{\mathsf{r}}}) \stackrel{\text{def}}{=} \top^{\mathsf{r}} \qquad \gamma^{\hat{\mathsf{r}}}_X(r) \stackrel{\text{def}}{=} r$$

$$\gamma^{\hat{\mathsf{r}}}_X(x : \hat{\boldsymbol{u}}) \stackrel{\text{def}}{=} x : \Lambda S \in \mathcal{S}.\, \textbf{let } \langle \hat{u}_i, \hat{u}_o \rangle = (\hat{\boldsymbol{u}} \circ \rho)(S) \textbf{ in } \langle \gamma^{\hat{\mathsf{r}}}_X(\hat{u}_i), \gamma^{\hat{\mathsf{r}}}_{X \cup \{x\}}(\hat{u}_o) \rangle$$

More details on the collapsed semantics including its abstract transformer are provided in the companion report [Pavlinovic et al. 2020, § C] .

7 PARAMETRIC DATA FLOW REFINEMENT TYPE SEMANTICS

At last, we obtain our parametric data flow refinement type semantics from the collapsed relational semantics by abstracting dependency relations over concrete constants by abstract relations drawn from some relational abstract domain. In § 7.2, we will then show that our data flow refinement type system introduced in § 4 is sound and complete with respect to this abstract semantics. Finally, in § 7.3 we obtain a generic type inference algorithm from our abstract semantics by using widening to enforce finite convergence of the fixpoint iteration sequence.

7.1 Type Semantics

Abstract domains. The abstract domains of our data flow refinement type semantics build on the set of types $\mathcal{V}^{\mathsf{t}}_X$ defined in § 4. Recall that $\mathcal{V}^{\mathsf{t}}_X$ is parametric with a set of abstract stacks $\hat{\mathcal{S}}$ and a complete lattice of basic refinement types $\langle \mathcal{R}^{\mathsf{t}}, \sqsubseteq^b, \bot^b, \top^b, \sqcup^b, \sqcap^b \rangle$, which can be viewed as a

union of sets $\mathcal{R}^{\mathsf{t}}_X$ for each scope X. We require that each $\mathcal{R}^{\mathsf{t}}_X$ forms a complete sublattice of $\mathcal{R}^{\mathsf{t}}$ and that there exists a family of Galois connections[3] $\langle \alpha^b_X, \gamma^b_X \rangle$ between $\mathcal{R}^{\mathsf{t}}_X$ and the complete lattice of dependency relations $\langle \mathcal{R}^{\mathsf{r}}_X, \subseteq, \emptyset, \mathcal{D}^{\mathsf{r}}_X, \cup, \cap \rangle$. For instance, for the domain $\mathcal{R}^{\mathsf{lia}}$ from Example 4.1, the concretization function γ^b_X is naturally obtained from the satisfaction relation for linear integer constraints.

We lift the partial order $\sqsubseteq^b$ on basic refinement types to a preorder $\sqsubseteq^{\mathsf{t}}$ on types as follows:

$$t_1 \sqsubseteq^{\mathsf{t}} t_2 \iff^{\text{def}} t_1 = \bot^{\mathsf{t}} \vee t_2 = \top^{\mathsf{t}} \vee (t_1, t_2 \in \mathcal{R}^{\mathsf{t}} \wedge t_1 \sqsubseteq^b t_2) \vee (t_1, t_2 \in \mathcal{T}^{\mathsf{t}} \wedge \forall \hat{S}.\ t_1(\hat{S}) \dot{\sqsubseteq}^{\mathsf{t}} t_2(\hat{S}))$$

By implicitly taking the quotient of types modulo α-renaming of dependency variables in function types we obtain a partial order that induces a complete lattice $\langle \mathcal{V}^{\mathsf{t}}_X, \sqsubseteq^{\mathsf{t}}, \bot^{\mathsf{t}}, \top^{\mathsf{t}}, \sqcup^{\mathsf{t}}, \sqcap^{\mathsf{t}} \rangle$. We lift this partial order point-wise to refinement type maps $\mathcal{M}^{\mathsf{t}} \stackrel{\text{def}}{=} \Pi \hat{n} \in \hat{\mathcal{N}}.\ \mathcal{V}^{\mathsf{t}}_{X_{\hat{n}}}$ and obtain a complete lattice $\langle \mathcal{M}^{\mathsf{t}}, \dot{\sqsubseteq}^{\mathsf{t}}, M^{\mathsf{t}}_{\bot}, M^{\mathsf{t}}_{\top}, \dot{\sqcup}^{\mathsf{t}}, \dot{\sqcap}^{\mathsf{t}} \rangle$.

Galois connection. The meaning of refinement types is given by a function $\gamma^{\mathsf{t}}_X : \mathcal{V}^{\mathsf{t}}_X \to \mathcal{V}^{\hat{\mathsf{r}}}_X$ that extends γ^b on basic refinement types. This function is then lifted to type maps as before:

$$\gamma^{\mathsf{t}}_X(\bot^{\mathsf{t}}) \stackrel{\text{def}}{=} \bot^{\hat{\mathsf{r}}} \qquad \gamma^{\mathsf{t}}_X(\top^{\mathsf{t}}) \stackrel{\text{def}}{=} \top^{\hat{\mathsf{r}}} \qquad \gamma^{\mathsf{t}}_X(x : \boldsymbol{t}) \stackrel{\text{def}}{=} x : \Lambda \hat{S}.\ \textbf{let}\ \langle t_i, t_o \rangle = \boldsymbol{t}(\hat{S})\ \textbf{in}\ \langle \gamma^{\mathsf{t}}_X(t_i), \gamma^{\mathsf{t}}_{X \cup \{x\}}(t_o) \rangle$$
$$\gamma^{\mathsf{t}}(M^{\mathsf{t}}) \stackrel{\text{def}}{=} \Lambda \hat{n}.\ (\gamma^{\mathsf{t}}_{X_{\hat{n}}} \circ M^{\mathsf{t}})(\hat{n})$$

Abstract domain operations. We briefly revisit the abstract domain operations on types introduced in § 4 and provide their formal specifications needed for the correctness of our data flow refinement type semantics.

We define these operations in terms of three simpler operations on basic refinement types. First, for $x, y \in X \cup \{\nu\}$ and $b \in \mathcal{R}^{\mathsf{t}}_X$, let $b[x = y]$ be an abstraction of the concrete operation that strengthens the dependency relations described by b with an equality constraint $x = y$. That is, we require $\gamma^b_X(b[x = y]) \supseteq \{d \in \gamma^b_X(b) \mid d(x) = d(y)\}$. Similarly, for $c \in \mathit{Cons} \cup \{\mathsf{F}\}$ we assume that $b[x = c]$ is an abstraction of the concrete operation that strengthens b with the equality $x = c$, i.e. we require $\gamma^b_X(b[x = c]) \supseteq \{d \in \gamma^b_X(b) \mid d(x) = c\}$. Lastly, we assume an *abstract variable substitution* operation, $b[x/\nu]$, which must be an abstraction of variable substitution on dependency relations: $\gamma^b_X(b[x/\nu]) \supseteq \{d[\nu \mapsto c, x \mapsto d(\nu)] \mid d \in \gamma^b_X(b), c \in \mathit{Cons}\}$. We lift these operations to general refinement types t in the expected way. For instance, we define

$$t[x = c] \stackrel{\text{def}}{=} \begin{cases} t[x = c] & \textbf{if}\ t \in \mathcal{R} \\ z : \Lambda \hat{S}.\ \langle \boldsymbol{t}(\hat{S})_{\mathsf{in}}[x = c], \boldsymbol{t}(\hat{S})_{\mathsf{out}}[x = c] \rangle & \textbf{if}\ t = z : \boldsymbol{t} \wedge x \neq \nu \\ t & \textbf{otherwise} \end{cases}$$

Note that in the second case of the definition, the fact that x is in the scope of t implies $x \neq z$.

