Intensional Datatype Refinement

With Application to Scalable Verification of Pattern-Match Safety

EDDIE JONES, University of Bristol, UK
STEVEN RAMSAY, University of Bristol, UK

The pattern-match safety problem is to verify that a given functional program will never crash due to non-exhaustive patterns in its function definitions. We present a refinement type system that can be used to solve this problem. The system extends ML-style type systems with algebraic datatypes by a limited form of structural subtyping and environment-level intersection. We describe a fully automatic, sound and complete type inference procedure for this system which, under reasonable assumptions, is worst-case linear-time in the program size. Compositionality is essential to obtaining this complexity guarantee. A prototype implementation for Haskell is able to analyse a selection of packages from the Hackage database in a few hundred milliseconds.

CCS Concepts: • **Theory of computation** → **Functional constructs**; **Program verification**; *Logic and verification.*

Additional Key Words and Phrases: higher-order program verification, refinement types

ACM Reference Format:
Eddie Jones and Steven Ramsay. 2021. Intensional Datatype Refinement: With Application to Scalable Verification of Pattern-Match Safety. *Proc. ACM Program. Lang.* 5, POPL, Article 55 (January 2021), 29 pages. https://doi.org/10.1145/3434336

1 INTRODUCTION

The *pattern match safety* problem asks, given a program with non-exhaustive (algebraic datatype) patterns in its function definitions, is it possible that the program crashes with a pattern-match exception? Consider the example Haskell code in Figure 1. This code defines the two main ingredients in a typical definition (see e.g. [Harrison 2009]) of conversion from arbitrary propositional formulas to propositional formulas in disjunctive normal form (represented as lists of lists of literals). Using these definitions, the conversion can be described as the composition dnf := nnf2dnf ∘ nnf.

Notice that the definition of nnf2dnf is partial: it is expected only to be used on inputs that are in negation normal form (NNF). Consequently, unless is can be verified that nnf always produces a formula without any occurrence of Imp or Not, then any application of dnf to an expression of type Fm *a* may result in a pattern match failure exception. In this paper we present a new refinement type system that can be used to perform this verification statically and automatically. Type inference is compositional and incremental so that it can be integrated with modern development environments: open program expressions can be analysed and only the parts of the code that are modified need to be re-analysed as changes are made.

Authors' addresses: Eddie Jones, Department of Computer Science, University of Bristol, UK; Steven Ramsay, Department of Computer Science, University of Bristol, UK.

2475-1421/2021/1-ART55
https://doi.org/10.1145/3434336

```
data L a =                  nnf (Lit (Atom x)) = Lit (Atom x)
  Atom a                    nnf (Lit (NegAtom x)) = Lit (NegAtom x)
  | NegAtom a               nnf (And p q) = And (nnf p) (nnf q)
                            nnf (Or p q) = Or (nnf p) (nnf q)
data Fm a =                 nnf (Imp p q) = Or (nnf (Not p)) (nnf q)
  Lit (L a)                 nnf (Not (Not p)) = nnf p
  | Not (Fm a)              nnf (Not (And p q)) = Or (nnf (Not p)) (nnf (Not q))
  | And (Fm a) (Fm a)       nnf (Not (Or p q)) = And (nnf (Not p)) (nnf (Not q))
  | Or (Fm a) (Fm a)        nnf (Not (Imp p q)) = And (nnf p) (nnf (Not q))
  | Imp (Fm a) (Fm a)       nnf (Not (Lit (Atom x))) = Lit (NegAtom x)
                            nnf (Not (Lit (NegAtom x))) = Lit (Atom x)

                            nnf2dnf (Lit a) = [[a]]
                            nnf2dnf (Or p q) = List.union (nnf2dnf p) (nnf2dnf q)
                            nnf2dnf (And p q) = distrib (nnf2dnf p) (nnf2dnf q)
                             where distrib xss yss =
                                     List.nub [ List.union xs ys | xs <- xss, ys <- yss ]
```

Fig. 1. Conversion to disjunctive normal form.

Contributions. Whilst there are other analyses in the literature that can also verify instances of the foregoing example ours is, as far as we are aware, the only to offer strong guarantees on predictability, which we believe to be key to the usability of such systems in practice.

- The analysis is characterised by the type system, which is a natural, yet expressive extension of ML-style type systems with algebraic datatypes. It combines polyvariance (through environment-level intersection) and path-sensitivity (through conditional match typing).
- The analysis runs in time that is, in the worst-case, linear in the size of the program (under reasonable assumptions on the size of types and the nesting of matching).

We do not know of any other system or reachability analysis combining *polyvariance*, *path-sensitivity*, an intuitive characterisation of completeness, and a linear-time guarantee on the overall worst-case complexity (in terms of program size). Furthermore, our prototype demonstrates excellent performance over a range of packages from Hackage, processing each in less than a second.

1.1 A Type System for Intensional Datatype Refinements

Sound and terminating program analyses are conservative: there are always programs without bugs that, nevertheless, cannot be verified. Identifying a large fragment for which the analysis is complete, i.e. a class of safe programs for which verification is guaranteed, allows the programmer to reason about the behaviour of the analysis on their code. In particular, when an analysis fails to verify a program that the user believes to be safe, it gives them an opportunity to take action, such as by programming more defensively, in order to put their program into the fragment and thus be certain of verification success.

However, for this to be most effective, the fragment must be easily understood by the average functional programmer. Our analysis is complete with respect to programs typable in a natural extension of ML-style type systems with algebraic datatypes. Indeed it is characterised by this

system: the force of Theorems 25 and 28 is to say that it forms a sound and complete inference procedure. The system is presented in full in Section 5, but the highlights are as follows:

(i) The datatype environment introduced by the programmer, e.g. L a and Fm a, is *completed*: every datatype whose definition can be obtained by erasing constructors from one of those given is added to the environment for the purpose of type assignment. These new datatypes are called *intensional refinements*. These additional types allow for the scrutinee of a match to be typed with a datatype that is more precise than the underlying type provided by the programmer. For example, the datatypes in Figure 2 are among the intensional refinements of Fm a, where **data** A a = Atom a is an intensional refinement of L a. Of course, the names of the datatypes are irrelevant.

(ii) There is a natural notion of subtyping between intensional refinement datatypes which is incorporated into the type system through an unrestricted subsumption rule. For example, Clause a and Cube a are both subtypes of the intensional refinement:

```
data NFm = Lit (L a) | Or (NFm a) (NFm a) | And (NFm a) (NFm a)
```

which is itself a subtype of Fm a. However, Clause a, Cube a and STLC a are all incomparable.

(iii) The typing rule for the case analysis construct, by which pattern matching is represented, enforces that matching is exhaustive with respect to the type of the scrutinee. This ensures that the analysis of matching is sound: programs for which the match is not exhaustive will not be typable. Moreover, the rule is *path-sensitive*, with the type of the match only depending on the types of the branches corresponding to the type of the scrutinee. For example, the following function can be assigned the type $(a \to b) \to \mathsf{Cube}\, a \to \mathsf{Cube}\, b$ and it can be assigned the type $(a \to b) \to \mathsf{Clause}\, a \to \mathsf{Clause}\, b$, but not the type $(a \to b) \to \mathsf{STLC}\, a \to \mathsf{STLC}\, b$ because it does not handle the constructor Imp.

```
map f (Lit (Atom x))   = Lit (Atom (f x))
map f (Lit (NegAtom x)) = Lit (NegAtom (f x))
map f (And p q) = And (map p) (map q)
map f (Or p q)  = Or (map p) (map q)
```

Path sensitivity is essential for handling typical use cases. Often a single large datatype is defined but, locally, certain parts of the program work within a fragment (e.g. only on clauses). Path sensitivity helps to ensure that transformations on values inside the fragment remain inside the correct datatype refinement — otherwise map could only advertise that it returns formulas in type NFm a. For example, Elm-style web applications typically define a single, global datatype of actions although the constituent pages may only be prepared to handle certain (overlapping) subsets locally.

(iv) Finally, refinement polymorphism, and hence *context-sensitivity*, is provided by allowing for environments that have more than a single refinement type binding for each free program variable, i.e. an environment-level intersection. For example, suppose trivial : Clause $a \to$ Bool checks a clause for complementary literals, sing : Cube $a \to$ Bool checks if a cube consists of a single conjunct, and rn : String $\to$ String performs a renaming of propositional atoms. Then the following expression[1] is well typed:

$$\lambda xy.\, \mathsf{trivial}\ (\mathsf{map}\ \mathsf{rn}\ x)\ ||\ \mathsf{sing}\ (\mathsf{map}\ \mathsf{rn}\ y)$$

This is because the typing environment contains *both* of the aforementioned types for map. Note: this is polymorphism in the class of formulas, not only in the type a of their atoms.

[1] The example is rather contrived, but we may rather imagine such combinations occurring in different parts of the program.

```
data Clause a =
  Lit (L a)
  | Or (Clause a) (Clause a)

data STLC a =
  Lit (A a)
  | And (STLC a) (STLC a)
  | Imp (STLC a) (STLC a)

data Cube a =
  Lit (L a)
  | And (Cube a) (Cube a)
```

Fig. 2. Some intensional refinements of Fm *a*.

To distinguish between the typing assigned to the program by the programming language (which we consider part of the input to the analysis) from the types that can be assigned in our extended system, we call the former the *underlying* typing of the program.

Characterising the analysis with a type system allows the programmer to reason about its behaviour using typings as a kind of *certificate*. Returning to the above example, the programmer can be certain that uses of dnf will be verifiably safe because they can synthesize the intensional datatype refinement NFm, and check the typings $\mathsf{nnf} : \mathsf{Fm}\ a \to \mathsf{NFm}\ a$ and $\mathsf{nnf2dnf} : \mathsf{NFm}\ a \to [[\mathsf{L}\ a]]$.

1.2 Compositionality and Complexity

Our analysis takes the form of a type inference procedure for the system described above. As is typical, inference proceeds by generating and solving typing constraints. The constraints are guarded inclusions, representing flow of data conditioned on the presence of certain constructors in datatypes along a program path.

A key goal of our work is to give some *guarantee* of the scalability of the analysis to large, real-world programs. We do this by ensuring that the whole of type inference – constraint generation and constraint solving – runs in time that is worst-case linear in the size of the program (assuming other parameters, such as the size of underlying types, are fixed).

We achieve this complexity guarantee by a careful exploitation of compositionality in the type inference algorithm. The key is to ensure that the size of the constraint set used to summarise the behaviour of a component c is independent of the number of components that c depends on.

The issues involved are the same for any kind of compositional program analysis so, to illustrate, consider some abstract program $p = p_1 \cdot p_2 \cdot \cdots \cdot p_n$ that has been broken down into n "components" p_i. In the interests of approaching the worst-case complexity in as simple a way as possible, assume that each component uses only the component immediately preceding it in the chain — for example, via a procedure call.

A compositional program analysis computes a summary S_i of the behaviour of each component i separately. For example, for constraint-based analyses, this is typically a set of constraints in a solved form (e.g. a constrained type scheme). For each component, the size of S_i can depend on the size of component i (i.e. its program text), but also the size of the summary S_{i-1} already computed for component $i-1$ on which it depends. By choosing the granularity of components to be small, or otherwise by making some reasonable assumption, we can regard the size of the program text of each component to be bounded by a constant. Hence, when we speak of program size, we will refer to the number of components, n. A consequence of this is that we may assume that the number of times that p_i uses p_{i-1} is bounded by some constant, say c.

When analysing the worst-case complexity of polyvariant analyses, like HM(X)-style type inference [Odersky et al. 1999], there is typically the possibility that the summary of component i may be duplicated c times inside the summaries of those components that depend on it, and thus we arrive at the (well-known) conclusion that the "summary" for the entry point of the chain S_n

may be of size exponential in n^2. For non-polyvariant, non-path-sensitive analyses, there is no duplication, but it is nevertheless typical that summaries are already quadratic in n: the cubic-time fragment of set constraints (see e.g. [Fähndrich and Aiken 1996; Fähndrich et al. 1998; Heintze 1994; Su et al. 2000]) is one example of this class.

Since this blow-up occurs even in typical cases, there is an extensive literature on powerful simplification techniques by which large summaries may *sometimes* be replaced by more concise equivalents, see particularly [Aiken et al. 1999; Dolan and Mycroft 2017; Fähndrich and Aiken 1996; Flanagan and Felleisen 1999; Pottier 2001; Rehof 1997; Trifonov and Smith 1996]. However, getting just the right combination and tuning of heuristics is difficult, and the initial implementation effort and subsequent maintenance is significant (e.g. regular benchmarking as the underlying programming language evolves). Moreover, one will always be able to find reasonable programs on which heuristic simplifications are not well tuned, the program analysis/type inference will stall, and the program's author will lose faith in the system.

By contrast, our system is designed to guarantee that the worst-case size of any S_i is independent of the summaries that it depends on, and hence of the program size (though it is exponential in the size of the largest underlying type). Therefore, with other parameters fixed, the size of each our summaries S_i is bounded by a constant.

