Manifest Contracts for Datatypes

Taro Sekiyama,
Atsushi Igarashi, and Yuki Nishida

Kyoto University

Data structures

- Data structures are crucial to design algorithms
- Efficient algorithms need fine-grained specifications on data structures

E.g., binary search

```
val binary_search :
      int -> int array -> int option
```

An argument array is required to be sorted!

Specifications for data structures

Two styles in giving specifications for data structures as types

- “Extrinsic” style

```
{ x:int list | sorted x }
```

- “Intrinsic” style

```
type sorted_list =
 | SNil
 | SCons of x:int *
          {xs:slist|(nil xs) or (x < head xs)}
```

Extrinsic style

- Specifications are given to plain structures
  - E.g., sorted lists are represented as

```
{ x:int list | sorted x }
```

where "`sorted x`" expresses that x is sorted

- Work so far:
  - subset types
  - flat contracts
  - etc.

Intrinsic style

- Specifications are given to data constructors
  - E.g., sorted lists are represented as

```
type slist =
| SNil
| SCons of x:int *
      { xs:slist | (nil xs) or (x < head xs) }
```

- Work so far:
  - inductive types [Pfenning & Paulin-Mohring 1989]
  - GADTs [Cheney & Hinze 2003; Xi et al. 2003]
  - etc.

Intrinsic style

- Specifications are given to data constructors
  - E.g., sorted lists are represented as

```
type slist =
| SNil
| SCons of x:int *
      { xs:slist | (nil xs) or (x < head xs) }
```

the head is less than the head of the tail (if any)

- Work so far:
  - inductive types [Pfenning & Paulin-Mohring 1989]
  - GADTs [Cheney & Hinze 2003; Xi et al. 2003]
  - etc.

Pros and cons of two styles as contracts

	Pros	Cons
Extrinsic style `{x:int list|sorted x}`	easy to write programs	poor information on substructures
Intrinsic style `type slist = …`	rich information on substructures	difficult to write programs

Our work: transformations to take the best of both worlds

E.g., tail parts of sorted lists are merely lists (no specs)

Checking specs ***dynamically*** can worsen asymptotic time complexity

	Pros	**Cons**
Extrinsic style `{x:int list|sorted x}`	easy to write programs	poor information on substructures
Intrinsic style `type slist = …`	rich information on substructures	difficult to write programs

Our work: transformations to take the best of both worlds

Extrinsic style can worsen time complexity

```
let tail
  (y : { x:int list | sorted x }) :
  { x:int list | sorted x } =
  match y with
  | []      -> []
  | z::zs -> zs
```

Extrinsic style can worsen time complexity

```
let tail
  (y : { x:int list | sorted x }) :
  { x:int list | sorted x } =
  match y with
  | []
  | z::zs -
```

returns the tail of an argument list (if any)

Extrinsic style can worsen time complexity

```
let tail
  (y : { x:int list | sorted x }) :
  { x:int list | sorted x } =
  match y with
  | []
  | z::zs -
```

returns the tail of an argument list (if any)

When *sorted* is checked dynamically, the asymptotic time complexity turns out to be ***O(length x)***

Intrinsic style can preserve time complexity

```
let tail (y : slist) : slist = match y with
  | SNil                     -> SNil
  | SCons (z, zs) -> zs
```

```
type slist =
  | SNil  of unit
  | SCons of x:int *
   { xs: slist | (nil xs) or (x < head xs) }
```

Intrinsic style can preserve time complexity

```
let tail (y : slist) : slist = match y with
  | SNil                    -> SNil
  | SCons (z, zs) ->
```

The type of zs is slist, so the asymptotic time complexity is ***O(1)***

```
type slist =
  | SNil  of unit
  | SCons of x:int *
   { xs: slist | (nil xs) or (x < head xs) }
```

E.g., tail parts of sorted lists are merely lists (no specs)

Checking specs *dynamically* can worsen asymptotic time complexity

	Pros	**Cons**
Extrinsic style { x:int list | sorted x }	easy to write programs	poor information on substructures
Intrinsic style type slist = …	rich information on substructures	difficult to write programs

E.g., tail parts of sorted lists are merely lists (no specs)

Checking specs *dynamically* can worsen asymptotic time complexity

	Pros	**Cons**
Extrinsic style { x:int list | sorted x }	easy to write programs	poor information on substructures
Intrinsic style type slist = …	rich information on substructures	difficult to write programs

1. Refining constructors is unfamiliar for programmers
2. Library problem, e.g., ***all*** list-functions cannot be applied to slist

int list -> int

```
head (SCons (1, SNil))
```

slist

Our ideas for better use of intrinsic style

1. ***Static translation*** from a type in extrinsic style to one in intrinsic style

   `{x:int list|sorted x}` ➔ `type slist = …`

2. ***Dynamic conversion*** between data structures in both styles

   `head (<int list <= slist>`$^{\ell}$ `SCons (1, SNil))`

How do the ideas encourage use of intrinsic style?

