DATA FLOW REFINEMENT TYPE INFERENCE TOOL

DRIFT^2

By

YUSEN SU

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

NEW YORK UNIVERSITY
Department of Computer Science

MAY 2020

To the Faculty of New York University:

The members of the Committee appointed to examine the thesis of YUSEN SU find it satisfactory and recommend that it be accepted.

Prof., Thomas Wies

Ph.D., Zvonimir Pavlinovic

ACKNOWLEDGMENT

I firstly would like to sincerely thank my thesis advisor, Thomas Wies. I appreciate your guidance and useful advice as I worked on this project in one and a half years.

I also wish to show my gratefulness to my second reader, Zvonimir Pavlinovic. Thomas's and his great efforts on the theoretical framework, which was created in his PhD dissertation, support my work.

As always, I would be glad to express my gratitude to my family, fiancee and friends for their support and encouragement.

ABSTRACT

Refinement types utilize logical predicate for capturing run-time properties of programs which can be used for program verification. Traditionally, SMT-based checking tools of refinement types such as the implementation of Liquid Types [1] require either heuristics or random sampling logical qualifiers to find the relevant logical predicates.

In this thesis, we describe the implementation of a novel algorithm proposed in Zvonimir Pavlinovic's PhD thesis "Leveraging Program Analysis for Type Inference" [2], based on the framework of abstract interpretation for inferring refinement types in functional programs. The analysis generalizes Liquid type inference and is parametric with the abstract domain used to express type refinements. The main contribution of this thesis is to achieve the process of instantiating our parametric type analysis and to evaluate the algorithm's precision and efficiency. Moreover, we describe a tool, called DRIFT^2, which allows users to select an abstract domain for expressing type refinements and to control the degree to which context-sensitive information is being tracked by the analysis.

Finally, our work compares the precision and efficiency of DRIFT^2 for different configurations of numerical abstract domains and widening operations [3]. In addition, we compare DRIFT^2 with existing refinement type inference tools. The experimental results show that our method is both effective and efficient in automatically inferring refinement types.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter One

Introduction & Background

To guarantee the reliability and re-usability of software systems, researchers have developed many formal methods techniques such as program analysis [4] and verification [5]. Building reliable software, however, still faces many challenges. For example, writing too much data into system memory yields data outside of the intended structure, which encounters the difficulty of maintenance. Reasoning about such errors requires software engineers with comprehensive knowledge of the syntax and semantics of programming languages, standard/third-party libraries, and development environments. It is impossible to expect all programmers to be experts on the study of the meaning of programming languages, but it brings an opportunity for researchers to deploy automated verification methods. In order to support efficient and expressive reasoning, one direction is using a type system [6] which gives a set of rules that assign a type to each program construct, such as variables, expressions, or functions. Typically, a type system provides primitive types such as `int` representing integer values and type constructors such as `array` representing more complex composite data structures. These types guide programmers to correctly manipulate data in their programs. This feature ultimately provides programmers with a syntax-directed way of reasoning about program behaviors. Though programmers may be unwilling to provide type information manually. This raises the question of how to omit type annotations in a program but still enable the compiler to do type checking.

Fortunately, researchers have developed a technique, called type inference, which gives us this opportunity. Type inference is a feature of programming languages that computes the type of an expression without type annotations. Particularly, Hindley-Milner type inference [7] is one of the type inference algorithms that is in common use. In fact, it is not just an algorithm that determines type of an expression at compile time, but it always infers the most general type of the expression[1]. When combined with static type inference, a type system allows a compiler to verify the type correctness based on the analysis of a program's text. This component enables the compiler to guarantee program correctness properties without replying on extensive type annotations provided by the programmer.

However, the declaration of types like `int array` is not enough for expressing either index checking or range checking. For example, consider the following OCaml program:

```
1 let x: int array = Array.make 100 0 in
2 for i = 1 to 100 do
3     x.(i) <- read_int ()
4 done
```

The program initializes a zero-valued integer array `x` with a length of `100`, and uses a for-loop to repeatedly replace each element number in `x` with a user-provided integer value. If the program always requires an array to store non-zero values, such type systems like OCaml type system could not indicate those important program invariants. Thus, to extend the layer of precision during program analysis and be able to reason about the program's run-time behaviors more accurately, we need to put additional information into type systems.

Refinement type systems [8] are able to express and verify richer properties of the values computed by the program. A refinement type enriches a simple type with a predicate (i.e. refinement predicate) that conveys additional constraints about the represented values

[1]Although the feature of parametric polymorphism is more important, we will not discuss it in this thesis.

in terms of a Boolean-valued expression. In particular, the refinement predicates can precisely express dependencies between inputs and outputs of functions.

For instance, in OCaml, we can include refinement type systems to capture constraints on arrays manipulated by a program that are sufficient to verify the absence of out-of-bounds access:

```
1 let ra = Array.make 4 0 in (* ra: {ν: Int Array | length ν = 4} *)
2 let x = 3 in (* x: {ν: Int | ν = 3} *)
3 Array.set ra 5 x (* Run-time Error! Array index out of bounds *)
```

The example firstly creates an array `ra` of length `4` with all elements initialized to `0`, and then aims to update that array with value `x` at index `5`. The above program fails during execution because it attempts to set an element with an index that exceeds the allowed length (i.e. length of `ra` is `4` which is less than the requested index, `5`). However, the refinement type system can be used to catch the error at compile time. As we presented in the comments, each refinement type of a term (array `ra`) combines a base type (`Int Array`) with a predicate ($\{\nu : \mathsf{Int\ Array} \mid \mathsf{length}\ \nu = 4\}$) over the special "value variable" ν which is used to describe the value of the term. The predicate specifies a set of values that captures a more expressive program invariant. Particularly, the refinement type of `ra` captures the set of indices that can be used for safely interacting with the array returned by `Array.make 4 0`.

At line 3, the refinement type of the function Array.set in the above example can be expressed as:

$$\mathsf{Array.set} :: (a : \alpha\ \mathsf{array}) \rightarrow (i : \{\nu : \mathsf{Int} \mid 0 \leq \nu < \mathsf{length}\ a\}) \rightarrow (e : \{\nu : \mathsf{Int} \mid \mathsf{true}\}) \rightarrow \mathsf{unit}$$

The type $(i : \{\nu : \mathsf{int} \mid 0 \leq \nu < \mathsf{length}\ a\})$ of array index parameter i augments a simple type Int with a predicate indicating that the index of the given array a must be in the range 0 to the length of the array (which is `4` for array `ra`).

Thus, the refinement type provides a static method to capture more fine-grained infor-

mation about the concrete values observed during program execution, instead of relying on expensive run-time checks. Although these advantages come at a cost of automatic inference, a refinement type that encodes details about execution time behavior, which allows program analysis to infer how loops and components behave for the verification in practice.

For now, the question is how to infer such refinement types automatically in practice. In this thesis, we focus our attention on the Liquid type family [8, 9, 10, 1] of refinement type inference algorithms.

1.1 Liquid Type Inference

We start our discussion with an overview of the Liquid type inference algorithm and the challenges of inferring precise Liquid types with existing approaches. We then present an overview of our approach by developing Liquid type system as abstract interpretation.

To motivate our technical development, consider the following illustrative OCaml program:

```
1 let rec sum n =
2   if n <= 0 then 0
3   else n + sum (n - 1)
```

The program defines a function sum which computes the sum of all natural numbers up to the given bound n.

We briefly discuss how Liquid type inference works on the above example. At first, the algorithm calls for Hindley-Milner Type Inference [7] to infer the basic shape of the refinement type for every subexpression of the program. For instance, the inferred type for the function sum is int $\rightarrow$ int. Next, the algorithm replaces inferred ML types with *templates* of refinement types. Each base type τ (e.g. int) is replaced by a refinement type of the form $\{\nu : \tau \mid \phi(\nu, \vec{x})\}$, where ν is a value variable with type τ, and $\phi(\nu, \vec{x})$ is a

placeholder refinement predicate. The form expresses a constraint relation between the value variable ν and other variables $\vec{x}$ in scope of the type. Liquid type inference also encodes a dependent type relation for each function (written as $x : \tau_1 \rightarrow \tau_2$). Concretely, the function type includes a fresh dependency variable x which stands for the function's parameter, an input type τ_1 of the function, and a result type τ_2 of the function. The scope of x is the result type τ_2, i.e., refinement predicates inferred for τ_2 can express dependencies on the input value of type τ_1 by referring to x. For example, Liquid type inference infers the augmented type for function sum as:

$$n : \{\nu : \mathsf{int} \mid \phi_1(\nu)\} \rightarrow \{\nu : \mathsf{int} \mid \phi_2(\nu, n)\}.$$

Next, the algorithm generates a system of Horn clauses that specifies the subtyping relationships imposed on the refinement predicates by the program *data flow*. For instance, the body of sum induces the following Horn clauses over the refinement predicates in sum's type:

$$\begin{aligned}
n > 0 \land \nu = n - 1 &\Rightarrow \phi_1(v) && \mid \textit{value flows to the recursive call} \\
n \leq 0 \land \nu = 0 &\Rightarrow \phi_2(\nu, n) && \mid \textit{stipulates the "then" expression} \\
n > 0 \land \phi_2(\nu_1, n - 1) \land \nu = \nu_1 + n &\Rightarrow \phi_2(\nu, n) && \mid \textit{stipulates the "else" expression} \\
\nu \geq 0 &\Rightarrow \phi_1(v) && \mid \textit{no constraint applies to the initial input}
\end{aligned}$$

The first clause models the data flow from the parameter n to the recursive call in the *else* branch of the conditional. The second and third clause capture the constraints on the result value returned by sum in the *then* and *else* branch. The final one catches the constraint on the input value which we impose on the external calls to sum.

The original algorithm for Liquid type inference solves the obtained Horn clauses using a fixpoint computation based on predicate abstraction [11, 12] to derive each refinement predicate ϕ_i. That is, the analysis assumes a given set of atomic predicates $Q = \{p_1(\nu, \vec{x}), \ldots, p_n(\nu, \vec{x})\}$, 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 [13, 14]. For instance, the implementation of liquid type inference in the tool DSolve [15] solves flow constraints by given $Q = \{0 \leq \nu, \star \leq \nu, \nu < \star\}$ where $\star$ is a placeholder variable that can be instantiated with program variables. The final type for function sum obtained this way is:

$$n : \{\nu : \mathsf{int} \mid \nu \geq 0\} \rightarrow \{\nu : \mathsf{int} \mid 0 \leq \nu \wedge n \leq \nu\}$$

where true in the predicate means function sum takes any integers n which evaluates to true as input.

1.1.1 Challenges in Inferring Precise Liquid Types

Now, assume we set the goal of the analysis higher than before, where we wish to infer a refinement constraints for the return type of the function sum which applies the output is no less than $2 * n - 1$. However, the liquid type inference fails to conclude this inductive constraint. Typically, one problem of using predicate abstraction to describe program invariants is that the analysis needs to guess supplemental predicates like $2 * n - 1 \leq \nu$ in advance.

Generally, 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". The derived Horn clause system already signifies an abstraction of the program's semantics and, hence, entails an inherent loss of precision for any subsequent analysis. To motivate this issue, consider the following program:

```
1 let apply f x = f x
2 let g y = 2 * y
3 let h y = - (2 * y)
```

```
4 let main z =
5   let v = if 0 <= z then apply g z else apply h z in
6   assert (0 <= v)
```

Note that the `assert` statement in the last line is safe. The templates for the refinement types of the top-level functions are as follows:

$$\texttt{apply} :: (y : \{v\colon \mathsf{int} \mid \phi_1(v)\} \to \{v\colon \mathsf{int} \mid \phi_2(v,y)\}) \to x : \{v\colon \mathsf{int} \mid \phi_3(v)\} \to \{v\colon \mathsf{int} \mid \phi_4(v,x)\}$$

$$\texttt{g} :: y : \{v\colon \mathsf{int} \mid \phi_5(v)\} \to \{v\colon \mathsf{int} \mid \phi_6(v,y)\}$$

$$\texttt{h} :: y : \{v\colon \mathsf{int} \mid \phi_7(v)\} \to \{v\colon \mathsf{int} \mid \phi_8(v,y)\}$$

and the key Horn clauses are:

$$\begin{aligned}
0 \le z \wedge v = z &\Rightarrow \phi_3(v) & \qquad \phi_5(y) \wedge v = 2y &\Rightarrow \phi_6(v,y)\\
0 \le z \wedge \phi_1(v) &\Rightarrow \phi_5(v) & \qquad 0 \le z \wedge \phi_6(v,y) &\Rightarrow \phi_2(v,y)\\
0 > z \wedge v = z &\Rightarrow \phi_3(v) & \qquad \phi_7(y) \wedge v = -(2y) &\Rightarrow \phi_8(v,y)\\
0 > z \wedge \phi_1(v) &\Rightarrow \phi_7(v) & \qquad 0 \le z \wedge \phi_8(v,y) &\Rightarrow \phi_2(v,y)\\
\phi_3(x) &\Rightarrow \phi_1(v) & \qquad \phi_2(v,x) &\Rightarrow \phi_4(v,x)
\end{aligned}$$

Note that the least solution of this system of Horn clauses satisfies $\phi_1(v) = \phi_3(v) = \phi_5(v) = \phi_7(v) = \mathsf{true}$ and $\phi_2(v,x) = \phi_4(v,x) = (v = 2x \vee v = -(2x))$. The solution for $\phi_4(v,x)$ also represents the most precise type refinement that can be inferred for the type of `v` declared on line 5. 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 which of the two functions `g` and `h` will be called in `apply`. All existing refinement type inference tools based on inferring standard Liquid types therefore fail to verify the above example.

