Refinement Types and Abstract Refinements

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

Abstract

Even well-typed programs can go wrong, by returning a wrong answer or throwing a run-time error. A popular response is to allow programmers use *refinement type systems* to express semantic specifications about programs. We study verification in such systems. On one hand, expressive refinement type systems require run-time checks or explicit proofs to verify specifications. On the other hand, less expressive type systems allow static and automatic proofs of the specifications. Finally, we present abstract refinement types, a means to enhance the expressiveness of a refinement 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, but not that it is always an integer with a certain property, say different than zero. Thus, the type system is unable to statically ensure the safety of critical operations, such as a division by i. Several authors have proposed the use of refinement types [14, 13, 29, 15] as a mechanism for enhancing the expressiveness of type systems.

Refinement types refine a vanilla type with a predicate. For example, one can give `i` the following type:

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

that describes a value `v` of type `Int`, while, the refinement constraints this value `v` to be different than `0`.

One can use this refinement type to define a safe division operator:

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

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

At the call side of `safeDiv` the type system should check that the real arguments do not violate its specification. For instance, `safeDiv 8 9` is safe, since `9` is always different that zero. But, `safeDiv 8 0` should create a type error. Apart from concrete values, `safeDiv` can be applied to arbitrary program expressions: `safeDiv n m` is safe only if `m` is an integer different than zero.

Refinement Function Types. Refinement function types[2, 13], allow the specification of the result to depend on the argument. A parameter is used to bind the argument

and can appear in the refinement of the result. As an example, we define a `pred` function, that takes as argument a positive integer `n` and returns the its predecessor. Refinement function types allow us to give `pred` a type that exactly captures this behaviour:

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

This type expresses that for each positive integer argument `n`, the result is an `Int` exactly equal to `n-1`. When `pred` is applied to a concrete value, the parameter `n` is substituted with this value. For example `pred 2 :: {v:Int | v = n-1}[2/n] = {v:Int | v = 1}`. Thus, for each concrete argument, the result should be the predecessor of this argument.

Verification. *Verification* is a procedure that takes as input a program, i.e., definitions for functions and values, and some specifications, i.e., refinement type signatures for functions and values, and decides whether the specifications hold for the program. Informally, it checks that each expression satisfies its type, for example, the `pred` definition actually returns the predecessor of its argument, or that at each function application the arguments satisfy the function's preconditions, as in the `safeDiv` example. If the specifications hold, the program is *Safe*, otherwise it is *Unsafe*.

Higher-order programming languages, such as ML or Haskell, treat functions as first order objects. Thus, one can use functions in refinements and create higher-order predicates. For instance, the following type

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

describes that an arbitrary functional argument should satisfy a predicate `terminates`. Reasoning in a higher-order logic is undecidable, thus if arbitrary program values appear in the refinements, the verification procedure is undecidable. As we shall see, if the refinement language is restricted, i.e., is less expressive, verification can be decidable.

The rest of this paper is organized as follows: In section 2, we present a core calculus that constitutes the base for many refinement type systems. In section 3, we describe reasoning in undecidable refinement type systems. In section 4, we present less expressive type systems that are decidable. In section 5 we present, how abstraction over refinements enhances expressiveness of decidable type systems. Finally, we conclude.

2 Preliminaries

To formally describe and compare type systems, we define a core calculus, following [13, 29, 15]. We refer to our calculus as λ_C, and in this section we present its syntax and type system.

2.1 Syntax

The syntax of expressions and types is summarized in Figure 1. λ_C expressions include variables, constants, typed λ-abstractions and function applications. Constants include primitive integers: $0, 1, 2, \ldots$ and primitive booleans: `true` or `false`, which take the basic types, integer and boolean, respectively. A basic type can be refined with a

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

Figure 1: **Syntax of λ_C**

predicate to construct a basic refinement type. Refinement types also contain function types, in which, a variable binds the argument, so that the result refinement can refer to it. The predicate p is not yet defined. As noted earlier, if p contains arbitrary program expressions, the type system is undecidable, but p can be restricted in such a way as to render the type system decidable. Finally, we define a typing environment Γ, that maps variables to their type, and will be used in the rules.

2.2 Typing

The typing rules used by λ_C are summarized in Figure 2.

Type Checking. Type checking rules state that an expression e has a type τ under an 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. We write $\Gamma \vdash e : \tau$ and create one rule for each program expression.