We proceed component by component, first generating constraints and immediately putting them into a solved form. However, computing a solved form so as to guarantee the above property is not straightforward. Our constraint solver, which is inspired by the resolution-based approach used in set constraint based program analysis [Aiken and Wimmers 1992, 1993; Aiken et al. 1994a; Heintze et al. 1992] may take time exponential in the size of its input.

We are able to guarantee a linear time complexity overall because our compositional approach ensures that each constraint set that is given to the solver is unrelated to the size of the program. The size of the constraint set generated for a given component depends only on the size of the *summaries* of the components it depends on — the solved forms — and each of these is bounded by a constant. Therefore, the size of any constraint set supplied to the solver is also bounded by a constant. Thus we solve a small (but exponential in the size of the underlying types) number of constraints at every program point, rather than an enormous (exponential in the size of the program) number of constraints when processing the program's entry point.

This works only because we show that our constraint sets in solved form have the following remarkable property, stated formally as Theorem 32.

> Suppose C is a set of constraints in solved form over variables V and let $I \subseteq V$ be arbitrary. Let $C{\restriction_I}$, called the *restriction of C to I* be those constraints in C in which occur *only* variables from I. Then every solution to $C{\restriction_I}$ can be extended to a solution of all C.

In the restriction $C \restriction_I$, entire constraints are culled, including those that involve a mixture of variables from I and $V \setminus I$. Such mixed constraints, intuitively, impose compatibility requirements on the different components of a solution to C. What is significant about the above property is that it guarantees not only that the part of the constraint set only concerned with $V \setminus I$ is internally consistent but, moreover, that the mixed constraints will be satisfiable no matter which solution to $C{\restriction_I}$ is chosen.

We exploit compositionality in order to choose a minimal set of variables I, *the interface*, with which to restrict constraint sets. The interface of a program expression in context $\Gamma \vdash e : T$ consists only of the those refinement variables that occur free in Γ and T. The size of the interface depends

[2]However, note that Gustavsson and Svenningsson [2001] show that this can be reduced to cubic complexity in the case of simple variable/variable constraints.

only on the size of the underlying type of e, the size of definitions of any datatypes occurring in that type and the nesting of pattern matching. Thus, if we make the (in our view, reasonable) assumption that the sizes of these quantities are bounded by a constant, so too is the size of the interface and, therefore, the size of any restricted constraint set — our component summary.

1.3 Implementation

Of course worst-case complexity is only part of the story, and especially so when the constant factors depend upon several assumptions. Hence, we have implemented our System in Haskell as a GHC Plugin and ran it on a selection of packages from the Hackage database. The plugin takes a Haskell package to be compiled and runs our type inference algorithm over the whole code to yield a constrained type assignment and a set of type errors. The average time taken to process each module is in the order of milliseconds and the results show very stark contrast between the number of refinement variables associated with the program points in the module (often > 10000) and the number of refinement variables in the interfaces (typically < 20).

1.4 Outline

The rest of the paper is structured as follows. In Section 2 we describe a Haskell-like functional programming language which forms the setting for our work. This is followed in Section 3 by our definitions of refinement. Then in Sections 4 and 5 by the definition of the type system that characterises the analysis. In Sections 6, 7 and 8 we present our analysis as a type inference algorithm, generating and solving constraints. We discuss the restriction operation and its complexity in Section 9 and we report on our implementation in Section 10. Finally, we conclude and discuss related work in Section 12.

2 LANGUAGE

Preliminaries. Given sets X and Y, let us write $X \to Y$ for the set of all functions from X to Y and $\mathsf{Map}\ X\ Y$ for the set of all finite maps between X and Y. As usual function arrows are assumed to associate to the right. Additionally, we define the indexing of function arguments, that is $(X_1 \to \cdots \to X_m \to Y)[i] = X_i$ for all $i \in [1..m]$. Given a family of sets Y_x indexed by $x \in X$, let us write $\Pi x \in X.\ Y_x$ for the subset of $X \to \bigcup_{x \in X} Y_x$ that contains only functions that are guaranteed to map each $x \in X$ to some element of Y_x and let us write $\Sigma x \in X.\ Y_x$ for the subset of $X \times \bigcup_{x \in X} Y_x$ in which the second component of each pair (x, y) is guaranteed to belong to Y_x. Given a family of sets Y_x indexed by $x \in X$, let us write $\coprod_{x \in X} Y_x$ for their disjoint sum and inj_x for each of the canonical injections.

Types. We assume a countable collection $\mathbb{A}$ of type variables, ranged over by α; a finite collection $\mathbb{B}$ of base types, ranged over by b, and a countable collection $\mathbb{D}$ of *algebraic datatype identifiers* ranged over by d. These can be thought as the names of first-order type constructors. Each datatype identifier has a fixed arity, and only forms a proper type when supplied with the appropriate number of type arguments. We refer to a datatype identifier with its argument as a datatype, and when it is clear from the context we will also write these as d.

$$\begin{array}{rrcl} (\textsc{Monotypes}) & T, U, V & ::= & \alpha \mid b \mid d\ \vec{T} \mid T_1 \to T_2 \\ (\textsc{Type schemes}) & S & ::= & T \mid \forall \alpha.\, S \end{array}$$

We write $\mathsf{Dt}\ D$ to stand for the set of all datatypes with datatype identifiers drawn from the set D. $\mathsf{Ty}\ D$ and $\mathsf{Sch}\ D$ are defined similarly for monotypes and schemes. We consider monotypes to be a trivial instance of type schemes where convenient. The purpose of distinguishing base types from datatypes is that the former may not be refined. For example, we will consider Int to be a base

type, Fm a datatype identifier and Fm Int a datatype. Type schemes are identified up to renaming of bound variables.

Lifting over types. Given a relation on datatypes $R \subseteq \mathsf{Dt}\ D_1 \times \mathsf{Dt}\ D_2$, we write $\mathsf{Ty}(R)$ for the relation on $\mathsf{Ty}\ D_1 \times \mathsf{Ty}\ D_2$ defined inductively by the following:

$$\frac{}{\mathsf{Ty}(R)(\alpha,\ \alpha)} \qquad \frac{}{\mathsf{Ty}(R)(b,\ b)} \qquad \frac{}{\mathsf{Ty}(R)(d_1,\ d_2)}\, R(d_1,\ d_2) \qquad \frac{\mathsf{Ty}(R)(T_3,\ T_1) \quad \mathsf{Ty}(R)(T_2,\ T_4)}{\mathsf{Ty}(R)(T_1 \to T_2,\ T_3 \to T_4)}$$

Expressions and modules. We assume a countable collection of term variables, ranged over by x, y, z, and variations. As well as a countable collection $\mathbb{K}$ of datatype constructors, ranged over by k. The arity of a constructor is denoted $\mathsf{Arity}(k)$. The expressions of the language are:

$$\begin{array}{rcl} m & ::= & \epsilon \mid m \cdot \langle x = e \rangle \\ e & ::= & c \mid k \mid e_1\ e_2 \mid e\ T \mid \lambda x : T.\, e \mid \Lambda\alpha.\, e \\ & & \mid \mathsf{case}\ e\ \mathsf{of}\ \{k_1\ \vec{x_1} \mapsto e_1 \mid \cdots \mid k_n\ \vec{x_n} \mapsto e_n\} \end{array}$$

Expressions are identified up to renaming of bound variables and we will adopt the Barendregt variable convention in order to retain a simple notation. Since we are defining a refinement type system, we will assume that the input program already has a typing assigned by the underlying type system of the programming language. We assume that this is manifest, in part, by the insertion of appropriate type abstraction $\Lambda\alpha.\, e$ and application $e\ T$ terms, and by the annotation of term abstractions with their argument type $\lambda x : T.\, e$. We also assume, as is the case for GHC Core, that pattern matching has been preprocessed into case expressions in which patterns are 1 level deep, i.e have the form $k\ x_1\ \cdots\ x_n$ for some constructor k. Since our analysis is path sensitive, this is not a restriction and we could perform an equivalent, but clumsier, development allowing for syntax containing nested patterns.

Modules m are simply a sequence of variable definitions $\langle x = e \rangle$ that may be empty ϵ. For simplicity of presentation, we allow recursive definitions but not mutually recursive definition sets.

Datatype environments. The meaning of datatypes is defined by an environment of datatype definitions. Each datatype definition introduces a new datatype identifier along with a collection of datatype constructors that can be used to build instances of the type.

Definition 1 (Datatype Environment). A *datatype environment* is a pair consisting of a set $D \subseteq \mathbb{D}$ of datatypes identifiers and a function $\Delta : D \to \mathsf{Map}\ \mathbb{K}\ (\mathsf{Sch}\ D)$ mapping each datatype identifier d to a finite map which records the associated constructors and their type. We assume these types only concern the type variables that appear in datatype's definition, and so is of shape: $\forall\vec{\alpha}.\, U_1 \to \cdots \to U_m \to d\ \vec{\alpha}$, where m is the constructor's arity. For convenience, and as the return type of a constructor is predetermined, we will often identify $\Delta(d)(k)$ with just the sequence of types corresponding to a constructor's arguments, i.e. $[U_1,\ ..,\ U_m]$ where $U_i = \Delta(d)(k)[i]$.

Since datatype environments are partial functions on $\mathbb{D}$, they inherit the natural partial order in which $\Delta_1 \subseteq \Delta_2$ just if the graph of the former is included in the graph of the latter. If $\Delta_1 \subseteq \Delta_2$ we say that Δ_1 is a *subenvironment* of Δ_2.

Note that the notion of subenvironment only concerns the datatypes that are defined in an environment and not the definitions of those datatypes (the constructors and their types), which will be treated by the notion of *refinement* in the sequel.

3 DATATYPE REFINEMENT

Henceforth we will fix a particular datatype environment $\underline{\Delta} : \underline{D} \to \mathsf{Map}\ \mathbb{K}\ (\mathsf{Sch}\ \underline{D})$ which we call the *underlying datatype environment*. We think of $\underline{\Delta}$ as the datatype environment that is provided by the programmer by their datatype definitions.

Example 2. We will use the following as running example of underlying datatype environment. Consider the datatype Lam of λ-terms with arithmetic using a locally nameless representation:

data Arith = Lit Int | Add | Mul
data Lam = Cst Arith | BVr Int | FVr String | Abs Lam | App Lam Lam

These datatypes are slightly artificial, but they allow us to illustrate several features of the definitions in one example. For simplicity, we will consider Int and String to be base types (which will, therefore, not be refined).

The underlying datatype environment contains definitions for all the datatypes declared by the programmer. Some datatype definitions require the definitions of other datatypes to be understood properly. For example, to understand Lam, one must also understand the definition of Arith since one is defined in terms of the other. There is a notion of a subenvironment that contains all and only those definitions that are needed to understand one particular datatype.

Definition 3 (Slice). Suppose $\Delta : D \to \mathsf{Map}\ \mathbb{K}\ (\mathsf{Sch}\ D)$. One can always construct a subenvironment of Δ by starting from a given datatype $d \in D$ and closing under transitive dependencies. Define the *slice of d through* Δ, written $\langle d \rangle_\Delta$, as the least subenvironment of Δ containing d.

For example, in the environment $\underline{\Delta}$ described in Example 2, we have $\langle \mathsf{Lam} \rangle_{\underline{\Delta}}$ being the whole environment, and $\langle \mathsf{Arith} \rangle_{\underline{\Delta}}$ being just the definition of Arith itself.

Definition 4 (Refinement). We say that a datatype environment $\Delta_1 : D \to \mathsf{Map}\ \mathbb{K}\ (\mathsf{Sch}\ D)$ is a *refinement* of a datatype environment $\Delta_2 : D \to \mathsf{Map}\ \mathbb{K}\ (\mathsf{Sch}\ D)$ just if the definitions of the datatypes in Δ_1 are bounded above by the definitions in Δ_2, that is: for all $d \in D$, $\Delta_1(d) \subseteq \Delta_2(d)$ (i.e. the graph of the first map is included in the graph of the second).

Suppose $\Delta_1(d)(k)$ is some type scheme S and Δ_1 is a refinement of Δ_2, then $\Delta_2(d)(k)$ is the same S. So, the refinements of $\Delta : D \to \mathsf{Map}\ \mathbb{K}\ (\mathsf{Sch}\ D)$ are in one-to-one correspondence with choices of constructors for each of the constituent datatypes. More precisely, every function $f : \Pi d \in D.\ \mathcal{P}(\mathsf{dom}\ \Delta(d))$ determines a refinement Δ_f satisfying:

$$\Delta_f(d)(k) = \begin{cases} \Delta(d)(k) & \text{if } k \in f(d) \\ \bot & \text{otherwise} \end{cases}$$

and each refinement arises in this way.

Example 5. The following refinement of the underlying environment from Example 2 describes a type of closed, applicative terms over linear arithmetic.

data Arith = Lit Int | Add Arith
data Lam = Cst Arith | App Lam Lam

This refinement is determined by the choice f_{LL} satisfying:

$$f_{LL}(\mathsf{Arith}) = \{\mathsf{Lit}, \mathsf{Add}\} \qquad f_{LL}(\mathsf{Lam}) = \{\mathsf{Cst}, \mathsf{App}\}$$

For the purpose of assigning types to the program, we construct a new datatype environment consisting of all possible refinements of the underlying environment supplied by the programmer[3].