We then define the function that yields the abstraction of a constant $c \in \mathit{Cons}$ as $[\nu = c]^{\mathsf{t}} \stackrel{\text{def}}{=} \top^b[\nu = c]$. The strengthening operation $t[x \leftarrow t']$ is defined recursively over the structure of types as follows:

$$t[x \leftarrow t'] \stackrel{\text{def}}{=} \begin{cases} \bot^{\mathsf{t}} & \textbf{if}\ t' = \bot^{\mathsf{t}} \\ t[x = \mathsf{F}] & \textbf{else if}\ t' \in \mathcal{T}^{\mathsf{t}} \\ t \sqcap^b t'[x/\nu] & \textbf{else if}\ t \in \mathcal{R} \\ z : \Lambda \hat{S}.\ \langle \boldsymbol{t}(\hat{S})_{\mathsf{in}}[x \leftarrow t'], \boldsymbol{t}(\hat{S})_{\mathsf{out}}[x \leftarrow t'] \rangle & \textbf{else if}\ t = z : \boldsymbol{t} \\ t & \textbf{otherwise} \end{cases}$$

[3] In fact, we can relax this condition and only require a concretization function, thus supporting abstract refinement domains such as polyhedra [Cousot and Halbwachs 1978].

$$
\begin{array}{l}
x : \boldsymbol{t}_1 \ltimes^{\mathsf{t}} x : \boldsymbol{t}_2 \stackrel{\text{def}}{=} \textbf{let}\, \boldsymbol{t} = \Lambda\hat{S}. \\
\quad \textbf{let}\, \langle t_{1i}, t_{1o}\rangle = \boldsymbol{t}_1(\hat{S});\ \langle t_{2i}, t_{2o}\rangle = \boldsymbol{t}_2(\hat{S}) \\
\quad \langle t'_{2i}, t'_{1i}\rangle = t_{2i} \ltimes^{\mathsf{t}} t_{1i} \\
\quad \langle t'_{1o}, t'_{2o}\rangle = t_{1o}[x \leftarrow t_{2i}] \ltimes^{\mathsf{t}} t_{2o}[x \leftarrow t_{2i}] \\
\quad \textbf{in}\, \langle\langle t'_{1i}, t_{1o} \sqcup^{\mathsf{t}} t'_{1o}\rangle, \langle t'_{2i}, t_{2o} \sqcup^{\mathsf{t}} t'_{2o}\rangle\rangle \\
\textbf{in}\, \langle x : \Lambda\hat{S}.\ \pi_1(\boldsymbol{t}(\hat{S})), x : \Lambda\hat{S}.\ \pi_2(\boldsymbol{t}(\hat{S}))\rangle
\end{array}
$$

$$
\begin{array}{l}
x : \boldsymbol{t} \ltimes^{\mathsf{t}} \bot^{\mathsf{t}} \stackrel{\text{def}}{=} \langle x : \boldsymbol{t}, x : \boldsymbol{t}_\bot\rangle \\
x : \boldsymbol{t} \ltimes^{\mathsf{t}} \top^{\mathsf{t}} \stackrel{\text{def}}{=} \langle \top^{\mathsf{t}}, \top^{\mathsf{t}}\rangle \\
t_1 \ltimes^{\mathsf{t}} t_2 \stackrel{\text{def}}{=} \langle t_1, t_1 \sqcup^{\mathsf{t}} t_2\rangle \\
\qquad \textbf{(otherwise)}
\end{array}
$$

Fig. 6. Abstract value propagation in the data flow refinement type semantics

$$
\begin{array}{l}
\mathsf{step}^{\mathsf{t}}[\![c_\ell]\!](\hat{E}, \hat{S}) \stackrel{\text{def}}{=} \\
\quad \textbf{do}\, \hat{n} = \ell\diamond\hat{E};\ \Gamma^{\mathsf{t}} \leftarrow \textbf{env}(\hat{E});\ t' \leftarrow \hat{n} := [\nu{=}c]^{\mathsf{t}}[\Gamma^{\mathsf{t}}] \\
\quad \textbf{return}\, t' \\
\\
\mathsf{step}^{\mathsf{t}}[\![x_\ell]\!](\hat{E}, \hat{S}) \stackrel{\text{def}}{=} \\
\quad \textbf{do}\, \hat{n} = \ell\diamond\hat{E};\ t \leftarrow !\hat{n};\ \hat{n}_x = \hat{E}(x);\ \Gamma^{\mathsf{t}} \leftarrow \textbf{env}(\hat{E}) \\
\qquad t' \leftarrow \hat{n}_x,\ \hat{n} := \Gamma^{\mathsf{t}}(x)[\nu{=}x]^{\mathsf{t}}[\Gamma^{\mathsf{t}}] \ltimes^{\mathsf{t}} t[\nu{=}x]^{\mathsf{t}}[\Gamma^{\mathsf{t}}] \\
\quad \textbf{return}\, t' \\
\\
\mathsf{step}^{\mathsf{t}}[\![(e_i\, e_j)_\ell]\!](\hat{E}, \hat{S}) \stackrel{\text{def}}{=} \\
\quad \textbf{do}\, \hat{n} = \ell\diamond\hat{E};\ \hat{n}_i = i\diamond\hat{E};\ \hat{n}_j = j\diamond\hat{E};\ t \leftarrow !\hat{n} \\
\qquad t_i \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_i]\!](\hat{E}, \hat{S});\ \textbf{assert}(t_i \in \mathcal{T}^{\mathsf{t}}) \\
\qquad t_j \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_j]\!](\hat{E}, \hat{S}) \\
\qquad t'_i,\ x : [i\,\hat{:}\,\hat{S} \triangleleft t'_j \rightarrow t'] = t_i \ltimes^{\mathsf{t}} x : [i\,\hat{:}\,\hat{S} \triangleleft t_j \rightarrow t] \\
\qquad t'' \leftarrow \hat{n}_i,\ \hat{n}_j,\ \hat{n} := t'_i,\ t'_j,\ t' \\
\quad \textbf{return}\, t''
\end{array}
$$

$$
\begin{array}{l}
\mathsf{step}^{\mathsf{t}}[\![(\lambda x.e_i)_\ell]\!](\hat{E}, \hat{S}) \stackrel{\text{def}}{=} \\
\quad \textbf{do}\, \hat{n} = \ell\diamond\hat{E};\ t \leftarrow \hat{n} := x : \boldsymbol{t}_\bot \\
\qquad t' \leftarrow \textbf{for}\, t\, \textbf{do}\, \mathsf{body}^{\mathsf{t}}(x, e_i, \hat{E}, t) \\
\qquad t'' \leftarrow \hat{n} := t' \\
\quad \textbf{return}\, t'' \\
\\
\mathsf{body}^{\mathsf{t}}(x, e_i, \hat{E}, t)(\hat{S}') \stackrel{\text{def}}{=} \\
\quad \textbf{do}\, \hat{n}_x = x\diamond\hat{E}\diamond\hat{S}';\ \hat{E}_i = \hat{E}.x : \hat{n}_x;\ \hat{n}_i = i\diamond\hat{E}_i \\
\qquad t_x \leftarrow !\hat{n}_x;\ t_i \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_i]\!](\hat{E}_i, \hat{S}') \\
\qquad x : [\hat{S}' \triangleleft t'_x \rightarrow t'_i], t' = \\
\qquad\quad x : [\hat{S}' \triangleleft t_x \rightarrow t_i] \ltimes^{\mathsf{t}} t|_{\hat{S}'} \\
\qquad \hat{n}_x,\ \hat{n}_i := t'_x,\ t'_i \\
\quad \textbf{return}\, t'
\end{array}
$$