The rule T-CONST uses a function tc that maps each primitive constant to its predefined type. The rule T-VAR checks the type of a variable, according to the environment Γ. The rule T-FUN checks the type of the function-body in the environment, extended with the argument of the function. Since the argument type is given, it could be any arbitrary type, say $\{v : b \mid 1\}$, which is invalid, as a base type is refined with the value 1, which can not be a valid predicate. A wellformedness rule is used to check that the argument type is *well-formed*, i.e., its refinements are valid predicate expressions. Finally, the rule T-APP checks that in an application $e_1\ e_2$ the expression e_1 has a function type whose argument is the type of the argument e_2. As we discussed in the introduction, in the final type, the parameter x should be replaced with the actual argument e_2.

Wellformedness rules. Wellformedness rules state that a type τ is well-formed under environment Γ, that is, the refinements in τ are boolean expressions in the environment Γ. We write $\Gamma \vdash \tau$ and create one rule for each type.

The rule WF-BASE checks that in a basic refinement type, the refinement is a valid boolean expression. The environment of this check is extended with the value that is refined; for instance, to check the validity of $\{v : \texttt{int} \mid v > 0\}$, we check that $v > 0$ is a boolean expression, in an environment where v is an integer value. The rule WF-FUN recursively applies the wellformedness rule to the argument type of the function, and to the result type, in an environment extended with the argument parameter.

Subtyping rules. Consider that the predefined type for the integer 2 is an integer that is exactly equal to 2. The type system can check,via the rule T-CONST, that $\emptyset \vdash 2 : \{v : \texttt{int} \mid v = 2\}$. If 2 is applied to a function that expects a positive integer, say $f :: x : \{v : \texttt{int} \mid v > 0\} \rightarrow \tau$, the type system should also check that $\emptyset \vdash 2 :$

$\{v : \texttt{int} \mid v > 0\}$. There are many ways for this check to succeed. We follow *syntactic subtyping*, in which subtyping reduces to implication checking. In our example, $v = 2 \Rightarrow v > 0$ implies $\{v : \texttt{int} \mid v = 2\} \preceq \{v : \texttt{int} \mid v > 0\}$.

In the general case, subtyping rules 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. We write $\Gamma \vdash \tau_1 \preceq \tau_2$ and create one rule for every type:

The rule $\prec$-BASE serves two purposes: Firstly, it checks that the basic type is the same in the two types. Moreover, it checks that under the environment Γ, the left hand side refinement implies the right hand side. The implication checking is enforced by a predicate `Valid` which varies between the systems that we will describe. The rule $\prec$-FUN relates two function types according to the contravariant rule.

In the rest of this paper, we will use the core calculus λ_C upon which we will build a subset of three typing systems[29, 15, 36].

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

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

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

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

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

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

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

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

Figure 2: **Static Semantics for λ_C**

3 Undecidable Systems

In systems where the refinement language can have arbitrary program expressions, higher order predicates can be expressed, thus the verification procedure is undecidable. There are many alternatives to reason in such languages, in this section we will present two of them: Firstly, we present *Interactive theorem proving*; where the proofs are statically provided by the user. Secondly, we present *Contracts Calculi*; where the specifications are checked at run time.

```
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)

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.
```