Definition 6 (The Intensional Refinement Environment). Given a family of datatype environments $\Delta_{i\in I} : D_i \to \mathsf{Map}\ \mathbb{K}\ (\mathsf{Sch}\ D_i)$ we define their coproduct as the environment:

$$\coprod_{i\in I} \Delta_i : (\coprod_{i\in I} D_i) \to \mathsf{Map}\ \mathbb{K}\ (\mathsf{Sch}\ (\coprod_{i\in I} D_i))$$

whose domain is simply a disjoint sum of sets (as defined in the preliminaries). The coproduct comes equipped with canonical injections inj_i satisfying $(\coprod_{i\in I} \Delta_i)(\mathsf{inj}_i d)(d)(k) = \mathsf{inj}_i(\Delta_i(d)(k))$ wherever the latter is properly defined.

The *intensional refinement environment*, written Δ^*, is the coproduct of all the refinements of $\underline{\Delta}$:

$$\Delta^* \coloneqq \coprod_{f\in I} \underline{\Delta}_f : D^* \to \mathsf{Map}\ \mathbb{K}\ (\mathsf{Sch}\ D^*)$$

where I is the set $\Pi\, \underline{d} \in \underline{D}.\ \mathcal{P}(\mathsf{dom}\ \underline{\Delta}(\underline{d}))$ of functions from underlying datatypes d to appropriate subsets of constructors. That is, the set of all possible refinements. We write the domain of this coproduct as D^*.

Note that, formally, the datatype identifiers whose definitions are given in the intensional refinement environment are of shape $\mathsf{inj}_\Delta\, d$ with $d \in \mathbb{D}$. This is convenient because one can read such a name as "the refinement of d whose definition is Δ". However, we will continue to use more friendly names (and Haskell notation) in our examples.

Example 7. The type of closed, applicative terms over linear arithmetic from Example 5 can be found in Δ^* alongside the type LArith of linear arithmetic constants:

$$\mathsf{LATm}_0 \coloneqq \mathsf{inj}_{f_{LL}}\ \mathsf{Lam} \qquad \mathsf{LArith} \coloneqq \mathsf{inj}_{f_{LA}}\ \mathsf{Arith}$$

the latter being defined by $f_{LA}(\mathsf{Arith}) = \{\mathsf{Lit}, \mathsf{Add}\}$ and $f_{LA}(\mathsf{Lam}) = \emptyset$. This datatype could equivalently be defined as $\mathsf{inj}_{f_{LL}}\ \mathsf{Arith}$ (or in many other ways). Other refinement datatypes defined include the type LLam_0 of closed λ-terms over linear arithmetic, the type ATm of applicative terms and the type ATm_0 of closed applicative terms, whose definitions we give in convenient notation:

data ATm_0 = Cst Arith | App ATm_0 ATm_0
data ATm = FVr String | Cst Arith | App ATm ATm
data LLam_0 = Cst LArith | BVr Int | Abs LLam_0 | App LLam_0 LLam_0

Definition 8 (Refinement Type). A type (scheme) $S \in \mathsf{Sch}\ D^*$ is said to be a *refinement type*. Refinement types come equipped with an *underlying type*, written $\mathcal{U}(S)$, which is a type (scheme) in $\mathsf{Sch}\ \underline{D}$ defined recursively as follows:

$$\begin{aligned}
\mathcal{U}(a) &= a \\
\mathcal{U}(b) &= b \\
\mathcal{U}(\mathsf{inj}_f d\ \vec{T}) &= d\ \overrightarrow{\mathcal{U}(T)} \\
\mathcal{U}(T_1 \to T_2) &= \mathcal{U}(T_1) \to \mathcal{U}(T_2) \\
\mathcal{U}(\forall a.\ S) &= \forall a.\ \mathcal{U}(S)
\end{aligned}$$

[3]It would suffice to take all the refinements of all the slices (which itself still includes some redundancy, but this would complicate the definitions for no practical gain)

$$\text{(SShape)}\ \frac{}{\vdash T_1 \not\sqsubseteq T_2} \;\Big|\; \mathcal{U}(T_1) \neq \mathcal{U}(T_2)$$

$$\text{(SMis)}\ \frac{}{\vdash d_1\ \overrightarrow{T_1} \not\sqsubseteq d_2\ \overrightarrow{T_2}} \;\Big|\; \mathsf{dom}(\Delta^*(d_1)) \not\subseteq \mathsf{dom}(\Delta^*(d_2))$$

$$\text{(SSim)}\ \frac{\vdash U_{1_i}[\overrightarrow{T_1/\alpha}] \not\sqsubseteq U_{2_i}[\overrightarrow{T_2/\beta}]}{\vdash d_1\ \overrightarrow{T_1} \not\sqsubseteq d_2\ \overrightarrow{T_2}} \;\Bigg|\; \begin{array}{l} m = \mathsf{Arity}(k),\ i \in [1..m] \\ \Delta^*(d_1)(k) = \forall \overrightarrow{\alpha}.\ U_{1_1} \to \cdots U_{1_m} \to d_1\ \overrightarrow{\alpha} \\ \Delta^*(d_2)(k) = \forall \overrightarrow{\beta}.\ U_{2_1} \to \cdots U_{2_m} \to d_2\ \overrightarrow{\beta} \end{array}$$

$$\text{(SArrL)}\ \frac{\vdash T_1' \not\sqsubseteq T_1}{\vdash T_1 \to T_2 \not\sqsubseteq T_1' \to T_2'} \qquad \text{(SArrR)}\ \frac{\vdash T_2 \not\sqsubseteq T_2'}{\vdash T_1 \to T_2 \not\sqsubseteq T_1' \to T_2'}$$

Fig. 3. The complement of the subtyping relation on monotypes.

In the following, we will assume that we are given a program equipped with a complete underlying typing, that is: every subterm M has an associated type $S \in \mathsf{Sch}\ \underline{D}$. Our task will be to find a new *refinement* typing: an assignment to each subterm M of a refinement type $S' \in \mathsf{Sch}\ D^*$ that has the same "shape" as the underlying type S of M in the sense that $\mathcal{U}(S') = S$.

4 SUBTYPING

Refinement induces a natural ordering on refinement datatypes according to which constructors are available in their definition. This ordering can then be lifted to all types built over those datatypes in the obvious way.

Definition 9 (Subtyping). The judgement $\vdash T_1 \sqsubseteq T_2$ is defined coinductively, using the inductive system of rules in Figure 3 to characterise the complement. We extend subtyping to two schemes that have the same quantifier prefix, writing: $\vdash \forall \overrightarrow{\alpha}.\ T_1 \sqsubseteq \forall \overrightarrow{\alpha}.\ T_2$ whenever $\vdash T_1 \sqsubseteq T_2$. We say that two types T_1 and T_2 are *subtype equivalent* and write $\vdash T_1 \equiv T_2$ just if $\vdash T_1 \sqsubseteq T_2$ and $\vdash T_2 \sqsubseteq T_2$.

Intuitively, refinement specifies the possible shapes of types that are then interrelated by subtyping. Refinement is a covariant treatment of arrow types, since we have $\mathcal{U}(T_1 \to T_2) = T_1' \to T_2'$ iff $\mathcal{U}(T_1) = T_1'$ and $\mathcal{U}(T_2) = T_2'$. On the other hand, as can be seen from the definition, subtyping interprets the argument type contravariantly. Consequently, there are some $\mathcal{U}(T) = \underline{T}$ for which $T \sqsubseteq \underline{T}$ and other $\mathcal{U}(T) = \underline{T}$ for which $\underline{T} \sqsubseteq T$ (viewing an underlying type as its own trivial refinement).

We give the definition coinductively because, as usual, there is a notion of simulation that arises naturally from our coalgebraic view of datatype environments. Consequently, it is most straightforward to think of the defining rules as providing a system in which to construct finite refutations of subtype inequalities $T_1 \not\sqsubseteq T_2$, which will, ultimately, fail to hold either because the types T_1 and T_2 have a different shape, or because T_1 provides some constructor that T_2 does not.

Example 10. Following the running example, the judgement $\vdash \mathsf{LATm}_0 \to \mathsf{String} \not\sqsubseteq \mathsf{ATm}_0 \to \mathsf{String}$ follows by a simple refutation:

$$\textsf{(SArrL)}\ \frac{\textsf{(SSim)}\ \dfrac{\textsf{(SMis)}\ \dfrac{}{\vdash \mathsf{Arith} \not\sqsubseteq \mathsf{LArith}}}{\vdash \mathsf{ATm}_0 \not\sqsubseteq \mathsf{LATm}_0}}{\vdash \mathsf{LATm}_0 \to \mathsf{String} \not\sqsubseteq \mathsf{ATm}_0 \to \mathsf{String}}$$

Conversely, we can use the coinduction principle to show that $\vdash T_1 \sqsubseteq T_2$, in which case we require a model[4] of $\sqsubseteq$ that contains $(T_1,\ T_2)$.

Example 11. For $\vdash \mathsf{ATm}_0 \to \mathsf{String} \sqsubseteq \mathsf{LATm}_0 \to \mathsf{String}$ we provide the following witness:

$$\left\{ \begin{array}{c} (\mathsf{ATm}_0 \to \mathsf{String}, \mathsf{LATm}_0 \to \mathsf{String}),\ (\mathsf{String}, \mathsf{String}),\ \\ (\mathsf{LATm}_0, \mathsf{ATm}_0),\ (\mathsf{LArith}, \mathsf{Arith}),\ (\mathsf{Int}, \mathsf{Int}) \end{array} \right\}$$

It can be easily verified that this set is a model of the defining rules for $\sqsubseteq$ and hence, by coinduction, is contained within it.

However, such models can be a bit unwieldy in general as the types involved get more complex. We can do better by observing that the definition can be approximated by a coinductive part, concerning datatypes, and an inductive part, by which a subtyping relationship between datatypes is lifted to all types. Consequently, we need only find a model of the coinductive part, which is much neater since it only concerns $\mathsf{Dt}\ D \times \mathsf{Dt}\ D$. The following can be shown by a straightforward coinduction.

Lemma 1 (Simulation). Let $R \subseteq \mathsf{Dt}\ D \times \mathsf{Dt}\ D$ and suppose that, for all $(d_1\ \overrightarrow{T_1},\ d_2\ \overrightarrow{T_2}) \in R$, and for all k such that $\Delta(d_1)(k)$ is defined:

- $\Delta(d_2)(k)$ is defined.
- And, moreover, $\mathsf{Ty}(R)(U_{1_i},\ U_{2_i})$ for each $i \in [1..\mathsf{Arity}(k)]$, where $\overrightarrow{U_1}$ and $\overrightarrow{U_2}$ are the argument types of $\Delta(d_1)(k)$ and $\Delta(d_2)(k)$ instantiated at $\overrightarrow{T_1}$ and $\overrightarrow{T_2}$ respectively.

Then it follows that $\mathsf{Ty}(R)$ is included in the subtype relation.

Using this result, it suffices to exhibit $R \coloneqq \{(\mathsf{LATm}_0, \mathsf{ATm}_0),\ (\mathsf{LArith}, \mathsf{Arith})\}$ in order to conclude e.g. $\vdash \mathsf{ATm}_0 \to \mathsf{String} \sqsubseteq \mathsf{LATm}_0 \to \mathsf{String}$. Intuitively, this witness determines the model $\mathsf{Ty}(R)$, which contains the model of Example 11.

5 REFINEMENT TYPE ASSIGNMENT

In this section, we present a refinement type system whose purpose is to exclude the possibility of pattern-match failure. To achieve this, the typing rule for pattern-matching requires that cases are exhaustive according to the type of the scrutinised expression. However, the system allows for all refinement datatypes and incorporates the above notion of subtyping, which allows for the scrutinised expression to be typed much more precisely than is possible in the underlying type system.

For the purpose of defining the refinement type system, we make some standard Hindley-Damas-Milner assumptions about the underlying type system, namely that type application happens immediately after introducing a variable of polymorphic type and type abstraction happens only at the point of definition. As a minor simplification, we assume that constants are monomorphic and write $\mathbb{C}(c)$ for the monotype assigned to c axiomatically. Since this is a refinement type system, we assume that all expressions have already been assigned an underlying type, which we will typically

[4]By which we mean a set of pairs of types satisfying all $\not\sqsubseteq$-defining rules.

$$\text{(TModE)}\ \frac{}{\Gamma \vdash \epsilon : \Gamma} \qquad \text{(TModD)}\ \frac{\Gamma \vdash m : \Gamma' \quad \Gamma' \cup \{x : T\} \vdash e : T' \quad \vdash T' \sqsubseteq T}{\Gamma \vdash m \cdot \langle x = \Lambda \vec{\alpha}.\, e \rangle : \Gamma' \cup \{x : \forall \vec{\alpha}.\, T\}}$$

Fig. 4. Typing for modules.

write with an underline to aid readability. We will only need to consult these underlying types when they appear in the syntax, e.g. abstraction and type application.

