Dependent Types and Abstract Refinements

Niki Vazou
nvazou@cs.ucsd.edu
University of California, San Diego

Abstract

Even well-types programs can go wrong by returning a wrong answer. A popular response to this problem is the use of dependent type systems to encode more expressive invariants about program values. In this paper we present two categories of dependent type systems. On one hand there exist expressive dependent type systems that require or run-time checks or explicit proves to verify their assertions On the other hand there exist less expressive type system that allow static and automatic proves of the assertions. Finally, we present abstract refinement types, a means to extend the expressiveness of a dependent type system without increasing its complexity/

1 Introduction

Functional programming languages, like ML and Haskell, come with strong static types system, which detects a lot of errors at compile-time and enhances code documentation.

The utility of these type systems stems from their ability to predict, at compile-time, invariants about the run-time values computed by the program. Unfortunately, classical type systems only capture relatively coarse invariants. For example, the system can express the fact that a variable `i` is of the type `Int`, meaning that it is always an integer, b ut not that it is always an integer within a certain range, say greater than zero. Thus, the type system is unable to statically ensure the safety of critical operations, such as a division by i. Instead, the language can only provide a weaker dynamic safety guarantee at the additional cost of high performance overhead. Several authors have proposed the use of dependent types [20] as a mechanism for enhancing the expressiveness of type systems [14, 27, 2, 22, 10].

Such a system uses the dependent type:

```
i :: {v:Int | v != 0}
```

that describes all values `v` if type `Int` and the refinements states that the value `v` should be different than `0`.

We can use this dependent type to define a safe division operator:

```
safeDiv :: Int -> {v:Int | v != 0} -> Int
```

This type expresses the fact that the division operator takes two `Int` arguments and returns an `Int`. Moreover restricts the second argument to be different that zero, so as to eliminate division by zero operations.

When the function `safeDiv` is actually used, the type system should check that the real arguments do not violate `safeDiv`'s specification.

So, it is always safe to have `safeDiv 8 9` since `9` is always different that zero. But, `safeDiv 8 0` should create a type error. Finally, if we apply as a second argument an arbitrary program expression : `safeDiv 8 n` the type system, in order to

accept this application, should infer than its type implies that `n` is an integer different than zero.

Dependent Function Types. Dependent types, allow the predicate of the result of a function to depend of function arguments. As an example, we can defined the `pred` function that returns the predecessor of its argument, and give it a very descriptive type:

```
pred :: n : Int -> {v:Int | v = n-1}
pred n = n-1
```

When we apply a concrete value to `pred`, this should substitute `n` in the result type. For example `pred 2 :: {v:Int | v = n-1}[2/n] = {v:Int | v = 1}`. With this means, the type of the result can have properties that depend on function arguments.

Decidability vs Expressiveness. If predicates contain arbitrary program expressions, then one can define an undecidable function `terminates`; that takes an function argument `f` and a value `x` and returns true if and only if the computation `f x` terminates:

```
terminates    :: f:(a -> b) -> x:a -> Bool
```

and use it inside type predicates:

```
f:(a->b) -> x: a ->  {v:Bool|terminates f x}
```

Clearly, a dependent type system, which allows arbitrary program expressions as predicates is undecidable. In this paper, we will describe both expressive and decidable systems. The rest of this paper is organized as follows: In section 2, we present a core calculus that constitutes the base for all the following systems. In section 3, we will describe how to reason with undesidable dependent type systems. In section 4, we will present a type system that restricts the predicate expressions and is statically decidable. In section 5 we will present, how abstraction over predicates can enhance expressiveness of decidable type systems.

2 Preliminaries

To be able to formally describe and compare the type systems, we will define a a core calculus λ_c, its syntax and type system:

Syntax. The syntax of expressions and types is summarized in Figure 8. λ_D expressions include variables, constants, λ-abstractions and function applications. Basic types can be integer or boolean. λ_D types can be dependent types and function types. The predicate p is not yet defined. As we noted earlier, if p contains arbitrary program expressions, the type system is undecidable. In dependent function types we use a variable to bind the argument, as it can be used in the type of the result.

Finally, we define a typing environment Γ, that maps variables to their type.

Static Typing.

There are three types of judgments:

- ***Wellformedness judgments*** ($\Gamma \vdash \tau$) state that a type τ is well-formed under environment Γ, that is, the refinements in τ are boolean expressions in the environment Γ.

Expressions	e	::=	$x \mid c \mid \lambda x : \tau . e \mid e\ e$
Predicates	p	::=	$\ldots$
Basic Types	b	::=	`int` $\mid$ `bool`
Dependent Types	τ	::=	$\{v : b \mid p\} \mid x : \tau \rightarrow \tau$
Typing Environment	Γ	::=	$\emptyset \mid x : \tau, \Gamma$

Figure 1: **Syntax of Expressions, Types and Schemas**

- ***Subtyping judgments*** ($\Gamma \vdash \tau_1 \preceq \tau_2$) state that the type τ_1 is a subtype of the type τ_2 under environment Γ, that is, when the free variables of τ_1 and τ_2 are bound to values described by Γ, the set of values described by τ_1 is contained in the set of values described by τ_2.
- ***Typing judgments*** ($\Gamma \vdash e : \tau$) state that the expression e has the type τ under environment Γ, that is, when the free variables in e are bound to values described by Γ, the expression e will evaluate to a value described by τ.

Wellformedness Rules. The wellformedness rules check that the refinements are indeed `bool`-valued expressions in the appropriate environment. The rule WF-BASE checks that the refinement e is boolean. The rule WF-FUN recursively applies the wellformdness rule to the argument type of the function, and to the result type, after it adds the argument binder with the correct type to the environment.

