Refinement Types

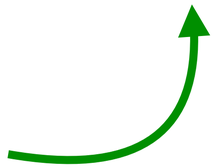
Niki Vazou
IMDEA Software Institute

Refinement Types

types refined with logical predicates

Existing Type: `(!!) :: [a] → Int → a`

Refinement Type: `(!!) :: xs:[a] → i:{Int | 0 ≤ i < len xs} → a`



Logical predicate
here encodes safe indexing

Light.hs

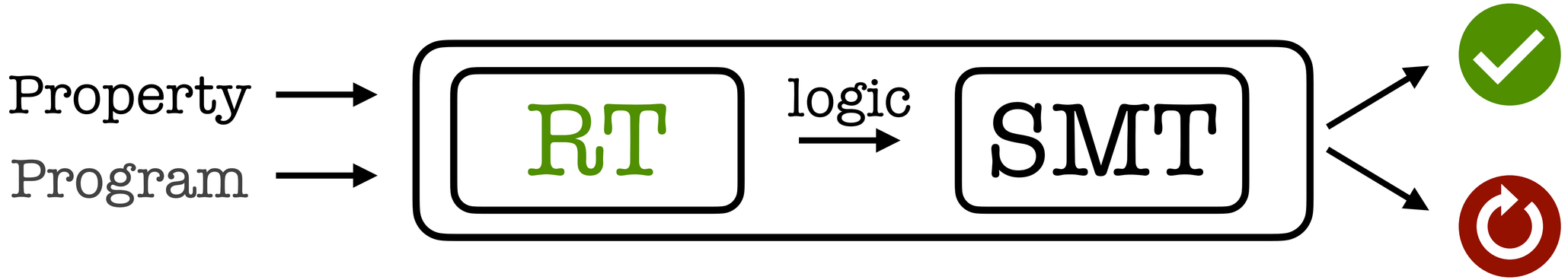
```haskell
module Light where

test :: [Int] -> Int -> Int
test xs i = xs !! i
```

PROBLEMS TERMINAL OUTPUT PORTS

TERMINAL ghcid

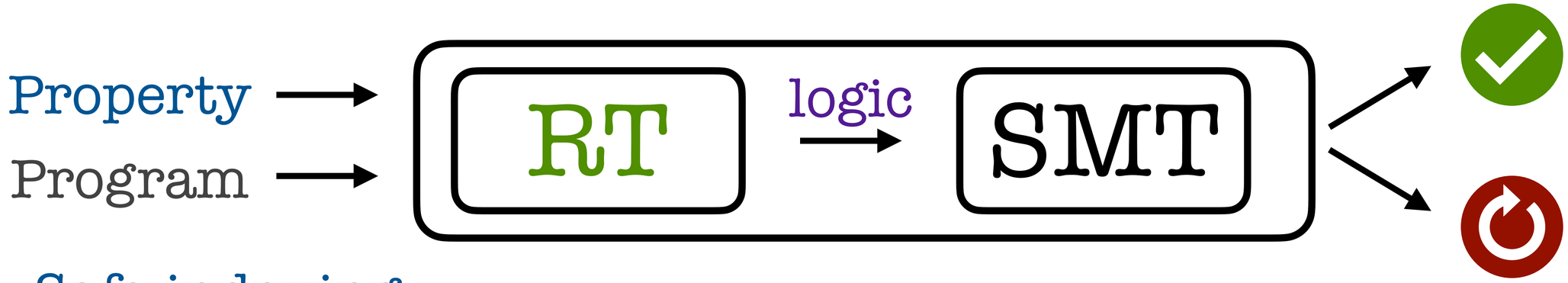
```
All good (4 modules, at 23:16:42)
```

Refinement Types



*SMT: A tool that automatically decides validity of logical formulas.

Refinement Types



Safe-indexing

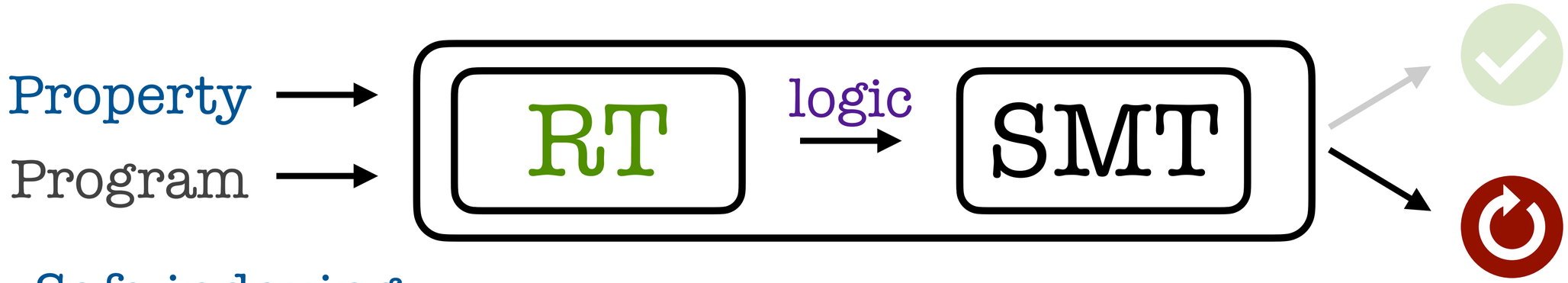
```
xs!!i
```

INFO

$\Rightarrow$

$0 \leq i < \text{len } xs$

Refinement Types



Safe-indexing

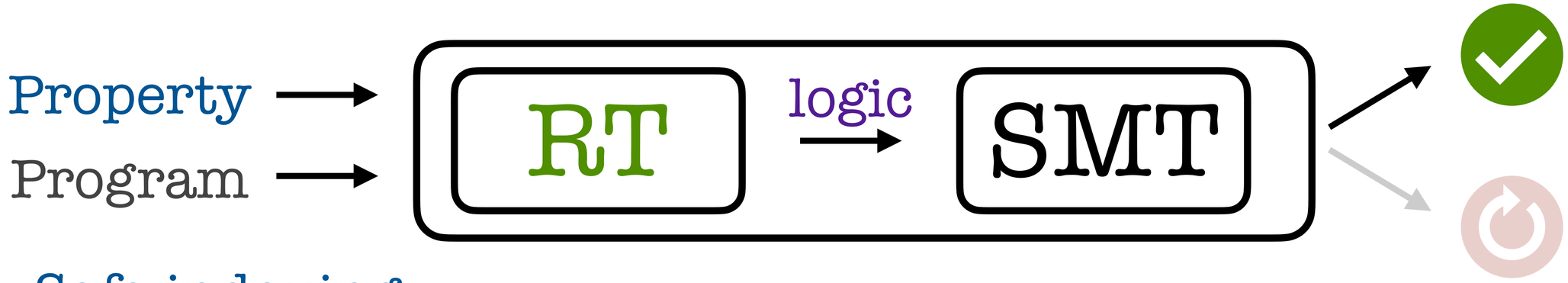
```
xs!!i
```

true

$\Rightarrow$

$0 \leq i < \text{len } xs$

Refinement Types

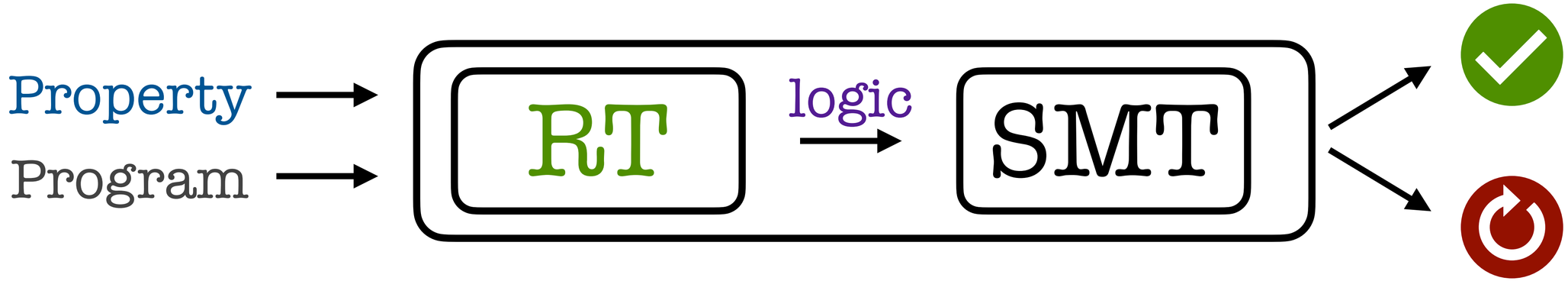


Safe-indexing

```
if 0 <= i && i < length xs
  then Just (xs!!i)
  else Nothing
```

$$0 \leq i < \text{len } xs$$
$$\Rightarrow$$
$$0 \leq i < \text{len } xs$$

Refinement Types

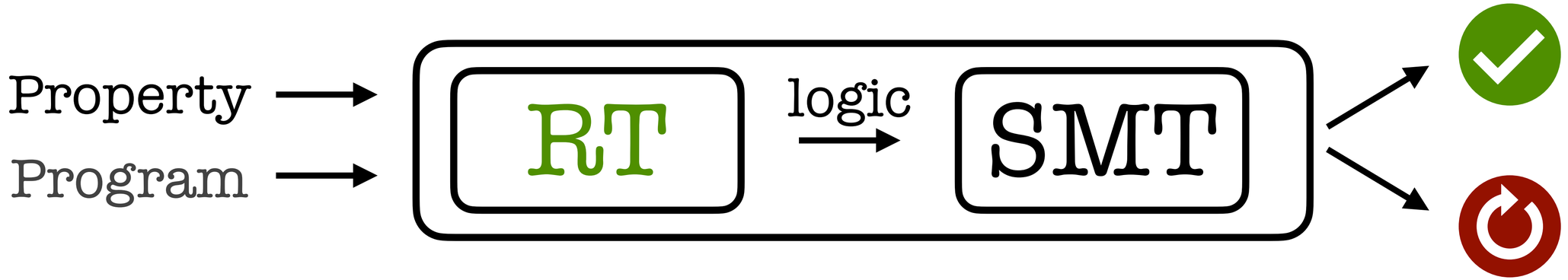



```
if 0 <= i && i < length xs
  then Just (xs!!i)
  else Nothing
```