Additionally, we relax the normal definition of a type environment from a function to a relation. Program variables may, therefore, have many types as long as they refine the same underlying type. This assumption is equivalent to allowing environment-level intersection types.

Definition 12 (Type assignment). A *type environment*, typically Γ or Δ, is a finite relation between program variables x and type schemes S, whose elements are typically written $x : S$. We require that $x : S_1 \in \Gamma$ and $x : S_2 \in \Gamma$ implies $\mathcal{U}(S_1) = \mathcal{U}(S_2)$. This ensures that $\mathcal{U}(\Gamma)$ can be defined in the obvious way. The type assignment system is divided into two sets of rules, for expressions (Figure 5) and for modules (Figure 4), defining judgements, respectively:

$$\Gamma \vdash e : T \qquad \Gamma \vdash m : \Delta$$

in which $\mathcal{U}(\Gamma) \vdash e : \mathcal{U}(T)$ and $\mathcal{U}(\Gamma) \vdash m : \mathcal{U}(\Delta)$ are the underlying typings, provided by the programming language, for the expression e and the module m respectively.

The system is conceptually similar to an underlying ML-style system, but note:

- Any suitable refinement datatype d can be used in order to type a datatype constructor or the scrutinee of a case statement.
- The notion of subtyping from the previous section is incorporated through a subsumption rule (recall that $\vdash T_1 \sqsubseteq T_2$ implies that T_1 and T_2 have the same shape according to $\mathcal{U}$).
- The pattern-matching rule is restricted by a condition requiring that cases are exhaustive.
- The branches of the case expression only need to be typed if the branch is reachable, incorporating path-sensitivity. This relaxation only makes sense for a refinement type system, because reachability is encoded by choosing an appropriate refinement d in the rule (TCase) . From an operational point of view it makes no difference to the set of computations expressible.
- Finally, everywhere a particular underlying type is required by the syntax, an arbitrary choice of refinement type of the appropriate shape can be made in its place.

As discussed in the introduction, allowing several types for each term ensures they can be used in different contexts. This approach is more lightweight than an intersection type system, and arguably easier for programmers to reason about if types are to be considered as certificates. When it comes to algorithmic inference, however, the non-deterministic aspect would be problematic. Instead, in Section 7, we rely on *refinement polymorphism* to summarise *every* typing of a variable in some environment compactly by a single constrained type scheme. The polymorphism of this kind is no different from that of the Hindley-Milner system, which could equally be viewed as an infinite intersection type system, or indeed allowing several typings of the same variable in an environment. Likewise, it is simpler to define polymorphic constructors and datatypes, than to consider each instantiation separately.

$$\text{(TVar)}\ \frac{}{\Gamma \vdash x\ \overrightarrow{\underline{T}} : T[\overrightarrow{S/\alpha}]} \;\Big|\; \begin{array}{l}\mathcal{U}(\overrightarrow{T}) = \overrightarrow{\underline{T}} \\ x : \forall \overrightarrow{\alpha}.\, T \in \Gamma\end{array} \qquad \text{(TCst)}\ \frac{}{\Gamma \vdash c : \mathbb{C}(c)}$$

$$\text{(TCon)}\ \frac{}{\Gamma \vdash k\ \overrightarrow{\underline{T}} : T[\overrightarrow{S/\alpha}]} \;\Big|\; \begin{array}{l}\mathcal{U}(\overrightarrow{T}) = \overrightarrow{\underline{T}} \\ k \in \mathsf{dom}\,(\Delta^*(d)) \\ \Delta^*(d)(k) = \forall \overrightarrow{\alpha}.\, T\end{array} \qquad \text{(TSub)}\ \frac{\Gamma \vdash e : T_1}{\Gamma \vdash e : T_2} \;\Big|\; \vdash T_1 \sqsubseteq T_2$$

$$\text{(TAbs)}\ \frac{\Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash \lambda x{:}\underline{T_1}.\, e : T_1 \to T_2} \;\Big|\; \begin{array}{l}\mathcal{U}(T_1) = \underline{T_1} \\ x \notin \mathsf{dom}\,\Gamma\end{array} \qquad \text{(TApp)}\ \frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

$$\text{(TCase)}\ \frac{\Gamma \vdash e : d\ \overrightarrow{T} \quad (\forall i \le m)\ \Gamma \cup \overrightarrow{\{x_i : \Delta^*(d)(k_i)[\overrightarrow{T/\alpha}]\}} \vdash e_i : T}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{|_{i=1}^{m}\ k_i\ \overrightarrow{x_i} \mapsto e_i\} : T} \;\Big|\; \mathsf{dom}(\Delta^*(d)) = \{k_1, \ldots, k_m\}$$

Fig. 5. Type assignment for expressions.

Example 13. Recall the refinements of Example 7 and consider the function cloSub, with underlying type List (String × Lam) → Lam → Lam, whose purpose is to close an applicative term by substituting closed terms everywhere.

```
cloSub m t =
 case t of
  FVr s  →  lkup m s
  Cst c  →  Cst c
  App u v →  App (cloSub m u) (cloSub m v)
```

To keep the example simple, we assume that the lookup function lkup has the following type: $\forall\alpha$. List (String × α) → String → α in the environment. Consequently, in our official syntax, e.g. the FVr case really contains an explicit type application: lkup Lam *m s*. Then the function cloSub can be assigned the refinement type[5]: List (String × ATm_0) → ATm → ATm_0. Thus expressing the fact that the application of a *closing* substitution to a arbitrary *applicative* term yields a *closed applicative* term.

This is possible due to a combination of the features of the system. First, observe that it is possible, in the abstraction rule, to assume that the bound variable *m* of underlying type List (String × Lam) has type List (String × ATm_0) in (TAbs) since it can easily be seen that the former is a refinement of the latter. Then it follows that lkup Lam *m i* can be assigned the type Atm_0 by choosing Atm_0 for *T* in (TVar) . Second, under the assumption that the bound variable *t* has refinement type Atm, it follows from (TCase) that the variable *c* that is bound by the case Cst *c* can be assigned the type Arith. Note that the rule (TCase) is applicable only because we have chosen the refinement ATm of Lam which guarantees that the input will not contain any abstractions. Then, in the body of the case, we can choose instead the more specific typing Cst : Arith → ATm_0. Similarly, *u* and *v* are assigned the type ATm so that the subexpressions cloSub *m u* and cloSub *m v* in the body of the

[5]Here, List and × can be understood as the trivial refinements of their namesakes, i.e. with all constructors available.

final case can be assigned the type ATm_0. Then the type of the body as a whole, and therefore the entire case analysis, is also Atm_0.

The central problem is typability, for closed expressions: given an underlying datatype environment $\underline{\Delta}$ and a closed module m which is typed in $\underline{\Delta}$, does there exist a refinement type assignment to the functions of m? Typically m will contain library functions whose source is not available to the system, but for which an underlying type is known. To incorporate such functions we interpret an underlying type environment $\underline{\Gamma}$ as containing trivial refinement types for each such function, i.e. each d occurring in such a type Γ denotes the refinement of d that makes available all constructors.

Definition 14 (Typability). A triple $\underline{\Delta}$, Γ and m constitutes a positive instance of the *refinement typability problem* just if there is a refinement type environment Γ' such that $\Gamma \vdash m : \Gamma'$. In such a case, we say that $\underline{\Delta}; \Gamma \vdash m$ is *refinement typable*.

The rest of the paper concerns the algorithmic solution of the typability problem.

6 CONSTRUCTOR SET CONSTRAINTS

We assume a countable set of *refinement variables*, ranged over by X, Y, Z and so on. The purpose of a refinement variable X is to represent a function in $\Pi d \in \underline{D}.\ \mathcal{P}(\mathsf{dom}\,\underline{\Delta}(d))$. As described in Section 3, such functions are in 1-1 correspondence with refinements of $\underline{\Delta}$. We will abuse notation and write X for both uses (thus the following rather strange-looking equation $X(d) = \mathsf{dom}(X(d))$ holds by interpreting each of the two occurrences of X according to its context.)

Definition 15 (Constraints). A *constructor set expression*, typically S, is either a finite set of constructors $\{k_1, \ldots, k_m\}$ or a pair $X(\underline{d})$ consisting of a refinement variable X and an underlying datatype $\underline{d}$. The underlying type of the constructor set expression is (partially) defined as follows:

$$\mathcal{U}(X(\underline{d})) = \underline{d} \qquad \mathcal{U}(\{k_1, \ldots, k_m\}) = \underline{d} \quad \text{if } \forall i \in [1..m].\ k_i \in \mathsf{dom}\,\underline{\Delta}(\underline{d})$$

We consider only those constructor set expressions for which the underlying type is defined. We write $\mathsf{FRV}(S)$ for the set of refinement variables occurring anywhere in S (which will either be empty or a singleton).

An *inclusion constraint* is an ordered pair of constructor set expressions, written (suggestively) as $S_1 \subseteq S_2$. When S_1 is a singleton $\{k\}$, we will rather write the pair as $k \in S_2$. We shall only consider inclusion constraints in which both set expressions have the same underlying type. The refinement variables of an inclusion constraint $\mathsf{FRV}(S_1 \subseteq S_2)$ are defined by extension from $\mathsf{FRV}(S_1)$ and $\mathsf{FRV}(S_2)$ in the obvious way.

A *conditional constraint*, hereafter just *constraint*, is a pair $\phi \ ?\ S_1 \subseteq S_2$ consisting of a set of inclusion constraints ϕ and an inclusion constraint $S_1 \subseteq S_2$. The set ϕ is called the *guard* and the inclusion $S_1 \subseteq S_2$ the *body*. We will only consider conditional constraints in which each element of the guard has shape $k \in X(\underline{d})$. When the guard of a constraint $\emptyset \ ?\ S_1 \subseteq S_2$ is trivial, we shall usually omit it and write only the body $S_1 \subseteq S_2$. The set of refinement variables $\mathsf{FRV}(\phi \ ?\ \mathsf{FRV}(S_1 \subseteq S_2))$ of a constraint is defined as usual.

Sometimes we shall guard a constraint set C, and write $\phi \ ?\ C$ for the set $\{\psi \cup \phi \ ?\ S_1 \subseteq S_2 \mid \psi \ ?\ S_1 \subseteq S_2 \text{ an element of } C\}$. We write $\mathsf{FRV}(C)$ for the set of refinement variables occurring in C.

Intuitively, an inclusion $S_1 \subseteq S_2$ is satisfied by any assignment to the refinement variables that makes S_1 included in S_2. A constraint $\phi \ ?\ S_1 \subseteq S_2$ is satisfied if either some inclusion in the guard is not satisfied or the body is satisfied.

Definition 16 (Satisfaction). A *constructor set assignment*, hereafter just *assignment*, is a total map θ taking each refinement variable X to a constructor choice function from $\Pi d \in \underline{D}.\ \mathcal{P}(\mathsf{dom}\,\underline{\Delta}(d))$.

$$\text{(ISBase)}\ \frac{}{\vdash b \sqsubseteq b \Longrightarrow \emptyset} \qquad \text{(ISTyVar)}\ \frac{}{\vdash \alpha \sqsubseteq \alpha \Longrightarrow \emptyset}$$

$$\text{(ISArr)}\ \frac{\vdash T_{21} \sqsubseteq T_{11} \Longrightarrow C_1 \quad \vdash T_{12} \sqsubseteq T_{22} \Longrightarrow C_2}{\vdash T_{11} \to T_{12} \sqsubseteq T_{21} \to T_{22} \Longrightarrow C_1 \cup C_2}$$

$$\text{(ISData)}\ \frac{(\forall ki.)\ \vdash \mathsf{inj}_X(U_i)[\overrightarrow{T_X/\alpha}] \sqsubseteq \mathsf{inj}_Y(U_i)[\overrightarrow{T_Y/\alpha}] \Longrightarrow C_{k_i}}{\vdash \mathsf{inj}_X\, d\, \overrightarrow{T_X} \sqsubseteq \mathsf{inj}_Y\, d\, \overrightarrow{T_Y} \Longrightarrow C} \quad \left| \begin{array}{l} \underline{\Delta}(\underline{d})(k) = \\ \quad \forall \overrightarrow{\alpha}.\, U_1 \to \cdots U_n \to d\ \overrightarrow{\alpha} \\ C = \{X(\underline{d}) \subseteq Y(\underline{d})\} \cup \\ \quad \bigcup_k \bigcup_{i=1}^{n} (k \in X(\underline{d}))\ ?\ C_{k_i} \end{array} \right.$$

Fig. 6. Inference for subtype inequalities.

The meaning of a constructor set expression S under an assignment θ is a set of constructors $\theta[\![S]\!]$ defined as follows:

$$\theta[\![X(\underline{d})]\!] = \theta(X)(\underline{d}) \qquad \theta[\![\{k_1, \ldots, k_m\}]\!] = \{k_1, \ldots, k_m\}$$

An inclusion constraint $S_1 \subseteq S_2$ is *satisfied* by an assignment θ, written $\theta \models S_1 \subseteq S_2$ just if $\theta[\![S_1]\!]$ is included in $\theta[\![S_2]\!]$. A constraint $\phi\ ?\ S_1 \subseteq S_2$ is *satisfied* by an assignment θ, written $\theta \models \phi\ ?\ S_1 \subseteq S_2$ just if, whenever $\theta \models k \in X(d)$ for every inclusion constraint $k \in X(d)$ in ϕ, then $\theta \models S_1 \subseteq S_2$.