Figure 3: The pred function in Coq

3.1 Interactive theorem Proving

One approach to verify that a program satisfies some specifications is the user to statically prove them. This approach is used by *interactive theorem provers*, such as NuPRL [9], Coq [5], F* [33], Agda [26] and Isabelle [25] 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 once again the `pred` function that computes the predecessor of a positive number.

```
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 Coq[1] the refinement type is defined in the standard library, and is syntactic sugar, for the type family `sig`. With this, one can define the `pred` function, as presented in Figure 3. The function `pred` takes an argument `s` which has two components: a natural number `n` and a proof `pf` that this number is positive. There is a case analysis on `s`: 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_refl`, the only constructor of equality to `S n'`.

In the call side of `pred`, the programmer should provide both the argument and a proof that it is positive. As an example, if `two_gt0` is a proof that two is greater than zero, we can have

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

This application typechecks, as Coq verifies that the argument satisfies `pred` precondition, or `two_gt0` is indeed a proof that 2 is greater than 0.

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

3.2 Contracts

Another approach to verify that a program satisfies some assertions is to 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 [22], an object-oriented programming language, totally adopted assertions and developed the "Design by Contract" philosophy [23].

Contracts are of the form: $\langle\{\mathtt{v}:\tau \mid p\}\rangle^l$. The refinement part, as usual, describes 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 $\langle\{\mathtt{v}:\mathtt{Int} \mid v > 0\}\rangle^l$ applied to two values, 2 and 0:

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

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".

In 2002, Findler and Felleisen [12] were the first 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 a "variance-contravariance" rule. Moreover, they allow dependent contracts, i.e. contracts that have the form of a refinement 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 [7] formally defined contract satisfaction on Findler and Felleisen system, 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. [18], Blume and McAllester [7], Chitil and Huch [8], Guha et al. [17], and Tobin-Hochstadt and Felleisen [34]. By contrast, *manifest contracts* are integrated into the type system, which tracks, for each value, the most recently checked contract. Hybrid types [13] are a well-known example in this style; others include the work of Ou et al. [27], Wadler and Findler [38], and Gronski et al. [16], Belo et al. [4] and Grennberg et al. [15]. In the rest subsection we discuss manifest contracts and present a core calculus for them.

3.2.1 Manifest Contracts

Manifest Contracts Systems[15], 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:

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

The system should statically verify that the value `n` is of the source type `Int`. After the cast, this value is treated as if it has the target type $\{\texttt{v:Int | v > 0}\}$. 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 application cast $(\langle \tau_{11} \texttt{ -> } \tau_{12} \Rightarrow \tau_{21} \texttt{ -> } \tau_{22} \rangle^l \texttt{ f})\ \texttt{v}$ will reduce to $\langle \tau_{12} \Rightarrow \tau_{22} \rangle^l\ (\texttt{f}\ (\langle \tau_{21} \Rightarrow \tau_{11} \rangle^l\ \texttt{v}))$ 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 once more consider the `pred` example. To get desired the type signature for `pred`, we have to wrap the function's definition in a type cast:

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

If we apply a positive number, say `2 :: {v:Int | v > 0}`, we will have the following computation:

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

The first line is `pred`'s definition. In the second line the rule for functional cast is applied. Then, the check that `2` is an integer succeeds, and `2` is applied to `pred'` so, we get `1`. Finally, this result is checked to be `1` and since this check succeeds the value is returned. If `pred'` was not returning the correct value, the program would raise a blame:

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

The evaluation is the same as in the previous example, up to the point where the `pred'` application returns. Here, the application returns `0`, thus the final check fails and the program raises a blame.

You may notice, that in both cases, `pred'` is applied to a positive integer. Since a positive integer is not a primitive type, the only way to get such a type is via a cast. Thus, for this application to statically typecheck, the argument should be wrapped in a cast. But, if we cast a non-positive value to be positive, then the cast will fail:

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

We saw that two distinct casts should be used to satisfy the functions pre- and post-conditions. These casts use different labels, with which we can track the source of failure, if any.

3.2.2 Formal Language

Lets now extend our core calculus λ_c to λ_{cc}, so that it supports manifest contracts. In the expressions of our language we add a blaming expression and a type cast. The refinement language includes any core expression. Everything else remains unchanged.

In the typing judgements we add two rules: a blame expression can have any well formed type, while a type cast 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 with 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 4: **Syntax** from λ_C to λ_{CC}

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 5: **Static Semantics** from λ_C to λ_{CC}

4 Decidable Type Systems