1. Programmers can obtain dynamically efficient datatypes easily, using **type translation** (the 1st idea)

`{x:int list|sorted x}` ➔ `type` slist = …

How do the ideas encourage use of intrinsic style?

1. Programmers can obtain dynamically efficient datatypes easily, using **type translation** (the 1st idea)

`{x:int list|sorted x}` ➔ `type slist = …`

2. They write programs using the generated datatypes

```
let tail (y : slist) : slist = …
```

How do the ideas encourage use of intrinsic style?

1. Programmers can obtain dynamically efficient datatypes easily, using **type translation** (the 1st idea)

```
{x:int list|sorted x} ➔ type slist = …
```

2. They write programs using the generated datatypes

```
let tail (y : slist) : slist = …
```

3. If needed, they can reuse functions for original data structures, using **dynamic conversion** (the 2nd idea)

```
head (<int list <= slist>ℓ (SCons (1, SNil)))
```

This Work

- We give a lambda-calculus based on *manifest contracts* and formalize the ideas in the calculus
  - Manifest contracts [Flanagan 2006; Greenberg et al. 2010] are a framework which can combine static and *dynamic* specification checkings
- We implement the dynamic conversion mechanism on OCaml, using the extensible preprocessor (Camlp4)
  - Available at http://goo.gl/VMhAv2

Contents

1. Manifest Contracts with Datatypes
    1. Refinement types
    2. Manifest Datatypes
2. Dynamic Type Conversion

```
<int list <= slist>ℓ
```

3. Syntactic Type Translation

```
{x:int list|sorted x} → type slist = …
```

Software contracts

- Specifications of program components
  - impossible to represent as *simple* types
- Dynamically enforced
  - written in an executable form, i.e., as programs

Manifest contracts [Flanagan 2006; Greenberg et al. 2010]

- Contracts are made "manifest" as part of types

Refinement types (a.k.a. subset types)

$$\{x{:}T \mid e\}$$

denotes the set of values of type *T* satisfying the *Boolean expression e*

E.g., `{x:int|0 < x}` means positive integers

- Contracts are checked statically or dynamically
  - This work concerns only dynamic checking

Manifest datatypes

- Data constructors are given contracts
  - This notion itself is not new

Sorted Lists (Manifest Datatype ver.)

```
type slist =
  | SNil  of unit
  | SCons of x:int *
        {xs:slist|(nil xs) or (x < head xs)}
```

Manifest datatypes

- Data constructors are given contracts
  - This notion itself is not new

Sorted Lists (Manifest Datatype ver.)

```
type slist =
  | SNil  of unit
  | SCons of x:int *
         {xs:slist|(nil xs) or (x < head xs)}
```

Contents

1. Manifest Contracts with Datatypes
   1. Refinement types
   2. Manifest Datatypes
2. Dynamic Type Conversion

```
<int list <= slist>ℓ
```

3. Syntactic Type Translation

```
{x:int list|sorted x} → type slist = …
```

Type conversion in manifest contracts

[Flanagan 2006]

$$<T_1 <= T_2>^{\ell}$$

checks that a value of T_2 works as T_1

E.g., conversion for base types

$$<\{x:int|0 < x\} <= int>^{\ell}\ 4 \longrightarrow 4$$

$$<\{x:int|0 < x\} <= int>^{\ell}\ 0 \longrightarrow \Uparrow\ell$$

checks that given integers are positive

4 is positive, but 0 is not; so exception $\Uparrow\ell$ is raised

Type conversion in manifest contracts

[Flanagan 2006]

$$<T_1 \; <= \; T_2>^{\ell}$$

checks that a value of T_2 works as T_1

E.g., conversion for dependent pair types

```
<x:int*{y:int| x < y} <= int*int>ℓ (4,6)
  → (4,<{y:int|4 < y} <= int>ℓ 6)
  → (4,6)
```

checks that the first integer is less than the second

Type conversion to sorted lists

```
< slist <= int list >ℓ
```

checks that integer lists are sorted

```
< slist <= int list >ℓ (1 :: [])
```

$\longrightarrow$

```
type slist =
| SNil  of unit
| SCons of x:int *
      { xs:slist | (nil xs) or (x < head xs) }
```