Definition 17 (Solutions). A *solution* to a constraint set C is an assignment θ satisfying every constraint in C, we write $\theta \models C$. We say that C is *solvable*, or *satisfiable*, just if it has a solution.

Remark 1. The full set constraint language is exactly the monadic class of first-order propositions [Bachmair et al. 1993]. By applying the translation of that paper, it can be shown that guarded constraints of the form laid out above are (monadic) Horn clauses with constructors simply interpreted as constants.

7 TYPE INFERENCE

Since our system is effectively syntax directed (the subsumption rule can be factored into the other syntax-directed rules), type inference follows a standard pattern of constraint generation and satisfiability checking (see e.g. [Odersky et al. 1999]). The constraints are subtype inequalities over refinement variables, but it is easily seen that, in our restricted setting, such inequalities are equivalent to conditional inclusion constraints between refinement variables and sets of datatype constructors. To enable this approach, we extend the language of types so to allow datatypes parametrised by refinement variables.

Definition 18 (Extended Types). The *extended types* are monotypes extended with datatypes built over refinement variables:

$$T, U, V ::= \cdots \mid \mathsf{inj}_X\, d\, \overrightarrow{T}$$

Note that the type arguments to an injected datatype identifier are also extended. Expressions of the form $(\mathsf{inj}_X\ \mathsf{List})(\mathsf{inj}_Z\ \mathsf{Int})$ are, therefore, well-formed. Recall from Section 3 that refinement datatype identifiers are of the form $\mathsf{inj}_\Delta\, d$, with d an underlying datatype identifier, and should be thought of as specifying the refinement of d whose datatype definition is given by Δ. The task of inference is to determine constraints on these Δ that enable a typing to be assigned and check that the constraints have a solution.

For convenience, we shall implicitly lift injections to any type, or sequence of types, written $\mathsf{inj}_X\, T$, so that the injection is distributed over datatypes in T. In the context of extended types,

we will associate a substitution action θT with each constructor set assignment θ by lifting the definition $\theta(\mathsf{inj}_X\, d) \coloneqq \mathsf{inj}_{\theta(X)}\, d$ homomorphically over all extended types. Finally, we write $\mathsf{FRV}(T)$ for the set of refinement variables occurring in injections in T.

We also adopt an extension of type schemes that are constrained:

Definition 19 (Constrained Type Scheme). We subsume the type scheme S by the *constrained type scheme*, which has shape: $\forall \vec{\alpha}.\, \forall \vec{X}.\, C \supset T$, where C is a constraint set and T is an *extended type*. We define $\mathsf{FRV}(\forall \vec{\alpha}.\, \forall \vec{X}.\, C \supset T) = (\mathsf{FRV}(C) \cup \mathsf{FRV}(T)) \setminus \vec{X}$. A *constrained type environment* is a finite mapping from program variables to constrained type schemes, whose elements are written $x : S$. We define $\mathsf{FRV}(\Gamma)$ in the obvious way.

As is typical, there is generally no "best" monotype solution to a set of inclusion constraints, so constrained type schemes give us an internal representation for the set of all types assignable to a module-level function. For example, assuming constant combinator K defined as usual, it can be seen that the module-level recursive function $f = \lambda x : \mathsf{Lam}.\, \mathrm{K}\ [\mathsf{Lam},\ \mathsf{Lam}]\ x\ (f\ (f\ x))$ can be assigned the constrained type scheme : $\forall XY.\, C \supset \mathsf{inj}_X\, \mathsf{Lam} \to \mathsf{inj}_Y\, \mathsf{Lam}$, with C:

$$X(\mathsf{Lam}) \subseteq Y(\mathsf{Lam}) \qquad Y(\mathsf{Lam}) \subseteq X(\mathsf{Lam})$$
$$\mathsf{Cst} \in X(\mathsf{Lam})\ ?\ X(\mathsf{Arith}) \subseteq Y(\mathsf{Arith}) \qquad \mathsf{Cst} \in Y(\mathsf{Lam})\ ?\ Y(\mathsf{Arith}) \subseteq X(\mathsf{Arith})$$

Intuitively, its input flows to its output and conversely, so we require $\mathsf{inj}_X\, \mathsf{Lam} \sqsubseteq \mathsf{inj}_Y\, \mathsf{Lam}$ and $\mathsf{inj}_Y\, \mathsf{Lam} \sqsubseteq \mathsf{inj}_X\, \mathsf{Lam}$ (which is encoded by the above set constraints when we view the refinements X and Y as functions specifying the choice of constructors). However, there is obviously no "best" instantiation of refinement variables X and Y.

Constrained type environments can be understood as compact descriptions of "ordinary" type environments (in the sense of Definition 12), which is made precise as follows.

Definition 20. Define $(\!|\Gamma|\!)$ for the type environment that can be obtained from the *closed* constrained type schemes in Γ, by instantiation of refinement quantifiers with every possible solution, that is, supposing Γ is closed: $(\!|\Gamma|\!) \coloneqq \{\ x : \forall \vec{\alpha}.\, \theta T \mid \theta \models C \wedge (x : \forall \vec{\alpha}.\, \forall \vec{X}.\, C \supset T) \in \Gamma\ \}$.

Typical presentations of type inference by constraint generation involve choosing fresh type variables, which are then constrained. Since we work with refinement types, it is more convenient to choose fresh refinement type templates, which are just refinement types that are everywhere parametrised by fresh refinement variables — in the setting of refinement types, at the point at which inference would choose a fresh type, the underlying shape of the type is already known. We write $\mathsf{Fresh}(X)$ to assert that X must be a fresh refinement variable (i.e. not already used in the current scope). We extend the notion to fresh types T of underlying shape $\underline{T}$.

Definition 21 (Fresh Types). We write $\mathsf{Fresh}_{\underline{T}}(T)$ for the following inductive predicate.

- For all $\alpha \in \mathbb{A}$, $\mathsf{Fresh}_\alpha(\alpha)$.
- For all $b \in \mathbb{B}$, $\mathsf{Fresh}_b(b)$.
- For $\underline{d} \in \underline{D}$, if $\mathsf{Fresh}_{\underline{T}}(T)$ for every T in $\vec{T}$, and $\mathsf{Fresh}(X)$ then $\mathsf{Fresh}_{\underline{d}}(\mathsf{inj}_X(\underline{d})\ \vec{T})$
- For all $T_1, T_2 \in \mathsf{Ty}\, D^*$, $\underline{T_1}, \underline{T_2} \in \mathsf{Ty}\, \underline{D}$, if $\mathsf{Fresh}_{\underline{T_1}}(T_1)$ and $\mathsf{Fresh}_{\underline{T_2}}(T_2)$ then $\mathsf{Fresh}_{\underline{T_1} \to \underline{T_2}}(T_1 \to T_2)$.

The definition guarantees that $\mathcal{U}(T) = \underline{T}$. We extend the notion to sequences of types, writing $\mathsf{Fresh}_{\vec{\underline{T}}}(\vec{T})$ to denote that the two sequences $\vec{T}$ and $\vec{\underline{T}}$ have the same length and are related pointwise by freshness.

Definition 22 (Inference). Inference is split into three parts: for subtyping (Figure 6), for expressions (Figure 7) and for modules (Figure 8) using three judgement forms, respectively:

$$\vdash T_1 \sqsubseteq T_2 \Longrightarrow C \qquad \Gamma \vdash e : \underline{T} \Longrightarrow T,\, C \qquad \Gamma \vdash m \Longrightarrow \Gamma',\, C$$

$$\text{(ICst)}\ \frac{}{\Gamma \vdash c : \underline{T} \Longrightarrow \underline{T},\ \emptyset}$$

$$\text{(ICon)}\ \frac{}{\Gamma \vdash k\ \vec{\underline{T}} : \underline{V} \Longrightarrow \mathsf{inj}_X\ \underline{V}\ \vec{T},\ \{k \in X(d)\}} \;\Big|\; \begin{array}{l} k \in \mathsf{dom}\,(\underline{\Delta}(\underline{d})) \\ \mathsf{Fresh}(X) \text{ and } \mathsf{Fresh}_{\vec{\underline{T}}}(\vec{T}) \end{array}$$

$$\text{(IVar)}\ \frac{}{\Gamma \vdash x\ \vec{\underline{T}} : \underline{V} \Longrightarrow U[\overrightarrow{Y/X}][\overrightarrow{T/\alpha}],\ C[\overrightarrow{Y/X}]} \;\Big|\; \begin{array}{l} x : \forall \vec{\alpha}.\, \forall \vec{X}.\, C \supset U \in \Gamma \\ \mathsf{Fresh}(\vec{Y}) \text{ and } \mathsf{Fresh}_{\vec{\underline{T}}}(\vec{T}) \end{array}$$

$$\text{(IAbs)}\ \frac{\Gamma \cup \{x : T_1\} \vdash e : \underline{T_2} \Longrightarrow T_2,\ C}{\Gamma \vdash \lambda x{:}\underline{T_1}.\, e : \underline{T_1} \to \underline{T_2} \Longrightarrow T_1 \to T_2,\ C} \;\Big|\; \mathsf{Fresh}_{\underline{T_1}}(T_1)$$

$$\text{(IApp)}\ \frac{\Gamma \vdash e_1 : \underline{T_1} \to \underline{T_2} \Longrightarrow T_1 \to T_2,\ C_1 \quad \Gamma \vdash e_2 : \underline{T_1} \Longrightarrow T_3,\ C_2 \quad \vdash T_3 \sqsubseteq T_1 \Longrightarrow C_3}{\Gamma \vdash e_1\ e_2 : \underline{T_2} \Longrightarrow T_2,\ C_1 \cup C_2 \cup C_3}$$

$$\text{(ICase)}\ \frac{\begin{array}{c}(\forall i \le m)\ \vdash T_i \sqsubseteq T \Longrightarrow C_i' \\ \Gamma \vdash e : \underline{d}\ \vec{\underline{T}} \Longrightarrow \mathsf{inj}_X\ \underline{d}\ \vec{T},\ C_0 \\ (\forall i \le m)\ \Gamma \cup \vec{x_i} : (\mathsf{inj}_X\ \vec{A})[\overrightarrow{T/\alpha}] \vdash e_i \Longrightarrow T_i,\ C_i\end{array}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{|_{i=1}^m\ k_i\ \vec{x_i} \mapsto e_i\} : \underline{T} \Longrightarrow T,\ C} \;\Big|\; \begin{array}{l} \mathsf{Fresh}_{\underline{T}\cdots\underline{T}}(T \cdot \vec{T_i}) \\ C = C_0 \cup\ \{X(\underline{d}) \subseteq \{k_1, \ldots, k_m\}\} \\ \qquad \cup \bigcup_{i=1}^m (k_i \in X(\underline{d})\ ?\ (C_i \cup C_i')) \\ \underline{\Delta}(\underline{d})(k_i) = \\ \quad \forall \vec{\alpha}.\, A_1 \to \cdots A_n \to d\ \vec{\alpha} \end{array}$$

Fig. 7. Inference for expressions.

Given two (extended) types T_1 and T_2 we infer a set of constraints C under which the former will be a subtype of the latter using the system of judgements $\vdash T_1 \sqsubseteq T_2 \Longrightarrow C$. For expressions in context $\Gamma \vdash e : \underline{T}$, we infer (extended) monotypes T and the constraints C under which they are permissible using a system of judgements of the form $\Gamma \vdash e : \underline{T} \Longrightarrow T,\ C$. In such judgements, Γ a constrained type environment, i.e. a finite map from term variables to constrained type schemes. We will omit the underlying type when not important. The rules are given in Figure 7. Constrained refinement type schemes are inferred for module-level definitions using a system of judgements of shape $\Gamma \vdash m \Longrightarrow \Gamma',\ C$. The definitions are given in Figure 8. The systems can be read algorithmically by regarding the quantities before the $\Longrightarrow$ as inputs the quantities afterwards as outputs (however, it should be noted that, assuming regular datatypes only, the subtyping relation must be computed by achieving a fixed point explicitly).