In 1991 Freeman and Pfennning [14] introduced a decidable refinement type system for a subset of ML. In their system, the programmer defines refinement types for the vanilla data types; for example, the vanilla list data type can be refined to describe *nil* lists, or *singleton* lists, i.e., lists with exactly one element. These definitions are used to construct a finite datatype lattice of each ML type; a singleton list or a nill list is also an vanilla list, thus both refined lists are less than the unrefined one in the lattice. The datatype lattice is a representation of the subtype relationship that is used in the refinement type inference algorithm. Since each lattice is finite, the subtyping relation is decidable.

Later, at [39], they extended their language to support linear arithmetic constraints; thus they could encode a list with length some integer n and reason about safety of list indexing. In this system, subtyping reduces to predicate implication and they used a variant of Fourier's method [28] for constraint solving. Finally, they created DML(C)[40], an extension of ML with refinement types, that supports array bound check elimination, redundant pattern matching clause removal, tag check elimination and untagged representation of datatypes. Refinements in DML(C) are restricted to a finite and decidable constrain domain C, which renders constraint solving, and thus subtyping decidable.

DML(C) is a practical programming language, in the sense that programs can often be annotated with very little internal change and the resulting constraint simplification problems can be solved efficiently in practice. Its disadvantage is that annotation burden is high for the programmer, as often 10-20 percent of the code is typing annotations. In order to encourage programmers to use refinement specifications in their programs, Ou et al. [27], proposed a language design and type system that allows programmers to add semantic specifications to program fragments bit by bit. More specifically, for certain program components the type checker verifies statically the refinement type specifications. The rest components are written as in any ordinary simply-typed programming language. When control passes between different components, data flowing from simply-typed code into refinement-typed code is checked dynamically to ensure that the invariants hold.

Another system that combines static verification with dynamic checks is presented in Flanagan's Hybrid Type Checking [13]. Flanagan's type system uses syntactic sybtyping to create implication, as discusses in 2. Moreover, he assumes an algorithm that decides the validity of the implications. For each implication the algorithm runs for limited time: if it answers unsafe, the program is unsafe, but if it does not terminate, a cast is added to postpone the check at runtime. Thus, his systems checks implications statically, whenever possible and dynamically, only when necessary.

In Liquid Types [29], implication checking always terminates in limited time, as implications belong to a decidable subset of first order logic. This is achieved by restricting the refinement language according to a finite set of qualifiers. With this technique, liquid type system allows type inference, as a means of decreasing the annotation burden. We present Liquid Types in the rest of the section.

Many systems discussed so far, including DML(C), Hybrid Type System and Liquid Types, use *syntactic subtyping* for constraint generation and SMT solvers for constraint solving. *Satisfiability Modulo Theories*(SMT) solvers solve implications for (fragments of) first-order logic plus various standard theories such as equality, real and integer (linear) arithmetic, uninterpreted functions, bit vectors, and (extensional) arrays. Some of the leading systems include CVC3[3], Yices [11], and Z3 [10].

With the advent of SMT solvers, the combination of syntactic subtyping for constraint generation and an SMT solver for constraint solving has been used in various systems: Mandelbaum et all [21], extended the domain of predicates to describe the state and the effects of the programs verified. Suter et all [32] increase the power of reasoning to support user defined recursive functions. Finally, Unno et all [35], created a relative complete system for higher-order functional programs.

Apart from syntactic subtyping, SMT solvers can be used in other refinement decidable systems: Dminor [6] uses *semantic subtyping* where subtyping is totally decided by first order implication checking, while HALO [37] uses *denotational semantics* to prove specification checking.

4.1 Liquid Types

In Liquid Types[29], Rondon et all restrict the refinement language according to a finite set of qualifiers, and achieve 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 to be verified. The system then infers *liquid types*, which are refinement types where the refinement predicates are conjunctions of the logical qualifiers. Type checking and inference are decidable for three reasons. First, they use a conservative but decidable notion of subtyping, where subtyping reduces to implication checks in a decidable logic. Each implication holds if and only if it 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 refinement 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.

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. Let $\mathbb{Q}$ be the set of logical qualifiers $\{0 < \star, \star \leq v, v < \star, v = \star + 1\}$. 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 $\mathbb{Q}^\star$ 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 $\{0 < x, x \leq v, y \leq v, v = n + 1, v < n\}$. A liquid type over $\mathbb{Q}$ is a refinement type where the refinement predicates are conjunctions of qualifiers from $\mathbb{Q}^\star$.

Type Inference. Type inference is performed in three steps: (1) The vanilla type of each expression is refined with liquid variables which represent the unknown refinements. (2) Syntactic subtyping is used to create implication constraints between the unknown variables and the concrete refinements. (3) A theorem prover is used to find the strongest conjunction of qualifiers in $\mathbb{Q}$ that satisfies the subtyping constraints.

To illustrate this procedure, 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 | kn} -> {v:Int | kp}
```