Subtyping Rules. The subtyping rules stipulate when the set of values described by type τ_1 is subsumed by the values described by τ_2. The rule $\prec$-BASE serves two purposes: Firstly, it checks that the basic type is the same in the two types, as in vanilla type systems Moreover, it checks that under the environment Γ, the left hand side refinement implies the right hand side. To do so, we use a `Valid` predicate which varies between the systems that we will describe. The rule $\prec$-FUN relates two function types according to the contravariant rule.

Type Checking Rules. The type checking rules check that an expression in of the correct type. The rule T-CONST checks the types of the constant expressions, according to their definitions. The rule $\{v : c \mid h\}$ecks the types of the variables, according to the environment Γ. The rule $c : h \rightarrow e$cks the types of the variables, according to the environment Γ. The rule T-FUN checks both that the argument type is well-formed and that the type of the function-body is correct in the environment, extended with the argument of the function. Finally, the rule T-APP checks that e_1 is has a function type and that its argument has the type of the argument e_2. Also, in the result type, the argument x is replaced with the actual argument e_2.

You may notice that the subtyping relation is not used in this basic rules, but as we stated, the core calculus will be extended to formalize the type systems that we will describe.

3 Undecidable Systems

As we said, if the refinement language can have arbitrary program expressions, we can not statically prove the soundness of a program. In this section we will present three

Well-Formedness $\boxed{\Gamma \vdash \tau}$

$$\frac{\Gamma, p : b \vdash e : \texttt{bool}}{\Gamma \vdash \{v : b \mid p\}} \text{ WF-BASE} \qquad \frac{\Gamma \vdash \tau_x \quad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x : \tau_x \to \tau} \text{ WF-FUN}$$

Subtyping $\boxed{\Gamma \vdash \tau \preceq \tau}$

$$\frac{(\Gamma, v : b \vdash \texttt{Valid}(p_1 \Rightarrow p_2))}{\Gamma \vdash \{v : b \mid p_1\} \preceq \{v : b \mid p_2\}} \prec\text{-BASE}$$

$$\frac{\Gamma \vdash \tau_{21} \preceq \tau_{11} \quad \Gamma, x_2 : \tau_{21} \vdash \tau_{12}[x_2/x_1] \preceq \tau_{22}}{\Gamma \vdash x_1 : \tau_{11} \to \tau_{12} \preceq x_2 : \tau_{21} \to \tau_{22}} \prec\text{-FUN}$$

Type Checking $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash c : tc\,(c)} \text{ T-CONST} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ T-VAR}$$

$$\frac{\Gamma, x : \tau_x \vdash e : \tau \quad \Gamma \vdash \tau_x}{\Gamma \vdash \lambda x : \tau_x . e : x : \tau_x \to \tau} \text{ T-FUN} \qquad \frac{\Gamma \vdash e_1 : x : \tau_x \to \tau \quad \Gamma \vdash e_2 : \tau_x}{\Gamma \vdash e_1\ e_2 : \tau[e_2/x]} \text{ T-APP}$$

Figure 2: **Static Semantics: Well-formedness, Subtyping and Type Checking**

ways to reason about such languages:

- Interactive theorem proving; where the proofs are statically created by the user
- Contracts Calculi; where the assertions are checked at run time
- Hybrid Type Checking; where the assertions are checked statically whenever possible and dynamically when necessary.

3.1 Interactive theorem Proving

One approach to verify that a program satisfies some specifications is to statically prove them. This approach is used by *interactive theorem provers*, such as Coq, Agda and Isabelle, that allow the expression of mathematical assertions, mechanically check proofs of these assertions, help to find formal proofs, and extract a certified program from the constructive proof of its formal specification.

As an example, consider a function `pred` in Coq, that computes the predecessor of a positive number, and has type signature

```
pred :: s:{n : nat | n > 0} -> {v:nat | s = S v}
```

This type signature says that if `pred` is called with a positive number `s`, it will return `s`'s predecessors. There are two different assertions that should be proved:

- The result of the function is the predecessor of the argument. At `pred`'s definitions the programmer should provide a proof that this assertion is indeed satisfied.
- The argument is a positive number. At each call side of `pred`, the user should provide a proof that its argument is positive.

In more details `pred` function can be defined in Coq as follows [1]

```
Definition pred (s : {n : nat | n > 0}) : {m : nat |
    proj1_sig s = S m} :=
  match s return {m : nat | proj1_sig s = S m} with
    | exist 0 pf => match zgtz pf with end
    | exist (S n') pf => exist _ n' (eq_refl (S n'))
  end.
```

where the refinement type is syntactic sugar, defined in the standard library, for the type family `sig`:

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> sig P
Notation
  "{ x : A | P }" := sig (fun x : A => P)
```

The function `pred` takes an argument `s` which has two components: a natural number `n` and a proof `pf` that this number is positive. Then, there is a case analysis on `s` and if `n` is zero, then the proof `pf` is used to reach a contradiction; thus this case can not occur. Otherwise, `n` has a predecessor, say `n'` and the function returns `n'` combined with a proof that its successor is equal to `n`. This proof is constructed by applying `eq_relf`, the only constructor of equality to `S n'`.

As we said, at the call side of `pred`, the programmer should provide both the argument and a proof that it is positive. As an example, we can have