1.1.2 Design Space of Liquid Type Inference

With refinement type system and the limitation of liquid type inference, in our draft paper [16], we present a new view on liquid type inference in terms of abstract interpretation. In particular, we constructed a *data flow refinement type analysis* by forming a sequence of Galois abstractions of new concrete higher-order program semantics [17, 18]. Our new semantics captures refinement properties for every program location. Especially for a function with several call sites, the semantics makes explicit how input data propagates backwards from the call sites of a function to the function definition and, conversely, how output data flows from the definition back to the call sites. Furthermore, the analysis is parametric with (1) the abstract domain used to express type refinements, (2) the choice of the widening operator used to enforce the sound termination of the analysis, and (3) the degree to which context-sensitive control flow information is being tracked.

1.2 Contribution

In this thesis, we have implemented a prototype of the parametric data flow refinement type analysis in a tool called DRIFT^2 (written in OCaml). Here is a brief overview of the contributions, as well as the main features of DRIFT^2.

- Our model moves away from particular representations of type refinements, which leads DRIFT^2 to support various abstract domains for expressing refinement types such as Octagons [19] and Polyhedra [20].

- Our tool allows its user to specify various widening strategies, used by the abstract interpretation, to ensure the termination of the analysis.

- We have implemented two versions of the analysis: a context-insensitive version in which all entries in functions are collapsed to a single one (as in liquid type inference)

and a 1-context-sensitive analysis that infers intersection function types, allowing each call site location to be checked against a different type.

Acknowledgement. This thesis is part of joint work with Zvonimir Pavlinovic and Thomas Wies [16].

1.3 Thesis Organization

Chapter 2 gives an overview of the tool, and contains technical details of our program semantics and abstract transformer. We also present a subset of OCaml program we analyzed as example. Chapter 3 shows the experiment process and results in detail. Chapter 4 describes some related tools for refinement type inference. Finally, Chapter 5 discusses the limitations of DRIFT^2 and potential future work.

Chapter Two

Design & Theory

In this chapter, we firstly present detailed overview of the design of DRIFT^2, a composite verifier for a subset of OCaml that computes a value map recording an inferred refinement type for each program's execution point. In order to enable the tool to be parametric with the abstract domain of type refinements, DRIFT^2 builds on the Apron library [21] which provides numerical abstract domains for program analysis: users of DRIFT^2 can select an abstract domain to verify a given OCaml program.

After that, we delineate our parametric data flow refinement type semantics, originally from our draft paper [16]. Next, we give the details of our abstract transformer and propagation in Section2.2. Finally, we show how our system infers refinement types for a given OCaml program.

The chapter is organized as follows: Section 2.1 introduces the components of DRIFT^2, including verification workflow and the target language of the refinement type inference method. Section 2.2 gives a program semantics used for our data flow refinement type analysis, as well as an example that describes how the analysis works.

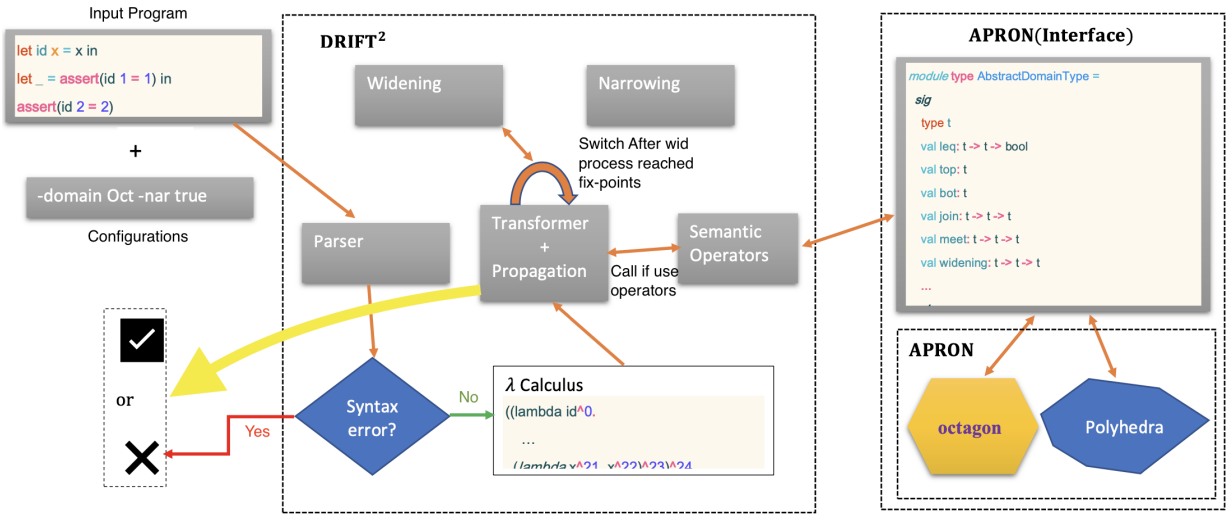

Figure 2.1 DRIFT^2 verification workflow.

2.1 Design

In this section, we briefly summarize the high-level design of DRIFT^2. For the analysis front end, we used a self-designed lexer and parser for parsing the input programs. We chose a subset of OCaml as target language. In the back ends of DRIFT^2, we designed two modules that provide interfaces for the various abstract domains and the implementation of the semantics domain. Finally, we implemented our transformer based on the input expressions and followed by the pseudo algorithms described from Fig. 2.4.

2.1.1 DRIFT^2 Verification Workflow

We start with a short interpretation of DRIFT^2 verification workflow, shown in Figure 2.1. Generally, DRIFT^2 takes an OCaml file and verifies the correctness of the program by preferring the analysis for a user-selected abstract domain and widening operator. For example, analyzing Program 1 (depicted in Fig. 2.6) yields a success message "The input program is safe".

Encoding

DRIFT^2 encodes (untyped) OCaml programs into the λ-calculus (defined in Sect.2.1.2). In particular, DRIFT^2 parses a conditional expression `if` e_0 `then` e_1 `else` e_2 as $e_0\,?\,e_1 : e_2$. The let binding expression `let` $x = e_1$ `in` e_2 evaluates into $(\lambda x.e_2)\ e_1$, and we use `let rec` $f\,x = e_1$ `in` e_2 as `let` $f = \mu f.\lambda x.\ e_1$ `in` e_2. An assert expression `assert`(a) is represented as $a\,?\,() : ()$. If users specify any predicates for the input function (usually main function), the parser also generates predefined nodes (definition see Section 2.2.1) and corresponding functions call as an entry of the analysis. Moreover, we encapsulate each intermediate point of a program's execution as a node and bundle the abstract value into that node.

Transformer

Our transformer are doing as follows. At first, we implicitly model the paths of the run-time values' flow as an execution map (definition on Section 2.2.1). The refinement information stored in the undetermined nodes are obtained by using a procedure to propagate the inferred types between nodes. Finally, our transformer repeatedly calculates the inputs - outputs relations until a fix-point is reached. In the end, DRIFT^2 checks the flow information inside the computed execution map, whether each assertion is safe according to the computed types.

Propagation

As a data-flow analysis tool, DRIFT^2 implements flow propagation for updating node's constraints. During the analysis, any abstract value stored at each node is predicted as data propagation between values at consecutive nodes in the execution paths. For example, the Figure. 2.2 describes how the abstract data is propagated between a function call-site and relative definition. Details about this process are discussed in Section 2.2.3.

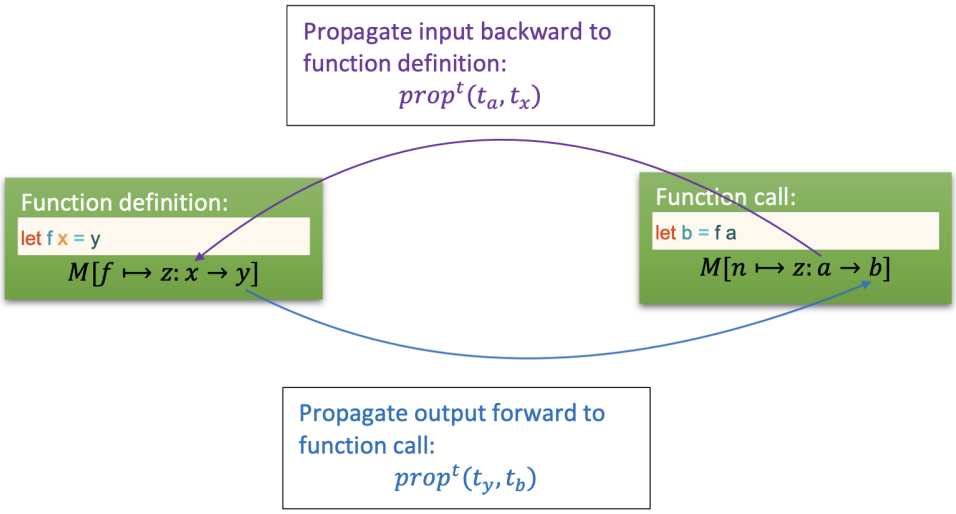

Figure 2.2 Data value propagation

Widening

As a program analysis based on abstract interpretation [17], we use widening and narrowing [22] for enforcing the convergence of the analysis in a finite number of iterations. We currently give four options to the user: widening with/without narrowing, and delay widening with/without narrowing. In particular, if the narrowing approach is permitted, DRIFT^2 will start it right after the widening steps reached the fixpoint. Both procedures ensure termination of the analysis.

Semantic Interface

We present an interface for the program semantics, introduced in Section 2.2.1, which includes several procedures associated with a semantics object. Concretely, the interface matches a set of operators over abstract values, including the most frequently used operators like inclusion $\sqsubseteq$ and equality test, join $\sqcup$ and meet $\sqcap$, and widening ∇ operators. Furthermore, we introduce operations for strengthening the refinement predicates with inferred refinement types of variables in the current environment, as well as for projecting

variables out of scope (details provided in Section 2.2.2).

APRON Interface

This part of design serves as module interface which is composed of a sequence of definitions above relational abstract lattices $\langle \mathcal{A}_X, \sqsubseteq^{\mathsf{a}}, \bot^{\mathsf{a}}, \top^{\mathsf{a}}, \sqcup^{\mathsf{a}}, \sqcap^{\mathsf{a}} \rangle$ (defined in Section 2.2.1). In addition, we construct several methods for testing arithmetic constraints and projecting variables from refinement predicates.

2.1.2 Target Language λ_{D}

Our target language λ_{D} is a call-by-value, higher-order functional variant of the λ-calculus with recursion. Here is the summary of the language syntax:

$$e \in \mathit{Exp} ::= \; c \;\mid\; x \;\mid\; e_1 \; \mathit{op} \; e_2 \;\mid\; \mu f.\lambda x.e \;\mid\; e_1 \; e_2 \;\mid\; e_0 \,?\, e_1 : e_2$$

$$x, f \in \mathit{Var} \quad c \in \mathit{Const} ::= 0, \ldots, \mathit{true}, \mathit{false}, () \quad \mathit{op} \in \mathit{Operator} ::= \; \{+, *, \leq, \&\&, \mathit{mod}, \ldots\}$$

Basically, λ_{D} expressions include constants (integers, Booleans, and void value), variables, arithmetic expressions, if-then-else expressions, (recursive) λ-abstractions, and function applications.

2.2 Theory

In this section, we present a parametric generalization of Liquid types, which we refer to as the *data flow refinement type* semantics. We also describe how the *data flow refinement type* semantics of an expression e can be obtained as a widening sequence of a generic abstract transformer.