Type conversion to sorted lists

```
< slist <= int list >ℓ
```

1. replaces each constructor of a given list to corresponding one of slist
2. checks the contract in the type of the constructor

```
< slist <= int list >ℓ (1 :: [])
```

$\longrightarrow$

```
type slist =
| SNil  of unit
| SCons of x:int *
      { xs:slist | (nil xs) or (x < head xs) }
```

Type conversion to sorted lists

```
< slist <= int list >^ℓ
```

1. replaces each constructor of a given list to corresponding one of slist
2. checks the contract in the type of the constructor

```
< slist <= int list >^ℓ (1 :: [])
```

```
→ SCons
```

```
type slist =
| SNil  of unit
| SCons of x:int *
       { xs:slist | (nil xs) or (x < head xs) }
```

Type conversion to sorted lists

```
< slist <= int list >ℓ
```

1. replaces each constructor of a given list to corresponding one of slist
2. checks the contract in the type of the constructor

```
< slist <= int list >ℓ (1 :: [])
  → SCons (<T <= int*int list>ℓ (1, []))
```

```
type slist =
| SNil  of unit
| SCons of x:int *
         { xs:slist | (nil xs) or (x < head xs) }
```

Type conversion to sorted lists

```
<slist ⇐ int list>ℓ (1 :: [])
→ SCons (<T ⇐ int * int list>ℓ (1, []))
```

```
type slist =
  | SNil    of unit
  | SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }
```

Type conversion to sorted lists

<slist ⇐ int list>$^\ell$ (1 :: [])

⟶ SCons (<T ⇐ int * int list>$^\ell$ (1, []))

type slist =
  | SNil of unit
  | SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }

Type conversion to sorted lists

<slist ⇐ int list>$^\ell$ (1 :: [])

→ SCons (<T ⇐ int * int list>$^\ell$ (1, []))

→ SCons (1,

<{ xs:slist | (nil xs) or (1 < head xs) } ⇐ int list>$^\ell$ [])

---

type slist =

| SNil of unit

| SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }

Type conversion to sorted lists

<slist ⇐ int list>$^{\ell}$ (1 :: [])

→ SCons (<T ⇐ int * int list>$^{\ell}$ (1, []))

→ SCons (1,

<{ xs:slist | (nil xs) or (1 < head xs) } ⇐ int list>$^{\ell}$ [])

→ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^{\ell}$

(<slist ⇐ int list>$^{\ell}$ []))

Split into two:
(1) conversion from integer lists to slist
(2) conversion to check the contract

type slist =
| SNil
| SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }

Type conversion to sorted lists

<slist ⇐ int list>$^\ell$ (1 :: [])

→ SCons (<T ⇐ int * int list>$^\ell$ (1, []))

→ SCons (1,

<{ xs:slist | (nil xs) or (1 < head xs) } ⇐ int list>$^\ell$ [])

→ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^\ell$

(1) (<slist ⇐ int list>$^\ell$ []))

Split into two:

(1) conversion from integer lists to slist

(2) conversion to check the contract

type slist =

| SNil

| SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }

Type conversion to sorted lists

<slist ⇐ int list>$^\ell$ (1 :: [])

→ SCons (<T ⇐ int * int list>$^\ell$ (1, []))

→ SCons (1,
  <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ int list>$^\ell$ [])

→ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^\ell$ (2)
  (<slist ⇐ int list>$^\ell$ [])) (1)

Split into two:
(1) conversion from integer lists to slist
(2) conversion to check the contract

type slist =
  | SNil
  | SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }

Type conversion to sorted lists

<slist ⇐ int list>$^{\ell}$ (1 :: [])

⟶ SCons (<T ⇐ int * int list>$^{\ell}$ (1, []))

⟶ SCons (1,

<{ xs:slist | (nil xs) or (1 < head xs) } ⇐ int list>$^{\ell}$ [])