Constrained type generation via these systems of rules follows a well established pattern for expressions and modules (see e.g. [Odersky et al. 1999] for a general treatment of the non-refinement case), so we concentrate on the inference rules for subtyping. Like the more standard inference rules for expressions and modules, the inference rules for subtyping generate a derivation tree and a system of constraints whose solution guarantees the correctness of the corresponding instance of the derivation tree. However, in the case of subtyping, the derivation tree is not a proof in the

$$\text{(IModE)}\ \frac{}{\Gamma \vdash \epsilon \Longrightarrow \Gamma}$$

$$\text{(IModD)}\ \frac{\Gamma \vdash m \Longrightarrow \Gamma' \quad \Gamma' \cup \{x : T\} \vdash e \Longrightarrow T',\ C_1 \quad \vdash T' \sqsubseteq T \Longrightarrow C_2}{\Gamma \vdash m \cdot \langle x : \forall \vec{\alpha}.\ \underline{T} = \Lambda \vec{\alpha}.\ e \rangle \Longrightarrow \Gamma' \cup \{x : \forall \vec{\alpha}.\ \forall \vec{X}.\ C_1 \cup C_2 \supset T\}} \;\Big|\; \begin{array}{l} \mathsf{Fresh}_{\underline{T}}(T) \\ \vec{X} = \mathsf{FRV}(T) \end{array}$$

Fig. 8. Inference for modules.

system of Figure 3, which is for the complement of the subtyping relation, but rather a proof that the solution constitutes a simulation in the sense of Lemma 1. For example, the conclusion of (ISData) yields the constraints $\{X(d) \subseteq Y(d)\} \cup \bigcup_{k \in \mathbb{K}} \bigcup_{i=1}^{n} (k \in X(\underline{d})) \ ?\ C_{k_i}$. The first part of this constraint encodes the first bullet of Lemma 1: the environment Δ^* at $\mathsf{inj}_Y\ d$ must include all constructors included by the same environment at $\mathsf{inj}_X\ d$. Since, for any refinement X, $\mathsf{dom}(\Delta^*(\mathsf{inj}_X\ d)) = X(d)$ (recall the notational abuse adopted at the start of Section 6), we arrive at $X(d) \subseteq Y(d)$. The second part of this constraint encodes the second bullet of the lemma: *if* $k \in \mathsf{dom}(\Delta^*(\mathsf{inj}_X\ d))$ (and, therefore, $k \in \mathsf{dom}(\Delta^*(\mathsf{inj}_Y\ d))$), then the corresponding argument types are again related — in inference we recursively infer constraints on the relationship between the types and guard the constraints by $k \in X(d)$.

Theorem 23 (Soundness and completeness of $\sqsubseteq$-inference). Let T_1 and T_2 be extended types and suppose $\vdash T_1 \sqsubseteq T_2 \Longrightarrow C$. Then, for all assignments θ: $\vdash \theta T_1 \sqsubseteq \theta T_2$ *iff* $\theta \models C$.

The following states the correctness of type inference for expressions in a closed environment (e.g. for module-level definitions). The appendix contains a proof for the general case.

Theorem 24 (Soundness and completeness of expression inference). Let Γ be a constrained type environment, e an expression, V an extended refinement, C a set of constraints and let $\Gamma \vdash e \Longrightarrow V,\ C$. Then, for all refinement types T: $(\!|\Gamma|\!) \vdash e : T$ *iff* $\exists \theta.\ \theta \models C \ \wedge\ \vdash \theta V \sqsubseteq T$.

Finally, we can state the overall correctness of inference for modules.

Theorem 25 (Soundness and completeness of module inference). Suppose Γ and Γ' are closed constrained type environments, m a module and $\Gamma \vdash m \Longrightarrow \Gamma'$. Then, for all type environments Δ:

$$(\!|\Gamma|\!) \vdash m : \Delta \quad \textit{iff} \quad (\!|\Gamma'|\!) \sqsubseteq \Delta$$

8 SATURATION

The solvability of constraints can be determined by a process of saturation under all possible consequences. This is a generalisation of the transitive closure of simple inclusion constraint graphs, and a particular instance of Horn clause resolution more generally. For our constraint language, saturated constraint sets have a remarkable property: they can be restricted to any subset of their variables whilst preserving solutions.

Definition 26 (Atomic constraints). A constraint is said to be *atomic* just if its body is one of the following four shapes:

$$X(\underline{d}) \subseteq Y(\underline{d}) \qquad X(\underline{d}) \subseteq \{k_1, \ldots, k_m\} \qquad k \in X(\underline{d}) \qquad k \in \emptyset$$

An atomic constraint is said to be *trivially unsatisfiable* if it is of shape $\emptyset\ ?\ k \in \emptyset$. A constraint set is said to be *trivially unsatisfiable* just if it contains a trivially unsatisfiable constraint.

$$\text{(Transitivity)}\ \frac{\phi\ ?\ S_1 \subseteq S_2 \quad \psi\ ?\ S_2 \subseteq S_3}{\phi \cup \psi\ ?\ S_1 \subseteq S_3}$$

$$\text{(Satisfaction)}\ \frac{\phi\ ?\ k \in X(d) \quad \psi,\ k \in X(d)\ ?\ S_1 \subseteq S_2}{\phi \cup \psi ?\ S_1 \subseteq S_2}$$

$$\text{(Weakening)}\ \frac{\phi\ ?\ X(d) \subseteq Y(d) \quad \psi,\ k \in Y(d)\ ?\ S_1 \subseteq S_2}{\phi \cup \psi,\ k \in X(d)\ ?\ S_1 \subseteq S_2}$$

Fig. 9. Saturation rules.

By applying standard identities of basic set theory, every constraint is equivalent to a set of atomic constraints. In particular, a constraint of the form $k \in \{k_1, \ldots, k_m\}$ is equivalent to the empty set of atomic constraints (i.e. can be eliminated) whenever k is one of the k_i.

Definition 27 (Saturated constraint sets). An atomic constraint set, i.e. one that only contains atomic constraints, is said to be *saturated* just if it is closed under the saturation rules in Figure 9. We write $\mathsf{Sat}(C)$ for the saturated atomic constraint set obtained by iteratively applying the saturation rules to C. Note, c.f. Remark 1, all three rules correspond to special cases of resolution.

The (Transitivity) rule closes subset inequalities under transitivity, but must keep track of the associated guards by taking the union. The (Satisfaction) rule allows for a guard atom $k \in X(d)$ to be dropped whenever the same atom constitutes the body of another constraint in the set (but the other guards from both must be preserved). Finally, the (Weakening) rule allows for replacing $Y(d)$ in a guard by $X(d)$ when it is known to be no larger, thus weakening the constraint. Saturation under these rules preserves and reflects solutions:

Theorem 28 (Saturation equivalence). For any assignment θ, $\theta \models C$ iff $\theta \models \mathsf{Sat}(C)$.

If there are no trivially unsatisfiable constraints in $\mathsf{Sat}(C)$, then a solution can be constructed as follows. For each variable X occurring in C, define a function θX as follows:

$$(\theta X)(\underline{d}) \coloneqq \{k \mid k \in X(d) \text{ is in } \mathsf{Sat}(C)\}$$

Then θ solves $\mathsf{Sat}(C)$. Conversely, if there is a trivially unsatisfiable constraint in $\mathsf{Sat}(C)$, then $\mathsf{Sat}(C)$ is unsolvable and, by the equivalence theorem, it follows that C has no solution either.

Theorem 29. C is unsatisfiable iff $\mathsf{Sat}(C)$ is trivially unsatisfiable.

9 RESTRICTION AND COMPLEXITY

In practice, having established that a constraint set is solvable, we are only interested in the solutions for a certain subset of the refinement variables. For example if, as we have seen, the constraints C describe a set of types $\{T\theta \mid \theta \text{ solves } C\}$ of the module level functions, then we may consider two solutions θ_1 and θ_2 to be the same whenever they agree on $\mathsf{FRV}(T)$. We call the free refinement variables of T, in this case, the *interface variables*.

Definition 30. Let C be a saturated constraint set and let I be some set of refinement variables, called the *interface variables*. Then define the *restriction of C to I*, written $C{\restriction_I}$, as the set $\{\phi\ ?\ S_1 \subseteq S_2 \in C \mid \mathsf{FRV}(\phi) \cup \mathsf{FRV}(S_1) \cup \mathsf{FRV}(S_2) \subseteq I\}$.

The restriction of C to I is quite severe, since it simply *discards* any constraint not solely comprised of interface variables. However, a remarkably strong property of the rules in Figure 9 is that, whenever C is solvable, *every* solution of $\mathsf{Sat}(C)\restriction_I$ may be extended[6] to a solution of $\mathsf{Sat}(C)$ (and therefore of C) — independent of the choice of I! Since every solution of $\mathsf{Sat}(C)$ trivially restricts to a solution of $\mathsf{Sat}(C)\restriction_I$ (the latter has fewer constraints over fewer variables), it follows that the solutions of $\mathsf{Sat}(C)\restriction_I$ are *exactly* the restrictions of the solutions of C.

Example 31. Consider the following constraint set C by way of an illustration:

$$\begin{array}{ll} \star\ \mathsf{Cst} \in X_1(\mathsf{Lam}) & X_1(\mathsf{Lam}) \subseteq X_2(\mathsf{Lam}) \\ \quad \mathsf{FVr} \in X_2(\mathsf{Lam})\ ?\ X_2(\mathsf{Lam}) \subseteq X_3(\mathsf{Lam}) & \star\ X_3(\mathsf{Lam}) \subseteq \{\mathsf{FVr}, \mathsf{Cst}\} \end{array}$$

This set is not saturated and, consequently, there is no guarantee that the restriction of this set to an interface results in a constraint system whose solutions can generally be extended to solutions of the original set C. For example, if we restrict this set to the interface $I = \{X_1, X_3\}$, the effect will be to retain the two starred constraints. But then

$$\theta(X)(\underline{d}) \coloneqq \begin{cases} \{\mathsf{Cst}, \mathsf{FVr}, \mathsf{App}\} & \text{if } X = X_1 \text{ and } \underline{d} = \mathsf{Lam} \\ \emptyset & \text{otherwise} \end{cases}$$

is a solution of $C\restriction_I$ that does not extend to any solution of C. However, after saturation, $\mathsf{Sat}(C)$ consists of the following:

$$\begin{array}{ll} \star\ \mathsf{Cst} \in X_1(\mathsf{Lam}) & X_1(\mathsf{Lam}) \subseteq X_2(\mathsf{Lam}) \\ \quad \mathsf{Cst} \in X_2(\mathsf{Lam}) & \mathsf{FVr} \in X_2(\mathsf{Lam})\ ?\ X_2(\mathsf{Lam}) \subseteq X_3(\mathsf{Lam}) \\ \quad \mathsf{FVr} \in X_2(\mathsf{Lam})\ ?\ X_1(\mathsf{Lam}) \subseteq X_3(\mathsf{Lam}) & \mathsf{FVr} \in X_2(\mathsf{Lam})\ ?\ \mathsf{Cst} \in X_3(\mathsf{Lam}) \\ \quad \mathsf{FVr} \in X_1(\mathsf{Lam})\ ?\ X_2(\mathsf{Lam}) \subseteq X_3(\mathsf{Lam}) & \star\ \mathsf{FVr} \in X_1(\mathsf{Lam})\ ?\ X_1(\mathsf{Lam}) \subseteq X_3(\mathsf{Lam}) \\ \star\ \mathsf{FVr} \in X_1(\mathsf{Lam})\ ?\ \mathsf{Cst} \in X_3(\mathsf{Lam}) & \star\ X_3(\mathsf{Lam}) \subseteq \{\mathsf{FVr}, \mathsf{Cst}\} \\ \quad \mathsf{FVr} \in X_2(\mathsf{Lam})\ ?\ X_2(\mathsf{Lam}) \subseteq \{\mathsf{FVr}, \mathsf{Cst}\} & \quad \mathsf{FVr} \in X_2(\mathsf{Lam})\ ?\ X_1(\mathsf{Lam}) \subseteq \{\mathsf{FVr}, \mathsf{Cst}\} \\ \quad \mathsf{FVr} \in X_1(\mathsf{Lam})\ ?\ X_2(\mathsf{Lam}) \subseteq \{\mathsf{FVr}, \mathsf{Cst}\} & \star\ \mathsf{FVr} \in X_1(\mathsf{Lam})\ ?\ X_1(\mathsf{Lam}) \subseteq \{\mathsf{FVr}, \mathsf{Cst}\} \end{array}$$

In particular, the constraint $\mathsf{FVr} \in X_1(\mathit{Lam})\ ?\ X_1(\mathit{Lam}) \subseteq X_3(\mathit{Lam})$ which will be retained in the restriction $\mathsf{Sat}(C)\restriction_I$, which consists of the starred constraints. Consequently, the above assignment θ is ruled out. Indeed, one can easily verify that every solution of $\mathsf{Sat}(C)\restriction_I$ extends to a solution of $\mathsf{Sat}(C)$ and hence of C.

Theorem 32 (Restriction/Extension). Suppose C is a saturated constraint set and I is a subset of variables. Let θ be a solution for $C\restriction_I$. Then there is a solution θ' for C satisfying, for all $X \in I$:

$$\theta'(X)(\underline{d}) = \theta(X)(\underline{d})$$

Although our inference procedure is compositional, i.e. it breaks modules down into top-level definitions, and terms down into sub-terms that can be analysed in isolation, this is no guarantee of its efficiency. As we have described it in Section 7, the number of constraints associated with a function definition depends on the size of the definition — constraints are generated at most syntax nodes and propagated to the root. In fact, as is well known for constrained type inference, the situation is worse than simply this, because a whole set of constraints is imported from the environment when inferring for a program variable x. The number of constraints associated with x will again depend on the size of the definition of x and the number of constraints associated with any functions that x depends on, and so the number of constraints can become exponential in the number of function definitions[7].

[6] In the sense of Theorem 32.