2.2.1 Parametric Data Flow Refinement Type Semantics

The details about abstract domain on this semantics is formalized in [16]. Here, we briefly summarize the semantics as follows:

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

$$\hat{\mathcal{N}}_e \stackrel{\text{def}}{=} Loc \times \hat{\mathcal{E}} \qquad \hat{\mathcal{N}}_x \stackrel{\text{def}}{=} \hat{\mathcal{E}} \times Var \times \hat{\mathcal{S}}$$

$$t \in \mathcal{V}^{\mathsf{t}}_X ::= \bot^{\mathsf{t}} \mid \top^{\mathsf{t}} \mid \mathcal{R}^{\mathsf{t}}_X \mid \mathcal{T}^{\mathsf{t}}_X \qquad R^{\mathsf{t}}_X \in \mathcal{R}^{\mathsf{t}}_X \stackrel{\text{def}}{=} \mathsf{B} \times \mathcal{A}_X$$

$$z : T^{\mathsf{t}}_X \in \mathcal{T}^{\mathsf{t}}_X \stackrel{\text{def}}{=} \Sigma z \in Var.\hat{\mathcal{S}} \to \mathcal{V}^{\mathsf{t}}_{X\setminus\{z\}} \times \mathcal{V}^{\mathsf{t}}_{X\cup\{z\}} \qquad M^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}} \stackrel{\text{def}}{=} \Pi\hat{n} \in \hat{\mathcal{N}}.\, \mathcal{V}^{\mathsf{t}}_{X_{\hat{n}}}$$

Semantics Domains

Abstract nodes, stacks, and environments We label (represented as *Loc*) each point of a program's execution, and associate it with an abstract *(execution) node*, $\hat{n} \in \hat{\mathcal{N}}$, which we redesign by using the concept from [23]. We propose two types of the abstract nodes - *expression nodes* $\hat{\mathcal{N}}_e$ and *variable nodes* $\hat{\mathcal{N}}_x$. Here, an expression node is a pair of a label ℓ and an abstract environment $\hat{E}$, denoted $\ell \diamond \hat{E}$, which encodes the execution information from a subexpression e^{ℓ} evaluated under the environment $\hat{E}$. An abstract environment $\hat{E}$, one collection kept a set of all bindings of variables in the scope, is a (finite) partial map association between variables and variable nodes. A variable node, a tuple of a variable x, an environment $\hat{E}$ and abstract stack $\hat{S}$, denoted $x \diamond \hat{E} \diamond \hat{S}$, is created at each execution point where an argument value is bound to a formal parameter x of a function at a call site. The environment $\hat{E}$ attached in the variable node is the environment at the point where the variable/function is defined, and the stack $\hat{S}$ is the *call site stack* that may capture the sequence of call site locations before this variable node was created. We write $\ell \hat{:} \hat{S}$ to denote the stack obtained from $\hat{S}$ by prepending ℓ. The purpose of using call site stacks is to disambiguate various calls to a function based on the function's arguments. We intuitively treat $\hat{S}$ as a parametric construction which allows programmer to define what kinds of information that need to store. For instance, if we select non-sensitive analysis

of a program, the stack $\hat{S}$ will be empty. If we choose 1-context-sensitive to obtain the most recent calling context information, the stack on the variable nodes will only store the call site location. More details about how we abstract stacks in DRIFT^2 can be found in Section 3.1.1.

Abstract values and execution maps There are four classes of abstract (data flow) values $t \in \mathcal{V}^{\mathsf{t}}_X$. First, the value $\bot^{\mathsf{t}}$ stands for nontermination or unreachability of a node, and the value $\top^{\mathsf{t}}$ models every possible abstract value. Second, every base refinement $\mathcal{R}^{\mathsf{t}}_X$ is a pair of a base type b and an abstract refinement relation a (drawn from a family of relational abstract domain $\langle \mathcal{A}_X, \sqsubseteq^{\mathsf{a}}, \bot^{\mathsf{a}}, \top^{\mathsf{a}}, \sqcup^{\mathsf{a}}, \sqcap^{\mathsf{a}} \rangle$ parameterized by abstract scopes X), denoted $\{v : \mathsf{b} \mid a\}$. That is, it describes the refinement type that contains a base type augmented with a concrete representation of refinement relations. Finally, we represent functions as *tables*. A table T^{t}_X maintains an input/output value dependency $T^{\mathsf{t}}(\hat{S}) = \langle t_i, t_o \rangle$ for each abstract call site stack $\hat{S}$. We denote t_i by $\pi_1(T^{\mathsf{t}}(\hat{S}))$ and t_o by $\pi_2(T^{\mathsf{t}}(\hat{S}))$. We use square bracket notation for tables and write $\hat{S} : t_1 \to t_2$ for a table entry that maps call stack $\hat{S}$ to the value pair $\langle t_1, t_2 \rangle$. We denote by $T^{\mathsf{t}}_{\bot^{\mathsf{t}}}$ the empty table that maps every call site stack to the pair $\langle \bot^{\mathsf{t}}, \bot^{\mathsf{t}} \rangle$. Hence, we use $[\hat{S} : t_1 \to t_2]$ as shorthand for the table $T^{\mathsf{t}}_{\bot^{\mathsf{t}}}[\hat{S} : t_1 \to t_2]$. We say that a table T^{t} has been called at a node inducing call site stack $\hat{S}$, denoted $\hat{S} \in T^{\mathsf{t}}$, if the associated input value is not $\bot$: $\pi_1(T^{\mathsf{t}}(\hat{S})) \neq \bot^{\mathsf{t}}$.

At last, the parametric data flow refinement type semantics computes *execution maps* $M^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}}$, which map nodes to values. An *execution map* is the primary structure on which the semantics operates. As we show in Fig.2.6, the analysis of a program evaluates program expressions in given environments and stores the corresponding nodes with the resulting values. We write $M^{\mathsf{t}}_{\bot}$ ($M^{\mathsf{t}}_{\top^{\mathsf{t}}}$) for the execution map that assigns $\bot^{\mathsf{t}}$ ($\top^{\mathsf{t}}$) to every node.

2.2.2 Abstract Domain Specification

In order to express each refinement predicate in refinement type R^{t}_{X}, we briefly introduce several relational numerical abstract domains that are implemented in Apron:

- Octagons [19]: Octagon is a lightweight numerical domain for static analysis over abstract interpretation. Concretely, given a finite set of variables $V_1, V_2, \ldots, V_n$, the octagon domain is a conjunction of constraints that only allows at most two variables to express. That is, the expressed constraints take the form $\pm V_i \pm V_j \leq c_{ij}$ for some V_i, V_j.

- Convex Polyhedra and Linear Equalities [20]: Polyhedra form an expressive numerical domain, which effectively capture precise relations over a set of variables $V_1, V_2, \ldots, V_n$. To express program properties, polyhedra use the following linear constraints: $\forall j \cdot \sum_i a_{ij} \pm V_i \leq b_j$.

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.

By parametric representing refinement predicate, users could easily adjust numerical domains according to the different types of programs for analysis.

Domain Operators

In this part, we briefly describe the most important operators that we used for our transformer. We define:

- strengthening: Liquid type inference introduced an environment strengthened operation that replaces all occurrences of the value variable ν embedded from the environment with the actual variable being refined [1], thereby we structurally define a strengthening operations over refinement types for our semantics. For in-

stance, given refinement values $t, t' \in \mathcal{V}_X^{\mathsf{t}}$ and a program variable x, the operation $t[x \leftarrow t'] = t \sqcap^{\mathsf{t}} t'[x/v]$ is used for strengthening the refinement type t with respect to the replacement of v by the value x in t'. Besides, we use the operation $t[v{=}x] = t \sqcap^{\mathsf{t}} \{v = x\}$ to strengthen t with an equality constraint $v = x$.

- projection: This operator is used frequently after each propagation is performed. This process ensures that variable scoping in a refinement constraint should not messed up the others, and it guarantees that the types inferred by our analysis to strictly follow the scoping rules of OCaml. In the following section, we use $\boldsymbol{proj}_X t$ to denote this rescoping operation, which removes all constraints of variables that not belong to the given environment set $\hat{E}$. Note that we replace $\hat{E}$ by a set of variables X here (i.e. $X = dom(\hat{E})$).

2.2.3 Value Propagation

The propagation process (shown in Fig. 2.3) is the key feature of our transformer. Function $\mathsf{prop}^{\mathsf{t}}$ takes two abstract values as input and propagates information between them. On the upper level, operation $\ltimes^{\mathsf{t}}$ computes the new abstract values over the least common scopes by extending the scope of an abstract value if necessary. Then it calls $\mathsf{prop}^{\mathsf{t}}$ for the value propagation. Finally, it uses projection operator to restore the scope of each result, which ensures the scopes of a value remain compatible.

The most interesting case is the propagation between two function types $z : T_1^{\mathsf{t}}$ and $z : T_2^{\mathsf{t}}$. At the top level, $z : T_2^{\mathsf{t}}$ usually represents a function's call-site, and $z : T_1^{\mathsf{t}}$ relative to the definition of that function. In this occasion, we use the strengthening operator to capture the dependency of the output values t_{1o} on the input value t_{2i} during propagation.

Note that the notation of X_t refers to the set of variables (scope) over which t ranges. Procedure ***lc_env*** calculates the least common scope of the two input abstract values. Function ***ext*** extends the scope of each abstract value to the given least common scope X.

$$
\begin{array}{l|l}
\mathsf{prop}^{\mathsf{t}}(z : T_1^{\mathsf{t}}, z : T_2^{\mathsf{t}}) \stackrel{\text{def}}{=} & \\
\quad \textbf{let}\ T^{\mathsf{t}} = \Lambda \hat{S}. & \\
\quad\quad \textbf{let}\ \langle t_{1i}, t_{1o} \rangle = T_1^{\mathsf{t}}(\hat{S})\textbf{;}\ \langle t_{2i}, t_{2o} \rangle = T_2^{\mathsf{t}}(\hat{S}) & \mathsf{prop}^{\mathsf{t}}(T^{\mathsf{t}}, \bot^{\mathsf{t}}) \stackrel{\text{def}}{=} \langle T^{\mathsf{t}}, T_{\bot}^{\mathsf{t}} \rangle \\
\quad\quad \langle t'_{2i}, t'_{1i} \rangle = \mathsf{prop}^{\mathsf{t}}(t_{2i}, t_{1i}) & \mathsf{prop}^{\mathsf{t}}(T^{\mathsf{t}}, \top^{\mathsf{t}}) \stackrel{\text{def}}{=} \langle \top^{\mathsf{t}}, \top^{\mathsf{t}} \rangle \\
\quad\quad \langle t'_{1o}, t'_{2o} \rangle = \mathsf{prop}^{\mathsf{t}}(t_{1o}[z \leftarrow t_{2i}], t_{2o}[z \leftarrow t_{2i}]) & \mathsf{prop}^{\mathsf{t}}(t_1, t_2) \stackrel{\text{def}}{=} \langle t_1, t_1 \sqcup^{\mathsf{t}} t_2 \rangle \\
\quad \textbf{in}\ \langle \langle t'_{1i}, t_{1o} \sqcup^{\mathsf{t}} t'_{1o} \rangle, \langle t'_{2i}, t_{2o} \sqcup^{\mathsf{t}} t'_{2o} \rangle \rangle & \qquad\qquad \textbf{(otherwise)} \\
\textbf{in}\ \langle z : \Lambda \hat{S}.\ \pi_1(T^{\mathsf{t}}(\hat{S})), z : \Lambda \hat{S}.\ \pi_2(T^{\mathsf{t}}(\hat{S})) \rangle &
\end{array}
$$

$$
\begin{array}{l}
t_1 \ltimes^{\mathsf{t}} t_2 \stackrel{\text{def}}{=} \\
\quad \textbf{let}\ X = \textit{lc\_env}(X_{t_1}, X_{t_2})\ \ \textbf{in} \\
\quad \textbf{let}\ t'_1, t'_2 = \textit{ext}_X(t_1, t_2)\ \ \textbf{in} \\
\quad \textbf{let}\ \langle t''_1, t''_2 \rangle = \mathsf{prop}^{\mathsf{t}}(t'_1, t'_2)\ \ \textbf{in} \\
\quad \langle \textit{proj}_{X_{t_1}} t''_1, \textit{proj}_{X_{t_2}} t''_2 \rangle
\end{array}
$$

Figure 2.3 Abstract value propagation in the refinement type semantics

2.2.4 Abstract Transformer

The (monadic) abstract transformer of the parametric data flow refinement type semantics is shown in Fig. 2.4 (the syntax of monad is given in Figure 2.5). The signature of our transformer is as follows:

$$\mathsf{step}^{\mathsf{t}} : \mathit{Exp} \to \mathcal{E}^{\mathsf{t}} \to \hat{\mathcal{S}} \to \mathcal{V}_X^{\mathsf{t}} \to \mathcal{M}^{\mathsf{t}} \to \mathcal{M}^{\mathsf{t}}$$

Concretely, when the parser interpreted the input program into the syntax of our target language λ_{D}. $\mathsf{step}^{\mathsf{t}}$ takes that expression as e along with an initial environment $\hat{E}$, an empty abstract stack $\hat{S}$, an unconstrained abstract value $t_{\text{æ}}$, and an initial execution map M^{t}. After the analysis completes one iteration, $\mathsf{step}^{\mathsf{t}}$ returns an (updated) execution map M^{t}. The definition of our transformer is formalized inductively over the structure of e.