Fig. 7. Abstract transformer for the data flow refinement type semantics

Finally, we lift $t[x \leftarrow t']$ to the operation $t[\Gamma^{\mathsf{t}}]$ that strengthens t with respect to a type environment Γ^{t} by defining $t[\Gamma^{\mathsf{t}}] \stackrel{\text{def}}{=} \bigsqcap^{\mathsf{t}}_{x \in \mathsf{dom}(\Gamma^{\mathsf{t}})} t[x \leftarrow \Gamma^{\mathsf{t}}(x)]$.

Abstract propagation and transformer. The propagation operation $\ltimes^{\mathsf{t}}$ on refinement types, shown in Fig. 6, is then obtained from $\ltimes$ in Fig. 5 by replacing all operations on concrete values with their counterparts on types. In a similar fashion, we obtain the new abstract transformer $\mathsf{step}^{\mathsf{t}}$ for the refinement type semantics from the concrete transformer step. We again use a state monad to hide the manipulation of type execution maps. The corresponding operations are variants of those used in the concrete transformer, which are summarized in Fig. 7. The abstract transformer closely resembles the concrete one. The only major differences are in the cases for constant values and variables. Here, we strengthen the computed types with the relational information about the variables in scope obtained from the current environment $\Gamma^{\mathsf{t}} = M^{\mathsf{t}} \circ \hat{E}$.

Abstract semantics. We identify abstract properties $\mathcal{P}^{\mathsf{t}}$ in the data flow refinement type semantics with type maps, $\mathcal{P}^{\mathsf{t}} \stackrel{\text{def}}{=} \mathcal{M}^{\mathsf{t}}$ and define $\gamma : \mathcal{P}^{\mathsf{t}} \rightarrow \mathcal{P}$, which maps abstract properties to concrete properties by $\gamma \stackrel{\text{def}}{=} \gamma^{\mathsf{r}} \circ \gamma^{\hat{\mathsf{r}}} \circ \gamma^{\mathsf{t}}$.

Lemma 7.1. *The function γ is a complete meet-morphism between $\mathcal{P}^{\mathsf{t}}$ and $\mathcal{P}$.*

It follows that γ induces a Galois connection between concrete and abstract properties. The data flow refinement semantics $\mathrm{C}^{\mathsf{t}}[\![\cdot]\!] : \mathit{Exp} \to \mathcal{P}^{\mathsf{t}}$ is then defined as the least fixpoint of $\mathsf{step}^{\mathsf{t}}$:

$$\mathrm{C}^{\mathsf{t}}[\![e]\!] \stackrel{\text{def}}{=} \mathrm{lfp}^{\dot{\sqsubseteq}^{\mathsf{t}}}_{M^{\mathsf{t}}_{\perp}} \Lambda M^{\mathsf{t}}.\ \textbf{let}\ \langle \_, M^{\mathsf{t}\prime}\rangle = \mathsf{step}^{\mathsf{t}}[\![e]\!](\epsilon,\epsilon)(M^{\mathsf{t}})\ \textbf{in}\ M^{\mathsf{t}\prime}$$

Theorem 7.2. *The refinement type semantics is sound, i.e.* $\mathrm{C}[\![e]\!] \subseteq \gamma(\mathrm{C}^{\mathsf{t}}[\![e]\!])$.

The soundness proof follows from the calculational design of our abstraction and the properties of the involved Galois connections.

We say that a type t is *safe* if it does not contain $\top^{\mathsf{t}}$, i.e. $t \neq \top^{\mathsf{t}}$ and if $t = x : \boldsymbol{t}$ then for all $\hat{S} \in \hat{\mathcal{S}}$, $\boldsymbol{t}(\hat{S})_{\mathsf{in}}$ and $\boldsymbol{t}(\hat{S})_{\mathsf{out}}$ are safe. Similarly, a type map M^{t} is safe if all its entries are safe. The next lemma states that safe type maps yield safe properties. It follows immediately from the definitions of the concretizations.

Lemma 7.3. *For all safe type maps* M^{t}, $\gamma(M^{\mathsf{t}}) \subseteq P_{\mathsf{safe}}$.

A direct corollary of this lemma and the soundness theorems for our abstract semantics is that any safe approximation of the refinement type semantics can be used to prove program safety.

Corollary 7.4. *For all programs* e *and safe type maps* M^{t}, *if* $\mathrm{C}^{\mathsf{t}}[\![e]\!] \dot{\sqsubseteq}^{\mathsf{t}} M^{\mathsf{t}}$, *then* e *is safe.*

7.2 Soundness and Completeness of Type System

It is worth to pause for a moment and appreciate the resemblance between the subtyping and typing rules introduced in § 4 on one hand, and the abstract propagation operator $\ltimes^{\mathsf{t}}$ and abstract transformer $\mathsf{step}^{\mathsf{t}}$ on the other hand. We now make this resemblance formally precise by showing that the type system exactly captures the safe fixpoints of the abstract transformer. This implies the soundness and completeness of our type system with respect to the abstract semantics.

We start by formally relating the subtype relation and type propagation. The following lemma states that subtyping precisely captures the safe fixpoints of type propagation.

Lemma 7.5. *For all* $t_1, t_2 \in \mathcal{V}^{\mathsf{t}}_X$, $t_1 <: t_2$ *iff* $\langle t_1, t_2\rangle = t_1 \ltimes^{\mathsf{t}} t_2$ *and* t_1, t_2 *are safe.*

We use this fact to show that any derivation of a typing judgment $\Gamma^{\mathsf{t}}, \hat{S} \vdash e : t$ represents a safe fixpoint of $\mathsf{step}^{\mathsf{t}}$ on e, and vice versa, for any safe fixpoint of $\mathsf{step}^{\mathsf{t}}$ on e, we can obtain a typing derivation. To state the soundness theorem we need one more definition: we say that a typing environment is *valid* if it does not map any variable to $\bot^{\mathsf{t}}$ or $\top^{\mathsf{t}}$.

Theorem 7.6 (Soundness). *Let* e *be an expression,* Γ^{t} *a valid typing environment,* $\hat{S}$ *an abstract stack, and* $t \in \mathcal{V}^{\mathsf{t}}$. *If* $\Gamma^{\mathsf{t}}, \hat{S} \vdash e : t$, *then there exist* $M^{\mathsf{t}}, \hat{E}$ *such that* M^{t} *is safe,* $\Gamma^{\mathsf{t}} = M^{\mathsf{t}} \circ \hat{E}$ *and* $\langle t, M^{\mathsf{t}}\rangle = \mathsf{step}^{\mathsf{t}}[\![e]\!](\hat{E}, \hat{S})(M^{\mathsf{t}})$.