Verification is

Case sensitive

Refinement Types



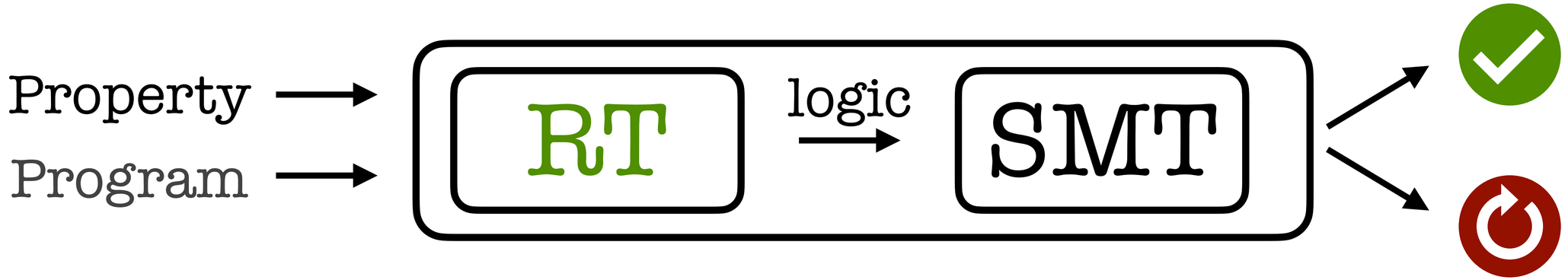

```
xs:[{v:Int | 0 /= v }]

42 `div` (xs!!i)
```

Verification is

Type-based

Case sensitive

Refinement Types



Verification is

Decidable

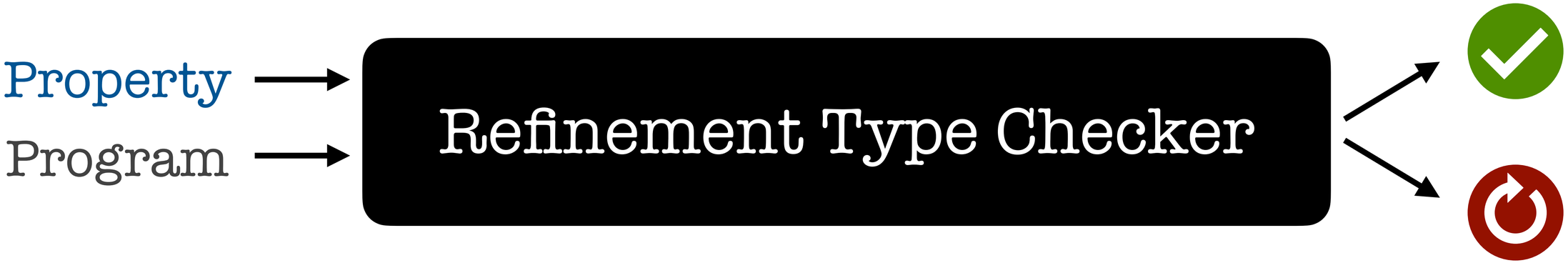
Type-based

Case sensitive

Logic is only
SMT decidable theories

*To avoid unpredictable SMT-verification (a.k.a. “the butterfly effect”)

Refinement Types are



Decidable

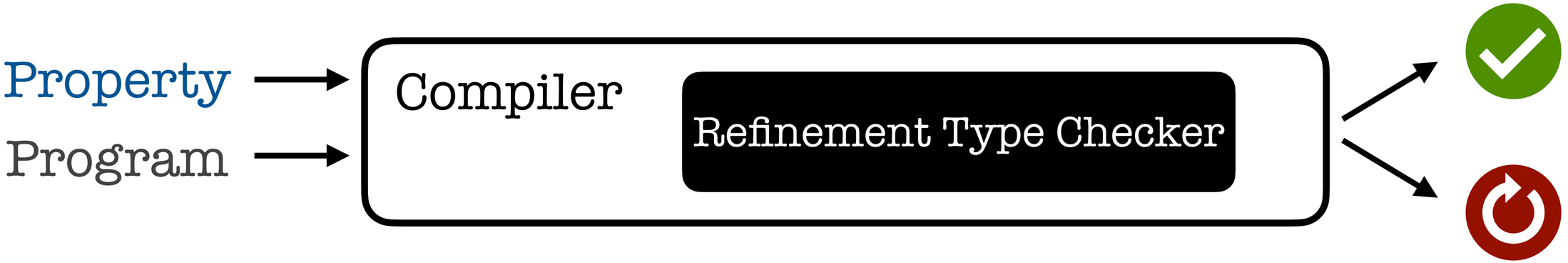
Type-based

Case sensitive

SMT-automated

Language Integrated

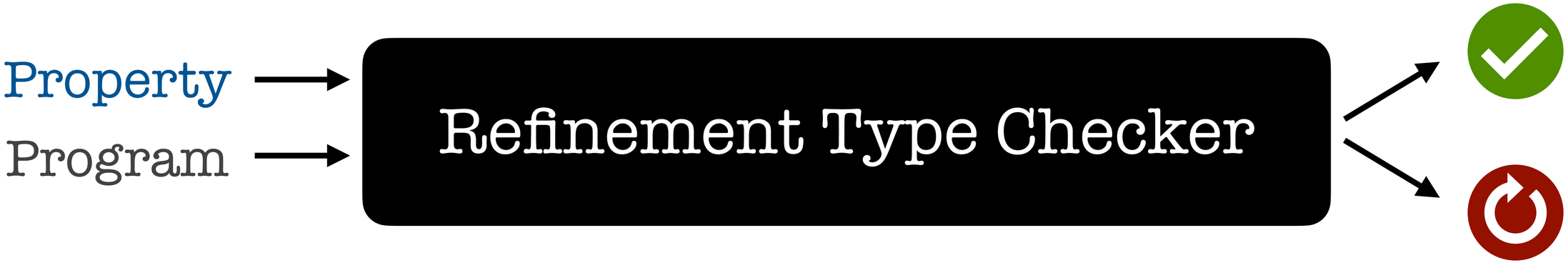
Language Integration



Refinement Type Checker is part of the compiler!

Editor fly-check integration;
Checking is part of the project build;
Cloud testing support; etc

Refinement Types are



Decidable

Type-based

Case sensitive

SMT-automated

Language Integrated

Refinement Types are designed to be practical!

The Design of Refinement Types

2014 Liquid Haskell

1991 Refinement Types

PLDI '91

Refinement Types for ML

Tim Freeman
tsf@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Frank Pfenning
fp@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We describe a refinement of ML's type system allowing the specification of recursively defined subtypes of user-defined datatypes. The resulting system of *refinement types* preserves desirable properties of ML such as decidability of type inference, while at the same time allowing more errors to be detected at compile-time. The type system combines abstract interpretation with ideas from the intersection type discipline, but remains closely tied to ML in that refinement types are given only to programs which are already well-typed in ML.

tended to the full Standard ML language.

To see the opportunity to improve ML's type system, consider the following function which returns the last cons cell in a list:

```
datatype α list = nil | cons of α * α list
fun lastcons (last as cons(hd,nil)) = last
  | lastcons (cons(hd,tl)) = lastcons tl
```

We know that this function will be undefined when called on an empty list, so we would like to obtain a type error at compile-time when `lastcons` is called with an argument of `nil`. Using refinement types this can be

PLDI '91

Refinement Types for ML

Tim Freeman
tsf@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Frank Pfenning
fp@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We describe a refinement of ML's type system allowing the specification of recursively defined subtypes of user-defined datatypes. The resulting system of *refinement types* preserves desirable properties of ML such as decidability of type inference, while at the same time allowing more errors to be detected at compile-time. The type system combines abstract interpretation with ideas from the intersection type discipline, but remains closely tied to ML in that refinement types are given only to programs which are already well-typed in ML.

1 Introduction

Standard ML [MTH90] is a practical programming language with higher-order functions, polymorphic types, and a well-developed module system. It is a statically typed language, which allows the compiler to detect many kinds of errors at compile time, thus leading to more reliable programs. Type inference is decidable and every well-typed expression has a principal type, which means that the programmer is free to omit type declarations (almost) anywhere in a program.

In this paper we summarize the design of a system of subtypes for ML which preserves the desirable properties listed above, while at the same time providing for specification and inference of significantly more precise type information. We call the resulting types *refinement types*, as they can be thought of as refining user-defined data types of ML. In particular, we do not extend the language of programs for ML (only the language of types) and, furthermore, we provide refined type information only for programs which are already well-typed in ML. In this preliminary report we only deal with an extension of Mini-ML [CDDK86], but we believe that the ideas described here can be further extended to the full Standard ML language.

To appear in ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation.

To see the opportunity to improve ML's type system, consider the following function which returns the last cons cell in a list:

```
datatype α list = nil | cons of α * α list
fun lastcons (last as cons(hd,nil)) = last
  | lastcons (cons(hd,tl)) = lastcons tl
```

We know that this function will be undefined when called on an empty list, so we would like to obtain a type error at compile-time when `lastcons` is called with an argument of `nil`. Using refinement types this can be achieved, thus preventing runtime errors which could be caught at compile-time. Similarly, we would like to be able to write code such as

```
case lastcons y of
     cons(x,nil) => print x
```

without getting a compiler warning. However, the ML type system does not distinguish singleton lists from lists in general, so when compiling this case statement ML compilers will issue a warning because it does not handle all possible forms of lists. Here, refinement types allow us to eliminate unreachable cases.

Attempting to take such refined type information into account at compile time can very quickly lead to undecidable problems. The key idea which allows us to circumvent undecidability is that subtype distinctions (such as singleton lists as a subtype of arbitrary lists) must be made explicitly by the programmer in the form of recursive type declarations. In the example above, we can declare the refinement type of singleton lists as

```
datatype α list = nil | cons of α * α list
rectype α singleton = cons (α, nil)
```

This `rectype` declaration instructs the type checker to distinguish singleton lists from other lists. The datatype constructor names `cons` and `nil` in the right-hand side of the `rectype` declaration stand for subtypes which one can think of as subsets. At any type α the type

1 Introduction

Standard ML [MTH90] is a practical programming language with higher-order functions, polymorphic types, and a well-developed module system. It is a statically typed language, which allows the compiler to detect many kinds of errors at compile time, thus leading to more reliable programs. Type inference is decidable and every well-typed expression has a principal type, which means that the programmer is free to omit type declarations (almost) anywhere in a program.

In this paper we summarize the design of a system of subtypes for ML which preserves the desirable properties listed above, while at the same time providing for specification and inference of significantly more precise type information. We call the resulting types *refinement types*, as they can be thought of as refining user-defined data types of ML. In particular, we do not extend the language of programs for ML (only the language of types) and, furthermore, we provide refined type information only for programs which are already well-typed in ML. In this preliminary report we only deal with an extension of Mini-ML [CDDK86], but we believe that the ideas described here can be further ex-

PLDI '91

Refinement Types for ML

Tim Freeman
tsf@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Frank Pfenning
fp@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We describe a refinement of ML's type system allowing the specification of recursively defined subtypes of user-defined datatypes. The resulting system of *refinement types* preserves desirable properties of ML such as decidability of type inference, while at the same time allowing more errors to be detected at compile-time. The type system combines abstract interpretation with ideas from the intersection type discipline, but remains closely tied to ML in that refinement types are given only to programs which are already well-typed in ML.