```
pred (exist _ 2 two_gt0)
```

where `two_gt0` is a proof that two is greater than zero.

Even this example seems tedious, interactive theorem prooving can be simplified using inference and tacticks. Though, the user still needs to provide proofs. We will discuss other systems, which remove this burden from the user.

3.2 Contracts

Another approach to verify that a program satisfies some assertions is dynamically check them. These assertions are called *contracts*, i.e., dynamically enforced pre- and post assertions that define formal, precise and verifiable interface specifications for software components. Their use in programming languages dates back to the 1970s; when Eiffel [13], an object-oriented programming language, totally adopted assertions and developed the "Design by Contract" philosophy [14].

Contracts are of the form: $\langle\{v : \tau \mid p\}\rangle^l$ and describe the values v, of type τ that satisfy the predicate p. The l superscript is a *blame label*, used to identify the source of failures. As an example, consider a contract for positive integers $\langle\{v : Int \mid v > 0\}\rangle^l$ applied to two values, 2 and 0:

$$\begin{aligned} \langle\{v : Int \mid v > 0\}\rangle^{l'}\ 2 &\quad\quad \rightarrow 2 \\ \langle\{v : Int \mid v > 0\}\rangle^{l}\ 0 &\quad\quad \rightarrow \Uparrow l \end{aligned}$$

If the check succeeds, as in the case for 2, then the application will return the value, so the first application just returns 2. If it fails, then the entire program will "blame" the label l, raising an uncatchable exception $\Uparrow l$, pronounced "blame l".

Assigning blame for contractual violations in higher-order languages is complex: The boundaries between cooperating components are more obscure than in the world with only first-order functions. A function may invoke a function passed to it at its call side. Accordingly, the blame for a corresponding contract violation must lie with the supplier of the bad value, no matter if the bad value was passed by directly applying a function or by applying a base value.

In 2002, Findler and Felleisen in [5] were the first to to create a system for higher order languages with contracts. In their system, the blame is properly assigned in the higher-order components of the program via the "variance-contravariance" rule. Moreover, they allow dependent contracts, i.e. contracts that have the form of a dependent function type, where the result can depend on the argument. Finally, they treat contracts as first class values, i.e., contracts are values that can be passed to and from functions

In 2004, Blume and McAllester, at [3] noted that in Findler and Felleisen system, the concept of contract satisfaction had not actually been defined formally, and they proved that their system is indeed sound and complete, The contract system is sound if whenever the algorithm blames a contract declaration, that contract declaration is actually wrong. Conversely, it is complete if blame on a expression is explained by the fact that the expression violates one of it contract interfaces.

Findler and Felleisen's work sparked a great interest in contracts, and in the following years a variety of related systems have been studied. Broadly, these come in two different sorts. In systems with *latent contracts*, types and contracts are orthogonal features. Examples of this style include Findler and Felleisen's original system, Hinze et al. [10], Blume and McAllester [3], Chitil and Huch [4], Guha et al. [9], and Tobin-Hochstadt and Felleisen [20]. By contrast, *manifest contracts* are integrated into the type system, which tracks, for each value, the most recently checked contract. Hybrid types [6] are a well-known example in this style; others include the work of Ou et al. [16], Wadler and Findler [22], and Gronski et al. [8], Belo et al. [2] and Grennberg et al. [7].

In the next subsection we will discuss manifest contracts and how we can extend out core language to support them.

3.2.1 Manifest Contracts

Manifest Contracts Systems, as presented in [7], use casts, $\langle \tau_s \Rightarrow \tau_t \rangle^l$ to convert values from the source type τ_s to the target type τ_t and raise $\Uparrow\ l$ if the cast fails.

As an example, consider a cast from integers to positives:

$$\langle Int \Rightarrow \{v : Int | v > 0\} \rangle^l\ n$$

The system should statically verify that the value n is of the source type Int. After the cast, the program can treat this value as if it has the target type $\{v : Int | v > 0\}$. At run-time, a check will be made that n is actually a positive integer and if it fails it will raise $\Uparrow\ l$.

To generalize, for base contracts, a cast will behave just like a check on the target type: applied to n, the cast either returns n or raises $\Uparrow\ l$.

A function cast

$$(\langle \tau_{11} \to \tau_{12} \Rightarrow \tau_{21} \to \tau_{22} \rangle^l\ f)\ v$$

will reduce to

$$\langle \tau_{12} \Rightarrow \tau_{22} \rangle^l\ (f\ ((\langle \tau_{21} \Rightarrow \tau_{11} \rangle^l)\ v))$$

wrapping the argument v in a (contravariant) cast between the domain types and wrapping the result of the application in a (covariant) cast between the codomain types.

To better understand how function casts work lets go back to our running example, `pred`:

$$\texttt{pred} :: n : \{v : Int | v > 0\} -> \{v : Int | v = n - 1\}$$

To get this type signature for `pred`, we have to wrap the function's definition in a type cast:

$$\texttt{pred'}\ x = x - 1$$
$$\texttt{pred} = \langle Int \to Int \Rightarrow x : \{v : Int | v > 0\} -> \{v : Int | v = x - 1\} \rangle^{lpred}\ \texttt{pred'}$$

Now, when we apply a positive number, say $2 :: \{v : Int | v > 0\}$, we will have the following computation:

$$\begin{aligned}
\texttt{pred}\ 2 &= (\langle Int \to Int \Rightarrow x : \{v : Int | v > 0\} -> \{v : Int | v = x - 1\} \rangle^{lpred}\ \texttt{pred'})\ 2 \\
&\to^{\star} (\langle Int \Rightarrow \{v : Int | v = 2 - 1\} \rangle^{lpred})\ (\texttt{pred'}\ (\langle \{v : Int | v > 0\} \Rightarrow Int \rangle^{lpred}\ 2)) \\
&\to^{\star} (\langle Int \Rightarrow \{v : Int | v = 2 - 1\} \rangle^{lpred})\ (\texttt{pred'}\ 2) \\
&\to^{\star} (\langle Int \Rightarrow \{v : Int | v = 2 - 1\} \rangle^{lpred})\ 1 \\
&\to^{\star} 1
\end{aligned}$$

Thus, if `pred'` was not returning the correct value, the program would raise a blame:

$$\texttt{pred'}\ x = 0$$
$$\begin{aligned}
\texttt{pred}\ 2 &= (\langle Int \to Int \Rightarrow x : \{v : Int | v > 0\} -> \{v : Int | v = x - 1\} \rangle^{lpred}\ \texttt{pred'})\ 2 \\
&\to^{\star} (\langle Int \Rightarrow \{v : Int | v = 2 - 1\} \rangle^{lpred})\ (\texttt{pred'}\ (\langle \{v : Int | v > 0\} \Rightarrow Int \rangle^{lpred}\ 2)) \\
&\to^{\star} (\langle Int \Rightarrow \{v : Int | v = 2 - 1\} \rangle^{lpred})\ (\texttt{pred'}\ 2) \\
&\to^{\star} (\langle Int \Rightarrow \{v : Int | v = 2 - 1\} \rangle^{lpred})\ 0 \\
&\to^{\star} \Uparrow\ lpred
\end{aligned}$$

You may notice, that in both cases, `pred'` is applied to a "casted" value $\langle \{v : Int | v > 0\} \Rightarrow Int \rangle^{lpred}\ 2$. This is a downcast: the system statically knows the source type of 2, i.e., $\{v : Int | v > 0\}$ and at runtime it is checked that 2 is an Int. Thus, for an application to statically typecheck, the argument should have a refined type and the only way to get a refined type is though a cast. But, if we cast a non-positive value to be positive, then this cast will fail:

$$\texttt{pred}\ ((\langle Int \Rightarrow \{v : Int | v > 0\} \rangle^{zero})\ 0) = \Uparrow\ zero$$

3.2.2 Formal Language

Lets now extend our core calculus λ_c to $\lambda_c c$, so that it supports manifest contracts. In the expressions of our language we should add a blaming expression and a type casting. As a refinement the language can use any core expression. Everything else remains unchanged.

In the typing judgements we add two rules: a blame expression can have any well formed types, while a type casting expression behaves as a function from the source to the target type. For a casting expression to typecheck, both types should be well formed and compatible, i.e., their unrefined types should be the same. We check this through a new compatibility judgement.

$$\begin{array}{rrcl} \textbf{\textit{Expressions}} & e & ::= & \ldots \mid \Uparrow\ l \mid \langle \tau \Rightarrow \tau \rangle^l \\ \textbf{\textit{Predicates}} & p & ::= & e \end{array}$$

Figure 3: **Syntax of Expressions, Types and Schemas**

Compatibility $\boxed{\tau_1 \| \tau_2}$

$$\frac{}{\{v : b \mid p_1\} \| \{v : b \mid p_2\}}\ \text{C-Base} \qquad \frac{\tau_{x_1} \| \tau_{x_2} \quad \tau_1 \| \tau_2}{x_1 : \tau_{x_1} \to \tau_1 \| x_2 : \tau_{x_2} \to \tau_2}\ \text{C-Fun}$$

Type Checking $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \Uparrow\ l : \tau}\ \text{T-Label} \qquad \frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2 \quad \tau_1 \| \tau_2}{\Gamma \vdash \langle \tau_1 \Rightarrow \tau_2 \rangle :: \tau_1 \to \tau_2}\ \text{T-Cast}$$

Figure 4: **Static Semantics: Well-formedness, Subtyping and Type Checking**

3.3 Hybrid Type Checking

The great disadvantage of contract calculi is that all contracts are checked in runtime, so type checking consumes cycles that otherwise would perform useful computation. Moreover limited coverage is provided: contracts are checked only for data values and code paths of actual execution. This disadvantages are eliminated in Flanagan's Hybrid Type Checking [6].

Instead of type casting, Flanagan used subtyping to convert values from one type to another. So, if we want to prove that 2 is a positive number, we have to prove that the following subtyping holds:

$$\{v : Int | v = 2\} <: \{v : Int | v > 0\}$$

This, in turn reduces to implication checking:

$$v = 2 \Rightarrow v > 0$$

which is easy to prove with some algorithm that uses linear arithmetic.

Flanagan's type system checks implications statically, whenever possible and dynamically, only when necessary. He assumes that there exists an algorithm that within limited time can conservatively approximate implications between predicates. Say that we know that $e :: \tau_s$ and we want to prove that $e :: \tau_t$, then we have to prove that the source type is subtype of the target, or $\tau_s \preceq \tau_t$. Subtyping is reduced to implication checking, thus implications of the form $e_s \Rightarrow e_t$ are created. Then, we apply the algorithm to each implication and there are three cases:

- The algorithm proves that all implications hold, $\vdash e_s \Rightarrow e_t$, which implies that $\tau_s \preceq \tau_t$ always holds.
- The algorithm proves that some implication does not hold, $\nvdash e_s \Rightarrow e_t$, which implies that the subtyping does not hold and the program is rejected as unsafe.
- The algorithm can not prove any of the above. Thus, the expression e is annotated with a type cast $\langle \tau_s \Rightarrow \tau_t \rangle$ to dynamically ensure that values returned by e are actually of the desired type τ_t.

So the program is transformed to include all required type casts, i.e., the ones that can not be statically proved.

Compared to manifest contacts, Hybrid type system has two advantages: First, type casts are automatically created, thus the annotation burden is lower for the programmer. Moreover, whenever possible, subtyping is statically checked, which leads to both limiting the run-time checks and broader coverage of the contract checks.

3.3.1 Formal Language

We will extend our core language λ_{cc} to λ_{ch}, so as to support hybrid type checking. The syntax of the language is not extended, but we need to add the subtyping rules. As we stated, the source program goes through a transformation and the necessary casts are added. Thus, every time the T-SUB rule is used, it is guaranteed that the algorithm can prove the subtyping relation.

The rule T-SUB is the one defined in the 2 where the `Valid` predicate refers to the algorithm used for implications checking.

Type Checking $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \tau_2 \quad \Gamma \vdash \tau_2 \preceq \tau_1 \quad \Gamma \vdash \tau_1}{\Gamma \vdash e : \tau_1} \text{ T-SUB}$$

Figure 5: **Static Semantics: Well-formedness, Subtyping and Type Checking**

4 Liquid Types

In this section we will present a dependent type system which is decidable. As we stated in the introduction, if the refinement language can contain arbitrary program expressions, then type checking is undecidable. In Liquid Types[17], Rondon et all restricted the refinement logic according to a predetermined finite set of qualifiers, which gives not only decidable type checking, but also automatic type inference.

The system takes as input a program and a finite set of *logical qualifiers* which are simple boolean predicates that encode the properties we care to verify. The system then infers liquid types, which are dependent types where the refinement predicates are conjunctions of the logical qualifiers. In the system, type checking and inference are decidable for three reasons. First, they use a conservative but decidable notion of subtyping, where they reduce the subtyping of arbitrary dependent types to a set of implication checks over base types, each of which is deemed to hold if and only if an embedding of the implication into a decidable logic yields a valid formula in the logic. Second, an expression has a valid liquid type derivation only if it has a valid unrefined type derivation, and the dependent type of every subexpression is a refinement of its vanilla type. Third, in any valid type derivation, the types of certain expressions must be liquid. Thus, inference becomes decidable, as the space of possible types is bounded.

4.1 Logical Qualifiers and Liquid Types.

A logical qualifier is a boolean-valued expression (i.e., predicate) over the program variables, the special value variable `v` which is distinct from the program variables, and the special placeholder variable $\star$ that can be instantiated with program variables. We can assume, that $\mathcal{Q}$ is the set of logical qualifiers $\{0 \leq v, \star \leq v, v < \star, v = \star + 1\}$. We say that a qualifier q matches the qualifier q' if replacing some subset of the free variables in q with $\star$ yields q'. For example, the qualifier $x \leq v$ matches the qualifier $\star \leq v$. We write $\mathcal{Q}^\star$ for the set of all qualifiers not containing $\star$ that match some qualifier in $\mathbb{Q}$. For example, when $\mathbb{Q}$is as defined as above, and x, y and n are program variables. $\mathbb{Q}^\star$ includes the qualifiers $\{0 \leq v, x \leq v, y \leq v, v = n + 1, v < n\}$. A liquid type over $\mathbb{Q}$is a dependent type where the refinement predicates are conjunctions of qualifiers from $\mathbb{Q}^\star$.

4.2 Overview

4.2.1 Type Inference

As an example consider our pred example