Theorem 7.7 (Completeness). *Let* e *be an expression,* $\hat{E}$ *an environment,* $\hat{S} \in \hat{\mathcal{S}}$, M^{t} *a type map, and* $t \in \mathcal{V}^{\mathsf{t}}$. *If* $\mathsf{step}^{\mathsf{t}}[\![e]\!](\hat{E}, \hat{S})(M^{\mathsf{t}}) = \langle t, M^{\mathsf{t}}\rangle$ *and* M^{t} *is safe, then* $\Gamma^{\mathsf{t}}, \hat{S} \vdash e : t$ *where* $\Gamma^{\mathsf{t}} = M^{\mathsf{t}} \circ \hat{E}$.

7.3 Type Inference

The idea for the generic type inference algorithm is to iteratively compute $\mathrm{C}^{\mathsf{t}}[\![e]\!]$, which captures the most precise typing for e as we have established above. Unfortunately, there is no guarantee that the fixpoint iterates of $\mathsf{step}^{\mathsf{t}}$ converge towards $\mathrm{C}^{\mathsf{t}}[\![e]\!]$ in finitely many steps. The reasons are two-fold. First, the domain $\mathcal{R}^{\mathsf{t}}$ may not satisfy the *ascending chain condition* (i.e. it may have infinite height). To solve this first issue, we simply assume that $\mathcal{R}^{\mathsf{t}}$ comes equipped with a family of widening operators $\triangledown^{\mathsf{t}}_X : \mathcal{R}^{\mathsf{t}}_X \times \mathcal{R}^{\mathsf{t}}_X \to \mathcal{R}^{\mathsf{t}}_X$ for its scoped sublattices. Recall that a widening operator for a complete lattice $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap\rangle$ is a function $\triangledown : L \times L \to L$ such that: (1) $\triangledown$ is an

upper bound operator, i.e., for all $x, y \in L$, $x \sqcup y \sqsubseteq x \triangledown y$, and (2) for all infinite ascending chains $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ in L, the chain $y_0 \sqsubseteq y_1 \sqsubseteq \ldots$ eventually stabilizes, where $y_0 \stackrel{\text{def}}{=} x_0$ and $y_i \stackrel{\text{def}}{=} y_{i-1} \triangledown x_i$ for all $i > 0$ [Cousot and Cousot 1977].

The second issue is that there is in general no bound on the depth of function tables recorded in the fixpoint iterates. This phenomenon can be observed, e.g., in the following program[4]:

```
1 let rec hungry x = hungry and loop f = loop (f ()) in
2 loop hungry
```

To solve this second issue, we introduce a *shape widening* operator that enforces a bound on the depth of tables. The two widening operators will then be combined to yield a widening operator for $\mathcal{V}^{\mathsf{t}}_X$. In order to define shape widening, we first define the *shape* of a type using the function $\mathsf{sh} : \mathcal{V}^{\mathsf{t}}_X \to \mathcal{V}^{\mathsf{t}}_X$ defined as follows:

$$\mathsf{sh}(\bot^{\mathsf{t}}) \stackrel{\text{def}}{=} \bot^{\mathsf{t}} \quad \mathsf{sh}(\top^{\mathsf{t}}) \stackrel{\text{def}}{=} \top^{\mathsf{t}} \quad \mathsf{sh}(r^{\mathsf{t}}) \stackrel{\text{def}}{=} \bot^{\mathsf{t}} \quad \mathsf{sh}(x : \boldsymbol{t}) \stackrel{\text{def}}{=} x : \Lambda\hat{S}.\, \langle \mathsf{sh}(\boldsymbol{t}(\hat{S})_{\mathsf{in}}), \mathsf{sh}(\boldsymbol{t}(\hat{S})_{\mathsf{out}})\rangle$$

A shape widening operator is a function $\triangledown^{\mathsf{sh}}_X : \mathcal{V}^{\mathsf{t}}_X \times \mathcal{V}^{\mathsf{t}}_X \to \mathcal{V}^{\mathsf{t}}_X$ such that (1) $\triangledown^{\mathsf{sh}}_X$ is an upper bound operator and (2) for every infinite ascending chain $t_0 \sqsubseteq^{\mathsf{t}}_X t_1 \sqsubseteq^{\mathsf{t}}_X \ldots$, the chain $\mathsf{sh}(t'_0) \sqsubseteq^{\mathsf{t}}_X \mathsf{sh}(t'_1) \sqsubseteq^{\mathsf{t}}_X \ldots$ stabilizes, where $t'_0 \stackrel{\text{def}}{=} t_0$ and $t'_i \stackrel{\text{def}}{=} t'_{i-1} \triangledown^{\mathsf{sh}}_X t_i$ for $i > 0$.

Example 7.8. The occurs check performed when unifying type variables in type inference for Hindley-Milner-style type systems serves a similar purpose as shape widening. In fact, we can define a shape widening operator that mimics the occurs check. To this end, suppose that each dependency variable x is tagged with a finite set of pairs $\langle \ell, \hat{S}\rangle$. We denote this set by $tag(x)$. Moreover, when computing joins over function types $x : \boldsymbol{t}$ and $y : \boldsymbol{t}$, first α-rename x, respectively, y by some fresh z such that $tag(z) = tag(x) \cup tag(y)$. We proceed similarly when applying $\ltimes^{\mathsf{t}}$ to function types. Finally, assume that each fresh dependency variable x generated by $\mathsf{step}^{\mathsf{t}}$ for the function type at a call expression $e_i\, e_j$ has $tag(x) = \{\langle i, \hat{S}\rangle\}$ where $\hat{S}$ is the abstract call stack at this point. Then to obtain $t_1 \triangledown^{\mathsf{sh}}_X t_2$, first define $t = t_1 \sqcup^{\mathsf{t}}_X t_2$. If t contains two distinct bindings of dependency variables x and y such that $tag(x) = tag(y)$, define $t_1 \triangledown^{\mathsf{sh}}_X t_2 = \top^{\mathsf{t}}$ and otherwise $t_1 \triangledown^{\mathsf{sh}}_X t_2 = t$. Clearly this is a shape widening operator if we only consider the finitely many tag sets that can be constructed from the locations in the analyzed program.

In what follows, let $\triangledown^{\mathsf{sh}}_X$ be a shape widening operator. First, we lift the $\triangledown^{\mathsf{t}}_X$ on $\mathcal{R}^{\mathsf{t}}$ to an upper bound operator $\triangledown^{\mathsf{ra}}_X$ on $\mathcal{V}^{\mathsf{t}}_X$:

$$t \triangledown^{\mathsf{ra}}_X t' \stackrel{\text{def}}{=} \begin{cases} t \triangledown^{\mathsf{t}}_X t' & \textbf{if } t, t' \in \mathcal{R}^{\mathsf{t}} \\ x : \Lambda\hat{S}.\ \textbf{let } \langle t_i, t_o\rangle = \boldsymbol{t}(\hat{S}); \langle t'_i, t'_o\rangle = \boldsymbol{t}'(\hat{S}) \\ \qquad \textbf{in } \langle t_i \triangledown^{\mathsf{ra}}_X t'_i, t_o \triangledown^{\mathsf{ra}}_{X\cup\{x\}} t'_o\rangle & \textbf{if } t = x : \boldsymbol{t} \wedge t' = x : \boldsymbol{t}' \\ t \sqcup^{\mathsf{t}} t' & \textbf{otherwise} \end{cases}$$

We then define $\triangledown^{\mathsf{t}}_X : \mathcal{V}^{\mathsf{t}}_X \times \mathcal{V}^{\mathsf{t}}_X \to \mathcal{V}^{\mathsf{t}}_X$ as the composition of $\triangledown^{\mathsf{sh}}_X$ and $\triangledown^{\mathsf{ra}}_X$, that is, $t \triangledown^{\mathsf{t}}_X t' \stackrel{\text{def}}{=} t \triangledown^{\mathsf{ra}}_X (t \triangledown^{\mathsf{sh}}_X t')$.