The analysis takes into account the control flow of the analyzed program by strengthening refinement predicates with information obtained from tests in conditional branches.

For this purpose, we introduce an abstract data value $t_{æ}$ as an additional parameters of the transformer, initialized to $\top^{\mathsf{t}}$. This parameter accumulates the information collected along the current control flow path to strengthen each refinement predicate on that path. For instance, after evaluating the conditional tests in the *if-then-else* expression, the results should be strengthened to each branch path so that the analysis takes all possible paths into account to over-approximate the behavior of the program. If a path is infeasible, all abstract values associated with nodes on that path are set to $\bot^{\mathsf{t}}$.

2.2.5 Transformer Notation and Explanation

To reduce notational clutter when presenting the abstract transformer, we use a state monad to compose the abstract transformer from simpler functions of type $M^{\mathsf{t}} \rightarrow M^{\mathsf{t}} \times t$. The definitions of the relevant monad transformers are found in Figure 2.5 using syntax inspired by Haskell's monad syntax.

We introduce several notations that we use in the abstract transformer. We denote the current binding of $\hat{n}$ in M^{t} (which is the abstract value t) by notation $t = M^{\mathsf{t}}(\hat{n})$ and $M^{\mathsf{t}}[\hat{n} \mapsto t]$ as an updates of execution map M^{t} at node $\hat{n}$ with a new value t. We write $z : [\hat{S} : t \rightarrow t']$ for a temporary table that maps abstract call stack $\hat{S}$ to the abstract value pair $\langle t, t' \rangle$, where z is a dependency variable to express relations between input t and output t'. We also denote $Bool^t$ as an unconstrained base refinement type over a Boolean base type. Other domain operators could be referred from the Section 2.2.2.

We now explain the abstract transformer in detail:

Constant $e = c_\ell$**.** At this step, we firstly create an constraint $[v = c]^{\mathsf{t}}$, abbreviated to $\{v : type(c) \mid v = c\}$ where $type(c)$ means the base type of the constant c, and strengthen that constraint with the flowed path information. Then we join the result with the original stored abstract value in the current node.

Variable $e = x_\ell$**.** Here, our transformer computes the value propagation between the

$\mathsf{step}^{\mathsf{t}}[\![c_\ell]\!](\hat{E},\hat{S},t_{æ}) \stackrel{\text{def}}{=}$

do $\hat{n} = \ell\diamond\hat{E}$ **;** $t' \leftarrow \hat{n} := [\nu = c]^{\mathsf{t}} \sqcap^{\mathsf{t}} t_{æ}$

return t'

$\mathsf{step}^{\mathsf{t}}[\![x_\ell]\!](\hat{E},\hat{S},t_{æ}) \stackrel{\text{def}}{=}$

do $\hat{n} = \ell\diamond\hat{E}$ **;** $\hat{n}_x = \hat{E}(x)$ **;** $t \leftarrow !\hat{n}$**;** $\Gamma \leftarrow \mathbf{env}(\hat{E})$

$t_x = \Gamma(x)[\nu = x]^{\mathsf{t}} \sqcap^{\mathsf{t}} t_{æ}$ **;** $t = t[\nu = x]^{\mathsf{t}} \sqcap^{\mathsf{t}} t_{æ}$

$t' \leftarrow \hat{n}_x,\ \hat{n} := t_x \ltimes^{\mathsf{t}} t$

return t'

$\mathsf{step}^{\mathsf{t}}[\![(e_i\ e_j)_\ell]\!](\hat{E},\hat{S},t_{æ}) \stackrel{\text{def}}{=}$

do $\hat{n} = \ell\diamond\hat{E}$ **;** $\hat{n}_i = i\diamond\hat{E}$ **;** $\hat{n}_j = j\diamond\hat{E}$ **;** $t \leftarrow !\hat{n}$

$t_i \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_i]\!](\hat{E},\hat{S},t_{æ})$ **; assert**$(t_i \in \mathcal{T}^{\mathsf{t}})$

$t_j \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_j]\!](\hat{E},\hat{S},t_{æ})$

$t'_i,\ z : [i \hat{:} \hat{S} : t'_j \to t'] = t_i \ltimes^{\mathsf{t}} z : [i \hat{:} \hat{S} : t_j \to t]$

$t'' \leftarrow \hat{n}_i,\ \hat{n}_j,\ \hat{n} := t'_i,\ t'_j,\ t'$

return t''

$\mathsf{step}^{\mathsf{t}}[\![(e_i\ ?\ e_j : e_k)_\ell]\!](\hat{E},\hat{S},t_{æ}) \stackrel{\text{def}}{=}$

do $\hat{n} = \ell\diamond\hat{E}$ **;** $\hat{n}_i = i\diamond\hat{E}$ **;** $\hat{n}_j = j\diamond\hat{E}$ **;** $\hat{n}_k = k\diamond\hat{E}$ **;** $t \leftarrow !\hat{n}$

$t_i \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_i]\!](\hat{E},\hat{S},t_{æ})$ **; assert**$(t_i \sqsubseteq^{t} Bool^{t})$

$t_{true},\ t_{false} = t_0[true^{\mathsf{t}}] \sqcap^{\mathsf{t}} t_{æ},\ t_0[false^{\mathsf{t}}] \sqcap^{\mathsf{t}} t_{æ}$

$t_j \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_j]\!](\hat{E},\hat{S},t_{true})$

$t_k \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_k]\!](\hat{E},\hat{S},t_{false})$

$t'_j,\ t' = t_j \ltimes^{\mathsf{t}} t$**;** $t'_k,\ t'' = t_k \ltimes^{\mathsf{t}} t$ **in**

$t''' \leftarrow \hat{n}_i,\ \hat{n}_j,\ \hat{n}_k,\ \hat{n} := t'_i,\ t'_j,\ t'_k,\ t' \sqcup^{\mathsf{t}} t''$

return t'''

$\mathsf{step}^{\mathsf{t}}[\![(e_i\ op\ e_j)_\ell]\!](\hat{E},\hat{S},t_{æ}) \stackrel{\text{def}}{=}$

do $\hat{n} = \ell\diamond\hat{E}$ **;** $\hat{n}_i = i\diamond\hat{E}$ **;** $\hat{n}_j = j\diamond\hat{E}$ **;** $t \leftarrow !\hat{n}$

$t_i \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_i]\!](\hat{E},\hat{S},t_{æ})$

$t_j \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_j]\!](\hat{E},\hat{S},t_{æ})$

$t_{op} = t^{\mathsf{t}}_{\top}[\nu = \hat{n}_i\ op\ \hat{n}_j][\hat{n}_i \leftarrow t_i][\hat{n}_j \leftarrow t_j]$

$\_,\ t' = t_{op} \ltimes^{\mathsf{t}} t$

$t'' \leftarrow \hat{n} := t' \sqcap^{\mathsf{t}} t_{æ}$

return t''

$\mathsf{step}^{\mathsf{t}}[\![(\mu f.\lambda x.e_i)_\ell]\!](\hat{E},\hat{S},t_{æ}) \stackrel{\text{def}}{=}$

do $\hat{n} = \ell\diamond\hat{E}$ **;** $t \leftarrow \hat{n} := x : T^{\mathsf{t}}_{\bot}$

$t' \leftarrow$ **for** $\hat{S}' \in t$ **do** $\mathsf{body}^{\mathsf{t}}(f, x, e_i, \hat{n}, \hat{E}, t_{æ}, t)$

$t'' \leftarrow \hat{n} := t'$

return t''

$\mathsf{body}^{\mathsf{t}}(f, x, e_i, \hat{n}, \hat{E}, t_{æ}, t)(\hat{S}') \stackrel{\text{def}}{=}$

do $\hat{n}_x = x\diamond\hat{E}\diamond\hat{S}'$ **;** $\hat{n}_f = f\diamond\hat{E}\diamond\hat{S}'$

$\hat{E}_i = \hat{E}.x : \hat{n}_x\ , f : \hat{n}_f$ **;** $\hat{n}_i = i\diamond\hat{E}_i$

$t_x \leftarrow !\hat{n}_x$ **;** $t_f \leftarrow !\hat{n}_f$

$t'_{æ} = t_{æ}[x \leftarrow t_x]$**;** $t_i \leftarrow \mathsf{step}^{\mathsf{t}}[\![e_i]\!](\hat{E}_i, \hat{S}', t'_{æ})$

$x : [\hat{S}' : t'_x \to t'_i], t' = x : [\hat{S}' : t_x \to t_i] \ltimes^{\mathsf{t}} t(\hat{S}')$

$t'', t'_f = t(\hat{S}') \ltimes^{\mathsf{t}} t_f$

$\_ \leftarrow \hat{n}_x,\ \hat{n}_f\ \hat{n}_i := t'_x,\ t'_f,\ t'_i$

return $t' \sqcup^{\mathsf{t}} t''$

Figure 2.4 Abstract transformer for data flow refinement semantics

$$
\begin{aligned}
!\hat{n} &\stackrel{\text{def}}{=} \Lambda M^{\mathsf{t}}.\, \langle M^{\mathsf{t}}, M^{\mathsf{t}}(\hat{n})\rangle \\
\hat{n} := t &\stackrel{\text{def}}{=} \Lambda M^{\mathsf{t}}.\, \textbf{let}\, t' = M^{\mathsf{t}}(\hat{n}) \sqcup^{\mathsf{t}} t\, \textbf{in}\, \langle M^{\mathsf{t}}[\hat{n} \mapsto t'], t'\rangle \\
\textbf{env}(\hat{E}) &\stackrel{\text{def}}{=} \Lambda M^{\mathsf{t}}.\, \langle M^{\mathsf{t}}, M^{\mathsf{t}} \circ \hat{E}\rangle \\
\textbf{for}\ \hat{S} \in T^{\mathsf{t}}\ \textbf{do}\ F &\stackrel{\text{def}}{=} \Lambda M^{\mathsf{t}}.\, \dot{\bigsqcup}^{\mathsf{t}}_{\hat{S}\in T^{\mathsf{t}}} F(\hat{S})(M^{\mathsf{t}}) \\
[\![\textbf{do}\,[]\textbf{return}\, t]\!] &\stackrel{\text{def}}{=} \Lambda M^{\mathsf{t}}.\, \langle M^{\mathsf{t}}, t\rangle \\
[\![\textbf{do}\, v \leftarrow F; bs\, \textbf{in}\, t]\!] &\stackrel{\text{def}}{=} \Lambda M^{\mathsf{t}}.\, \textbf{let}\, \langle M^{\mathsf{t}'}, u\rangle = F(M^{\mathsf{t}})\textbf{in} \\
&\qquad\qquad \textbf{if}\, M^{\mathsf{t}'} = M^{\mathsf{t}}_{\top}\, \textbf{then}\, \langle M^{\mathsf{t}}_{\top}, \top^{\mathsf{t}}\rangle\, \textbf{else}\, [\![\textbf{do}\, bs\, \textbf{return}\, t[u/v]]\!](M^{\mathsf{t}'}) \\
[\![\textbf{do}\, v = F; bs\, \textbf{return}\, t]\!] &\stackrel{\text{def}}{=} \Lambda M^{\mathsf{t}}.\, \textbf{let}\, \langle M^{\mathsf{t}'}, u\rangle = F\, \textbf{in}\, [\![\textbf{do}\, bs\, \textbf{return}\, t[u/v]]\!](M^{\mathsf{t}}) \\
[\![\textbf{do}\, \textbf{assert}(P); bs\, \textbf{in}\, t]\!] &\stackrel{\text{def}}{=} \Lambda M^{\mathsf{t}}.\, \textbf{if}\, P\, \textbf{then}\, [\![\textbf{do}\, bs\, \textbf{return}\, t]\!](M^{\mathsf{t}})\, \textbf{else}\, \langle M^{\mathsf{t}}_{\top}, \top^{\mathsf{t}}\rangle
\end{aligned}
$$

Figure 2.5 Monad transformers for abstract state machine

abstract variable node $\hat{n}_x$ binding x and the current expression node $\hat{n}$ where x is used. The values t and t_x represent binding of those nodes respectively. The computation is achieved by using propagation procedure defined in Fig. 2.3. The function $\ltimes^{\mathsf{t}}$ takes these values as input and propagates information between them. After propagation, we update the bindings to the new values t'_x and t' into M^{t}.