```
pred n = n - 1
```

The liquid type for `pred` can be inferred in three steps:

(Step 1) By Hindley-Milner algorithm, we can infer that `pred` has the type `Int -> Int`. Using this type, we create a template for the liquid type of `pred`,

```
pred :: n:{v:Int | k_n} -> {v:Int | k_p}
```

where `k_n` and `k_p` are liquid type variables representing the unknown refinements for the argument n and the body of `pred`, respectively.

(Step 2)If moreover we assume:

```
(-) :: x:Int -> y:Int -> {v:Int | v = x - y}
```

The the result of `pred` has a type

```
{v:Int | v = x - y}[x/n][y/1] = {v:Int | v = n - 1}
```

But this should be subtype of the template type for the result. So, we get the subtyping constraint

```
n - 1 :: {v:Int | v = n - 1} <: {v:Int | k_p}
```

which reduces to the implication constraint

$$v = n - 1 \Rightarrow k\_p$$

(Step 3)Since the program is "open", i.e., there are no calls to `pred`, we assign `k_p` true, meaning that any integer arguments can be passed, and use a theorem prover to find the strongest conjunction of qualifiers in $\mathbb{Q}$that satisfies the subtyping constraints. The algorithm infers that $v = n - 1$ is the strongest solution for `k_p`. By substituting the solution for `k_p` into the template for `pred`, our algorithm infers

```
pred :: n:Int -> {v:Int | v = n-1}
```

4.2.2 Type Checking

As one may notice the inferred type signature of `pred` does not constrain the type of the argument. This happens because the code for pred produces no such constraint. We could manually specify a type for `pred`