1 Introduction

Standard ML [MTH90] is a practical programming language with higher-order functions, polymorphic types, and a well-developed module system. It is a statically typed language, which allows the compiler to detect many kinds of errors at compile time, thus leading to more reliable programs. Type inference is decidable and every well-typed expression has a principal type, which means that the programmer is free to omit type declarations (almost) anywhere in a program.

In this paper we summarize the design of a system of subtypes for ML which preserves the desirable properties listed above, while at the same time providing for specification and inference of significantly more precise type information. We call the resulting types *refinement types*, as they can be thought of as refining user-defined data types of ML. In particular, we do not extend the language of programs for ML (only the language of types) and, furthermore, we provide refined type information only for programs which are already well-typed in ML. In this preliminary report we only deal with an extension of Mini-ML [CDDK86], but we believe that the ideas described here can be further extended to the full Standard ML language.

To appear in ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation.

To see the opportunity to improve ML's type system, consider the following function which returns the last cons cell in a list:

```
datatype α list = nil | cons of α * α list
fun lastcons (last as cons(hd,nil)) = last
  | lastcons (cons(hd,tl)) = lastcons tl
```

We know that this function will be undefined when called on an empty list, so we would like to obtain a type error at compile-time when `lastcons` is called with an argument of `nil`. Using refinement types this can be achieved, thus preventing runtime errors which could be caught at compile-time. Similarly, we would like to be able to write code such as

```
case lastcons y of
     cons(x,nil) => print x
```

without getting a compiler warning. However, the ML type system does not distinguish singleton lists from lists in general, so when compiling this case statement ML compilers will issue a warning because it does not handle all possible forms of lists. Here, refinement types allow us to eliminate unreachable cases.

Attempting to take such refined type information into account at compile time can very quickly lead to undecidable problems. The key idea which allows us to circumvent undecidability is that subtype distinctions (such as singleton lists as a subtype of arbitrary lists) must be made explicitly by the programmer in the form of recursive type declarations. In the example above, we can declare the refinement type of singleton lists as

```
datatype α list = nil | cons of α * α list
rectype α singleton = cons (α, nil)
```

This `rectype` declaration instructs the type checker to distinguish singleton lists from other lists. The datatype constructor names `cons` and `nil` in the right-hand side of the `rectype` declaration stand for subtypes which one can think of as subsets. At any type α the type

PLDI '91

Refinement Types for ML

Tim Freeman
tsf@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Frank Pfenning
fp@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We describe a refinement of ML's type system allowing the specification of recursively defined subtypes of user-defined datatypes. The resulting system of *refinement types* preserves desirable properties of ML such as decidability of type inference, while at the same time allowing more errors to be detected at compile-time. The type system combines abstract interpretation with

```
datatype α list = nil | cons of α * α list
rectype α singleton = cons (α, nil)
```

Standard ML [MTH90] is a practical programming language with higher-order functions, polymorphic types, and a well-developed module system. It is a statically typed language, which allows the compiler to detect many kinds of errors at compile time, thus leading to more reliable programs. Type inference is decidable and every well-typed expression has a principal type, which means that the programmer is free to omit type declarations (almost) anywhere in a program.

In this paper we summarize the design of a system of subtypes for ML which preserves the desirable properties listed above, while at the same time providing for specification and inference of significantly more precise type information. We call the resulting types *refinement types*, as they can be thought of as refining user-defined data types of ML. In particular, we do not extend the language of programs for ML (only the language of types) and, furthermore, we provide refined type information only for programs which are already well-typed in ML. In this preliminary report we only deal with an extension of Mini-ML [CDDK86], but we believe that the ideas described here can be further extended to the full Standard ML language.

To appear in ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation.

To see the opportunity to improve ML's type system, consider the following function which returns the last cons cell in a list:

```
datatype α list = nil | cons of α * α list
fun lastcons (last as cons(hd,nil)) = last
  | lastcons (cons(hd,tl)) = lastcons tl
```

```
case lastcons y of
     cons(x,nil) => print x
```

without getting a compiler warning. However, the ML type system does not distinguish singleton lists from lists in general, so when compiling this case statement ML compilers will issue a warning because it does not handle all possible forms of lists. Here, refinement types allow us to eliminate unreachable cases.

Attempting to take such refined type information into account at compile time can very quickly lead to undecidable problems. The key idea which allows us to circumvent undecidability is that subtype distinctions (such as singleton lists as a subtype of arbitrary lists) must be made explicitly by the programmer in the form of recursive type declarations. In the example above, we can declare the refinement type of singleton lists as

```
datatype α list = nil | cons of α * α list
rectype α singleton = cons (α, nil)
```

This `rectype` declaration instructs the type checker to distinguish singleton lists from other lists. The datatype constructor names `cons` and `nil` in the right-hand side of the `rectype` declaration stand for subtypes which one can think of as subsets. At any type α the type

Refinement Types are data type definitions

The Design of Refinement Types

Refinement Types are data type definitions

2014 Liquid Haskell

1991 Refinement Types

PLDI '91

Refinement Types for ML

Tim Freeman
tsf@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Frank Pfenning
fp@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We describe a refinement of ML's type system allowing the specification of recursively defined subtypes of user-defined datatypes. The resulting system of *refinement types* preserves desirable properties of ML such as decidability of type inference, while at the same time allowing more errors to be detected at compile-time. The type system combines abstract interpretation with ideas from the intersection type discipline, but remains closely tied to ML in that refinement types are given only to programs which are already well-typed in ML.

tended to the full Standard ML language.

To see the opportunity to improve ML's type system, consider the following function which returns the last cons cell in a list:

```
datatype α list = nil | cons of α * α list
fun lastcons (last as cons(hd,nil)) = last
  | lastcons (cons(hd,tl)) = lastcons tl
```

We know that this function will be undefined when called on an empty list, so we would like to obtain a type error at compile-time when `lastcons` is called with an argument of `nil`. Using refinement types this can be

The Design of Refinement Types

Refinement Types are data type definitions

2017 Extensible Datasort Refinements

2014 Liquid Haskell

1991 Refinement Types

PLDI '91

Refinement Types for ML

Tim Freeman
tsf@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Frank Pfenning
fp@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We describe a refinement of ML's type system allowing the specification of recursively defined subtypes of user-defined datatypes. The resulting system of *refinement types* preserves desirable properties of ML such as decidability of type inference, while at the same time allowing more errors to be detected at compile-time. The type system combines abstract interpretation with ideas from the intersection type discipline, but remains closely tied to ML in that refinement types are given only to programs which are already well-typed in ML.

tended to the full Standard ML language.

To see the opportunity to improve ML's type system, consider the following function which returns the last cons cell in a list:

```
datatype α list = nil | cons of α * α list
fun lastcons (last as cons(hd,nil)) = last
  | lastcons (cons(hd,tl)) = lastcons tl
```

We know that this function will be undefined when called on an empty list, so we would like to obtain a type error at compile-time when `lastcons` is called with an argument of `nil`. Using refinement types this can be

ESOP'17

Extensible Datasort Refinements

Jana Dunfield(✉)

University of British Columbia, Vancouver, Canada
jd169@queensu.ca

Abstract. Refinement types turn typechecking into lightweight verification. The classic form of refinement type is the datasort refinement, in which datasorts identify subclasses of inductive datatypes.

Existing type systems for datasort refinements require that all the refinements of a type be specified when the type is declared; multiple refinements of the same type can be obtained only by duplicating type definitions, and consequently, duplicating code.

We enrich the traditional notion of a signature, which describes the inhabitants of datasorts, to allow *re-refinement* via signature extension, without duplicating definitions. Since arbitrary updates to a signature can invalidate the inversion principles used to check case expressions, we develop a definition of signature well-formedness that ensures that extensions maintain existing inversion principles. This definition allows different parts of a program to extend the same signature in different ways, without conflicting with each other. Each part can be type-checked independently, allowing separate compilation.

The Design of Refinement Types

2017 Extensible Datasort Refinements

2014 Liquid Haskell

1991 Refinement Types

PLDI '91

Refinement Types for ML

Tim Freeman
tsf@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Frank Pfenning
fp@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We describe a refinement of ML's type system allowing the specification of recursively defined subtypes of user-defined datatypes. The resulting system of *refinement types* preserves desirable properties of ML such as decidability of type inference, while at the same time allowing more errors to be detected at compile-time. The type system combines abstract interpretation with ideas from the intersection type discipline, but remains closely tied to ML in that refinement types are given only to programs which are already well-typed in ML.

tended to the full Standard ML language.

To see the opportunity to improve ML's type system, consider the following function which returns the last cons cell in a list:

```
datatype α list = nil | cons of α * α list
fun lastcons (last as cons(hd,nil)) = last
  | lastcons (cons(hd,tl)) = lastcons tl
```

We know that this function will be undefined when called on an empty list, so we would like to obtain a type error at compile-time when `lastcons` is called with an argument of `nil`. Using refinement types this can be