Lemma 7.9. *$\triangledown^{\mathsf{t}}_X$ is a widening operator.*

We lift the widening operators $\triangledown^{\mathsf{t}}_X$ pointwise to an upper bound operator $\dot{\triangledown}^{\mathsf{t}} : \mathcal{M}^{\mathsf{t}} \times \mathcal{M}^{\mathsf{t}} \to \mathcal{M}^{\mathsf{t}}$ and define a widened data flow refinement semantics $\mathrm{C}^{\mathsf{t}}_{\triangledown}[\![\cdot]\!] : Exp \to \mathcal{P}^{\mathsf{t}}$ as the least fixpoint of the widened iterates of $\mathsf{step}^{\mathsf{t}}$:

$$\mathrm{C}^{\mathsf{t}}_{\triangledown}[\![e]\!] \stackrel{\text{def}}{=} \mathrm{lfp}^{\dot{\sqsubseteq}^{\mathsf{t}}}_{M^{\mathsf{t}}_{\bot}} \Lambda M^{\mathsf{t}}.\ \textbf{let } \langle \_, M^{\mathsf{t}\prime}\rangle = \mathsf{step}^{\mathsf{t}}[\![e]\!](\epsilon, \epsilon)(M^{\mathsf{t}}) \textbf{ in } (M^{\mathsf{t}} \mathbin{\dot{\triangledown}^{\mathsf{t}}} M^{\mathsf{t}\prime}) \tag{1}$$

[4]We here assume that recursive functions are encoded using the Y combinator.

The following theorem then follows directly from Lemma 7.9 and [Cousot and Cousot 1977].

Theorem 7.10. *The widened refinement type semantics is sound and terminating, i.e., for all programs e, $\mathrm{C}^{\mathsf{t}}_{\triangledown}[\![e]\!]$ converges in finitely many iterations. Moreover, $\mathrm{C}^{\mathsf{t}}[\![e]\!] \mathrel{\dot{\sqsubseteq}^{\mathsf{t}}} \mathrm{C}^{\mathsf{t}}_{\triangledown}[\![e]\!]$.*

Our parametric type inference algorithm thus computes $\mathrm{C}^{\mathsf{t}}_{\triangledown}[\![e]\!]$ iteratively according to (1). By Theorem 7.10 and Corollary 7.4, if the resulting type map is safe, then so is e.

8 IMPLEMENTATION AND EVALUATION

We have implemented a prototype of our parametric data flow refinement type inference analysis in a tool called Drift[5]. The tool is written in OCaml and builds on top of the Apron library [Jeannet and Miné 2009] to support various numerical abstract domains of type refinements. We have implemented two versions of the analysis: a context-insensitive version in which all entries in tables are collapsed to a single one (as in Liquid type inference) and a 1-context-sensitive analysis that distinguishes table entries based on the most recent call site locations. For the widening on basic refinement types, we consider two variants: plain widening and widening with thresholds [Blanchet et al. 2007]. Both variants use Apron's widening operators for the individual refinement domains.

The tool takes programs written in a subset of OCaml as input. This subset supports higher-order recursive functions, operations on primitive types such as integers and Booleans, as well as lists and arrays. We note that the abstract and concrete transformers of our semantics can be easily extended to handle recursive function definitions directly. As of now, the tool does not yet support user-defined algebraic data types. Drift automatically checks whether all array accesses are within bounds. In addition, the tool supports the verification of user-provided assertions. Type refinements for lists can express constraints on both the list's length and its elements.

To evaluate Drift we conducted two experiments that aim to answer the following questions:

(1) What is the trade-off between efficiency and precision for different instantiations of our parametric analysis framework?
(2) How does our new analysis compare with other state-of-the-art automated verification tools for higher-order programs?

Benchmarks and Setup. We collected a benchmark suite of OCaml programs by combining several sources of programs from prior work and augmenting it with our own new programs. Specifically, we included the programs used in the evaluation of the DOrder [Zhu et al. 2016] and R_Type [Champion et al. 2018a] tools, excluding only those programs that involve algebraic data types or certain OCaml standard library functions that our tool currently does not yet support. We generated a few additional variations of some of these programs by adding or modifying user-provided assertions, or by replacing concrete test inputs for the program's `main` function by unconstrained parameters. In general, the programs are small (up to 86 lines) but intricate. We partitioned the programs into five categories: first-order arithmetic programs (FO), higher-order arithmetic programs (HO), higher-order programs that were obtained by reducing program termination to safety checking (T), array programs (A), and list programs (L). All programs except two in the T category are safe. We separated these two erroneous programs out into a sixth category (E) which we augmented by additional unsafe programs obtained by modifying safe programs so that they contain implementation bugs or faulty specifications. The benchmark suite is available in the tool's Github repository. All our experiments were conducted on a desktop computer with an Intel(R) Core(TM) i7-4770 CPU and 16 GB memory running Linux.

[5] https://github.com/nyu-acsys/drift/

Table 1. Summary of Experiment 1. For each benchmark category, we provide the number of programs within that category in parentheses. For each benchmark category and configuration, we list: the number of programs successfully analyzed **(succ)** and the total accumulated running time across all benchmarks in seconds **(total)**. In the E category, an analysis run is considered successful if it flags the type error/assertion violation present in the benchmark. The numbers given in parenthesis in the total rows indicate the number of benchmarks that failed due to timeouts (if any). These benchmarks are excluded from the calculation of the cumulative running times. The timeout threshold was 300s per benchmark.

Benchmark category		Configuration											
	Version	context-insensitive						1-context-sensitive					
	Domain	Oct		Polka strict		Polka loose		Oct		Polka strict		Polka loose	
	Widening	w	tw	w	tw	w	tw	w	tw	w	tw	w	tw
FO (73)	succ	25	39	39	51	39	51	33	46	47	58	47	59
loc: 11	total	10.40	46.94	17.37	46.42(1)	15.82	42.88(1)	67.38	129.80	92.37	138.00(1)	87.27	138.33
HO (62)	succ	33	48	42	55	42	55	42	51	48	60	48	60
loc: 10	total	8.53	49.97	14.97	60.67	14.03	54.71	83.56	282.57	119.10	345.03	112.05	316.18
T (80)	succ	72	72	72	72	72	72	79	79	78	79	78	79
loc: 44	total	806.24	842.70	952.53	994.52	882.21	924.00	1297.13(1)	1398.25(1)	1467.09(1)	1566.11(1)	1397.71(1)	1497.28(1)
A (13)	succ	6	6	8	8	8	8	8	8	11	11	11	11
loc: 17	total	4.30	23.43	7.66	25.04	7.19	23.38	17.99	41.66	28.07	47.77	26.84	45.94
L (20)	succ	8	14	10	14	10	14	11	18	10	18	10	18
loc: 16	total	1.62	12.02	4.00	12.52	3.45	10.97	5.73	27.01	11.46	29.01	10.32	26.43
E (17)	succ	17	17	17	17	17	17	17	17	17	17	17	17
loc: 21	total	8.04	14.89	12.72	19.44	12.01	18.28	17.90	28.76	24.96	34.24	23.75	32.71

Experiment 1: Comparing different configurations of Drift. We consider the two versions of our tool (context-insensitive and 1-context-sensitive) and instantiate each with two different relational abstract domains implemented in Apron: Octagons [Miné 2007] (Oct), and Convex Polyhedra and Linear Equalities [Cousot and Halbwachs 1978] (Polka). For the Polka domain we consider both its *loose* configuration, which only captures non-strict inequalities, as well as its *strict* configuration which can also represent strict inequalities. We note that the analysis of function calls critically relies on the abstract domain's ability to handle equality constraints precisely. We therefore do not consider the interval domain as it cannot express such relational constraints. For each abstract domain, we further consider two different widening configurations: standard widening (w) and widening with thresholds (tw). For widening with thresholds [Blanchet et al. 2007], we use a simple heuristic that chooses the conditional expressions in the analyzed program as well as pair-wise inequalities between the variables and numerical constants in scope as threshold constraints.