Concretely, if t_x is a base refinement type contained refinement constraints and t is still $\bot^{\mathsf{t}}$, then we simply unions t_x with t and remains t_x unchanged. Of greater significance here is that when the variable x represents a function. The propagation will performs data flow under tables. Thus, we follows the idea from figure 2.2 where input information in the table t flow backward to t_x, together with outputs depended on these inputs flowing forward from t_x to t. In case that t is still $\bot^{\mathsf{t}}$ because no inputs accumulated here, we initialize that value as an empty table $T^{\mathsf{t}}_{\bot^{\mathsf{t}}}$ and leave t_x unchanged. Otherwise, for the condition that both t_x and t are tables, we propagate inputs and outputs by calling $\ltimes^{\mathsf{t}}$

recursively for every call site $\hat{S}_{\mathsf{cs}} \in t$. The recursive call for the propagation of the inputs inverts the direction because we require to update the information at t_x.

Function application $e = (e_i\, e_j)_\ell$**.** In this case of the expression, we firstly take a transition $step$ for evaluating e_i and extract the binding value t_i stored at the corresponding expression node $\hat{n}_i$ from the updated map M_i^{t}. If t_i is not a table, which means the base type of e_i is not a function, then we return the error map M_ω^{t} for this unsafe call. If t_i is a table, we then evaluate e_j to obtain the new map M_j^{t} and abstract value t_j at the associated expression node $\hat{n}_j$. We omitted conditions for checking whether t_j is neither $\bot^{\mathsf{t}}$ nor $\top^{\mathsf{t}}$ on the transformer. However, if so, we returns M_j^{t} if t_j is $\bot^{\mathsf{t}}$, and updates M_j^{t} for t to $\top^{\mathsf{t}}$ if t_j is $\top^{\mathsf{t}}$. Otherwise, we need to propagate the information between t, t_j and t_i, where input value for the call is t_j and return value is t. To do this, we construct a temporary table as $z : [i \hat{:} \hat{S} : t_j \rightarrow t]$, where $i \hat{:} \hat{S}$ is the concatenation of the parameter $\hat{S}$, whereby the extended stack will be the representation of function call. Then we propagate information between this table and call site value t_i. Clearly, we propagate the input information t_j backward to the input of t_i and the output of t_j forward to t. That is, after the propagation, t_i' encloses the information the information flowed from the input argument. Once the call site $\hat{n}_i$ stores the evaluated outputs from the function definition, the result information will finally propagate advance to the result expression node which is $\hat{n}$.

If-then-else $e = (e_i \,?\, e_j : e_k)_\ell$**.** We compute to get the updated map M_i^{t} which stores the value t_i associated with the expression node $\hat{n}_i$. If t_i is unreachable for now ($t_i = \bot^{\mathsf{t}}$), we return M_i^{t}. We return an error map $M_{\top^{\mathsf{t}}}^{\mathsf{t}}$ if t_i is not a Boolean refinement type where we use $Bool^t$ for representing supremum over Boolean refinement type. Then we evaluate the then-branch to obtain the value t_j over expression node $\hat{n}_j$ by strengthening the path of flowed data $t_{\text{æ}}$ with true part of constraints over t_i. The else branch is handled analogously. For $t_{\text{æ}}$, the strengthening process flows path constraints along with the execution path, so the extracted values t_{true} and t_{false} are based on which branch we evaluate. Finally, we respectively propagate the information between the expression node $\hat{n}$ and each branch

node. The result value over if-then-else expression node $\hat{n}$ is the join of the propagated values t', t'' from the above two computations.

Binary operation $e = (e_i \; op \; e_j)_\ell$**.** Here, we evaluate each operand to obtain the latest value t_i and t_j at the corresponding expression node $\hat{n}_i$ and $\hat{n}_j$. Next, we aggregate the values t_i, t_j from the two operands node $\hat{n}_i, \hat{n}_j$, plus a constraints for operation, which establishes the mathematical relation between two operands. Finally, we propagate the accumulated value (represented as t_{op}) forward to the abstract value t associated with the expression node $\hat{n}$. t' contains the abstract value obtained by abstractly evaluating op on t_i and t_j.

(Recursive) function $e = (\mu f.\lambda x.e_i)_\ell$**.** For the function expression node $\hat{n}$, we first look up the stored table T^{t}, denoted by an abstract value t in the abstract transformer, representing the current approximation of the function represented by e, and then walk through each abstract call stack $\hat{S}'$ where an input has already been back-propagated to T^{t} (if the input part of the table is $\bot^{\mathsf{t}}$, return M^{t}). Here, for each call site over table T^{t}, we perform several computations as follows. At first, we still report $M^{\mathsf{t}}_{\top^{\mathsf{t}}}$ if the stored call site information of the table $T^{\mathsf{t}}(\hat{S})$ is $\top^{\mathsf{t}}$. Otherwise, based on the call site $\hat{S}'$, we first create a variable node n_x for storing parameter value t_x and another variable node n_f for storing recursive information t_f if the function is recursive. Note that, the value bound to the node n_x might be a table if e is a high-order function. In addition, we add bindings for parameter variable x and the variable f for recursive call to e into the given environment $\hat{E}$. We also updates the path information $t_{æ}$ by pushing the value t_x over the variable node n_x, unless the value is a base refinement type. Note those defined nodes are unique in reference to $\hat{S}'$.

After those preliminaries, we propagate information between the values, combined by creating a singleton table over call site $\hat{S}'$, stored at the node n_x for the input parameter and at the node $\hat{n}_i$ for the body e_i, and the table T^{t}. Concretely, if the input value, which is bound to the argument of the expression node n at $\hat{S}'$, contains information, we propagate that backward to the corresponding environment node n_x. This captures how a function

receives a parameter from its caller. At the same time, if the body value $t_i = M^{\mathsf{t}}(\hat{n}_i)$ exists, we propagate that value forward to the output of T^{t} at $\hat{S}'$. We also computes t_i by calling $\mathsf{step}^{\mathsf{t}}$ with an updated environment $\hat{E}_i$, an abstract call site stack $\hat{S}'$, and a augmented flow value $t'_{\text{æ}}$, which represents the abstract value of the body e_i.

Furthermore, the final propagation performs interchange of information between the table T^{t} and the table stored at recursive call site node $\hat{n}_f$. Concretely, the information of recursive call propagates backward to the input of T^{t}, whereas the output of T^{t} at $\hat{S}_{\mathsf{cs}}$ flows forward to the recursive call. The last propagation allows T^{t} to pick up inputs coming from recursive calls to $\hat{n}_f$ and evaluate them in the next iterations. This step is omitted if the function is non-recursive.

After these two propagation steps, we store each updated value back to the given map M^{t} and obtain a new map M_1^{t}. Particularly, we obtained two values t, t' for the table T^{t}, which ultimately performs a relational join over these two new values before updating the binding. Overall, the updated maps obtained for all call sites are then joined into a single map.

2.2.6 Widening and Abstract Semantics

Widening To establish the convergence of the fix-point iteration for the type analysis, we need a widening operator for the domain of refinement types $\mathcal{V}_X^{\mathsf{t}}$. We construct such an operator from the widening operator $\triangledown_X^{\mathsf{a}}$ on the domains of refinement relations and a *shape widening* operator. In order to define the latter, we first define the *shape* of a type using the function $\mathsf{sh} : \mathcal{V}_X^{\mathsf{t}} \to \mathcal{V}_X^{\mathsf{t}}$

$$\mathsf{sh}(\bot^{\mathsf{t}}) \stackrel{\text{def}}{=} \bot^{\mathsf{t}} \quad \mathsf{sh}(\top^{\mathsf{t}}) \stackrel{\text{def}}{=} \top^{\mathsf{t}} \quad \mathsf{sh}(R^{\mathsf{t}}) \stackrel{\text{def}}{=} \bot^{\mathsf{t}} \quad \mathsf{sh}(z : T^{\mathsf{t}}) \stackrel{\text{def}}{=} z : \Lambda\hat{S}.\, \langle \mathsf{sh}(\pi_1(T^{\mathsf{t}}(\hat{S}))), \mathsf{sh}(\pi_2(T^{\mathsf{t}}(\hat{S}))) \rangle$$

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

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

$$t \nabla_X^{\mathsf{ra}} t' \stackrel{\text{def}}{=} \begin{cases} \{v : \mathsf{b} \mid a \nabla_X^{\mathsf{a}} a'\} & \textbf{if } t = \{v : \mathsf{b} \mid a\} \wedge t' = \{v : \mathsf{b} \mid a'\} \\ z : \Lambda \hat{S}.\ \textbf{let } \langle t_i, t_o \rangle = T^{\mathsf{t}}(\hat{S}) \textbf{; } \langle t_i', t_o' \rangle = T^{\mathsf{t}'}(\hat{S}) & \textbf{if } t = z : T^{\mathsf{t}} \wedge t' = z : T^{\mathsf{t}'} \\ \qquad \textbf{in } \langle t_i \nabla_{X \setminus \{z\}}^{\mathsf{ra}} t_i', t_o \nabla_{X \cup \{z\}}^{\mathsf{ra}} t_o' \rangle & \\ t \sqcup_X^{\mathsf{t}} t' & \textbf{otherwise} \end{cases}$$

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

Abstract semantics We now define the abstract semantics $\mathbf{S}^{\mathsf{t}}[\![e]\!]$ of a program e as the least fixpoint of the widened iterates of $\mathsf{step}^{\mathsf{t}}$ over the complete lattice of execution maps:

$$\mathbf{S}^{\mathsf{t}}[\![e]\!] \stackrel{\text{def}}{=} \mathbf{lfp}_{M_\perp^{\mathsf{t}}}^{\dot{\sqsubseteq}^{\mathsf{t}}} \Lambda M^{\mathsf{t}}.\ (M^{\mathsf{t}} \mathbin{\dot{\nabla}^{\mathsf{t}}} \mathsf{step}^{\mathsf{t}}[\![e]\!](\hat{E}_\epsilon)(\hat{S}_\epsilon)(\top^{\mathsf{t}})(M^{\mathsf{t}}))$$

Here, $\hat{E}_\epsilon$ is the empty environment and $\hat{S}_\epsilon$ denotes the empty abstract stack.

2.2.7 Example

We now explain the analysis through an example. Consider the program in Fig. 2.6. Our data flow refinement type analysis infers data flow invariants represented by execution maps. During the program execution, we inherently know that function `id` takes two individual data flow paths. Therefore, at each call-site, our semantics may trace the context information by storing input and output values separately. Recording the calling context can be parameterized as an abstract stack $\hat{S}$ inbuilt in the table semantics, which improves precision of the analysis.

In particular, for context-insensitive version, we integrate call-site information into a single function node. The resulting table for function `id` is shown on the left-hand side of Fig. 2.6. The information from two calls at location a and d has been collapsed. In this version, the tables only contain a single entry for all call sites. For the 1-context-sensitive

```
1 let id x = x in
2 let _ = assert((id^a 1^b)^c = 1) in
3 assert((id^d 2^e)^f = 2)
```

context-insensitive

$$\begin{aligned}
&\texttt{id} \mapsto [z : \{v : Int \mid v \geq 1; v \leq 2\} \\
&\qquad \rightarrow \{v : Int \mid v = z; v \geq 1; v \leq 2\}] \\
&a \mapsto [z : \{v : Int \mid v = 1\} \\
&\qquad \rightarrow \{v : Int \mid v = z; v = 1\}] \\
&d \mapsto [z : \{v : Int \mid v = 2\} \\
&\qquad \rightarrow \{v : Int \mid v = z; v = 2\}] \\
&b, c \mapsto \{v : Int \mid v = 1\} \\
&e, f \mapsto \{v : Int \mid v = 2\}
\end{aligned}$$

1-context-sensitive

$$\begin{aligned}
&\texttt{id} \mapsto [z_a : \{v : Int \mid v = 1\} \\
&\qquad \rightarrow \{v : Int \mid v = z_a; v = 1\}; \\
&\quad z_d : \{v : Int \mid v = 2\} \\
&\qquad \rightarrow \{v : Int \mid v = z_d; v = 2\}] \\
&a \mapsto [z_a : \{v : Int \mid v = 1\} \\
&\qquad \rightarrow \{v : Int \mid v = z_a; v = 1\}] \\
&d \mapsto [z_d : \{v : Int \mid v = 2\} \\
&\qquad \rightarrow \{v : Int \mid v = z_d; v = 2\}] \\
&b, c \mapsto \{v : Int \mid v = 1\} \\
&e, f \mapsto \{v : Int \mid v = 2\}
\end{aligned}$$

Figure 2.6 Program 1 and portion of its execution map obtained with the Polyhedra domain for a context-insensitive and 1-context-sensitive analysis.

version (see the right-hand side of Fig. 2.6), the table of function `id` captures input-output data based on each of these two paths. At this time, the tables store flow information separately for each the call site location. Like the example we given, the table of function `id` stored at locations a and d now contains information for each of the two call sites $\texttt{id}^a$ and $\texttt{id}^b$ in the program.