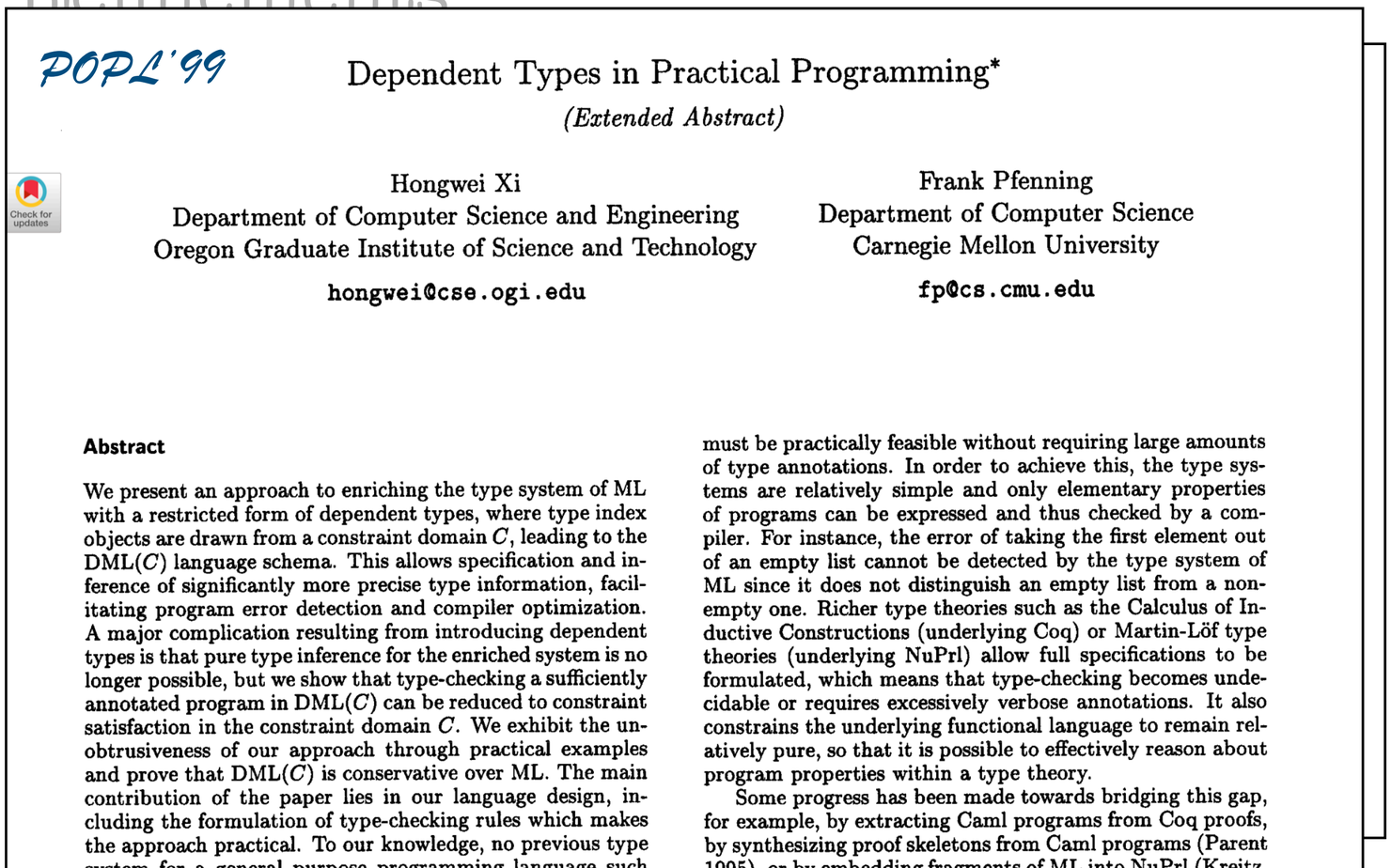
The Design of Refinement Types

2017 Extensible Datasort Refinements

2014 Liquid Haskell

1999 Dependent ML

1991 Refinement Types



POPL'99

Dependent Types in Practical Programming*

(Extended Abstract)

Hongwei Xi
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
hongwei@cse.ogi.edu

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu

Abstract

We present an approach to enriching the type system of ML with a restricted form of dependent types, where type index objects are drawn from a constraint domain C, leading to the DML(C) language schema. This allows specification and inference of significantly more precise type information, facilitating program error detection and compiler optimization. A major complication resulting from introducing dependent types is that pure type inference for the enriched system is no longer possible, but we show that type-checking a sufficiently annotated program in DML(C) can be reduced to constraint satisfaction in the constraint domain C. We exhibit the unobtrusiveness of our approach through practical examples and prove that DML(C) is conservative over ML. The main contribution of the paper lies in our language design, including the formulation of type-checking rules which makes the approach practical. To our knowledge, no previous type system for a general purpose programming language such must be practically feasible without requiring large amounts of type annotations. In order to achieve this, the type systems are relatively simple and only elementary properties of programs can be expressed and thus checked by a compiler. For instance, the error of taking the first element out of an empty list cannot be detected by the type system of ML since it does not distinguish an empty list from a non-empty one. Richer type theories such as the Calculus of Inductive Constructions (underlying Coq) or Martin-Löf type theories (underlying NuPrl) allow full specifications to be formulated, which means that type-checking becomes undecidable or requires excessively verbose annotations. It also constrains the underlying functional language to remain relatively pure, so that it is possible to effectively reason about program properties within a type theory.

Some progress has been made towards bridging this gap, for example, by extracting Caml programs from Coq proofs, by synthesizing proof skeletons from Caml programs (Parent 1995), or by embedding fragments of ML into NuPrl (Kreitz

POPL'99

Dependent Types in Practical Programming*

(Extended Abstract)

Hongwei Xi
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
hongwei@cse.ogi.edu

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu

Abstract

We present an approach to enriching the type system of ML with a restricted form of dependent types, where type index objects are drawn from a constraint domain C, leading to the DML(C) language schema. This allows specification and inference of significantly more precise type information, facilitating program error detection and compiler optimization. A major complication resulting from introducing dependent types is that pure type inference for the enriched system is no longer possible, but we show that type-checking a sufficiently annotated program in DML(C) can be reduced to constraint satisfaction in the constraint domain C. We exhibit the unobtrusiveness of our approach through practical examples and prove that DML(C) is conservative over ML. The main contribution of the paper lies in our language design, including the formulation of type-checking rules which makes the approach practical. To our knowledge, no previous type system for a general purpose programming language such as ML has combined dependent types with features including datatype declarations, higher-order functions, general recursions, let-polymorphism, mutable references, and exceptions. In addition, we have finished a prototype implementation of DML(C) for an integer constraint domain C, where constraints are linear inequalities (Xi and Pfenning must be practically feasible without requiring large amounts of type annotations. In order to achieve this, the type systems are relatively simple and only elementary properties of programs can be expressed and thus checked by a compiler. For instance, the error of taking the first element out of an empty list cannot be detected by the type system of ML since it does not distinguish an empty list from a non-empty one. Richer type theories such as the Calculus of Inductive Constructions (underlying Coq) or Martin-Löf type theories (underlying NuPrl) allow full specifications to be formulated, which means that type-checking becomes undecidable or requires excessively verbose annotations. It also constrains the underlying functional language to remain relatively pure, so that it is possible to effectively reason about program properties within a type theory.

Some progress has been made towards bridging this gap, for example, by extracting Caml programs from Coq proofs, by synthesizing proof skeletons from Caml programs (Parent 1995), or by embedding fragments of ML into NuPrl (Kreitz, Hayden, and Hickey 1998). In this paper, we address the issue of designing a type system for practical programming in which a restricted form of dependent types is available, allowing more program invariants to be captured by types. We conservatively refine the type system of ML by allowing some dependencies, without destroying desirable properties

POPL'99

Dependent Types in Practical Programming*

(Extended Abstract)

Hongwei Xi
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
hongwei@cse.ogi.edu

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu

```
datatype 'a list = nil | cons of 'a * 'a list
typeref 'a list of nat with (* indexing the datatype 'a list with nat *)
    nil <| 'a list(0)
  | cons <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)
```

A major complication resulting from introducing dependent types is that pure type inference for the enriched system is no longer possible, but we show that type-checking a sufficiently annotated program in DML(C) can be reduced to constraint satisfaction in the constraint domain C. We exhibit the unobtrusiveness of our approach through practical examples and prove that DML(C) is conservative over ML. The main contribution of the paper lies in our language design, including the formulation of type-checking rules which makes the approach practical. To our knowledge, no previous type system for a general purpose programming language such as ML has combined dependent types with features including datatype declarations, higher-order functions, general recursions, let-polymorphism, mutable references, and exceptions. In addition, we have finished a prototype implementation of DML(C) for an integer constraint domain C, where constraints are linear inequalities (Xi and Pfenning

ductive constructions (underlying Coq) or Martin-Löf type theories (underlying NuPrl) allow full specifications to be formulated, which means that type-checking becomes undecidable or requires excessively verbose annotations. It also constrains the underlying functional language to remain relatively pure, so that it is possible to effectively reason about program properties within a type theory.

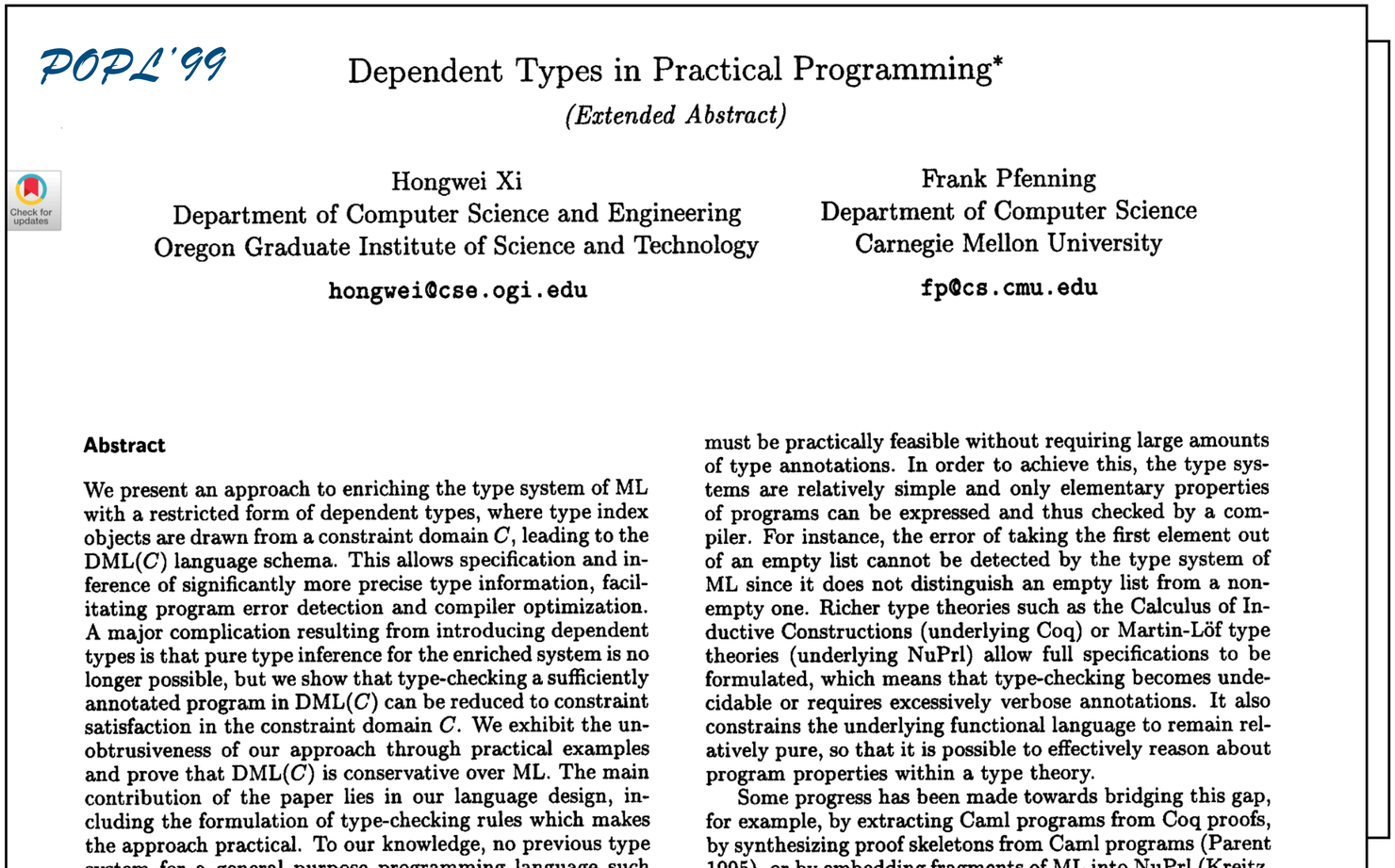
Some progress has been made towards bridging this gap, for example, by extracting Caml programs from Coq proofs, by synthesizing proof skeletons from Caml programs (Parent 1995), or by embedding fragments of ML into NuPrl (Kreitz, Hayden, and Hickey 1998). In this paper, we address the issue of designing a type system for practical programming in which a restricted form of dependent types is available, allowing more program invariants to be captured by types. We conservatively refine the type system of ML by allowing some dependencies, without destroying desirable properties