Table 1 summarizes the results of the experiment. First, note that all configurations successfully flag all erroneous benchmarks (as one should expect from a sound analysis). Moreover, the context-sensitive version of the analysis is in general more precise than the context-insensitive one. The extra precision comes at the cost of an increase in the analysis time by a factor of 1.8 on average. The 1-context-sensitive version with Polka loose/tw performed best, solving 244 out of 265 benchmarks. There are two programs for which some of the configurations produced timeouts. However, each of these programs can be successfully verified by at least one configuration. As expected, using Octagon is more efficient than using loose polyhedra, which in turn is more efficient than strict polyhedra. We anticipate that the differences in running times for the different domains will be more pronounced on larger programs. In general, one can use different domains for different parts of the program as is common practice in static analyzers such as Astrée [Cousot et al. 2009].

We conducted a detailed analysis of the 20 benchmarks that Drift could not solve using any of the configurations that we have considered. To verify the 16 failing benchmarks in the FO and HO categories, one needs to infer type refinements that involve either non-linear or non-convex constraints, neither of which is currently supported by the tool. This can be addressed e.g. by further increasing context-sensitivity, by using more expressive domains such as interval polyhedra [Chen et al. 2009], or by incorporating techniques such as trace partitioning to reduce loss of precision due

to joins [Mauborgne and Rival 2005]. The two failing benchmarks in the array category require the analysis to capture universally quantified invariants about the elements stored in arrays. However, our implementation currently only tracks array length constraints. It is relatively straightforward to extend the analysis in order to capture constraints on elements as has been proposed e.g. in [Vazou et al. 2013].

We further conducted a more detailed analysis of the running times by profiling the execution of the tool. This analysis determined that most of the time is spent in the strengthening of output types with input types when propagating recursively between function types in $\ltimes^{\mathsf{t}}$. This happens particularly often when analyzing programs that involve applications of curried functions, which are currently handled rather naively, causing a quadratic blowup that can be avoided with a more careful implementation. Notably, the programs in the T category involve deeply curried functions. This appears to be the primary reason why the tool is considerably slower on these programs. Moreover, the implementation of the fixpoint loop is still rather naive as it calls $\ltimes^{\mathsf{t}}$ even if the arguments have not changed since the previous fixpoint iteration. We believe that by avoiding redundant calls to $\ltimes^{\mathsf{t}}$ the running times can be notably improved.

Experiment 2: Comparing with other Tools. Overall, the results of Experiment 1 suggest that the 1-context-sensitive version of Drift instantiated with the loose Polka domain and threshold widening provides a good balance between precision and efficiency. In our second experiment, we compare this configuration with several other existing tools. We consider three other automated verification tools: DSolve, R_Type, and MoCHi. DSolve is the original implementation of the Liquid type inference algorithm proposed in [Rondon et al. 2008] (cf. § 2). R_Type [Champion et al. 2018a] improves upon DSolve by replacing the Houdini-based fixpoint algorithm of [Rondon et al. 2008] with the Horn clause solver HoIce [Champion et al. 2018b]. HoIce uses an ICE-style machine learning algorithm [Garg et al. 2014] that, unlike Houdini, can also infer disjunctive refinement predicates. We note that R_Type does not support arrays or lists and hence we omit it from these categories. Finally, MoCHi [Kobayashi et al. 2011] is a software model checker based on higher-order recursion schemes, which also uses HoIce as its default back-end Horn clause solver. We used the most recent version of each tool at the time when the experiments were conducted and we ran all tools in their default configurations. More precise information about the specific tool versions used can be found in the tools' Github repository.

We initially also considered DOrder [Zhu et al. 2016] in our comparison. This tool builds on the same basic algorithm as DSolve but also learns candidate predicates from concrete program executions via machine learning. However, the tool primarily targets programs that manipulate algebraic data types. Moreover, DOrder relies on user-provided test inputs for its predicate inference. As our benchmarks work with unconstrained input parameters and we explicitly exclude programs manipulating ADTs from our benchmark set, this puts DOrder decidedly at a disadvantage. To keep the comparison fair, we therefore excluded DOrder from the experiment.

Table 2 summarizes the results of our comparison. Drift and MoCHi perform similarly overall and significantly better than the other two tools. In particular, we note that only Drift and MoCHi can verify the second program discussed in § 2. The evaluation also indicates complementary strengths of these tools. In terms of number of verified benchmarks, Drift performs best in the HO, A, and L categories with MoCHi scoring a close second place. For the FO and T categories the roles are reversed. In the FO category, MoCHi benefits from its ability to infer non-convex type refinements, which are needed to verify some of the benchmarks in this category. Nevertheless there are five programs in this category that only Drift can verify. Unlike our current implementation, MoCHi does not appear to suffer from inefficient handling of deeply curried functions, which leads

Table 2. Summary of Experiment 2. In addition to the cumulative running time for each category, we provide the average **(avg)** and median **(med)** running time per benchmark (in s). Timeouts are reported as in Table 1. The timeout threshold was 300s per benchmark across all tools. In the **(succ)** column, we additionally provide in parentheses the number of benchmarks that were only solved by that tool (if any).

Bench-mark cat.	Drift				R_Type				DSolve				MoCHi			
	succ	full	avg	med	succ	full	avg	med	succ	full	avg	med	succ	full	avg	med
FO (73)	59(5)	138.33	1.89	0.26	44	5.78(15)	0.08	0.06	49	16.27	0.22	0.12	**62**	332.39(9)	4.55	21.50
HO (62)	**60**	316.18	5.10	1.77	49	9.19(5)	0.15	0.03	41	9.76	0.16	0.24	58	276.37(4)	4.46	15.18
T (80)	79	1497.28(1)	18.72	0.00	73	13.18	0.16	0.04	30	26.26	0.33	0.45	**80**	41.56	0.52	0.11
A (13)	**11**	45.94	3.53	3.70	-	-	-	-	8	7.15	0.55	0.77	9(1)	2.44	0.19	0.10
L (20)	**18(2)**	26.43	1.32	1.23	-	-	-	-	8	6.10	0.30	0.26	17	455.28(3)	22.76	19.81
E (17)	**17**	32.71	1.92	0.39	**17**	1.22	0.07	0.05	14	8.22	0.48	0.17	14	95.95(3)	5.64	43.41

to significantly better cumulative running times in the T category. On the other hand, Drift is faster than MoCHi in the L category.