[7] Although it is known that this can be avoided by a clever representation in the case of constraints that are only simple variable/variable inclusions [Gustavsson and Svenningsson 2001].

Let us fix N to be the number of module-level function definitions, K the maximum number of constructors associated with any datatype and D the maximum number of datatypes associated with any slice (for Lam, this is 2). A simple analysis of the shape of constraints yields the bound:

Lemma 2. There are $O(Kv^2D \cdot 2^{vKD+K})$ atomic constraints over v refinement variables.

Suppose $\Gamma \vdash e \Longrightarrow T,\ C$. As it stands, the number of variables v occurring in C will depend upon the size of e and the size of every definition that e depends on. We use restriction to break this dependency between the size of constraint sets and the size of the program. We compute $C\restriction_I$ for each constraint set C generated by an inference (i.e. at every step) with the interface I taken to be the free variables of the context $\mathsf{FRV}(\Gamma) \cup \mathsf{FRV}(T)$. Consequently, the number of variables v occurring in $C\restriction_I$ only depends on the number of refinement variables that are free in Γ and T. Since all the refinement variables of module level function types are generalised by (IModD) (assuming, as is usual, that inference for modules occurs in a closed environment), it follows that the only free refinement variables in Γ are those introduced during the inference of e, as a result of inferring under abstractions and case expressions. Thus v becomes independent of the number of function definitions.

Moreover, if we assume that function definitions are in β-normal form and that the maximum case expression nesting depth within any given definition is fixed (i.e. does not grow with the size of the program) then it follows that the number of refinement variables free in the environment is bounded by the size of the underlying type of e. Clearly, the number of free refinement variables in T is also bounded by its underlying type.

Consequently, for a constraint set C arising by an inference $\Gamma \vdash e \Longrightarrow T,\ C$ and then restricted to its context, the number of constraints given by Lemma 2 only depends on the size of the underlying types assigned to the e and, in the case of datatypes, the size of their definitions (slices). If we consider scaling our analysis to larger and larger programs to mean programs consisting of more and more functions, with bounded growth in the size of types and the size of individual function definitions then we may reasonably consider all these parameters fixed. Recall that N is the number of function definitions in the module. We have:

Theorem 33. Under the assumption that the size of types and the size of individual function definitions is bounded, the complexity of type inference is $O(N)$.

10 IMPLEMENTATION

We implemented a prototype of our inference algorithm for Haskell as a GHC plugin. The user can run our type checker as another stage of compilation with an additional command line flag. In addition to running the type checker on individual modules, an interface binary file is generated, enabling other modules to use the constraint information in separate compilations. It is available from: https://github.com/bristolpl/intensional-datatys.

Our plugin processes GHC's core language [Sulzmann et al. 2007], which is significantly more powerful than the small language presented here. Specifically, it must account for higher-rank types (including existentials), casts and coercions, type classes. We have not implemented a treatment of these features in our prototype and so any occurrences are not analysed. Furthermore, we disallow empty refinements of single-constructor datatypes (e.g. records). This relatively small departure from the theory is a substantial improvement to the efficiency of the tool due to the number of records and newtypes that are found in typical Haskell programs.

Since we do not analyse the dependencies of packages, datatypes that are defined outside the current package are treated as base types and not refined The resulting analysis provides a certificate of safety for some package modulo the safe use of its dependencies.

In addition to missing cases, the tool uses the results of internal analyses in GHC to identify pattern matching cases that will throw an exception. For example, the following code will be considered as potentially unsafe.

```
nnf2dnf (Lit a) = [[a]]
nnf2dnf (Or p q) = List.union (nnf2dnf p) (nnf2dnf q)
nnf2dnf (And p q) = distrib (nnf2dnf p) (nnf2dnf q)
nnf2dnf _ = error "Impossible case!"
```

10.1 Performance

We recorded benchmarks on a 2.20GHz Intel®Core™i5-5200U with 4 cores and 8.00GB RAM. We used the following selection of projects from the Hackage database:

- `aeson` is a performant JSON serialisation library.
- The `containers` package provides a selection of classic functional data structures such as sets and finite maps. The Data.Sequence module from this package contains machine generated code that lacks the typical modularity and structure of hand written code. For example, it contains an automatically generated set of 6 mutually recursive functions[8], each with a complex type and deeply nested matching. The corresponding interface is in excess of 80 refinement variables. This module could not be processed to completion in a small amount of time and so we have omitted it from the results. We will explore how best to process examples that violate our complexity assumptions in follow-up work.
- `extra` is a collection of common combinators for datatypes and control flow.
- `fgl` (Functional Graph Library) provides an inductive representation of graphs.
- `haskeline` is a command-line interface library
- `parallel` is Haskell's default library for parallel programming
- `sbv` is an SMT based automatic verification tool for Haskell programs.
- The `time` library contains several representations of time, clocks and calendars.
- `unordered-containers` provides hashing-based containers, for either performant code or datatypes without a natural ordering.

For each module we recorded the average time elapsed in milliseconds across 10 runs and the number of top-level definitions (N). We note both the total number of refinement variables generated during inference (V) and the largest interface (I). The contrast between these two figures gives some indication of how intractable the analysis may become be without the restriction operator. Naturally, constant factors will vary considerably between modules (not in correspondence with their size) and so our results also include the number of constructors (K) that appear in the largest datatype, and the number of datatypes (D) in the largest slice.

The benchmarks in Figure 1 provide a summary of the results for each project, i.e. the total time taken[9] , the total number of top-level definitions, the total number of refinement variables, the maximum interface size, the largest number of constructors associated with a datatype, and the largest slice. The full dataset can be found in the appendices and a virtual machine image for recreating the benchmarks can be downloaded from: https://doi.org/10.5281/zenodo.4072906.

Figure 1 also contains the number of warnings found in each packages. However, many of them stem from the same incomplete pattern. For example, 70 of the warnings from the sbv package are located in one function. All of these warnings were due to the tools limited, and thus extremely conservative, approach to handle features of GHC outside of its scope, such as typeclasses and encapulsation via the module system, so we are optimistic about future work.

[8]Since they are mutually recursive, they are processed together before generalisation and thus act as a single complex type.
[9]The total time taken is the sum of the time taken to analyse each module independently doesn't include start up costs etc.

Table 1. Benchmark Summaries

Name	N	K	V	D	I	Warnings	Time (ms)
aeson	728	13	20466	6	14	0	79.37
containers	1792	5	25237	2	23	18	118.26
extra	332	3	5438	3	7	0	61.53
fgl	700	2	18403	2	12	8	94.32
haskeline	1384	15	29389	19	27	0	111.67
parallel	110	1	959	2	18	0	10.18
pretty	222	8	3675	4	16	11	23.86
sbv	5076	44	171869	49	46	79	518.91
time	484	7	9753	6	10	9	134.16
unordered-containers	474	5	7761	3	24	2	30.56

These packages were selected to test the tool in a range of contexts and at scale. We did not find any true positives, but it is not surprising since large packages with many downloads on Hackage are likely to be quite mature.

11 RELATED WORK

The goal of our system is to automatically, statically verify that a given program is free of pattern match exceptions, and we have phrased it as a type inference procedure for a certain refinement type system with recursive datatype constraints. We have shown that it works well in practice, although a more extensive investigation is needed. Our primary motivation has been to ensure predictability by giving concrete guarantees on its expressive power and algorithmic complexity.

Recursive types, subtyping and set constraints. Our work sits within a large body of literature on recursive types and subtyping. As a type system, ours is not directly comparable to others in the literature: on the one hand, the intensional refinement restriction is quite severe, but on the other we allow for path sensitivity. One of the first works to consider subtyping in the setting of recursive types was that of Amadio and Cardelli [1993]. They proposed an exponential time procedure for subtype checking, but this was later improved to quadratic by Kozen, Palsberg, and Schwartzbach [1995]. Neither of these works gave a treatment of the combination with polymorphism, which is the subject of e.g. Castagna and Xu [2011]; Dolan and Mycroft [2017]; Hoang and Mitchell [1995]; Pottier [1998]. However, to the best of our knowledge, all the associated type inference algorithms are exponential time in the size of the program. In particular, Hoang and Mitchell [1995] shows that a general formulation of typing with recursive subtyping constraints has a PSPACE-hard typability problem. However, we mention as a counterpoint that when constraints are restricted to simple variable-variable inequalities, Gustavsson and Svenningsson [2001] show that there is a cubic-time algorithm. Being based on unification, inference for polymorphic variants is efficient [Garrigue 2002], but Castagna, Petrucciani, and Nguyen [2016] point out instances where programmers find the results to be unpredictable. None of the above allow for path-sensitive treatment of matching.

Our main inspiration has been the seminal body of literature of work on set constraints in program analysis, see particularly Aiken, Wimmers, and Lakshman [1994b], Aiken [1999] and Heintze [1992], and in particular, the line of work on making the cubic-time fragments scale in practice [Fähndrich and Aiken 1996; Fähndrich et al. 1998; Heintze 1994; Su et al. 2000]. Through an impressive array of sophisticated optimisations, the fragment can be made to run efficiently on many programs. However, the fundamental worst-case complexity is not changed and implementing and

tuning heuristics requires a large engineering effort. Moreover, this fragment does not accommodate path sensitivity.

An interesting new approach to full set constraints language is that of Eremondi [2019], who attempts to use SMT to circumvent the extremely high worst-case complexity in some practical cases. However, experiments are limited to programs less than a few hundred lines.

Many of the analyses and or type inference procedures discussed so far are compositional, i.e. parts of the program are analysed independently to yield summaries of their behaviour and then the summaries are later combined. However, it has been frequently observed that compositionality does not lead to scalability if the summaries are themselves large and complicated. In particular, it is not uncommon for "summaries" that grow with the square of the size of the program in the worst case. This has led to many works that attempt to simplify summaries, typically according to ingenious heuristics [Aiken et al. 1999; Dolan and Mycroft 2017; Fähndrich and Aiken 1996; Flanagan and Felleisen 1999; Pottier 2000, 2001; Rehof 1997; Trifonov and Smith 1996]. Since our primary motivation was predictability, we have designed our system so that heuristics are avoided[10]: in particular the size of summaries (i.e. constrained type schemes) only depends only on the size of the underlying types and not the size of the program. It is plausible that many of these heuristic optimisations are nevertheless applicable in order to help improve the overall efficiency. Note also that, if we are not concerned with a compositional analysis, then our class of constraints can be checked for solvability using a linear time algorithm due to Rehof and Mogensen [1999]. However, as explained in the introduction, compositionality is essential to obtaining *overall* linear time complexity.

Refinement types. Refinement types originate with the works of Freeman and Pfenning [1991] and Xi and Pfenning [1999]. Their distinguishing feature is that they attempt to assign types to program expressions for which an *underlying type* is already available. Typically, as here, the refinement type is also required to respect the shape of the underlying type. One can use this restriction, as in *loc cit* to ensure some independence of the the size of the type from the size of the program. However, as remarked in the final section, the constant factors are enormous since there is unrestricted intersection and union of refinements of the same underlying type which is represented explicitly.

The work of Freeman and Pfenning [1991] requires that the programmer declare the universe of refinement types up-front (where our universe is determined automatically as a completion of the underlying datatype environment). A disadvantage of this requirement is that it burdens the programmer with a kind of annotation that they would rather not have to clutter their program with, in many simple cases. A great advantage is that, by defining a refinement datatype explicitly, the programmer can indicate formally in the code her intention that a certain invariant is (somehow) important within a certain part of the program. It seems like a very fruitful idea to allow the programmer this freedom also in our system and we are actively working on an extension to allow for this as part of our future work. In particular, we would like to take advantage of several new advances in this line that relieve a lot of programmer burden, such as those of Dunfield [2007, 2017].

An incredibly fruitful recent evolution of refinement types are the Liquid Types of Rondon, Kawaguci, and Jhala [2008] (see especially Vazou, Bakst, and Jhala [2015] for a version with constrained type schemes) and similar systems (e.g. those of Terauchi [2010]; Unno and Kobayashi [2009]). Such technology is already accessible to the benefit of the average programmer through the Liquid Haskell system of Vazou, Seidel, Jhala, Vytiniotis, and Peyton-Jones [2014]. Due to the rich expressive power of these systems, which typically include dependent product, efficient and

[10]Heuristic-based optimisations can be the enemy of predictability since small changes in the program can lead to great changes in performance if the change causes the program to fall outside of the domain on which the heuristic is tuned.

fully-automatic type inference is not typically a primary concern and predictability can be ensured by liberal use of annotations.

Pattern match safety and model checking. The pattern match safety problem was also addressed by Mitchell and Runciman [2008], which was used to verify a number of small Haskell programs and libraries. The expressive power and algorithmic complexity are, however, unclear.