```
pred :: n:{v:Int | v > 0} -> {v:Int | n - 1}
```

Then then first step of the inference will be skipped and the body of the function will be type-checked against the given signature.

So, in the second step, as before, we will create an implication

$$n - 1 \Rightarrow n - 1$$

. That can trivially be verified in the third step.

We have use the above type signature and we call `pred` with some integer vale, say 2 then in the call site the constraint $v = 2 \Rightarrow v > 0$ will be generated, that can easily be verified. On the other hand, if we call it with zero, we will get the constraint $v = 0 \Rightarrow v > 0$, that does not hold, so our program will be unsafe.

4.3 Applications of Liquid Types

Liquid Types, as introduced in [17] used OCaml as target language and were used to verify properties such as array bound checking. In [11] they were extended with recursive and polymorphic refinements to enable static verification of complex data structures. For example they proved that a list is sorted or that a tree is a Binary Search Tree. Liquid Types can be used to verify properties even in imperative languages. In [18], Rondon et. all. present a refinement type system for C based on Liquid Types to verify memory safety properties, like the absence of array bounds violations and null-dereferences. In [12], Kawaguchi et. all. present Liquid Effects, a type-and-effect system based on refinement types which allows for fine-grained, low-level, shared memory multithreading while statically guaranteeing that a program is deterministic.

$$\textbf{\textit{Predicates}} \qquad p ::= \quad \text{true} \mid q \mid p \wedge p \,,\; q \in \mathbb{Q}^{\star}$$

Figure 6: **Syntax of Expressions, Types and Schemas**

4.4 Formal Language

Once again, we would like to extend λ_c such as to support liquid type checking.

As we said earlier, the refinements in the language can not contain arbitrary expressions, but they are limited to conjuctions of the logical qualifiers, which form a finite set.

They use a subtyping is the one defined in 2 but in this setting, the subtyping relation is decidable because their refinement language is restricted and can be checked with a SMT solver.

Type Checking $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \tau_2 \qquad \Gamma \vdash \tau_2 \preceq \tau_1 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash e : \tau_1} \;\text{T-SUB}$$

Figure 7: **Static Semantics: Well-formedness, Subtyping and Type Checking**

5 Abstract Refinement Types

In this section, we present abstract refinement types which enable quantification over the refinements of data- and function-types. The key insight is that we can avail of quantification while preserving SMT-based decidability, simply by encoding refinement parameters as uninterpreted propositions within the refinement logic. We illustrate how this mechanism yields a variety of sophisticated means for reasoning about programs, including: index-dependent refinements for reasoning about key-value maps, recursive refinements for reasoning about recursive data types, and inductive refinements for reasoning about higher-order traversal routines.

5.1 The key idea

Consider the monomorphic `max` function on `Int` values. We can give `max` a refinement type, stating that its result is greater or equal than both its arguments:

```
max      :: x:Int -> y:Int -> {v:Int | v >= x && v >= y}
max x y = if x > y then x else y
```

With this type signature, if we apply `max` to two positive integers, say `n` and `m`, we can get that the result is grater or equal to both of them, as `max n m :: {v:Int | v >=n && v >=m}`. But we can not reason about other properties: If we apply `max` to two even numbers, can not verify that the result is also even. Thus, even though we have the information on the input, we lose it on the result.

To solve this problem, we introduce *abstract refinements* which let us quantify or parameterize a type over its constituent refinements. For example, we can type `max` as

```
max :: forall <p::Int->Bool>. Int<p> -> Int<p> -> Int<p>
```

where `Int<p>` is an abbreviation for the refinement type `{v:Int | p(v)}`. Intuitively, an abstract refinement `p` is encoded in the refinement logic as an *uninterpreted function symbol*, which satisfies the *congruence* axiom [15]

$$\forall \overline{X}, \overline{Y} : (\overline{X} = \overline{Y}) \Rightarrow P(\overline{X}) = P(\overline{Y})$$

It is trivial to verify, with an SMT solver, that `max` enjoys the above type: the input types ensure that both `p(x)` and `p(y)` hold and hence the returned value in either branch satisfies the refinement `{v:Int | p(v)}`, thereby ensuring the output type.

In a call site, we simply instantiate the *refinement* parameter of `max` with the concrete refinement after which type checking proceeds as usual. As an example, suppose that we call `max` with two even numbers:

`n :: {v:Int | v % 2 = 0}`, `m :: {v:Int | v % 2 = 0}`,

Then, the abstract refinement can be instantiated with a concrete predicate `{\v -> v % 2 = 0}`, which will give `max` the type `max [{\v -> v % 2 = 0}] :: {v:Int | v % 2 = 0} -> {v:Int | v % 2 = 0} -> {v:Int | v % 2 = 0}` Since both `n` and `m` are even numbers we can verify that the preconditions hold, so the result will also be even: `max [{\v -> v % 2 = 0}] n m :: {v:Int | v % 2 = 0}`.

This is the basic concept of abstract refinements, which , as we will see have many applications.

5.2 Inductive Refinements

Consider a `loop` function that takes a function `f`, an integer `n`, a base case `z` and applies the function `f` to the `z`, `n` times:

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n     = go (i+1) (f i acc)
                 | otherwise = acc
```

We can use inductive reasoning for this function, as

- For any *loop invariant* `R :: (Int, a)` that relates the loop iteration `i` with the accumulator `acc`
- **Base Case** If `R(0, z)` holds, thus if `z` satisfies the loop iteration at `0`.
- **Inductive Step** If `R(i, acc)`⇒`R(i+1, f i acc)` holds; thus, if the function `f` preserves the loop invariant.
- **Conclusion** Then `R(n, loop f n z)` holds; thus `loop`'s result satisfies the invariant at `n`.

We can use an abstract refinement `r :: Int -> a -> Prop`, that relates the loop iteration `i` with the accumulator `acc`, to encode the loop invariant `R`. With this, we give `loop` a type that actually encodes induction:

```
loop :: forall <r :: Int -> a -> Prop> .
           f : (i:Int -> a<r i> -> a<r (i+1)>)
        -> n : {v:Int | n >=0}
        -> z : a<r 0>
        -> a<r n>
```

This type says that for any invariant `r`, if the function `f` preserves the invariant, `n` is a natural number and `z` satisfies the invariant at `0`, then `loop`'s result will satisfy the invariant at `n`.