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

(Step 2) We assume a descriptive type for minus:

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

and use it to construct type of `pred`'s result:

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

This type should be subtype of the template type of the body:

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

The above subtype reduces to the following constraint:

$$\texttt{v = n - 1} \Rightarrow \texttt{kp}$$

(Step 3)Since the program is "open", i.e., there are no calls to `pred`, we assign `kp` true, meaning that any integer argument 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 `kp`. By substituting the solution for `kp` into the template for `pred`, the algorithm infers

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

Type Checking. As one may notice the inferred type signature of `pred` does not constrain the type of the argument. This is correct, as `pred`'s definition does not constrain its argument. One could give `pred` a more precise type, say:

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

The system can verify that this type holds, following a procedure similar to the one for type inference:

The first step can be skipped, since there exists a concrete type for `pred`. The body of the function will be type-checked against the given signature. In the second step, as before, we construct the type of the body to be `{v:Int | n - 1}` and constraint this type to be subtype of `pred`'s result, or

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

This subtyping reduces to a trivial implication $\texttt{v = n - 1} \Rightarrow \texttt{v = n - 1}$ that can be proven in the third step.

Given the above type signature if `pred` is called with some positive integer value, say `2`, then in the call site the constraint $v = 2 \Rightarrow v > 0$ will be generated, that can be statically verified. On the other hand, if it is called with a non-positive value, say `0`, we will get the unsatisfied constraint $v = 0 \Rightarrow v > 0$, so the program will be unsafe.

4.2 Applications of Liquid Types

Liquid Types, as introduced in [29], used OCaml as target language and were used to verify array bound checking. One year later[19], they were extended with recursive and polymorphic refinements to enable static verification of complex data structures; among which list sortedness or Binary Tree ordering. Liquid Types were used to verify properties even in imperative languages. Low-level Liquid Types[30] is 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. Finally, Liquid Effects[20], is 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.

4.3 Formal Language

We extend the core calculus λ_C to λ_L, a calculus that supports liquid type checking.

The crucial difference between the previous systems, is that the refinement language can not contain arbitrary expressions, but is constrained to conjunctions of the logical qualifiers, which form a finite set, as shown in Figure 6.

Static typing uses syntactic subtyping, as defined in Section 2. In this setting, the subtyping relation is decidable because the refinement language, and thus the implications created, refer to a decidable logic. Finally, the `Valid` relation is evaluated using the z3[10] SMT solver.

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

Figure 6: **Syntax** from λ_C to λ_L

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** from λ_C to λ_L

5 Abstract Refinement Types

As we saw so far, refinement type systems fall in two categories. Expressive types systems, like the ones presented in section 3, are statically undecidable; while decidable systems, like those discussed in section 4 should restrict the refinement language in a subset on first order logic. In this section, we present abstract refinement types[36], a means to increase expressiveness of a refinement system, while preserving SMT-based decidability. The key insight is that we avail quantification over the refinements of data- and function-types, 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: inductive refinements for reasoning about higher-order traversal routines, compositional refinements for reasoning about function composition, index-dependent refinements for reasoning about key-value maps, and recursive refinements for reasoning about recursive data types.

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 some arbitrary property: If we apply `max` to two even numbers, can not verify that the result is also even. Thus, even though we have the information that both arguments are even 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 [24]

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

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 | even v}
m :: {v:Int | even v}
```

Then, the abstract refinement can be instantiated with a concrete predicate `even`, which will give `max` the type

```
max [even] ::
{v:Int | even v} -> {v:Int | even v} -> {v:Int | even v}
```

where the expression in brackets is the refinement instantiation. Since both `n` and `m` are even numbers we can apply them to the above function, and get an even result:

```
max [even] n m :: {v:Int |  even v}
```

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

5.2 Inductive Refinements

As a first application we present, how abstract refinements allow us to formalize induction within the type system.

Consider a `loop` function that takes as arguments 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
```

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 f n z
  where f i acc = acc + 1
```

The accumulator is initialized with `z` and at each `loop`'s iteration it is increased by `1`. So, at the `i`th iteration, the accumulator is equal to `z+i`. There will be `n` iterations, thus he final result will be `z+n`. This reasoning constitutes an inductive proof that characterizes `loop`'s behaviour, but it is unclear how to give `loop` a (first-order) refinement type that captures this behaviour. Hence, it has not been possible to verify that `incr` result actually adds its two arguments.

Typing loop. Abstract refinements allow us to solve this problem, while remaining within the boundaries of SMT-based decidability. We give `loop` the following type:

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

The trick is to qualify over the invariant `r` that `loop` establishes between the loop iteration and the accumulator. Then the type signature encodes induction on natural numbers: (1) `n` should be a natural number, thus a non-negative integer, (2) the base case `z` should satisfy the invariant at 0, (3) in the inductive step, `f` uses the the old accumulator to create the new one, thus if the accumulator satisfies the invariant on the iteration `i`, the new one, as constructed by `f` should satisfy the invariant at `i+1`. If all these hold, we conclude that the result satisfies the invariant at `n`. This scheme is not novel[5]; what is new is the encoding, via uninterpreted predicate symbols in a SMT-decidable refinement type system.

Using loop. We can use this expressive type of `loop` to verify inductive properties of user functions:

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

Where the expression in brackets denotes the instantiation of the abstract refinements. For purpose of illustration we make abstract refinement instantiation explicit,

but in [36] it can be automatically inferred via liquid typing. Moreover, in [36] we present how this inductive reasoning can be applied to reason about structural inductive properties in lists.

5.3 Function Composition

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

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

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

In a first order refinement system we can verify that the function's behaviour is captured by its type. However, consider an alternative definition that uses function composition `(.):: (b -> c)-> (a -> b)-> a -> c`.