Chapter Three

DRIFT^2 & Experiments

In this chapter, we firstly discuss important aspect of DRIFT^2's implementation. We present the specific instantiations of our parametric data flow refinement type semantics that we implemented in DRIFT^2. In particular, we consider two specific choices for the representation of abstract stacks $\hat{S}$ which control the level of context sensitivity. The first choice yields a context-insensitive analysis that works similarly to Liquid type inference. The second choice tracks the most recent call site in $\hat{S}$, yielding a 1-context-sensitive analysis. Section 3.1.1 gives implementation-level details about these choices.

Next, we present performance evaluation by testing DRIFT^2 on OCaml programs. Our evaluation consists of two parts. The first part shows the results of an experiment that compares different configurations of DRIFT^2 in terms of the selecting abstract domains of type refinements, choices of widening/narrowing strategies, and levels of context sensitivities. In the second part, we compare DRIFT^2 with four state-of-the-art verification tools for OCaml programs: R_Type, DOrder, DSolve and MoCHI.

3.1 Implementation Details

Our implementation is composed of two parts. First, we utilized the Apron library [21] to implement various abstract domains for expressing basic refinement types, and pro-

vided utilities for the type semantics to manipulate and filter the propagated type information. Second, we implemented two versions of the abstract transformer for the context-insensitive and 1-context-sensitive analysis.

3.1.1 Syntax and Semantics Over Implementation

We first introduce how DRIFT^2 represents the set of constants of λ_{D}. We also give a special base type that encodes an integer array in λ_{D}.

In the remaining parts, we show how we act for the abstract stack $\hat{S}$ during the implementation of the data flow refinement type semantics. For now, we only give two well-defined semantics over implementation stage. Note that, the most notations (see Section 2.2.1) are inherited from the parametric data flow refinement type semantics.

Constants Built Into λ_{D}

On the implementation stage, we include three *basic types of constants*: integers, Boolean and unit values, and several predefined functions which encode several array operations. The semantics of each constant c belonging to a unique base type should be captured

precisely. In λ_{D}, we gives a set of constants as:

$$
\begin{aligned}
\texttt{1} &: \{v : Int \mid v = 1\} \\
\texttt{true} &: \{v : Bool \mid \texttt{TRUE:}v = 1 \ , \ \texttt{FALSE:}\bot^{\mathsf{a}}\} \\
\texttt{false} &: \{v : Bool \mid \texttt{TRUE:}\bot^{\mathsf{a}} \ , \ \texttt{FALSE:}v = 0\} \\
\texttt{()} &: Unit \\
\texttt{Array.make} &: zm\colon \{v : Int \mid v \geq 0\} \rightarrow ex\colon \{v : Int \mid true\} \\
&\qquad \rightarrow \{v : Int\ Array\ (l) \mid l \geq 0\ ;\ zm = l;\} \\
\texttt{Array.length} &: zl\colon \{v : Int\ Array\ (l) \mid l \geq 0;\} \rightarrow \{v : Int \mid v = l\} \\
\texttt{Array.get} &: zg\colon \{v : Int\ Array\ (l) \mid l \geq 0;\} \rightarrow zi\colon \{v : Int \mid v \geq 0\ ;\ v < l\} \\
&\qquad \rightarrow \{v : Int \mid true\} \\
\texttt{Array.set} &: zs\colon \{v : Int\ Array\ (l) \mid l \geq 0;\} \rightarrow zi\colon \{v : Int \mid v \geq 0\ ;\ v < l\} \\
&\qquad \rightarrow ex\colon \{v : Int \mid true\} \rightarrow Unit
\end{aligned}
$$

Note that, we present Boolean constant as an overappoximation of *true* and *false* parts because not all abstract domains allow the analysis to perform a negation operation over linear constraints precisely. Besides, we construct an array by using `Array.make` operation which takes a positive value zm as the length of that array and an initial value ex for each element of the array. `Array.length` requires an input array zl and returns the length of the given array. To access the elements of the array, we use `Array.get` and `Array.set`. Both operations require the query index zi to be within the bounds of the array.

Context-insensitive Semantic Domains

For the context insensitive program analysis, the function types do not differentiate between calling context, which means different call sites calling the same function refer to the same dataflow fact. Based on our semantics, the table only contains a single entry.

Thus, the semantic domains are simplified as follows:

- We remove the representation of abstract stack $\hat{S}$ and related constructions because all call-sites for a function (including recursive calls) will be collapsed into a single table entry. That is, the table type of the revised semantics is

$$z : T_X^{\mathsf{t}} \in \mathcal{T}_X^{\mathsf{t}} \stackrel{\text{def}}{=} \Sigma z \in Var. \mathcal{V}_{X\setminus\{z\}}^{\mathsf{t}} \times \mathcal{V}_{X\cup\{z\}}^{\mathsf{t}}$$

- We simplify expression nodes to $\hat{n}_s \in \hat{\mathcal{N}}_s \stackrel{\text{def}}{=} Loc$ and execution maps to $M^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}} \stackrel{\text{def}}{=} \Pi \hat{n}_s \in \hat{\mathcal{N}}_s.\ \mathcal{V}_{X_{\hat{n}_s}}^{\mathsf{t}}$. Note that, during the analysis, we pass an environment map $\hat{E}$ as a parameter (see Sect. 2.2.4), so the concise representation of a node $\hat{n}_s$ on the execution map M^{t} is equivalent to the node $\hat{n}$ regardless of the environment map $\hat{E}$.

1-Context-Sensitive Semantic Domains

For the 1-context-sensitive program analysis, the abstract stacks $\hat{\mathcal{S}}$ only record the most recent call-site. That is, we define $\hat{\mathcal{S}} \stackrel{\text{def}}{=} Loc$. The table type is now containing the calling context by recording the most recent call-site location. This is a significant changes that improves precision of the analysis. In the actual implementation, we further simplified the representation of nodes $\hat{n}$ by omitting environment $\hat{E}$ in a similar way as for the context-insensitive analysis.

Summary

Based on the two semantics we instantiated, the abstract nodes ($\hat{\mathcal{N}}$ & $\hat{\mathcal{N}}_s$) and the abstract values $\mathcal{V}_X^{\mathsf{t}}$ are defined by appropriate OCaml datatype. Both execution maps $\mathcal{M}^{\mathsf{t}}$ and environment maps $\hat{\mathcal{E}}$ are implemented using OCaml's map data structure. The reason we used the express representation of nodes (i.e. $\hat{\mathcal{N}}_s$) is to reduce the cost of key comparison on the map operations.

3.1.2 Transformer and Widening Over Implementation

We build our abstract value propagation and abstract transformer over the definition of data flow refinement type semantics. The implementation strictly follows the definition in Fig.2.3 and Fig.2.4. On the top level, we add a function that iterates the abstract transformer until a fix point is reached, and apply widening and/or narrowing at each iteration.

Moreover, to apply widening operation ∇^{t}_X at each iteration of $\mathsf{step}^{\mathsf{t}}$, we directly use domain widening operator ∇^{a}_X through the Apron library which includes a common interface over different abstract numeric domains. However, we do not implement the shape widening operator ∇^{sh}_X explicitly because we perform Hindley-Milner Type Inference upfront. Only if Hindley-Milner Type Inference succeeds do we proceed with the analysis. This guarantees that the analysis will terminate.

3.2 Experiments

In this section, we experimentally compare the efficiency and practicability of refinement type tools by analyzing refinement type checking with a collection of benchmarks for OCaml. To evaluate DRIFT^2 we conduct 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?

Our first experiment includes the comparison of the precision and performance of DRIFT^2 for different configurations of the abstract domains. The second experiment compares DRIFT^2 with other refinement type inference tools. All our experiments were conducted on a personal laptop with Intel(R) Core(TM) i9-9800H and 32 GB memory running Linux.

3.2.1 Benchmarks

To find a sufficient amount of untyped high-order OCaml programs, we generates the benchmarks for DRIFT^2 by integrating the benchmarks from DOrder [24] and the test cases from R_Type [25] that were previously proved and tested safe for analysis. We excluded any programs related to algebraic data types (ADTs), because DRIFT^2 currently does not support ADTs. Moreover, we added some erroneous benchmarks. The details are as follows:

- **Array Programs (A)** include several functions showing analysis to infer refinement types over bounds of elements/length in array;
- **First Order Programs (FO)** include numerous functions performing first-order behaviors;
- **High Order Programs (HO)** contain more compound high-order functions that either contain more of less functions as parameter or return a function as its output;
- **Erroneous Programs (E)** either contain implementation bugs or faulty specifications, including implementation errors and assertion errors.

Particularly, R_Type does not consider any analysis related to array, the representation of an array from its benchmarks is a function with indexed assertion check. For the following experiments, we created several logic-equivalent array benchmarks based on the format of R_Type given.

Moreover, DRIFT^2 is designed and implemented based on data flow analysis where the information gathered from the input program is about to calculate possible set of values at various program locations. The trigger of this analysis typically given by an entry of the function call. However, we apply numerical abstract domain (consists of the relational lattices) as part of analysis, which supports type refinements over abstract scope (i.e. use $\{v : Int \mid \top^a\}$ to express arbitrary integer input). We also examined those comparison

tools which introduced our test benchmarks. More details about this discussion could be found in Chapter 4. Based on these observations, we revised and dispatched benchmarks as two suites, instead of using those kinds of benchmarks directly.

Benchmark Suite A - Unconstrained Test Inputs

In this category, for all benchmarks, we ignore to present any main function calls as an entry of analysis in the code. Taken program 1 from Fig. 2.6 as an example:

```
let id x = x
let main (n:int(*-:{v:Int | true}*)) =
    assert(id n = n)
```

Note that, *true* predicate is uninformative, which means function **main** accepts any integer n as input. DRIFT[2] use the input refinement predicates, comment right after each parameter of the main function, as unconstrained inputs.

Benchmark Suite B - Concrete Test Inputs

For each test case, we keep the program logic as the same as benchmark suite A, except we now introduce several call site within the main function. For instance, here is how we re-construct program 1 (see Fig. 2.6) over the benchmark suite B:

```
let id x = x
let main_p (n:int) = assert(id n = n)
let main (w:unit) =
  let _ = main_p 1 in
  let _ = main_p 2 in
  ()
let _ = main ()
```

3.2.2 Experiment one: comparing different configurations of DRIFT^2

We use the two versions of our tool (context-insensitive and 1-context-sensitive) introduced on Section 2.2.1. and instantiate each with two different relational abstract domains implemented in Apron: Octagons (Oct), and Convex Polyhedra and Linear Equalities (Polka). For each abstract domain, we further consider three different widening configurations: widening without narrowing (w), widening with narrowing (wn), and delay widening with narrowing (dwn). For delay widening we use a fixed delay bound of 300 iterations, except for polyhedra where widening may kick in earlier if a certain threshold on the number of constraints per refinement is exceeded.

Benchmark Suite A

Table 3.1 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 2-3. The results further indicate that there is no clear winner among the different numerical abstract domains for type refinements. This highlights the benefit of a parametric analysis that allows one to easily swap out one refinement domain for another. One could combine the best-performing domains via a reduced product construction to obtain a domain that performs strictly better than any individual domain. However, we have not yet implemented this.