⟶ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^{\ell}$

(<slist ⇐ int list>$^{\ell}$ []))

---

type slist =

| SNil of unit

| SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }

Type conversion to sorted lists

<slist ⇐ int list>$^\ell$ (1 :: [])

→ SCons (<T ⇐ int * int list>$^\ell$ (1, []))

→ SCons (1,

<{ xs:slist | (nil xs) or (1 < head xs) } ⇐ int list>$^\ell$ [])

→ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^\ell$

(<slist ⇐ int list>$^\ell$ []))

type slist =

SNil of unit

| SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }

Type conversion to sorted lists

<slist ⇐ int list>$^{\ell}$ (1 :: [])

→ SCons (<T ⇐ int * int list>$^{\ell}$ (1, []))

→ SCons (1,

<{ xs:slist | (nil xs) or (1 < head xs) } ⇐ int list>$^{\ell}$ [])

→ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^{\ell}$

(<slist ⇐ int list>$^{\ell}$ []))

→ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^{\ell}$ SNil)

---

type slist =
  | SNil of unit
  | SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }

Type conversion to sorted lists

<slist ⇐ int list>$^{\ell}$ (1 :: [])

⟶ SCons (<T ⇐ int * int list>$^{\ell}$ (1, []))

⟶ SCons (1,
<{ xs:slist | (nil xs) or (1 < head xs) } ⇐ int list>$^{\ell}$ [])

⟶ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^{\ell}$
(<slist ⇐ int list>$^{\ell}$ []))

⟶ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^{\ell}$ SNil)

“nil xs” means xs = SNil

---

type slist =
| SNil of unit
| SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }

Type conversion to sorted lists

<slist ⇐ int list>$^{\ell}$ (1 :: [])

⟶ SCons (<T ⇐ int * int list>$^{\ell}$ (1, []))

⟶ SCons (1,
<{ xs:slist | (nil xs) or (1 < head xs) } ⇐ int list>$^{\ell}$ [])

⟶ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^{\ell}$
(<slist ⇐ int list>$^{\ell}$ []))

⟶ SCons (1, <{ xs:slist | (nil xs) or (1 < head xs) } ⇐ slist>$^{\ell}$ SNil)

⟶ SCons (1, SNil)

---

type slist =
| SNil of unit
| SCons of x:int * { xs:slist | (nil xs) or (x < head xs) }

Nontrivial example: list_containing0

≈ lists containing 0

```
type list_containing0 =
    | C1 of int * list_containing0
    | C2 of { x:int | x = 0 } * int list
```

Nontrivial example: list_containing0

≈ lists containing 0

```
type list_containing0 =
  | C1 of int * list_containing0
  | C2 of { x:int | x = 0 } * int list
```

- No constructors corresponding to []

Nontrivial example: list_containing0

≈ lists containing 0

```
type list_containing0 =
  | C1 of int * list_containing0
  | C2 of { x:int | x = 0 } * int list
```

- No constructors corresponding to []
- Two constructors corresponding to (::)

Either C_1 or C_2 has to be chosen dynamically

- Formal semantics: an oracle choice function
- Implementation: trial-and-error (backtracking)

Formalization

- We formalize a manifest calculus with manifest datatypes, following the syntactic approach [Belo et al. 2011]
  - The calculus supports dynamic conversion between manifest datatypes
- We prove type soundness via progress and subject reduction
  - Exceptions are legitimate results
- We fix a few technical flaws in the previous work

Contents

1. Manifest Contracts with Datatypes
   1. Refinement types
   2. Manifest Datatypes
2. Dynamic Type Conversion

   $\langle \text{int list} \Leftarrow \text{slist} \rangle^{\ell}$

3. Syntactic Type Translation

   { x:int list | sorted x } ➔ type slist = …

Syntactic type translation

Refinement Type → Manifest Datatype

Syntactic type translation

Returns whether x contains 0

{ x:int list | contains0 x }

Refinement Type → Manifest Datatype

Running example

An implementation of contains0 is:

```
let contains0 y = match y with
  | []     -> false
  | x::xs -> if x = 0 then true else contains0 xs
```

Ideas of translation

1. To collect guard conditions on execution paths reaching true from branches for [] and (::)

```
let contains0 y = match y with
  | []     -> false
  | x::xs -> if x = 0 then true else contains0 xs
```