```
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.

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

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

The trick is once again to quantify the type over refinements we care about. This time, we use two abstract refinements: the refinement `p` of the result of the first function `f` and the refinement `q` of the result of the second function `g`. For any argument `x`, we use an existential to bind the intermediate result to `z = g x`, so `z` satisfies `q` at `x`, and the result satisfies `p` at the intermediate result.

Using function composition. With this type for function composition, user functions get the concrete refinement of the final result to be the composition of the two refinements of the argument functions.

Back to `plusminus` example, with the appropriate refinement 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}
```

The result type asserts that there exists a value `z`, which is indeed the intermediate result, with the property `z = x - m`. With this, the final result is equal to `z + n`. If our logic supports equality , as SMT solvers do, we can verify that the final result is indeed equal to `(x - m)+ n`. In other words, we can verify the desired 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. A first step towards using vectors is to supply the appropriate types for vector operations, like set, get and empty. This usually means qualifying over the domain and the range of the vectors. Then, the programmer has to specify interesting vector properties, as we did for the Fibonacci memoization, or the null terminating string. Finally, the system can verify that user function, that transforms vectors preserve these properties. This procedure is applied in [36], where using appropriate types for vector operations, we reason about functions that transform Null Terminating Strings or efficiently compute a Fibonacci number.

5.5 Recursive Invariants

Finally, 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 [31]). 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 Implementation

We implemented abstract refinements in HSolve, a tool that uses liquid types as a base refinement system. HSolve takes an input a haskell source code and some specifications. If it can prove that the code satisfies the specifications, it returns *Safe* combined

with the code annotated with the inferred refinement types. Otherwise, it returns *UnSafe*, combined with the source location that could not be verified.

In the tool abstract refinements appear only on type specifications and are transparent for the haskell code. Refinement abstraction inserts a variable which is treated as an uninterprented functions symbol by the SMT solver. In refinement application the concrete refinement is inferred via liquid type variables; thus the user should not explicitly annotate the code with refinement instantiations.

We use the tool to verify some benchmarks, among which ghc official sorting algorithm on lists and two ghc libraries (Data.Map.Base and Data.Set.Splay) that reason about Binary Search Trees. The specification invariants are usually simple, as abstract refinements greatly simplify writing specifications for the majority of interface or public functions. Moreover, the verification procedure required only a few code modifications, which include reordering arguments or inserting ghost variables, to make explicit values over which various invariants depend.

5.7 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.

Since we extended our expressions the relevant typing rules should be added: The refinement abstraction expression is typed as an refinement abstraction type: the abstract refinement is treated as a variable and the checking proceeds in a straightforward way. In the refinement application, the abstract refinement π is replaced with a concrete one over the type τ. A formal definition of this substitution can be found in our paper.

Similarly, since we extended our types, the wellformdness and subtyping rules should be extended. In both cases, the abstract refinement is added in the environment and the check proceeds in a straightforward way.

We 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 run-type.

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

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

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

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

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

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

Figure 9: **Static Semantics** from λ_C to λ_A

6 Conclusion

In this report we presented various refinement 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 refinement type system to increase expressiveness without increasing complexity.

References

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

[2] L. Augustsson. Cayenne - a language with dependent types. In *ICFP*, 1998.

[3] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

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

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[6] Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. Semantic subtyping with an smt solver. In *ICFP*, 2010.

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

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

[9] R.L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[10] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[11] B. Dutertre and L. De Moura. Yices SMT solver. http://yices.csl.sri.com/.

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

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

[14] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.

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

[16] 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.

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

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

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

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

[21] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *ICFP*, pages 213–225, 2003.

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

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

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

[25] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[26] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, SE-412 96 Göteborg, Sweden, September 2007.

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

[28] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

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

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

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

[32] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.

[33] N. Swamy, J. Chen, C. Fournet, P-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278, 2011.

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

[35] Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL*, pages 75–86, 2013.

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

[37] Dimitrios Vytiniotis, Simon L. Peyton Jones, Koen Claessen, and Dan Rosén. Halo: haskell to logic through denotational semantics. In *POPL*, pages 431–442, 2013.

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

[39] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.

[40] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.