Benchmark Suite B

By using the benchmark suite B, as shown in Table 3.2, DRIFT^2 remains sound analysis on the erroneous benchmarks. In addition to context sensitivity, our analysis also remains the experimental phenomena as the first part. For timing, DRIFT^2 would require additional

Benchmark category		Configuration																	
	Version	context-insensitive									1-context-sensitive								
	Domain	Oct			Polka strict			Polka loose			Oct			Polka strict			Polka loose		
	Widening	w	wn	dwn	w	wn	dwn	w	wn	dwn	w	wn	dwn	w	wn	dwn	w	wn	dwn
	succ	21	21	27(1)	32	32	31	32	32	31	25	25	29	36	36	32	36	36	31
	total	4.08	4.68	12.70	7.27	9.20	21.91	6.68	7.93	20.09	9.95	12.37	29.82	15.85	19.33	35.31(2)	16.15	17.97	35.92(2)
HO (38)	avg.	0.11	0.12	0.33	0.19	0.24	0.58	0.18	0.21	0.53	0.26	0.33	0.78	0.42	0.51	0.93	0.42	0.47	0.95
loc: 8	mean	0.22	0.24	0.62	0.34	0.47	1.55	0.30	0.36	1.49	0.55	0.69	1.46	0.81	0.98	2.26	0.84	0.89	2.00
	succ	25	25	43(6)	42	42	44	42	42	47(2)	33	33	50(7)	46	46	48	46	46	49
	total	18.10	20.51	58.37	24.44	29.27	52.99(1)	22.50	27.04	50.59(1)	49.44	57.67	120.27	54.77	69.06	97.59(3)	61.17	66.88	109.78(4)
FO (70)	avg.	0.26	0.29	0.83	0.35	0.42	0.76	0.32	0.39	0.72	0.71	0.82	1.72	0.78	0.99	1.39	0.87	0.96	1.57
loc: 11	mean	3.33	3.58	6.43	4.49	5.00	6.18	4.00	4.42	4.43	10.78	11.84	12.11	10.15	11.95	12.06	11.94	12.08	12.63
	succ	8	8	8	11	11	11	11	11	11	8	8	8	11	11	11	11	11	11
	total	7.74	8.55	23.41	9.95	11.43	17.60	9.33	10.81	14.25	20.67	23.64	42.42	23.63	31.34	31.67	22.90	26.10	32.59
A (13)	avg.	0.60	0.66	1.80	0.77	0.88	1.35	0.72	0.83	1.10	1.59	1.82	3.26	1.82	2.41	2.44	1.76	2.01	2.51
loc: 17	mean	0.96	1.08	2.11	1.15	1.27	1.75	1.09	1.23	1.31	3.34	3.53	4.04	3.39	4.32	4.37	3.47	3.68	4.17
	succ	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	total	23.97	25.36	27.57	17.68	20.07	27.26	16.80	17.88	21.41	41.49	45.45	54.78	30.11	33.74	40.69	30.74	32.47	43.69
E (15)	avg.	1.60	1.69	1.84	1.18	1.34	1.82	1.12	1.19	1.43	2.77	3.03	3.65	2.01	2.25	2.71	2.05	2.16	2.91
loc: 16	mean	4.26	4.62	4.24	3.36	3.72	4.12	3.18	3.24	3.18	7.03	7.47	7.39	6.29	6.35	6.45	6.26	6.45	6.79

Table 3.1 Summary of Experiment 1 over Benchmark Suite A. For each benchmark category, we provide the number of programs within that category in parenthesis. For each benchmark category and configuration, we list: the number of programs successfully analyzed **(succ)**, the total accumulated running time across all benchmarks **(total)**, the average running time per benchmark **(avg.)**, and the mean running time per benchmark **(mean)**. All running times are in seconds. The numbers given in parentheses after **(total)** indicate the number of benchmarks we failed due to runtime error. The running time for these benchmarks is not included in the total. For the successfully verified benchmarks, we additionally provide in parentheses the number of benchmarks that were only solved by that specific configuration across all configurations of the same version of the tool. We omit these two values in case they are 0.

times if a program code contains a various amount of main procedure calls, because our analysis would need more iterations to update the constraints based on those arguments. For the array's benchmarks, our analysis failed more test cases than the previous experiment. The reason is that we did not apply a dependency between the bounds of an array and a requested length during the array construction.

Benchmark category		Configuration																	
	Version	context-insensitive									1-context-sensitive								
	Domain	Oct			Polka strict			Polka loose			Oct			Polka strict			Polka loose		
	Widening	w	wn	dwn	w	wn	dwn	w	wn	dwn	w	wn	dwn	w	wn	dwn	w	wn	dwn
	succ	24	24	28(1)	33	33	31	33	33	31	28	28	32	33	33	33	32	32	33
	total	4.19	3.99	8.39	6.68	7.80	15.83(1)	6.45	7.79	14.36(1)	12.45	13.77	27.19	20.40	20.71	38.53(1)	16.09(1)	18.26(1)	34.85(1)
HO (38)	avg.	0.11	0.11	0.22	0.18	0.21	0.42	0.17	0.20	0.38	0.33	0.36	0.72	0.54	0.55	1.01	0.42	0.48	0.92
loc: 12	mean	0.31	0.17	0.50	0.29	0.33	1.22	0.27	0.31	1.03	0.71	0.75	1.73	1.10	1.07	2.73	0.89	0.97	2.51
	succ	22	22	39(2)	40	40	47	40	40	49(1)	27	27	49(7)	45	45	50	45	45	53
	total	12.81	15.25	34.42	24.19	28.74	44.46(4)	24.63	27.57	45.72(4)	55.66	61.33	102.44	79.55(1)	81.64(1)	112.51(5)	65.93(1)	75.50(1)	109.30(4)
FO (70)	avg.	0.18	0.22	0.49	0.35	0.41	0.64	0.35	0.39	0.65	0.80	0.88	1.46	1.14	1.17	1.61	0.94	1.08	1.56
loc: 16	mean	1.89	2.13	2.30	4.54	4.98	5.68	4.64	4.73	5.72	8.90	9.44	7.83	12.64	11.68	14.45	10.21	10.87	11.74
	succ	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	total	5.94	6.77	15.14	8.91	10.38	14.03	9.07	9.91	10.38	20.79	23.29	36.35	30.27	30.51	33.96	26.53	27.97	28.83
A (13)	avg.	0.46	0.52	1.16	0.69	0.80	1.08	0.70	0.76	0.80	1.60	1.79	2.80	2.33	2.35	2.61	2.04	2.15	2.22
loc: 22	mean	0.97	1.05	2.08	1.17	1.34	1.58	1.21	1.27	1.29	3.35	3.56	4.01	3.94	3.83	4.25	3.65	3.58	3.63
	succ	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	total	22.87	24.11	22.48	18.90	20.25	23.92	19.32	19.35	21.30	47.23	51.58	56.19	47.32	43.69	53.40	41.21	41.19	48.23
E (15)	avg.	1.52	1.61	1.50	1.26	1.35	1.59	1.29	1.29	1.42	3.15	3.44	3.75	3.15	2.91	3.56	2.75	2.75	3.22
loc: 20	mean	3.84	3.98	3.38	3.35	3.46	3.43	3.41	3.32	3.15	9.62	10.65	8.89	12.04	10.34	11.30	10.02	9.89	9.85

Table 3.2 Summary of Experiment 1 for Benchmark Suite B.

Summary

Compared with the widening configurations, delayed to set widening improved the results, but the choice to set delay bound is a trade-off between the complexity of the analysis and the precision of the result. The additional defeated cases shows that our tested abstract domains were unable to infer non-convex properties or to express any non-linear constraints. Notwithstanding the configuration, applying widening after each step may cause precision loss , especially when the tool analyzes and holds the prove property (i.e. loop invariant). We believe the precision could be boosted if we use widening with thresholds or apply a more 'recoverable' narrowing process to enforce convergence. Test cases related to examining elements inside an array also failed in our analysis because we disregard for an extension of the refinement type system, although this inference is achievable by introducing parametric on refinement predicate variables [26].

We further conducted a more detailed analysis of the running times by profiling the execution of the tool. This analysis determined that in both versions, most of the time is spent in the projection operation of the underlying numerical abstract domain.

This operation is called when type refinements are rescoped after a call to $\mathsf{prop}^{\mathsf{t}}$. This happens particularly often when analyzing programs that involve applications of curried functions, which are currently handled rather naively. We believe that the running times can be improved notably by avoiding unnecessary rescoping and handling applications of fully applied curried functions as if they took all of their arguments at once.

3.2.3 Experiment 2: Comparing with other Tools

Overall, the results of Experiment 1 suggest that the 1-context-sensitive version of DRIFT^2 instantiated with the Polka domain and undelayed widening provides a good balance between precision and efficiency. In our second experiment, we compare this configuration with several other existing tools. We consider four other automated verification tools: DSolve, DOrder, R_Type, and MoCHi. DSolve is the original implementation of the Liquid type inference algorithm proposed in [14]. DOrder [24] builds on the same basic algorithm as DSolve but augments the simple syntactic heuristics for guessing candidate refinement predicates with a machine learning algorithm that learns such predicates from concrete program executions. R_Type improves upon this further with a more sophisticated ICE-style learning algorithm [27] and additionally replaces the Houdini-based fixpoint algorithm of [14] with a general Horn clause solver. In our experiments, we instantiate R_Type with the interpolation-based Horn clause solver Spacer [28], which is part of Z3 [29]. Finally, MoCHi [30] is a model checker for higher-order programs based on higher-order recursion schemes. We note that MoCHi is not limited to simple refinement types but can also infer intersection types.

Benchmark Suite A

Table 3.3 summarizes the results of our comparison. DRIFT^2 solves significantly more benchmarks than all other tools in the important category of higher-order programs.

Bench-mark cat.	DRIFT^2				R_Type				DOrder				DSolve				MoCHi			
	succ	full	avg	mean	succ	full	avg	mean	succ	full	avg	mean	succ	full	avg	mean	succ	full	avg	mean
HO (38)	36	15.85	0.42	0.81	28	2.91(3)	0.08	0.09	1	5.07	0.13	0.13	29	14.53	0.38	0.42	32	57.53(6)	1.51	3.93
FO (70)	46(3)	54.77	0.78	10.15	46(1)	7.38(13)	0.11	0.85	0	8.30	0.12	0.17	48(1)	30.80	0.44	1.33	53	170.59(15)	2.44	7.12
A (13)	11	23.63	1.82	3.39	4	1.57(2)	0.12	0.17	1	2.18	0.17	0.20	8	10.32	0.79	1.08	9(1)	4.50	0.35	0.80
E (15)	15	30.11	2.01	6.29	15	1.93	0.13	0.18	3	1.77	0.12	0.12	14	10.77	0.72	1.28	11	7.84(4)	0.52	1.72

Table 3.3 Summary of Experiment 2 on Benchmark Suite A.

Although DRIFT^2 does not surpass the hits of other tools in the first order category, we are still comparable to the other tools if we postponed widening before several iterations. In the remaining two categories, DRIFT^2 performs equally well as the best tool in each category in terms of solved benchmarks, though requiring more time. Notably, DOrder could not perform correctly if the input program does not contain the call entries.

Benchmark Suite B

The result by using Benchmark B has been summarized on Table 3.4. For the second half comparison of experiment 2, DRIFT^2 keeps the advantage of context-sensitive analysis. Although we presented this suite of benchmarks which includes some concrete test inputs, DOrder failed to parse some of them for analysis. Some error cases during experiment are caused by interpretation error. To the best of our knowledge, we allows DOrder to conduct test runs of the programs as much as possible, but the result we collected is not our expected. Overall, the comparison results indicate that our approach guarantees the termination of analysis by applying widening/narrowing process.

Bench-mark cat.	DRIFT^2				R_Type				DOrder				DSolve				MoCHi			
	succ	full	avg	mean	succ	full	avg	mean	succ	full	avg	mean	succ	full	avg	mean	succ	full	avg	mean
HO (38)	33(1)	20.67	0.54	1.11	27	18.52(4)	0.49	5.67	9	59.38(2)	1.56	9.46	31(1)	17.88	0.47	0.56	32(2)	69.67(6)	1.83	9.70
FO (70)	45(2)	90.71(1)	1.30	15.54	50(3)	47.23(10)	0.67	4.61	39(1)	40.01(6)	0.57	2.23	50	42.33	0.60	1.75	46(2)	157.28(24)	2.25	7.53
A (13)	7	35.24	2.71	4.62	5(1)	19.35(7)	1.49	3.21	4	11.91(1)	0.92	3.17	8(2)	12.87	0.99	1.04	9(2)	4.71	0.36	0.64
E (15)	15(1)	49.27	3.28	11.79	13	2.72(2)	0.18	0.27	4	12.14	0.81	2.35	14	14.14	0.94	1.64	5	2.53(10)	0.17	0.50

Table 3.4 Summary of Experiment 2 on Benchmark Suite B.