The Design of Refinement Types

2014 Liquid Haskell

1999 Dependent ML

1991 Refinement Types



POPL'99

Dependent Types in Practical Programming*

(Extended Abstract)

Hongwei Xi
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
hongwei@cse.ogi.edu

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu

Abstract

We present an approach to enriching the type system of ML with a restricted form of dependent types, where type index objects are drawn from a constraint domain C, leading to the DML(C) language schema. This allows specification and inference of significantly more precise type information, facilitating program error detection and compiler optimization. A major complication resulting from introducing dependent types is that pure type inference for the enriched system is no longer possible, but we show that type-checking a sufficiently annotated program in DML(C) can be reduced to constraint satisfaction in the constraint domain C. We exhibit the unobtrusiveness of our approach through practical examples and prove that DML(C) is conservative over ML. The main contribution of the paper lies in our language design, including the formulation of type-checking rules which makes the approach practical. To our knowledge, no previous type system for a general purpose programming language such must be practically feasible without requiring large amounts of type annotations. In order to achieve this, the type systems are relatively simple and only elementary properties of programs can be expressed and thus checked by a compiler. For instance, the error of taking the first element out of an empty list cannot be detected by the type system of ML since it does not distinguish an empty list from a non-empty one. Richer type theories such as the Calculus of Inductive Constructions (underlying Coq) or Martin-Löf type theories (underlying NuPrl) allow full specifications to be formulated, which means that type-checking becomes undecidable or requires excessively verbose annotations. It also constrains the underlying functional language to remain relatively pure, so that it is possible to effectively reason about program properties within a type theory.

Some progress has been made towards bridging this gap, for example, by extracting Caml programs from Coq proofs, by synthesizing proof skeletons from Caml programs (Parent

The Design of Refinement Types

2014 Liquid Haskell

2006 ATS

1999 Dependent ML

1991 Refinement Types

Applied Type System: An Approach to Practical Programming with Theorem-Proving

Hongwei Xi*
Boston University, Boston, MA 02215, USA
(*e-mail:* hwxi@cs.bu.edu)

Abstract

The framework Pure Type System (**PTS**) offers a simple and general approach to designing and formalizing type systems. However, in the presence of dependent types, there often exist certain acute problems that make it difficult for **PTS** to directly accommodate many common realistic programming features such as general recursion, recursive types, effects (e.g., exceptions, references, input/output), etc. In this paper, Applied Type System (**ATS**) is presented as a framework for designing and formalizing type systems in support of practical programming with advanced types (including dependent types). In particular, it is demonstrated that **ATS** can readily accommodate a

The Design of Refinement Types

2014 Liquid Haskell

2006 Hybrid Types

1999 Dependent ML

1991 Refinement Types

POPL'06

Hybrid Type Checking

Cormac Flanagan
Department of Computer Science
University of California, Santa Cruz
cormac@cs.ucsc.edu

Abstract

Traditional static type systems are very effective for verifying basic interface specifications, but are somewhat limited in the kinds specifications they support. Dynamically-checked contracts can enforce more precise specifications, but these are not checked until run time, resulting in incomplete detection of defects.
Hybrid type checking is a synthesis of these two approaches that enforces precise interface specifications, via static analysis where possible, but also via dynamic checks where necessary. This paper explores the key ideas and implications of hybrid type checking, in the context of the simply-typed λ-calculus with arbitrary refinements of base types.

Categories and Subject Descriptors D.3.1 [*Programming Languages: Formal Definitions and Theory*]: specification and verification

In contrast, *dynamic contract checking* [30, 14, 26, 19, 24, 27, 36, 25] provides a simple method for checking more expressive specifications. Dynamic checking can easily support precise specifications, such as:

- Subrange types, *e.g.*, the function `printDigit` requires an integer in the range [0,9].
- Aliasing restrictions, *e.g.*, `swap` requires that its arguments are distinct reference cells.
- Ordering restrictions, *e.g.*, `binarySearch` requires that its argument is a sorted array.
- Size specifications, *e.g.*, the function `serializeMatrix` takes as input a matrix of size n by m, and returns a one-dimensional array of size $n \times m$.
- Arbitrary predicates: an interpreter (or code generator) for a

an argument of 911. Using refinement types this can be

Syntax

$$S, T ::= x : S \to T$$

$$\mid \quad \{x : B \mid t\}$$

boolean term

2. The Language λ^H

This section introduces a variant of the simply-typed λ-calculus extended with casts and with precise (and hence undecidable) refinement types. We refer to this language as λ^H.

2.1 Syntax of λ^H

The syntax of λ^H is summarized in Figure 3. Terms include variables, constants, functions, applications, and casts . The cast $\langle S \triangleright T \rangle\ t$ dynamically checks that the result of t is of type T (in a manner similar to coercions [38], contracts [13, 14], and to type casts in languages such as Java [20]). For technical reasons, the cast also includes that static type S of the term t. Type casts are annotated with associated labels $l \in \mathit{Label}$, which are used to map run-time errors back to locations in the source program. Applications are also annotated with labels, for similar reasons. For clarity, we omit these labels when they are irrelevant.

The λ^H type language includes dependent function types [10], for which we use the syntax $x : S \to T$ of Cayenne [4] (in preference to the equivalent syntax $\Pi x : S.\ T$). Here, S is the domain type of the function and the formal parameter x may occur in the range type T. We omit x if it does not occur free in T, yielding the standard function type syntax $S \to T$.

We use B to range over base types, which includes at least `Bool` and `Int`. As in many languages, these base types are fairly coarse and cannot, for example, denote integer subranges. To overcome this limitation, we introduce *base refinement types* of the form

$$\{x : B \mid t\}$$

Here, the variable x (of type B) can occur within the boolean term or *predicate* t. This refinement type denotes the set of constants c of type B that satisfy this predicate, *i.e.*, for which the term $t[x := c]$ evaluates to true. Thus, $\{x : B \mid t\}$ denotes a subtype of B, and we use a base type B as an abbreviation for the trivial refinement type $\{x : B \mid \texttt{true}\}$.

Hybrid Type Checking. Flanagan. PLDI'06

Denotations*

$$[\![x : S \to T]\!] \doteq \{e \mid \ldots, \forall s \in [\![S]\!], e\ s \in [\![T[x/s]]\!]\}$$

$$[\![\{x : B \mid t\}]\!] \doteq \{e \mid \ldots, e \hookrightarrow^* v \Rightarrow t[v/x] \hookrightarrow^* \mathsf{true}\}$$

"If e goes to a value v, then the predicate t should be true."

$$42 \in [\![\{x : \mathsf{Int} \mid 0 \leq x\}]\!]$$

* or logical relations

Hybrid Type Checking. K. Knowles and C. Flanagan. TOPLAS'09

Implicit subtyping

$$\dfrac{\dfrac{\dfrac{\forall x : \mathsf{Int} \,.\, x = 42 \hookrightarrow^* \mathsf{true} \Rightarrow 0 \leq x \hookrightarrow^* \mathsf{true}}{x : \mathsf{Int} \vdash x = 42 \Rightarrow 0 \leq x} \quad \dfrac{}{\varnothing \vdash 42 : \{x : \mathsf{Int} \mid x = 42\}} \quad \dfrac{}{\varnothing \vdash \{x : \mathsf{Int} \mid x = 42\} \leq \{x : \mathsf{Int} \mid 0 \leq x\}}}{\varnothing \vdash 42 : \{x : \mathsf{Int} \mid 0 \leq x\}}}{42 \in [\![\{x : \mathsf{Int} \mid 0 \leq x\}]\!]}$$

Hybrid Type Checking. K. Knowles and C. Flanagan. TOPLAS'09

Implicit subtyping

$$\frac{\forall \theta \in [\![\Gamma, x : \mathsf{b}]\!] \,.\, \theta \cdot t_1 \hookrightarrow^* \mathsf{true} \Rightarrow \theta \cdot t_2 \hookrightarrow^* \mathsf{true}}{\Gamma, x : \mathsf{b} \vdash t_1 \Rightarrow t_2}$$

$$\frac{\Gamma, x : \mathsf{b} \vdash t_1 \Rightarrow t_2}{\Gamma \vdash \{x : \mathsf{b} \mid t_1\} \leq \{x : \mathsf{b} \mid t_2\}}$$

Subtyping is undecidable!

Hybrid Type Checking. K. Knowles and C. Flanagan. TOPLAS'09

The Design of Refinement Types

2014 Liquid Haskell

2008 Liquid Types

2006 Hybrid Types

1991 Refinement Types

PLDI '08

Liquid Types *

Patrick M. Rondon Ming W. Kawaguchi Ranjit Jhala

University of California, San Diego

{prondon,mwookawa,jhala}@cs.ucsd.edu

Abstract

We present *Logically Qualified Data Types*, abbreviated to *Liquid Types*, a system that combines *Hindley-Milner* type inference with *Predicate Abstraction* to automatically infer dependent types precise enough to prove a variety of safety properties. Liquid types allow programmers to reap many of the benefits of dependent types, namely static verification of critical properties and the elimination of expensive run-time checks, without the heavy price of manual annotation. We have implemented liquid type inference in DSOLVE, which takes as input an OCAML program and a set of logical qualifiers and infers dependent types for the expressions in the OCAML program. To demonstrate the utility of our approach, we describe experiments using DSOLVE to statically verify the safety of array accesses on a set of OCAML benchmarks that were previously annotated with dependent types as part of the DML project. We show that when used in conjunction with a fixed set of array bounds checking qualifiers, DSOLVE reduces the amount of manual annotation required for proving safety from 31% of program text to under 1%.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings*

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 within a certain range, say between 1 and 99. Thus, the type system is unable to statically ensure the safety of critical operations, such as a division by `i`, or the accessing of an array `a` of size 100 at an index `i`. Instead, the language can only provide a weaker dynamic safety guarantee at the additional cost of high performance overhead.