Ideas of translation

1. To collect guard conditions on execution paths reaching true from branches for [] and (::)

There are no execution paths reaching true from the branch for []

```
let contains0 y = match y with
  | []     -> false
  | x::xs -> if x = 0 then true else contains0 xs
```

Ideas of translation

1. To collect guard conditions on execution paths reaching true from branches for [] and (::)

There are two execution paths for (::):
(1) x = 0 (2) x <> 0 & contains0 xs

There are no execution paths reaching true from the branch for []

```
let contains0 y = match y with
  | []     -> false
  | x::xs -> if x = 0 then true else contains0 xs
```

Ideas of translation

2. The new datatype has one constructor for each execution path; the contract of it is conjunction of the guard conditions for the path

```
let contains0 y = match y with
  | []      -> false
  | x::xs -> if x = 0 then true else contains0 xs
```

Ideas of translation

2. The new datatype has one constructor for each execution path; the contract of it is conjunction of the guard conditions for the path

The new datatype has two constructors:

(1) C_1 : { x:int | x = 0 } * int list

(2) C_2 : { x:int | x <> 0 } * { xs:int list | contains0 xs }

```
let contains0 y = match
  | []    -> false
  | x::xs -> if x = 0 then true else contains0 xs
```

The guard conditions for (::)

(1) x = 0 (2) x <> 0 & contains0 xs

Ideas of translation

3. The recursive call becomes type-level recursion

The argument type of C_2 transforms from

{ x:int | x <> 0 } * { xs:int list | contains0 xs }

to

{ x:int | x <> 0 } * list_containing0

```
let contains0 y = match y with
  | []      -> false
  | x::xs -> if x = 0 then true else contains0 xs
```

Resulting datatype

{ x:int list | contains0 x }

```
let contains0 y = match y with
  | []     -> false
  | x::xs -> if x = 0 then true
             else contains0 xs
```

type list_containing0’ =
  | C_1 of { x:int | x = 0 } * int list
  | C_2 of { x:int | x <> 0 } * list_containing0’

Formalization

- We formalize translation for only integer lists
  - Generalization would be possible but cumbersome
- We prove its correctness: the generated datatype is equivalent to the original refinement type
  - Dynamic conversion between a refinement type and the new datatype always succeeds in both directions

E.g., dynamic checks with

<list_containing0’ ⇐ { x:int list | contains0 x }>$^{\ell}$

<{ x:int list | contains0 x } ⇐ list_containing0’>$^{\ell}$

always succeeds!

FAQ about translation

Q. Is the generated datatype dynamically efficient representation?

A. Yes, it is at least as efficient as the original refinement type

– Conversion to the new datatype involves the same computation as checking the contract in the refinement type

Q. What predicate functions does translation work well for?

A. Ones written in the fold form (at least)

We discuss these in the paper in more details

In the paper …

- Manifest datatypes abstracted over value variables
  - The translation algorithm supports that form
- Discussion on extension of type translation to other data structures, e.g., trees
- Formalization and proofs
  - Our manifest calculus and syntactic type translation
- A prototype implementation of our calculus (without type translation)

http://goo.gl/VMhAv2

Related work (1)

Lazy Contract Checking for Immutable Data Structures (Findler et al., IFL ‘07)

- The first work (as far as I know) that discussed pros and cons of extrinsic and intrinsic styles
- They attempted to resolve the inefficiency problem of extrinsic style by introducing lazy contract checking

Related work (2)

Ornamental Algebras, Algebraic Ornaments

(McBride, ‘12)

Refining Inductive Types

(Atkey et al., LMCS ’12)

- They have studied systematic derivation of inductive datatypes
- They don’t concern dynamic aspects of datatypes

Future work

Issue: reusing functions with type conversion could need a significant computation cost

head (<int list $\Leftarrow$ slist>$^{\ell}$ x)

The asymptotic time complexity is O(length x), not O(1)

Approaches:

- **Lazy Contract Checking for Immutable Data Structures** (Findler et al., IFL '07)
- A new calculus where such conversions are available for free

Conclusion

- We formulate specs in extrinsic and intrinsic styles as refinement types and manifest datatypes, resp.
- We give two ways to take the best of both worlds
  - Dynamic type conversion
  - Syntactic type translation
- We propose a manifest calculus with manifest datatypes to formalize our ideas

Prototype is available at http://goo.gl/VMhAv2

Thank you!

Questions?

Slowly, please ☺