We note that there are two benchmarks in the FO category and four benchmarks in the A category for which MoCHi produces false alarms. However, this appears to be due to the use of certain language features that are unsupported by the tool (such as OCaml's `mod` operator).

None of the tools produced unsound results in the E category. The failing benchmarks for MoCHi are due to timeouts and the ones for DSolve are due to crashes. The considered timeout was 300 seconds per benchmark for all tools. Across all benchmarks, R_Type timed out on 20 and MoCHi on 19 programs. Drift timed out on only one benchmark in the T category. We attribute the good responsiveness of Drift to the use of infinite refinement domains with widening in favor of the counterexample-guided abstraction refinement approach used by R_Type and MoCHi, which does not guarantee termination of the analysis.

9 RELATED WORK

Refinement type inference. Early work on refinement type systems supported dependent types with unions and intersections based on a *modular* bidirectional type checking algorithm [Dunfield and Pfenning 2003, 2004; Freeman and Pfenning 1991]. These algorithms require some type annotations. Instead, the focus of this paper is on fully automated refinement type inference algorithms that perform a whole program analysis. Many existing algorithms in this category can be obtained by instantiating our parametric framework. Specifically, Liquid type inference [Rondon et al. 2008; Vazou et al. 2015, 2013, 2014] performs a context-insensitive analysis over a monomial predicate abstraction domain of type refinements. Similarly, Zhu and Jagannathan [2013] propose a 1-context sensitive analysis with predicate abstraction, augmented with an additional counterexample-guided refinement loop, an idea that has also inspired recent techniques for analyzing pointer programs [Toman et al. 2020]. Our work generalizes these algorithms to arbitrary abstract domains of type refinements (including domains of infinite height) and provides parametric and constructive soundness and completeness results for the obtained type systems. Orthogonal to our work are extensions of static inference algorithms with data-driven approaches for inferring refinement predicates [Champion et al. 2018a; Zhu et al. 2015, 2016]. Gradual Liquid type inference [Vazou et al. 2018b] addresses the issue of how to apply whole program analysis to infer modular specifications and improve error reporting. Our framework is in principle compatible with this approach. We note that Galois connections are used in [Garcia et al. 2016; Kazerounian et al. 2018; Lehmann and Tanter 2017; Vazou et al. 2018b; Vekris et al. 2016] to relate dynamic gradual refinement types and static refinement types. However, the resulting gradual type systems are not calculationally constructed as abstract interpretations of concrete program semantics.

Semantics of higher-order programs. The techniques introduced in [Jagannathan and Weeks 1995] and [Plevyak and Chien 1995] use flow graphs to assign concrete meaning to higher-order programs. Nodes in these graphs closely relate to the nodes in our data flow semantics. Their semantics represents functions as expression nodes storing the location of the function expression. Hence, these semantics have to make non-local changes when analyzing function applications. Our data flow semantics treats functions as tables and the concrete transformer is defined structurally on program syntax, which is more suitable for deriving type inference analyses. The idea of modeling functions as tables that only track inputs observed during program execution was first explored in the minimal function graph semantics [Jones and Mycroft 1986; Jones and Rosendahl 1997]. However, that semantics does not explicitly model data flow in a program, and is hence not well suited for constructing refinement type inference algorithms. There is a large body of work on control and data flow analysis of higher-order programs and the concrete semantics they overapproximate [Cousot and Cousot 1994; Horn and Might 2011; Jones and Andersen 2007; Midtgaard 2012; Mossin 1998; Nielson and Nielson 1997]. However, these semantics either do not capture data flow properties or rely on non-structural concrete transformers. An interesting direction for future work is to reconcile our collapsed semantics with control flow analyses that enjoy *pushdown precision* such as [Gilray et al. 2016; Reps 1998; Vardoulakis and Shivers 2011a,b].

Temporal logic for higher-order programs [Okuyama et al. 2019] and higher-order recursion schemes [Kobayashi et al. 2011; Ong 2015] provide alternative bases for verifying functional programs. Unlike our framework, these approaches use finite state abstractions and model checking. In particular, they rely on abstraction refinement to support infinite height data domains, giving up on guaranteed termination of the analysis. On the other hand, they are not restricted to proving safety properties.

Types as abstract interpretations. The formal connection between types and abstract interpretation is studied by Cousot [1997]. The paper shows how to construct standard (modular) polymorphic type systems via a sequence of abstractions of denotational call-by-value semantics. Similarly, Monsuez [1992, 1993, 1995a,b] uses abstract interpretation to design polymorphic type systems for call-by-name semantics and model advanced type systems such as System F. Gori and Levi [2002, 2003] use abstract interpretation to design new type inference algorithms for ML-like languages by incorporating more precise widening operators for analyzing recursive functions. None of these works address refinement type inference. Harper introduces a framework for constructing dependent type systems from operational semantics based on the PER model of types [Harper 1992]. Although this work does not use abstract interpretation, it views types as an overapproximation of program behaviors derived using a suitable notion of abstraction.

10 CONCLUSION

In this work, we systematically develop a parametric refinement type systems as an abstract interpretation of a new concrete data flow semantics. This development unveils the design space of refinement type analyses that infer data flow invariants for functional programs. Our prototype implementation and experimental evaluation indicate that our framework can be used to implement new refinement type inference algorithms that are both robust and precise.

ACKNOWLEDGMENTS

This work is funded in parts by the National Science Foundation under grants CCF-1350574 and CCF-1618059. We thank the anonymous reviewers for their feedback on an earlier draft of this paper.

REFERENCES

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Vincenzo Arceri, Martina Olliaro, Agostino Cortesi, and Isabella Mastroeni. 2019. Completeness of Abstract Domains for String Analysis of JavaScript Programs. In *Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11884)*, Robert M. Hierons and Mohamed Mosbah (Eds.). Springer, 255–272. https://doi.org/10.1007/978-3-030-32505-3_15

Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.* 72, 1-2 (2008), 3–21. https://doi.org/10.1016/j.scico.2007.08.001

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2007. A Static Analyzer for Large Safety-Critical Software. *CoRR* abs/cs/0701193 (2007). arXiv:cs/0701193 http://arxiv.org/abs/cs/0701193

Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018a. ICE-Based Refinement Type Discovery for Higher-Order Functional Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10805)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 365–384. https://doi.org/10.1007/978-3-319-89960-2_20

Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. 2018b. HoIce: An ICE-Based Non-linear Horn Clause Solver. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 146–156. https://doi.org/10.1007/978-3-030-02768-1_8

Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. 2009. Interval Polyhedra: An Abstract Domain to Infer Interval Linear Relationships. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5673)*, Jens Palsberg and Zhendong Su (Eds.). Springer, 309–325. https://doi.org/10.1007/978-3-642-03237-0_21

Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent types for JavaScript. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 587–606. https://doi.org/10.1145/2384616.2384659

Patrick Cousot. 1997. Types as abstract interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 316–331.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 269–282.