In an exciting development, several authors have proposed the use of *dependent types* [20] as a mechanism for enhancing the expressivity of type systems [14, 27, 2, 22, 10]. Such a system can express the fact

$$\texttt{i} :: \{\nu{:}\texttt{int} \mid 1 \leq \nu \wedge \nu \leq 99\}$$

which is the usual type `int` together with a *refinement* stating that the run-time value of `i` is an always an integer between 1 and 99. Pfenning and Xi devised DML, a practical way to integrate such types into ML, and demonstrated that they could be used to recover

Implicit subtyping

$$\frac{\text{SMT-Valid}\ (\llbracket \Gamma, x : \mathsf{b} \rrbracket \wedge \llbracket t_1 \rrbracket \Rightarrow \llbracket t_2 \rrbracket)}{\Gamma, x : \mathsf{b} \vdash t_1 \Rightarrow t_2}$$

$$\frac{\Gamma, x : \mathsf{b} \vdash t_1 \Rightarrow t_2}{\Gamma \vdash \{x : \mathsf{b} \mid t_1\} \leq \{x : \mathsf{b} \mid t_2\}}$$

$\llbracket \cdot \rrbracket$ :: expression $\rightarrow$ decidable logic

Liquid Types. P. Rondon, M. Kawaguchi, and R. Jhala. PLDI'08

Implicit subtyping

$$\frac{\text{SMT-Valid}\ (\llbracket \Gamma, x : \mathsf{b} \rrbracket \wedge \llbracket t_1 \rrbracket \Rightarrow \llbracket t_2 \rrbracket)}{\Gamma, x : \mathsf{b} \vdash t_1 \Rightarrow t_2}$$

$$\frac{\Gamma, x : \mathsf{b} \vdash t_1 \Rightarrow t_2}{\Gamma \vdash \{x : \mathsf{b} \mid t_1\} \leq \{x : \mathsf{b} \mid t_2\}}$$

If user provided specs are from a decidable logic, then type checking and inference is decidable!

Liquid Types. P. Rondon, M. Kawaguchi, and R. Jhala. PLDI'08

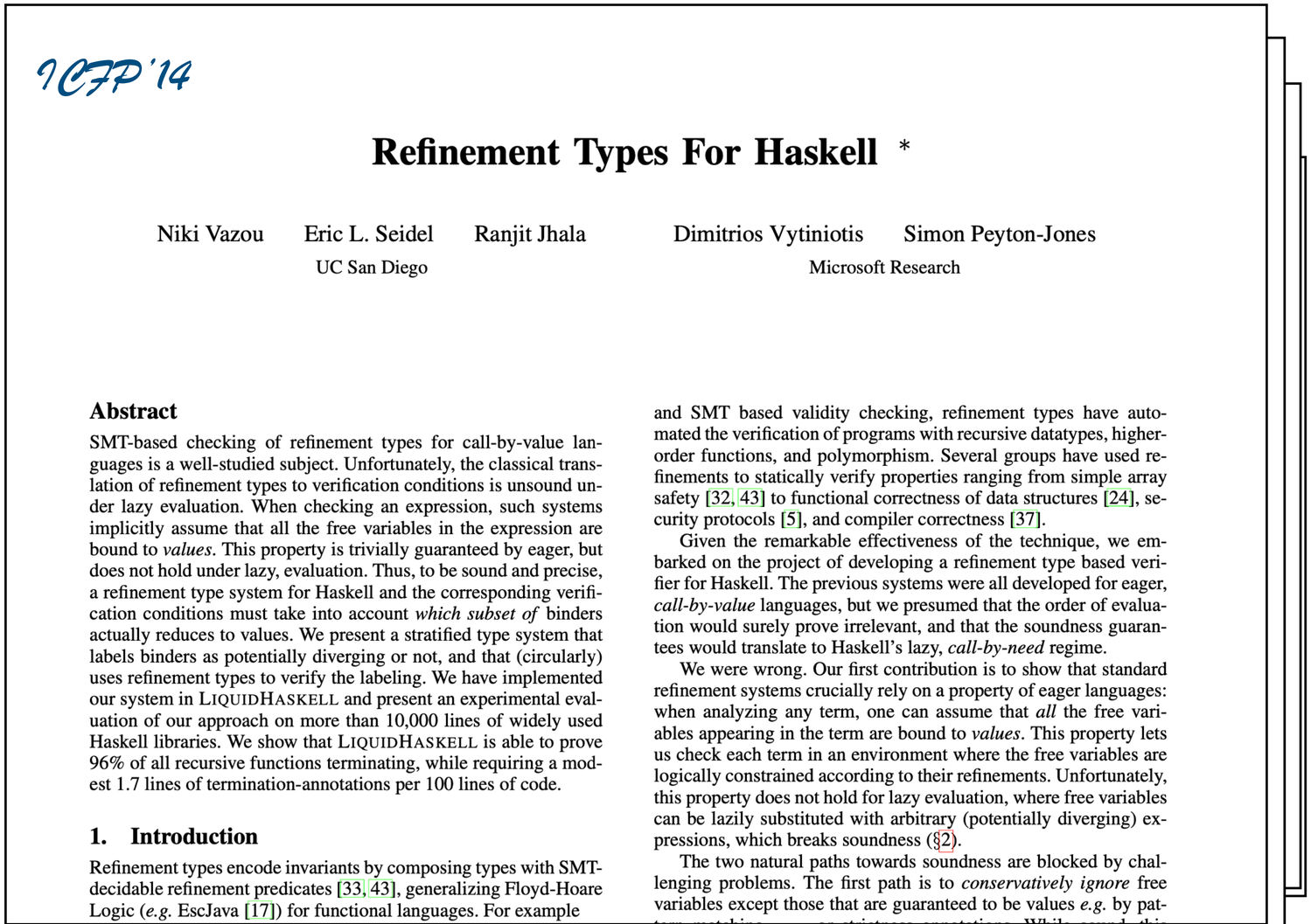
The Design of Refinement Types

2014 Liquid Haskell

2008 Liquid Types

2006 Hybrid Types

1991 Refinement Types



ICFP '14

Refinement Types For Haskell *

Niki Vazou Eric L. Seidel Ranjit Jhala
UC San Diego

Dimitrios Vytiniotis Simon Peyton-Jones
Microsoft Research

Abstract

SMT-based checking of refinement types for call-by-value languages is a well-studied subject. Unfortunately, the classical translation of refinement types to verification conditions is unsound under lazy evaluation. When checking an expression, such systems implicitly assume that all the free variables in the expression are bound to *values*. This property is trivially guaranteed by eager, but does not hold under lazy, evaluation. Thus, to be sound and precise, a refinement type system for Haskell and the corresponding verification conditions must take into account *which subset of* binders actually reduces to values. We present a stratified type system that labels binders as potentially diverging or not, and that (circularly) uses refinement types to verify the labeling. We have implemented our system in LIQUIDHASKELL and present an experimental evaluation of our approach on more than 10,000 lines of widely used Haskell libraries. We show that LIQUIDHASKELL is able to prove 96% of all recursive functions terminating, while requiring a modest 1.7 lines of termination-annotations per 100 lines of code.

1. Introduction

Refinement types encode invariants by composing types with SMT-decidable refinement predicates [33, 43], generalizing Floyd-Hoare Logic (*e.g.* EscJava [17]) for functional languages. For example

and SMT based validity checking, refinement types have automated the verification of programs with recursive datatypes, higher-order functions, and polymorphism. Several groups have used refinements to statically verify properties ranging from simple array safety [32, 43] to functional correctness of data structures [24], security protocols [5], and compiler correctness [37].

Given the remarkable effectiveness of the technique, we embarked on the project of developing a refinement type based verifier for Haskell. The previous systems were all developed for eager, *call-by-value* languages, but we presumed that the order of evaluation would surely prove irrelevant, and that the soundness guarantees would translate to Haskell's lazy, *call-by-need* regime.

We were wrong. Our first contribution is to show that standard refinement systems crucially rely on a property of eager languages: when analyzing any term, one can assume that *all* the free variables appearing in the term are bound to *values*. This property lets us check each term in an environment where the free variables are logically constrained according to their refinements. Unfortunately, this property does not hold for lazy evaluation, where free variables can be lazily substituted with arbitrary (potentially diverging) expressions, which breaks soundness (§2).

The two natural paths towards soundness are blocked by challenging problems. The first path is to *conservatively ignore* free variables except those that are guaranteed to be values *e.g.* by pat-

ICFP'14

Refinement Types For Haskell *

Niki Vazou Eric L. Seidel Ranjit Jhala
UC San Diego

Dimitrios Vytiniotis Simon Peyton-Jones
Microsoft Research

Abstract

SMT-based checking of refinement types for call-by-value languages is a well-studied subject. Unfortunately, the classical translation of refinement types to verification conditions is unsound under lazy evaluation. When checking an expression, such systems implicitly assume that all the free variables in the expression are bound to *values*. This property is trivially guaranteed by eager, but does not hold under lazy, evaluation. Thus, to be sound and precise, a refinement type system for Haskell and the corresponding verification conditions must take into account *which subset of* binders actually reduces to values. We present a stratified type system that labels binders as potentially diverging or not, and that (circularly) uses refinement types to verify the labeling. We have implemented our system in LIQUIDHASKELL and present an experimental evaluation of our approach on more than 10,000 lines of widely used Haskell libraries. We show that LIQUIDHASKELL is able to prove 96% of all recursive functions terminating, while requiring a modest 1.7 lines of termination-annotations per 100 lines of code.

1. Introduction

Refinement types encode invariants by composing types with SMT-decidable refinement predicates [33, 43], generalizing Floyd-Hoare Logic (*e.g.* EscJava [17]) for functional languages. For example and SMT based validity checking, refinement types have automated the verification of programs with recursive datatypes, higher-order functions, and polymorphism. Several groups have used refinements to statically verify properties ranging from simple array safety [32, 43] to functional correctness of data structures [24], security protocols [5], and compiler correctness [37].

Given the remarkable effectiveness of the technique, we embarked on the project of developing a refinement type based verifier for Haskell. The previous systems were all developed for eager, *call-by-value* languages, but we presumed that the order of evaluation would surely prove irrelevant, and that the soundness guarantees would translate to Haskell's lazy, *call-by-need* regime.

We were wrong. Our first contribution is to show that standard refinement systems crucially rely on a property of eager languages: when analyzing any term, one can assume that *all* the free variables appearing in the term are bound to *values*. This property lets us check each term in an environment where the free variables are logically constrained according to their refinements. Unfortunately, this property does not hold for lazy evaluation, where free variables can be lazily substituted with arbitrary (potentially diverging) expressions, which breaks soundness (§2).