Now, consider a user function `incr` that uses `loop` and at each iteration increases the accumulator by one:

```
incr :: Int -> Int -> Int
incr n z = loop g n z
  where g i acc = acc + 1
```

In this case, the invariant is that at each iteration `i`, the accumulator is equal to `i + z`, or $R(i, acc) \Leftrightarrow acc = i + z$. If we instantiate the abstract refinement in `loop` with this concrete refinement, we get a concrete refinement type for `loop`:

```
loop [\i acc -> acc = i + z]
        :: f : (i:Int -> {v:a | v = i+z} -> {v:a | v = i
           +1+z})
        -> n : {v:Int | n >=0}
        -> z : {v:a | v = z}
        -> {v:a | v = n + z}
```

We can prove that `loop`'s precondition is satisfied by `g`, thus, we can apply it to `loop` and get `incr`'s type:

```
incr :: n : {v:Int | n >= 0}
     -> z : Int
     -> {v:Int | v = n + z}
```

This type better describes `incr`'s behaviour, as it states that if we apply `incr` any natural number `n` and any integer `z`, we will get `n + z`.

5.3 Function Composition

As a next example, we will see how one can use abstract refinements to reason about function composition.

Consider a `plusminus` function that composes a plus and a minus function:

```
plusminus :: n:Int -> m:Int -> x:Int -> {v:Int | v = (x -
    m) + n}
plusminus n m x = (x - m) + n
```

However, consider an alternative definition that uses function composition `(.)`

```
plusminus n m x = plus . minus
  where plus  x = x + n
        minus x = x - m
```

It is unclear how to give `(.)` a first order refinement type that expresses that the result can be refined with the composition of the refinements of both arguments results.

To solve this problem, we can use abstract refinements and give `(.)` a type:

```
(.) :: forall < p :: b -> c -> Prop
              , q :: a -> b -> Prop>.
       f : (x:b -> c<p x>)
    -> g : (x:a -> b<q x>)
    -> x : a
    -> exists[z:b<q x>]. c<p z>
```

This type abstracts over the refinement `p` of the result of the first function `f` and the refinement `q` of the result of the second function `g` and for any argument `x`, the intermediate result is binded to `z = g x`, so `z` satisfies `q` at `x`, and returns a value that satisfies `p` at the intermediate result.

So back to `plusminus` example, with the appropriate instantiation we get the concrete refinement type for function composition:

```
(.) [{\x v -> v = x + n}, {\x v -> v = x - m}]
    :: f : (x:b -> {v:c | v = x+n})
    -> g : (x:a -> {v:b | v = x-m})
    -> x : a
    -> exists[z:{v:b | v = x-m}]. {v:c | v = z+n}
```

With this type, it is straightforward to prove the type of `plusminus`.

5.4 Index-Dependent Invariants

Next, we illustrate how abstract invariants allow us to specify and verify index-dependent invariants of key-value maps. To this end, we encode *extensible vectors* as functions from `Int` to some generic range `a`. Formally, we specify vectors as

```
data Vec a <dom :: Int -> Bool, rng :: Int -> a -> Bool>
  = V (i:Int<dom> -> a <rng i>)
```

Here, we are parameterizing the definition of the type `Vec` with *two* abstract refinements, `dom` and `rng`, which respectively describe the *domain* and *range* of the vector. That is, `d` describes the set of *valid* indices, and `r` specifies an invariant relating each `Int` index with the value stored at that index.

Describing Vectors. With this encoding, we can descibe various vectors. To start with we can have vectors of `Int` defined on positive integers with values equal to their index:

```
Vec <{\v -> v > 0}, {\_ v -> v = x}> Int
```

Or a vector that is defined only on index 1 with value 12:

```
Vec <{\v -> v > 0}, {\_ v -> v = x}> Int
```

As a more interesting example, we can define a *Null Terminating String* with length `n`, as a vector of `Char` defined on a range `[0, n)` with its last element equal to the terminating character:

```
Vec <{\v -> 0 <= v < n }, {\i v -> i = n-1 => v = '\0'}>
    Char
```

Finally, we can encode a Fibonacci memoization vector, that is defined on positive integers and its value on index `i` is either zero or the `i`th Fibonacci, and we can use this vector to efficiently compute a Fibonacci number:

```
Vec <{\v -> 0 <= v }, {\i v -> v != 0 => v = fib(i)}>
    Char
```

Using Vectors. To use the vectors, first we have to type vector operations, like set, get and empty, the appropriate types, which actually means to abstract over the domain and the range predicates. Then, we have to specify the properties we are interested about, as we did for the Fibonacci memoization, or the null terminating string. Finally, we can verify that user function, that transforms the vestors preserve these properties.

5.5 Recursive Invariants

Next, we turn our attention to recursively defined datatypes, and show how abstract refinements allow us to specify and verify high-level invariants that relate the elements of a recursive structure. Consider the following refined definition for lists:

```
data [a] <p :: a -> a -> Bool> where
  []  :: [a]<p>
  (:) :: h:a -> [a<p h>]<p> -> [a]<p>
```

The definition states that a value of type `[a]<p>` is either empty (`[]`) or constructed from a pair of a *head* `h::a` and a *tail* of a list of `a` values *each* of which satisfies the refinement `(p h)`. Furthermore, the abstract refinement `p` holds recursively within the tail, ensuring that the relationship `p` holds between *all* pairs of list elements.