Summary

One disadvantage of widening/narrowing revealed by the comparison is that DRIFT^2 cannot check for assertion violations until the analysis reaches its final fixpoint. This is because widening may overshoot, causing an assertion to be spuriously violated until the analysis recovers from this precision loss during narrowing. In contrast, tools based on abstract refinement techniques such as Spacer used in R_Type can often detect unsafe programs more quickly. This is reflected by the lower running times of R_Type in the E category. We further note that none of the tools produced unsound results in this category. The failing benchmarks for DSolve and MoCHi are due to timeouts, respectively, a run-time error in the analysis. The considered timeout was 5 minutes per benchmarks for all tools. In contrast, DRIFT^2 always terminated before the timeout. We attribute this to the use of widening/narrowing.

Chapter Four

Related Work

There is a large literature on refinement type based verification tools. Here, we only discuss the two most relevant approaches.

DSolve [15] is a verification tool that performs liquid type inference [1], and it builds refinement predicates over conjunctions of qualifiers. The logical qualifiers are user-specified shapes of predicates which the analysis the algorithm then instantiates. Generally, after performing type inference over the input program, DSolve generates qualified constraints by liquid subtyping relations, and then solves them through a theorem prover inspired by predicate abstraction [11, 12]. Although DSolve is an automated verification tool as it can use default qualifiers as hints, this mechanism is inadequate to meet all kinds of proofs of safety, especially for inferring refinement types of high order programs.

DOrder [24] is one of older tool that uses a random sampling approach to infer refinement predicates. This technique allows them to automatically infer refinement types for high-order fuctional programs. During learning invariant process, it uses a binary classification strategy and builds a classifier to determine invariants based on examples from program input and conuterexamples obtained from SMT solver, which finally synthesize the invariants from samples into refinement predicate. However, DOrder still requires the user to provide good test inputs for the input program. Compared with our analysis, DRIFT^2 directly allows its users to specify refinement properties of input parameters

through dedicated comments. In addition, DOrder lacks inferring inductive invariants of recursive functions with different calling contexts. DRIFT^2 avoids this problem by using the abstract call site stack $\hat{S}$ which preserves context information to prevent loss of precision.

R_Type [25] is a tool proposed after DOrder, which works by using implication constraints to discover inductive invariants and combining negative examples from the ICE framework [27, 31]. R_Type is also a machine learning based verification tool that use sampling-based approaches to infer type refinement. As a result, R_Type performs better than DOrder in our experiment. Additionally, R_Type infers arbitrary Boolean combinations of qualifiers as refinement predicates, this allows it to express constraints such as non-convex relations. While the approach is successful in revealing the above situations, it does no longer guarantee the termination of the analysis. Instead, DRIFT^2 utilizes widening and narrowing [22] approaches to obtain a safe approximation of the program semantics in finite time.

MoCHI [30] is a high-order model checker based on the model checking of higher-order recursion schemes [32]. To verify higher-order functional programs with infinite data domains (e.g. integers), MoCHI reduces such programs to higher-order Boolean programs by using predicate abstraction [12]. In addition, MoCHI uses a technique, called Counterexample-Guided Abstraction Refinement (CEGAR) [33], which attempts to discover new predicates refined from predicate abstraction if previous predicates are not sufficient to verify the program. One disadvantage of this approach is the discovered counterexamples for inferring an invariant may never be strong enough to prove program safety, which ultimately leads to infinitely many abstraction refinement steps; thus, MoCHI does not guarantee the termination of the analysis.

Chapter Five

Conclusions & Future Work

5.1 Limitations and Future Work

We now discuss some limitations to our current implementation. First, for the polyhedra domain, we start widening once any length of the inferred constraints exceeds 15 because Apron gave undefined behavior. This solution is not ideal, but the run time behavior of our analysis is not robust otherwise. We have manually analyzed several benchmarks that caused this problem, and the error is related to using meet and join operations. One reason could be that the calculation of matrix constraints leads to numerical overflows if the coefficients are extremely large.

As of now, the tool does not yet support lists or user-defined algebraic data types. Moreover, while DRIFT^2 automatically checks whether all array accesses are within bounds, it is unable to capture quantified constraints about array elements. However, the benchmarks include assertions that check bounds of element stored in arrays. Therefore, our next goal is to add support for these datatypes and more expressive assertions.

Additionally, we plan to implement several general improvements to the current version of the tool. In particular, we plan to improve the running times by avoiding unnecessary variable projections, by applying curried functions in an "uncurried" way. Second, to make DRIFT^2 more expressive, we plan to extend our system to support recursive

datatypes such as lists. Third, we wish to avoid using delay widening as a configuration because it is difficult to guess appropriate delay bounds. One approach for widening configuration is using widening with thresholds [34, 35], which helps us to capture upper bounds of iterations during the analysis of recursive functions.

5.2 Conclusions

In this thesis, we presented an implementation of a new parametric data flow refinement type inference analysis. Our experimental evaluation indicates that our approach guarantees the termination of the analysis by applying widening. The parametric modeling abstract stack $\hat{\mathcal{S}}$ also permits programmers to design and implement the degree of approximations by calling contexts. Overall, we conclude that DRIFT^2 compares very well to the state of the art, demonstrating that our approach promises to provide a solid foundation for implementing robust refinement type inference algorithms.

REFERENCES

[1] 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, editors. ACM, 159–169. DOI: 10.1145/1375581.1375602. https://doi.org/10.1145/1375581.1375602.

[2] Zvonimir Pavlinovic. 2019. *Leveraging Program Analysis for Type Inference*. English. PhD thesis, 289. ISBN: 9781085678834. http://proxy.library.nyu.edu/login?url=https://search.proquest.com/docview/2290954364?accountid=12768. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2019-10-12.

[3] Patrick Cousot and Radhia Cousot. 1992. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings* (Lecture Notes in Computer Science). Maurice Bruynooghe and Martin Wirsing, editors. Volume 631. Springer, 269–295. DOI: 10.1007/3-540-55844-6_142. https://doi.org/10.1007/3-540-55844-6_142.

[4] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated. ISBN: 3642084745.

[5] Nissim Francez. 1992. *Program Verification*. (1st edition). Addison-Wesley Longman Publishing Co., Inc., USA. ISBN: 0201416085.

[6] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press. ISBN: 978-0-262-16209-8.

[7] Luís Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*. Richard A. DeMillo, editor. ACM Press, 207–212. DOI: 10.1145/582153.582176. https://doi.org/10.1145/582153.582176.

[8] 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-

itor. ACM, 268–277. DOI: 10.1145/113445.113468. https://doi.org/10.1145/113445.113468.

[9] Kenneth Knowles and Cormac Flanagan. 2007. Type reconstruction for general refinement types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings* (Lecture Notes in Computer Science). Rocco De Nicola, editor. Volume 4421. Springer, 505–519. DOI: 10.1007/978-3-540-71316-6_34. https://doi.org/10.1007/978-3-540-71316-6_34.

[10] Kodai Hashimoto and Hiroshi Unno. 2015. Refinement type inference via horn constraint optimization. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings* (Lecture Notes in Computer Science). Sandrine Blazy and Thomas P. Jensen, editors. Volume 9291. Springer, 199–216. DOI: 10.1007/978-3-662-48288-9_12. https://doi.org/10.1007/978-3-662-48288-9_12.

[11] Tilak Agerwala and Jayadev Misra. 1978. Assertion Graphs for Verifying and Synthesizing Programs. Technical report 83. University of Texas, Austin.

[12] Susanne Graf and Hassen Saïdi. 1997. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings* (Lecture Notes in Computer Science). Orna Grumberg, editor. Volume 1254. Springer, 72–83. DOI: 10.1007/3-540-63166-6_10. https://doi.org/10.1007/3-540-63166-6_10.

[13] 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). José Nuno Oliveira and Pamela Zave, editors. Volume 2021. Springer, 500–517. DOI: 10.1007/3-540-45251-6_29. https://doi.org/10.1007/3-540-45251-6_29.

[14] 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). Renate A. Schmidt, editor. Volume 5663. Springer, 214–229. DOI: 10.1007/978-3-642-02959-2_18. https://doi.org/10.1007/978-3-642-02959-2_18.

[15] Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. 2010. Dsolve: safety verification via liquid types. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings* (Lecture Notes in Computer Science). Tayssir Touili, Byron Cook, and Paul B. Jackson, editors. Volume 6174.

Springer, 123–126. DOI: 10.1007/978-3-642-14295-6_12. https://doi.org/10.1007/978-3-642-14295-6_12.

[16] Anonymous Author. Design space of liquid type inference. unpublished, preprint on webpage at https://cs.nyu.edu/wies/publ/drift-draft.pdf, (N.D.).

[17] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors. ACM, 238–252. DOI: 10.1145/512950.512973. https://doi.org/10.1145/512950.512973.

[18] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*. Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors. ACM Press, 269–282. DOI: 10.1145/567752.567778. https://doi.org/10.1145/567752.567778.

[19] Antoine Miné. 2007. The octagon abstract domain. *CoRR*, abs/cs/0703084. arXiv: cs/0703084. http://arxiv.org/abs/cs/0703084.

[20] 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, editors. ACM Press, 84–96. DOI: 10.1145/512760.512770. https://doi.org/10.1145/512760.512770.

[21] 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). Ahmed Bouajjani and Oded Maler, editors. Volume 5643. Springer, 661–667. DOI: 10.1007/978-3-642-02658-4_52. https://doi.org/10.1007/978-3-642-02658-4_52.

[22] Patrick Cousot and Radhia Cousot. 1991. Comparison of the galois connection and widening/narrowing approaches to abstract interpretation. In *Actes JTASPEFL'91 (Bordeaux, France), October 1991, Laboratoire Bordelais de Recherche en Informatique (LaBRI), Proceedings* (Series Bigre). Michel Billaud, Pierre Castéran, Marc-Michel Corsini, Kaninda Musumbu, and Antoine Rauzy, editors. Volume 74. Atelier Irisa, IRISA, Campus de Beaulieu, 107–110.

[23] 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*. ACM, San Francisco, California, USA,

393–407. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199536. http://doi.acm.org/10.1145/199448.199536.

[24] 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, editors. ACM, 400–411. DOI: 10.1145/2784731.2784766. https://doi.org/10.1145/2784731.2784766.

[25] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018. 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). Dirk Beyer and Marieke Huisman, editors. Volume 10805. Springer, 365–384. DOI: 10.1007/978-3-319-89960-2_20. https://doi.org/10.1007/978-3-319-89960-2_20.

[26] 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). Matthias Felleisen and Philippa Gardner, editors. Volume 7792. Springer, 209–228. DOI: 10.1007/978-3-642-37036-6_13. https://doi.org/10.1007/978-3-642-37036-6_13.

[27] 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). Armin Biere and Roderick Bloem, editors. Volume 8559. Springer, 69–87. DOI: 10.1007/978-3-319-08867-9_5. https://doi.org/10.1007/978-3-319-08867-9_5.

[28] Arie Gurfinkel and Nikolaj Bjørner. 2019. The science, art, and magic of constrained horn clauses. In *21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2019, Timisoara, Romania, September 4-7, 2019*. IEEE, 6–10. DOI: 10.1109/SYNASC49474.2019.00010. https://doi.org/10.1109/SYNASC49474.2019.00010.

[29] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (Lecture Notes in Computer Science). C. R. Ramakrishnan and Jakob Rehof, editors. Volume 4963. Springer, 337–340. DOI: 10.1007/978-3-540-78800-3_24. https://doi.org/10.1007/978-3-540-78800-3_24.

[30] 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, editors. ACM, 222–233. DOI: 10.1145/1993498.1993525. https://doi.org/10.1145/1993498.1993525.

[31] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. 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, editors. ACM, 499–512. DOI: 10.1145/2837614.2837664. https://doi.org/10.1145/2837614.2837664.

[32] C.-H. Luke Ong. 2006. On model-checking trees generated by higher-order recursion schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 81–90. DOI: 10.1109/LICS.2006.38. https://doi.org/10.1109/LICS.2006.38.

[33] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. John Launchbury and John C. Mitchell, editors. ACM, 58–70. DOI: 10.1145/503272.503279. https://doi.org/10.1145/503272.503279.

[34] Axel Simon and Andy King. 2006. Widening polyhedra with landmarks. In *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings* (Lecture Notes in Computer Science). Naoki Kobayashi, editor. Volume 4279. Springer, 166–182. DOI: 10.1007/11924661_11. https://doi.org/10.1007/11924661_11.

[35] Bogdan Mihaila, Alexander Sepp, and Axel Simon. 2013. Widening as abstract domain. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings* (Lecture Notes in Computer Science). Guillaume Brat, Neha Rungta, and Arnaud Venet, editors. Volume 7871. Springer, 170–184. DOI: 10.1007/978-3-642-38088-4_12. https://doi.org/10.1007/978-3-642-38088-4_12.