The two natural paths towards soundness are blocked by challenging problems. The first path is to *conservatively ignore* free variables except those that are guaranteed to be values *e.g.* by pat-

The Design of Refinement Types

2018 Refinement Reflection

2014 Liquid Haskell

2008 Liquid Types

2006 Hybrid Types

1991 Refinement Types

POPL'18

Refinement Reflection: Complete Verification with SMT

NIKI VAZOU, University of Maryland, USA
ANISH TONDWALKAR, University of California, San Diego, USA
VIKRAMAN CHOUDHURY, Indiana University, USA
RYAN G. SCOTT, Indiana University, USA
RYAN R. NEWTON, Indiana University, USA
PHILIP WADLER, University of Edinburgh and Input Output HK, UK
RANJIT JHALA, University of California, San Diego, USA

We introduce *Refinement Reflection*, a new framework for building SMT-based deductive verifiers. The key idea is to reflect the code implementing a user-defined function into the function's (output) refinement type. As a consequence, at *uses* of the function, the function definition is instantiated in the SMT logic in a precise fashion that permits decidable verification. Reflection allows the user to write *equational proofs* of programs just by writing other programs *e.g.* using pattern-matching and recursion to perform case-splitting and induction. Thus, via the propositions-as-types principle, we show that reflection permits the *specification* of arbitrary functional correctness properties. Finally, we introduce a proof-search algorithm called *Proof by Logical Evaluation* that uses techniques from model checking and abstract interpretation, to completely automate equational reasoning. We have implemented reflection in Liquid Haskell and used it to verify that the widely used instances of the Monoid, Applicative, Functor, and Monad typeclasses actually satisfy key algebraic laws required to make the clients safe, and have used reflection to build the first library that actually verifies

POPL'18

Refinement Reflection: Complete Verification with SMT

NIKI VAZOU, University of Maryland, USA
ANISH TONDWALKAR, University of California, San Diego, USA
VIKRAMAN CHOUDHURY, Indiana University, USA
RYAN G. SCOTT, Indiana University, USA
RYAN R. NEWTON, Indiana University, USA
PHILIP WADLER, University of Edinburgh and Input Output HK, UK
RANJIT JHALA, University of California, San Diego, USA

We introduce *Refinement Reflection*, a new framework for building SMT-based deductive verifiers. The key idea is to reflect the code implementing a user-defined function into the function's (output) refinement type. As a consequence, at *uses* of the function, the function definition is instantiated in the SMT logic in a precise fashion that permits decidable verification. Reflection allows the user to write *equational proofs* of programs just by writing other programs *e.g.* using pattern-matching and recursion to perform case-splitting and induction. Thus, via the propositions-as-types principle, we show that reflection permits the *specification* of arbitrary functional correctness properties. Finally, we introduce a proof-search algorithm called *Proof by Logical Evaluation* that uses techniques from model checking and abstract interpretation, to completely automate equational reasoning. We have implemented reflection in Liquid Haskell and used it to verify that the widely used instances of the Monoid, Applicative, Functor, and Monad typeclasses actually satisfy key algebraic laws required to make the clients safe, and have used reflection to build the first library that actually verifies assumptions about associativity and ordering that are crucial for safe deterministic parallelism.

Deep.hs

```
8
9
10  {-@ rightId :: xs:[a] -> { xs ++ [] = xs } @-}
11  rightId :: [a] -> ()
12  rightId = undefined
13
14
15
16
17
```

PROBLEMS TERMINAL OUTPUT PORTS

TERMINAL ghcid

```
All good (4 modules, at 23:43:04)
```

What can be expressed?

By Default: Logic is only SMT decidable theories

e.g., linear arithmetic, uninterpreted functions, data types, etc

Direct to express: sortedness, safe-indexing

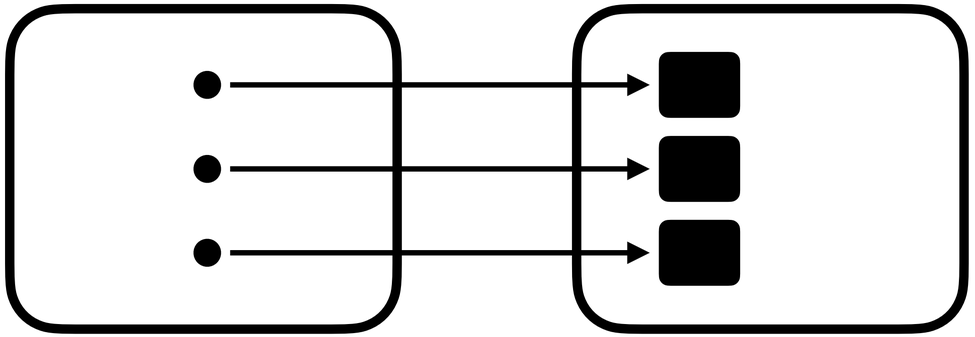
With R. Reflection: Any equational reasoning

Also expressable: Domain-specific properties

Secure Web Applications

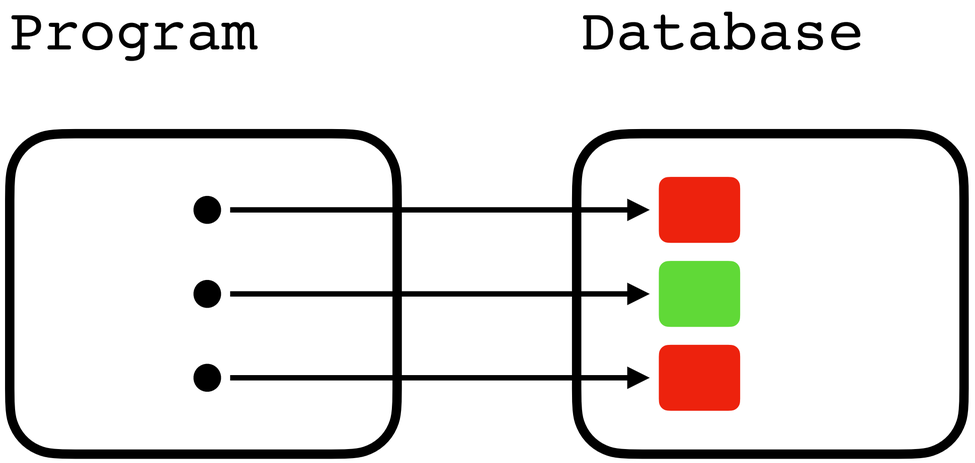
Program

Database



Programs manipulate data

Secure Web Applications




```
l:Label = Secret | Public
```

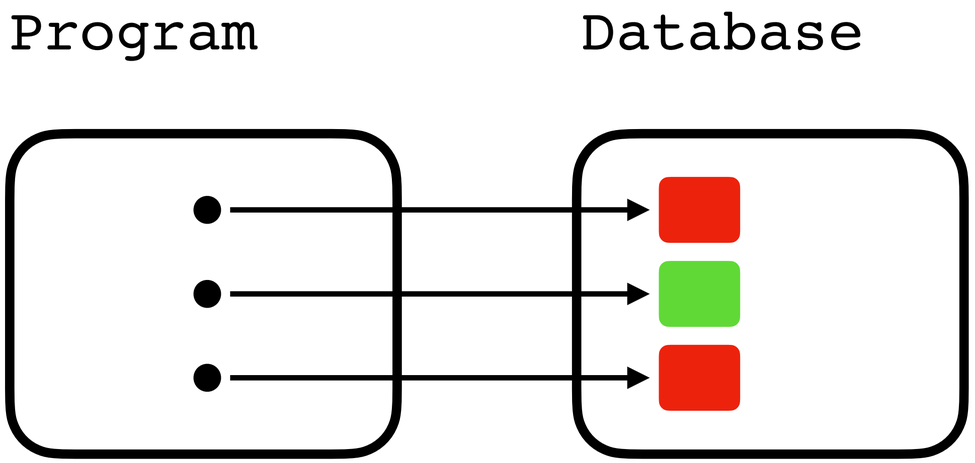
Programs manipulate data

Data are protected with policies

Noninterference:
Different labels cannot interfere

e.g., public programs cannot depend on secret data

Secure Web Applications




```
l:Label = Secret | Public
```

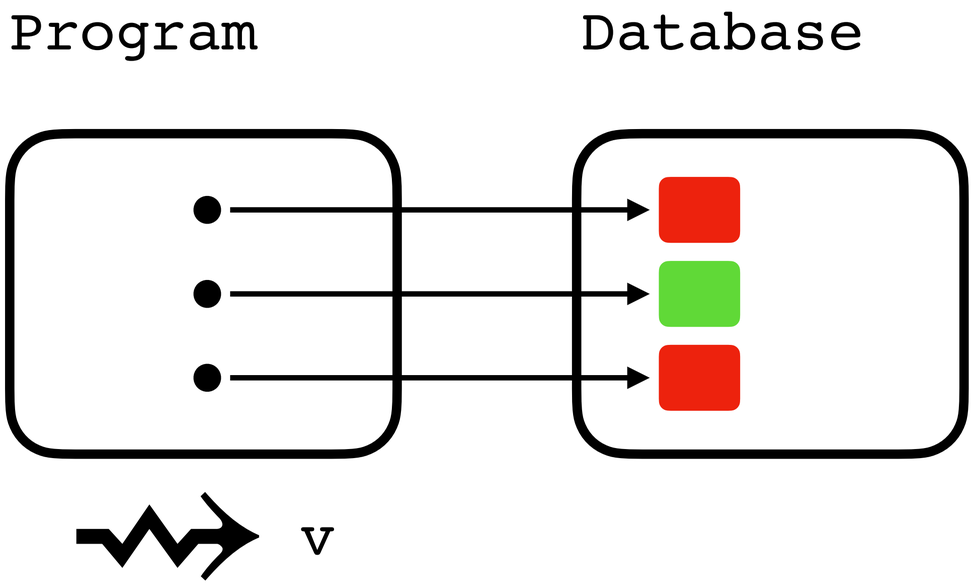
LWeb

Runtime checks to ensure noninterference

Refinement Types to prove noninterference of LWeb

LWeb: Information flow security for multi-tier web applications, by Parker, Vazou, and Hicks. POPL'19.

Noninterference

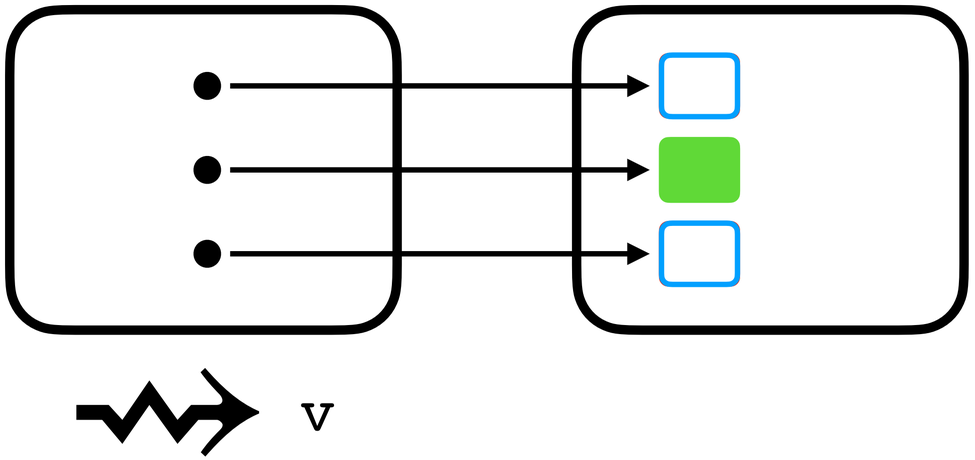


Execution preserves low view:

If we erase all the protected data, then execution remains the same.