Safety problems are within the scope of higher-order model checking (Kobayashi [2013]; Kobayashi and Ong [2009]; Ong [2006]) and a system for verifying pattern match safety, built on higher-order model checking was presented in [Ong and Ramsay 2011]. Higher-order model checking approaches reduce verification problems to model checking problems on a certain infinite tree generated by a higher-order grammar. Although the higher-order model checking problem is linear-time in the size of the grammar, the constant factors are enormous because, formally, it is n-EXPTIME complete (tower of exponentials of height n) with n the type-theoretic order of the functions in the grammar. Moreover, many of the transformations from program to grammar incur a large blow-up in size. Two promising evolutions of higher-order model checking are the approach of Kobayashi, Tsukada, and Watanabe [2018] based on the higher-order fixpoint logic of Viswanathan and Viswanathan [2004] and that of Cathcart Burn, Ong, and Ramsay [2017] via higher-order constrained Horn clauses.

Contract checking. Like pattern-match safety, static contract checking problems such as those considered by Xu, Jones, and Claessen [2009], Vytiniotis, Jones, Claessen, and Rosén [2013] and Nguyen, Tobin-Hochstadt, and Van Horn [2014] typically also reduce to reachability. However, giving guarantees on scalability via worst-case complexity does not seem to be a priority for this area and experiments are correspondingly limited to programs of only a few hundred lines.

Pattern match coverage checking. A related problem is the pattern match *coverage checking* problem, which asks, with respect to the type of the function: if a given set of patterns is exhaustive, non-overlapping and irredundant (a classic paper on this subject is that of Maranget [2007], but see Graf, Jones, and Scott [2020] for more recent developments). To illustrate the difference between the two problems: a program containing the following definition is always a no-instance of the coverage checking problem, since $f :: \mathsf{Bool} \to \mathsf{Bool}$ has non-exhaustive patterns:

$$f\ \mathsf{True} = \mathsf{False}$$

However, such a program may or may not be a no-instance of the pattern-match safety problem, since it depends on how f is actually used in the rest of the program. Like all safety problems, the latter can be reduced to reachability: is there an execution of the program that reaches a call of the form $f\ \mathsf{False}$. If no execution of the program containing this definition ever calls f with False, then this program will be a yes-instance of the safety problem, i.e. pattern-match safe.

On the one-hand, if every pattern-matching expression covers all cases, then the program is already safe, since no execution can trigger a pattern-match violation. On the other, a program may be safe and yet not cover every case in its patterns — indeed these are really the focus from a program verification perspective. The proliferation of exotic kinds of pattern allowed in a complex language such as Haskell (e.g. pattern synonyms [Pickering et al. 2016]), mean that coverage checking may sometimes benefit from reasoning about program executions in a localised way. However, as the authors of [Graf et al. 2020] point out, it is "unreasonable to expect a coverage checking algorithm to prove [a property of arbitrary program executions]".

12 CONCLUSION

We have presented a new extension of ML-style typing with intensional refinements of algebraic datatypes. Since type inference is fully automatic, the system can be used as a program analysis for verifying the pattern-match safety problem. Viewed this way, it incorporates polyvariance and path-sensitivity and yet we have shown that, under reasonable assumptions, the worst-case time complexity is linear in the size of the program. To achieve this we have shifted exponential complexity associated with HM(X)-style inference from the size of the program to the *size of the types of the program*. Moreover, we have shown that our assumptions on the size of types are reasonable in practice (equivalently: that the constant factors are not prohibitive) by demonstrating excellent performance of a prototype.

This was only possible because we have made a compromise on the space of invariants that we can synthesize: typings are built from *intensional* refinements. Although our analysis is polyvariant and path-sensitive, these features are ultimately limited by the shape of these refinements. We have given examples, (e.g. in Figure 2 and Example 7) of datatypes for which there are many useful intensional refinements that are expressible in our system. However, there are also common datatypes for which there are no useful intensional refinements. For example, the four intensional refinements of the datatype of lists correspond to: the empty type, the type of infinite lists, the type containing only the empty list and the original list datatype. However, none of these is especially useful in practice, and one would much rather have a refinement like the type of non-empty lists.

In any fully automatic program analysis, there will always be some compromise on expressivity. We believe it is important that one can understand, before using the tool on a program, whether the compromise will be a real limitation. The power of our analysis is characterised as a type system that can be understood by programmers familiar with usual ML-style typing. If the (user believes that their) program would be typable in this system (i.e. there exist intensional refinements of the datatypes under which a typing can be assigned), then the analysis will be able to verify it. Note that the user does not need to know anything about constraints, which occur only in the inference algorithm, in order to determine this. For example, if the user believes that the safety of their program relies on a invariant to do with the non-emptiness of lists then, since non-emptiness is not an expressible refinement of lists, they should not expect the program to be verifiable.

Our future work concerns such cases. We would like to enable the user to specify their own non-intensional refinements of datatypes to extend the space of expressible program invariants. For example, using some syntax, the programmer could indicate that the non-emptiness of lists is an important refinement. Under the hood, the system can extend the original definition of lists in such a way that (a) the new definition is extensionally equivalent to the original, but (b) non-empty lists is now an intensional refinement. The following is an example of such a redefinition:

```
List  a = Nil | Cons a (List' a)
List' a = Nil | Cons a (List' a)
```

From which the non-empty list refinement arises by erasing Nil from the List datatype. This would put the trade-off between expressive power and efficiency in the hands of the user, and since the type system is familiar, they are well equipped to reason about when it makes sense.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of the Engineering and Physical Sciences Research Council (EP/T006579/1) and the National Centre for Cyber Security via the UK Research Institute in Verified Trustworthy Software Systems. We thank our colleague Matthew Pickering for a lot of good Haskell advice and for helping us safely navigate the interior of the Glasgow Haskell Compiler.

REFERENCES

Alexander Aiken. 1999. Introduction to set constraint-based program analysis. *Science of Computer Programming* 35, 2 (1999), 79–111. https://doi.org/10.1016/S0167-6423(99)00007-6

Alexander Aiken and Edward L. Wimmers. 1992. Solving Systems of Set Constraints (Extended Abstract). In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992*. 329–340.

Alexander Aiken and Edward L. Wimmers. 1993. Type Inclusion Constraints and Type Inference. In *FPCA*. 31–41.

Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. 1994a. Soft Typing with Conditional Types. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*. 163–173.

Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. 1994b. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, 163–173. https://doi.org/10.1145/174675.177847

Alexander Aiken, Edward L. Wimmers, and Jens Palsberg. 1999. Optimal Representations of Polymorphic Types with Subtyping. *Higher-Order and Symbolic Computation* 12, 3 (1999), 237–282. https://doi.org/10.1023/A:1010056315933

Roberto M. Amadio and Luca Cardelli. 1993. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.* 15, 4 (1993), 575–631. https://doi.org/10.1145/155183.155231

Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. 1993. Set constraints are the monadic class. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 75–83.

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 2016. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, 378–391. https://doi.org/10.1145/2951913.2951928

Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. Association for Computing Machinery, 94–106. https://doi.org/10.1145/2034773.2034788

Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. 2017. Higher-order constrained horn clauses for verification. *Proc. ACM Program. Lang.* 2, POPL (2017), Article 11. https://doi.org/10.1145/3158099

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery, 60–72. https://doi.org/10.1145/3009837.3009882

Joshua Dunfield. 2007. Refined typechecking with Stardust. In *Proceedings of the 2007 workshop on Programming languages meets program verification*. Association for Computing Machinery, 21–32. https://doi.org/10.1145/1292597.1292602

Joshua Dunfield. 2017. Extensible Datasort Refinements. In *European Symposium on Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, 476–503. https://doi.org/10.1007/978-3-662-54434-1_18

Joseph Eremondi. 2019. Set Constraints, Pattern Match Analysis, and SMT. In *Trends in Functional Programming - 20th International Symposium, TFP 2019, Vancouver, BC, Canada, June 12-14, 2019, Revised Selected Papers (Lecture Notes in Computer Science)*, William J. Bowman and Ronald Garcia (Eds.), Vol. 12053. Springer, 121–141. https://doi.org/10.1007/978-3-030-47147-7_6

Manuel Fähndrich and Alexander Aiken. 1996. Making Set-Constraint Based Program Analyses Scale. In *First Workshop on Set Constraints at CP'96*.

Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. Association for Computing Machinery, 85–96. https://doi.org/10.1145/277650.277667

Cormac Flanagan and Matthias Felleisen. 1999. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.* 21, 2 (1999), 370–416. https://doi.org/10.1145/316686.316703

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. Association for Computing Machinery, 268–277. https://doi.org/10.1145/113445.113468

Jacques Garrigue. 2002. Simple Type Inference for Structural Polymorphism. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*.

Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. 2020. Lower your guards: a compositional pattern-match coverage checker. *Proc. ACM Program. Lang.* 4, ICFP (2020), 107:1–107:30. https://doi.org/10.1145/3408989

Jörgen Gustavsson and Josef Svenningsson. 2001. Constraint Abstractions. In *Symposium on Programs as Data Objects*, Olivier Danvy and Andrzej Filinski (Eds.). Springer Berlin Heidelberg, 63–83.

John Harrison. 2009. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press.

Nevin Heintze. 1994. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM conference on LISP and functional programming*. Association for Computing Machinery, 306–317. https://doi.org/10.1145/182409.182495

Nevin Heintze, Spiro Michaylov, and Peter Stuckey. 1992. CLP($\mathbb{R}$) and some electrical engineering problems. *Journal of Automated Reasoning* 9, 2 (1992), 231–260.

Nevin Charles Heintze. 1992. *Set based program analysis.* Thesis.

My Hoang and John C. Mitchell. 1995. Lower bounds on type inference with subtypes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, 176–185. https://doi.org/10.1145/199448.199481

Naoki Kobayashi. 2013. Model Checking Higher-Order Programs. *J. ACM* 60, 3 (2013), Article 20. https://doi.org/10.1145/2487241.2487246

N. Kobayashi and C. L. Ong. 2009. A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes. In *IEEE Symposium on Logic In Computer Science*. 179–188. https://doi.org/10.1109/LICS.2009.29

Naoki Kobayashi, Takeshi Tsukada, and Keiichi Watanabe. 2018. Higher-Order Program Verification via HFL Model Checking. In *European Symposium on Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, 711–738.

Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. 1995. Efficient recursive subtyping. *Mathematical Structures in Computer Science* 5, 1 (1995), 113–125. https://doi.org/10.1017/S0960129500000657

Luc Maranget. 2007. Warnings for pattern matching. *J. Funct. Program.* 17, 3 (2007), 387–421. https://doi.org/10.1017/S0956796807006223

Neil Mitchell and Colin Runciman. 2008. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*. Association for Computing Machinery, 49–60. https://doi.org/10.1145/1411286.1411293

Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 139–152. https://doi.org/10.1145/2628136.2628156

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *TAPOS* 5, 1 (1999), 35–55.

C.-H. Luke Ong and Steven J. Ramsay. 2011. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, 587–598. https://doi.org/10.1145/1926385.1926453

C. L. Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *IEEE Symposium on Logic in Computer Science*. 81–90. https://doi.org/10.1109/LICS.2006.38

Matthew Pickering, Gergo Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 80–91. https://doi.org/10.1145/2976002.2976013

François Pottier. 1998. *Type inference in the presence of subtyping: from theory to practice.* Thesis.

François Pottier. 2000. A Versatile Constraint-Based Type Inference System. *Nordic J. of Computing* 7, 4 (Dec. 2000), 312–347.

François Pottier. 2001. Simplifying Subtyping Constraints: A Theory. *Information and Computation* 170, 2 (2001), 153–183. https://doi.org/10.1006/inco.2001.2963

Jakob Rehof. 1997. Minimal typings in atomic subtyping. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, 278–291. https://doi.org/10.1145/263699.263738

Jakob Rehof and Torben Æ. Mogensen. 1999. Tractable constraints in finite semilattices. *Science of Computer Programming* 35, 2 (1999), 191 – 221.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 159–169. https://doi.org/10.1145/1375581.1375602

Zhendong Su, Manuel Fähndrich, and Alexander Aiken. 2000. Projection merging: reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, 81–95. https://doi.org/10.1145/325694.325706

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

Tachio Terauchi. 2010. Dependent types from counterexamples. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, 119–130. https://doi.org/10.1145/1706299.1706315

Valery Trifonov and Scott Smith. 1996. Subtyping constrained types. In *Static Analysis Symposium*, Radhia Cousot and David A. Schmidt (Eds.). Springer Berlin Heidelberg, 349–365.

Hiroshi Unno and Naoki Kobayashi. 2009. Dependent type inference with interpolants. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*. Association for Computing Machinery, 277–288. https://doi.org/10.1145/1599410.1599445

Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, 48–61. https://doi.org/10.1145/2784731.2784745

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. Association for Computing Machinery, 269–282. https://doi.org/10.1145/2628136.2628161

Mahesh Viswanathan and Ramesh Viswanathan. 2004. A Higher Order Modal Fixed Point Logic. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, 512–528.

Dimitrios Vytiniotis, Simon L. Peyton Jones, Koen Claessen, and Dan Rosén. 2013. HALO: haskell to logic through denotational semantics. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 431–442. https://doi.org/10.1145/2429069.2429121

Hongwei Xi and Frank Pfenning. 1999. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, 214–227. https://doi.org/10.1145/292540.292560

Dana N. Xu, Simon L. Peyton Jones, and Koen Claessen. 2009. Static contract checking for Haskell. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 41–52. https://doi.org/10.1145/1480881.1480889