Patrick Cousot and Radhia Cousot. 1994. Invited Talk: Higher Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis. In *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*, Henri E. Bal (Ed.). IEEE Computer Society, 95–112. https://doi.org/10.1109/ICCL.1994.288389

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. 2009. Why does Astrée scale up? *Formal Methods Syst. Des.* 35, 3 (2009), 229–264. https://doi.org/10.1007/s10703-009-0089-6

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96. https://doi.org/10.1145/512760.512770

Joshua Dunfield and Frank Pfenning. 2003. Type Assignment for Intersections and Unions in Call-by-Value Languages. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2620)*, Andrew D. Gordon (Ed.). Springer, 250–266. https://doi.org/10.1007/3-540-36576-1_16

Joshua Dunfield and Frank Pfenning. 2004. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 281–292. https://doi.org/10.1145/964001.964025

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March*

12-16, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2021), José Nuno Oliveira and Pamela Zave (Eds.). Springer, 500–517. https://doi.org/10.1007/3-540-45251-6_29

Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 268–277. https://doi.org/10.1145/113445.113468

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 429–442. https://doi.org/10.1145/2837614.2837670

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 691–704. https://doi.org/10.1145/2837614.2837631

Roberta Gori and Giorgio Levi. 2002. An Experiment in Type Inference and Verification by Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation, Third International Workshop, VMCAI 2002, Venice, Italy, January 21-22, 2002, Revised Papers (Lecture Notes in Computer Science, Vol. 2294)*, Agostino Cortesi (Ed.). Springer, 225–239. https://doi.org/10.1007/3-540-47813-2_16

Roberta Gori and Giorgio Levi. 2003. Properties of a Type Abstract Interpreter. In *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2575)*, Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay (Eds.). Springer, 132–145. https://doi.org/10.1007/3-540-36384-X_13

Robert Harper. 1992. Constructing Type Systems over an Operational Semantics. *J. Symb. Comput.* 14, 1 (1992), 71–84. https://doi.org/10.1016/0747-7171(92)90026-Z

David Van Horn and Matthew Might. 2011. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Commun. ACM* 54, 9 (2011), 101–109. https://doi.org/10.1145/1995376.1995400

Suresh Jagannathan and Stephen Weeks. 1995. A Unified Treatment of Flow Analysis in Higher-order Languages. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA). ACM, 393–407. https://doi.org/10.1145/199448.199536

Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 661–667. https://doi.org/10.1007/978-3-642-02658-4_52

Neil D. Jones and Nils Andersen. 2007. Flow Analysis of Lazy Higher-order Functional Programs. *Theor. Comput. Sci.* 375, 1-3 (April 2007), 120–136. https://doi.org/10.1016/j.tcs.2006.12.030

Neil D. Jones and Alan Mycroft. 1986. Data Flow Analysis of Applicative Programs Using Minimal Function Graphs. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. ACM Press, 296–306. https://doi.org/10.1145/512644.512672

Neil D. Jones and Mads Rosendahl. 1997. Higher-Order Minimal Function Graphs. *J. Funct. Log. Program.* 1997, 2 (1997). http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/1997/A97-02/A97-02.html

Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. 2018. Refinement Types for Ruby. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*, Isil Dillig and Jens Palsberg (Eds.). Springer, 269–290. https://doi.org/10.1007/978-3-319-73721-8_13

Se-Won Kim and Kwang-Moo Choe. 2011. String Analysis as an Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6538)*, Ranjit Jhala and David A. Schmidt (Eds.). Springer, 294–308. https://doi.org/10.1007/978-3-642-18275-4_21

Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 222–233. https://doi.org/10.1145/1993498.1993525

Shuvendu K. Lahiri and Shaz Qadeer. 2009. Complexity and Algorithms for Monomial and Clausal Predicate Abstraction. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5663)*, Renate A. Schmidt (Ed.). Springer, 214–229. https:

//doi.org/10.1007/978-3-642-02959-2_18

Nico Lehmann and Éric Tanter. 2017. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 775–788. http://dl.acm.org/citation.cfm?id=3009856

Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3444)*, Shmuel Sagiv (Ed.). Springer, 5–20. https://doi.org/10.1007/978-3-540-31987-0_2

Jan Midtgaard. 2012. Control-flow analysis of functional programs. *ACM Comput. Surv.* 44, 3 (2012), 10:1–10:33. https://doi.org/10.1145/2187671.2187672

Antoine Miné. 2007. The Octagon Abstract Domain. *CoRR* abs/cs/0703084 (2007). arXiv:cs/0703084 http://arxiv.org/abs/cs/0703084

Bruno Monsuez. 1992. Polymorphic typing by abstract interpretation. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 217–228.

Bruno Monsuez. 1993. Polymorphic types and widening operators. In *Static Analysis*. Springer, 267–281.

Bruno Monsuez. 1995a. System F and abstract interpretation. In *International Static Analysis Symposium*. Springer, 279–295.

Bruno Monsuez. 1995b. Using abstract interpretation to define a strictness type inference system. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 122–133.

Christian Mossin. 1998. Higher-Order Value Flow Graphs. *Nord. J. Comput.* 5, 3 (1998), 214–234.

Hanne Riis Nielson and Flemming Nielson. 1997. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 332–345. https://doi.org/10.1145/263699.263745

Yuya Okuyama, Takeshi Tsukada, and Naoki Kobayashi. 2019. A Temporal Logic for Higher-Order Functional Programs. In *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11822)*, Bor-Yuh Evan Chang (Ed.). Springer, 437–458. https://doi.org/10.1007/978-3-030-32304-2_21

Luke Ong. 2015. Higher-Order Model Checking: An Overview. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. IEEE Computer Society, 1–15. https://doi.org/10.1109/LICS.2015.9

Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. 2020. Data Flow Refinement Type Inference. *CoRR* abs/2011.04876 (2020). arXiv:2011.04876 http://arxiv.org/abs/2011.04876

John Plevyak and Andrew A. Chien. 1995. Iterative Flow Analysis.

Thomas W. Reps. 1998. Program analysis via graph reachability. *Inf. Softw. Technol.* 40, 11-12 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 46–59. http://dl.acm.org/citation.cfm?id=3009885

John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 684–714. https://doi.org/10.1007/978-3-030-44914-8_25

Hiroshi Unno and Naoki Kobayashi. 2009. Dependent type inference with interpolants. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, António Porto and Francisco Javier López-Fraguas (Eds.). ACM, 277–288. https://doi.org/10.1145/1599410.1599445

Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. 2013. Automating relatively complete verification of higher-order functional programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 75–86. https://doi.org/10.1145/2429069.2429081

Dimitrios Vardoulakis and Olin Shivers. 2011a. CFA2: a Context-Free Approach to Control-Flow Analysis. *Log. Methods Comput. Sci.* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:3)2011

Dimitrios Vardoulakis and Olin Shivers. 2011b. Pushdown flow analysis of first-class control. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 69–80. https://doi.org/10.1145/2034773.2034785

Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 48–61. https://doi.org/10.1145/2784731.2784745

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018a. Theorem proving for all: equational reasoning in liquid Haskell (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 132–144. https://doi.org/10.1145/3242744.3242756

Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 209–228. https://doi.org/10.1007/978-3-642-37036-6_13

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 269–282. https://doi.org/10.1145/2628136.2628161

Niki Vazou, Éric Tanter, and David Van Horn. 2018b. Gradual liquid type inference. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 132:1–132:25. https://doi.org/10.1145/3276502

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 310–325. https://doi.org/10.1145/2908080.2908110

Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. ACM, 61–78. https://doi.org/10.1145/91556.91592

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 214–227. https://doi.org/10.1145/292540.292560

He Zhu and Suresh Jagannathan. 2013. Compositional and Lightweight Dependent Type Inference for ML. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7737)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, 295–314. https://doi.org/10.1007/978-3-642-35873-9_19

He Zhu, Aditya V. Nori, and Suresh Jagannathan. 2015. Learning refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 400–411. https://doi.org/10.1145/2784731.2784766

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically learning shape specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 491–507. https://doi.org/10.1145/2908080.2908125