Noninterference

ε l Program Database



Execution preserves low view:

If we erase all the protected data, then execution remains the same.

```
nonInterference :: l:Label → p1:Program → p2:Program
                → { ε l p1 == ε l p2 }
                → { ε l (exec p1) == ε l (exec p2) }
```

Secure Web Applications

LWeb

A framework that enforces label-based, dynamic, information flow policies.

First major proof in Liquid Haskell **5.5K LoC** and revealed two design bugs.

LWeb: Information flow security for multi-tier web applications,
by Parker, Vazou, and Hicks. POPL'19.

Domain Specific Properties

Secure Web Applications

Resource Usage

Distributed Applications

Liquidate your asserts, by Handley, Vazou, and Hutton. POPL'20

Verifying Replicated Data Types, by Liu, Parker, Kuper, Hicks, and Vazou. OOPSLA'20

What can be expressed?

Decidable Properties

Safe-indexing

Domain Specific Properties

Secure Web Applications

Resource Usage

Distributed Applications

Refinement Types are

- ✓ Practical
- ✓ Expressive
- General
- Sound

Refinement Types are

- ✓ Practical
- ✓ Expressive
- General
- Sound

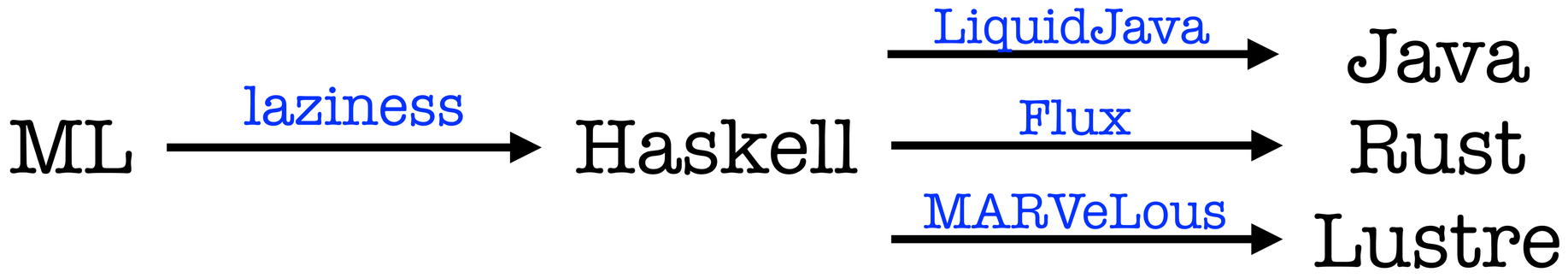
Are Refinement Types General?

Flux: Liquid Types for Rust, by Lehmann, Vazou, and Jhala. PLDI'23

Are Refinement Types General?

Usab. Oriented Design of Liquid Types for Java. Gamboa, Canelas, Timperley, Fonseca, ICSE 2023

Are Refinement Types General?



The principles are general,
but should be adjusted to each language.

Refinement Types: A tutorial, by Jhala and Vazou.
Synchronous Programming with Refinement Types. Chen et al. ICFP'24
Foundations and Trends in Programming Languages'21.

Refinement Types

- ✓ Practical
- ✓ Expressive
- ✓ General
- Sound

Refinement Types

- ✓ Practical
- ✓ Expressive
- ✓ General
- Sound

Are Refinement Types Sound?

A type system is sound when
it only accepts programs that cannot get stuck.

Soundness: If $\vdash e_o : t$ and $e_o \hookrightarrow^* e$,
then either e is a value or $e \hookrightarrow e_i$.

Soundness of Refinement Types

```
{-@ soundness :: e0:Expr -> t:Type -> Prop (HasType Empty e0 t)
              -> e:Expr -> Prop (EvalsTo e0 e)
              -> Either { isValue e } (ei::Expr, Prop (Step e ei)) @-}
```

Soundness: If $\vdash e_o : t$ and $e_o \hookrightarrow^* e$,
then either e is a value or $e \hookrightarrow e_i$.

Mechanizing Refinement Types
by Borkowski, Vazou, and Jhala, POPL'24.

Soundness of Refinement Types

```
{-@ soundness :: e0:Expr -> t:Type -> Prop (HasType Empty e0 t)
              -> e:Expr -> Prop (EvalsTo e0 e)
              -> Either { isValue e } (ei::Expr, Prop (Step e ei)) @-}
```

HasType **models** refinement type checking.
Defined by **data propositions** (inductive predicates).
Proved in ~10K lines of (Liquid) Haskell.

Mechanizing Refinement Types
by Borkowski, Vazou, and Jhala, POPL'24.

Soundness of Refinement Types

```
{-@ soundness :: e0:Expr -> t:Type -> Prop (HasType Empty e0 t)
              -> e:Expr -> Prop (EvalsTo e0 e)
              -> Either { isValue e } (ei::Expr, Prop (Step e ei)) @-}
```

```
soundness _e0 t e0_has_t _e e0_evals_e = case e0_evals_e of
  Refl e0 → progress e0 t e0_has_t -- e0 = e
  AddStep e0 e1 e0_step_e1 e e1_eval_e → -- e0 → e1 →* e
   soundness e1 t (preservation e0 t e0_has_t e1 e0_step_e1) e e1_eval_e
```

Mechanizing Refinement Types
by Borkowski, Vazou, and Jhala, POPL'24.

Refinement Types

- ✓ Practical
- ✓ Expressive
- ✓ General
- ✓ Sound

Unsoundness in Refinement Types

Soundness is proved in a model refinement type checker.

>**5** unsoundness reports*/year in Liquid Haskell

*An example of a program being accepted while it violated the property

Unsoundness in Refinement Types

Soundness is proved in a model refinement type checker.

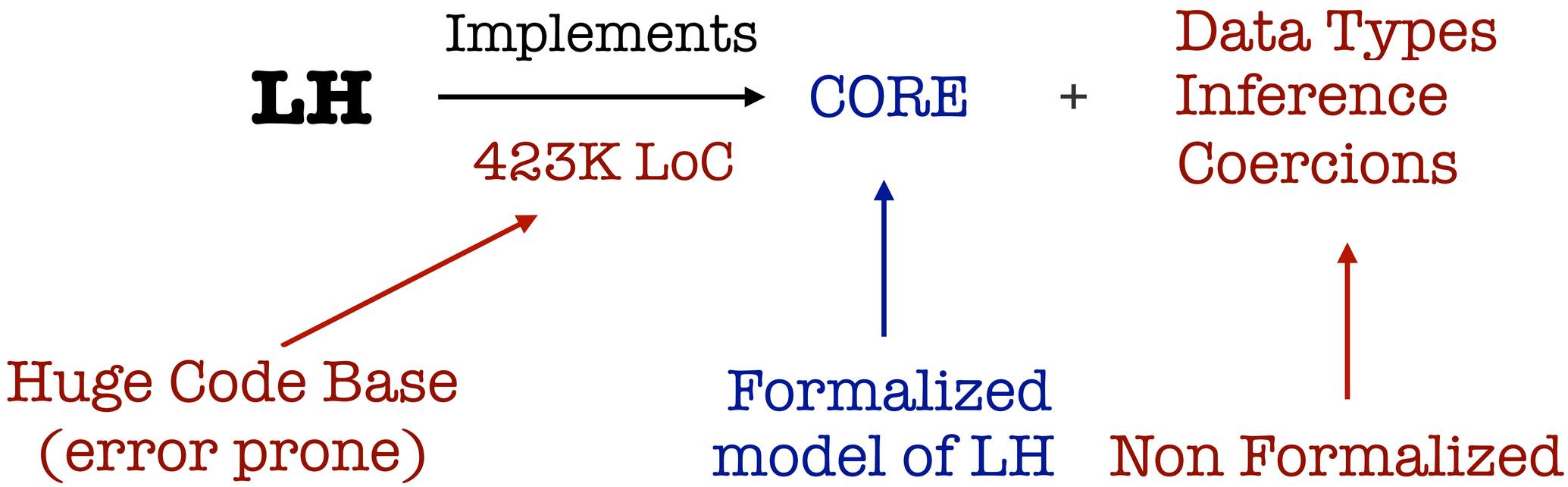
>**5** unsoundness reports /year in Liquid Haskell

Too many in comparison to sound verifiers:

<**1** unsoundness reports/year in Coq*.

*Coq: or Rocq type-theory based theorem prover, designed to be sound

Liquid Haskell is not Sound

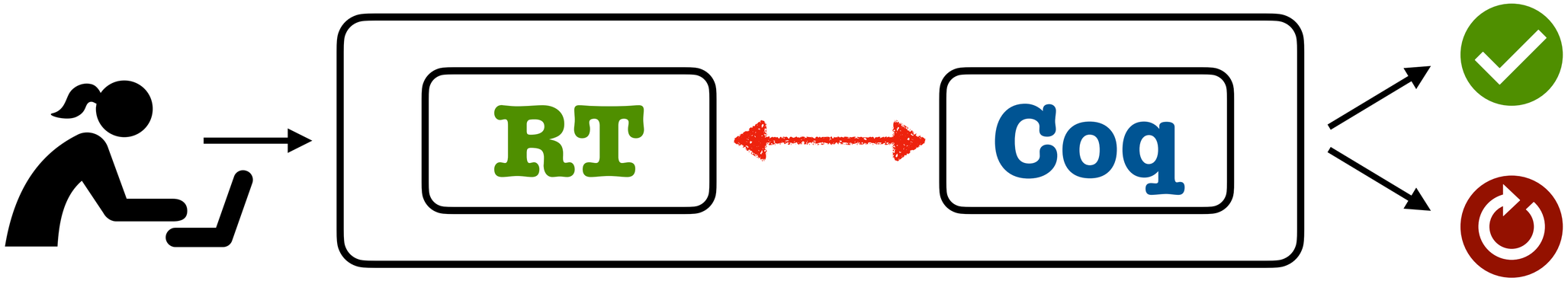


Coq is Designed to Be Sound

Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq
by Sozeau, Boulier, Forster, Tabareau and Winterhalter, POPL'20.

RT	vs.	**Coq**
Practical		Sound
On top of a PL		Implements a Logic
Implicit proofs		Explicit proofs
$42 : \{x : \mathsf{Int} \mid 0 \leq x\}$		$\mathsf{exist}(\_\ 42\ \mathsf{pt}) : \{x : \mathsf{Int} \mid 0 \leq x\}$

GOAL: Sound & Practical Verification



User writes RT on their favourite programming language!

Proofs are certified by Coq.

→ Early work presented at TYPES'23.

Explicit Refinement Types. J.E. Ghalayini and N. Krishnaswami. ICFP'23.

Refinement Types

- ✓ Practical
- ✓ Expressive
- ✓ General
-  Sound

Thanks!

Refinement Types

- ✓ Practical
- ✓ Expressive
- ✓ General
- ✗ Sound