Thus, by plugging in appropriate concrete refinements, we can define the following aliases, which correspond to the informal notions implied by their names:

```
type IncrList a = [a]<{\h v -> h <= v}>
type DecrList a = [a]<{\h v -> h >= v}>
type UniqList a = [a]<{\h v -> h != v}>
```

That is, `IncrList a` (resp. `DecrList a`) describes a list sorted in increasing (resp. decreasing) order, and `UniqList a` describes a list of *distinct* elements, *i.e.,* not containing any duplicates. We can use the above definitions to verify

```
[1, 2, 3, 4] :: IncrList Int
[4, 3, 2, 1] :: DecrList Int
[4, 1, 3, 2] :: UniqList Int
```

More interestingly, we can verify that the usual algorithms produce sorted lists:

```
insertSort :: (Ord a) => [a] -> IncrList a
insertSort []     = []
insertSort (x:xs) = insert x (insertSort xs)

insert :: (Ord a) => a -> IncrList a -> IncrList a
insert y []       = [y]
insert y (x:xs)
  | y <= x        = y : x : xs
  | otherwise     = x : insert y xs
```

Thus, abstract refinements allow us to *decouple* the definition of the list from the actual invariants that hold. This, in turn, allows us to conveniently reuse the same underlying (non-refined) type to implement various algorithms unlike, say, singleton-type based implementations which require up to three different types of lists (with three different "nil" and "cons" constructors [19]). This, makes abstract refinements convenient for verifying complex sorting implementations like that of `Data.List.sort` which, for efficiency, use lists with different properties (*e.g.,* increasing and decreasing).

5.6 Formal Language

We suggest that any refinement system can be extended with abstract refinements without increasing its complexity. First of all, the syntax should be extended to support refinement abstraction and application: If refinement abstraction, we abstract from an expression e the refinement π with type τ, while in refinement application we instantiate an abstrast refinement with a concrete one p that may have some parameters $\bar{x}$. Then, the predicates of the language should be extended to include abstract refinements, applies to program expressions The types of the language should also be extended to include refinement abstraction. Finally, two typing rules should be added for the two new expressions: The refinement abstraction expression is typed as an refinement abstraction type, in a straightforward way. In the refinement application, the abstract refinement π is replaced with a concrete one over the type τ

We have to note that Abstract refinements can be treated as uninterprented functions in the implication checking algorithm, thus the complexity of the system is not increased. Moreover, they appear only in the types, thus they can be erased in runtype. Finally, in [21] refinement abstraction and application can be inferred, thus the user does not have to actually alter the program and annotate it with explicit refinement instantiations.

$$
\begin{array}{rrcl}
\textbf{\textit{Expressions}} & e & ::= & \ldots \mid \Lambda\pi : \tau.e \mid e\,[\lambda\overline{x : \tau_x}.p] \\
\textbf{\textit{Predicates}} & p & ::= & \ldots \mid \pi\,\overline{e} \\
\textbf{\textit{Types}} & \tau & ::= & \ldots \mid \forall\pi : \tau.\tau
\end{array}
$$

Figure 8: **Syntax of Expressions, Types and Schemas**

Type Checking $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma, \pi : \tau_\pi \vdash e : \tau \quad \Gamma \vdash \tau}{\Gamma \vdash \Lambda\pi : \tau_\pi.e : \forall \pi : \tau_\pi.\tau} \text{ T-PGEN} \qquad \frac{\Gamma \vdash e : \forall \pi : \tau_\pi.\tau \quad \Gamma \vdash \lambda\overline{x : \tau_x}.p : \tau_\pi}{\Gamma \vdash e\,[\lambda\overline{x : \tau_x}.p] : \tau[\pi \rhd \lambda\overline{x : \tau_x}.p]} \text{ T-PINST}$$

Figure 9: **Static Semantics: Well-formedness, Subtyping and Type Checking**

6 Conclusion

In this report we presented various dependent type systems. We started with type systems where the predicates can be arbitrary expressions. Even though these systems are expressive, the assertions formed can not be statically verified. Thus to reason about such systems, the user should provide explicit proves for the assertions, as in interactive theorem proving, or the assertions can be verified at run time, as in contracts calculi. Next we presented Liquid Types, a dependent type system which restricts the predicate language, thus both type checking and inference is decidable. Finally, we presented Abstract Refinement Types, a means which can be used in a dependent type system to increase expressiveness without increasing complexity.

References

[1] *Certified Programming with Dependent Types*. MIT Press, 2013.

[2] J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *ESOP*, pages 18–37, 2011.

[3] Matthias Blume and David A. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4-5):375–414, 2006.

[4] Olaf Chitil and Frank Huch. Monadic, prompt lazy assertions in haskell. In *APLAS*, pages 38–53, 2007.

[5] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, 2002.

[6] C. Flanagan. Hybrid type checking. In *POPL*. ACM, 2006.

[7] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. *JFP*, 22(3):225–274, May 2012.

[8] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[9] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *DLS*, pages 29–40, 2007.

[10] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *FLOPS*, pages 208–225, 2006.

[11] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, pages 304–315, 2009.

[12] Ming Kawaguchi, Patrick Maxim Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic parallelism via liquid effects. In *PLDI*, pages 45–54, 2012.

[13] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[14] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[15] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[16] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

[17] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[18] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *POPL*, pages 131–144, 2010.

[19] T. Sheard. Type-level computation using narrowing in omega. In *PLPV*, 2006.

[20] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *POPL*, pages 395–406, 2008.

[21] N. Vazou, P. Rondon, and R. Jhala. Abstract refinements. In *ESOP*, 2013.

[22] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, pages 1–16, 2009.