Manifest Contracts for OCaml

Yuki Nishida Atsushi Igarashi

Kyoto University

ACM SIGPLAN Workshop on ML, September 2015

Software Contracts

[Meyer *Object-Oriented Software Construction*, Prentice Hall, 1997]

Specifications of a module interface

- embedded into program code
- used for verification

Example of Contracts

`let sqrt (x : int) = ...`

Pre-Condition: $0 \leq x$

Post-Condition: $\texttt{ret}^2 \leq x < (\texttt{ret} + 1)^2$

Manifest Contracts

[Greenberg, Pierce, and Weirich POPL 2010; Flanagan POPL 2006]

Contracts are described by the following new types

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

positive integers $\{x{:}\texttt{int} \mid x > 0\}$

non-zero integers $\{x{:}\texttt{int} \mid x \neq 0\}$

Dependent function types: $x{:}T_1 \to T_2$

negation function $x{:}\texttt{int} \to \{y{:}\texttt{int} \mid x + y = 0\}$

Manifest Contracts

[Greenberg, Pierce, and Weirich POPL 2010; Flanagan POPL 2006]

underlying type

Contracts are described by the following new types

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

positive integers $\{x{:}\texttt{int} \mid x > 0\}$

non-zero integers $\{x{:}\texttt{int} \mid x \neq 0\}$

Dependent function types: $x{:}T_1 \to T_2$

negation function $x{:}\texttt{int} \to \{y{:}\texttt{int} \mid x + y = 0\}$

Manifest Contracts

[Greenberg, Pierce, and Weirich POPL 2010; Flanagan POPL 2006]

underlying type

boolean expression

Contracts are described by the following new types

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

positive integers $\{x{:}\texttt{int} \mid x > 0\}$

non-zero integers $\{x{:}\texttt{int} \mid x \neq 0\}$

Dependent function types: $x{:}T_1 \to T_2$

negation function $x{:}\texttt{int} \to \{y{:}\texttt{int} \mid x + y = 0\}$

Manifest Contracts

[Greenberg, Pierce, and Weirich POPL 2010; Flanagan POPL 2006]

parameter

underlying type

boolean expression

Contracts are described by the following new types

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

positive integers $\{x{:}\texttt{int} \mid x > 0\}$

non-zero integers $\{x{:}\texttt{int} \mid x \neq 0\}$

Dependent function types: $x{:}T_1 \to T_2$

negation function $x{:}\texttt{int} \to \{y{:}\texttt{int} \mid x + y = 0\}$

Example of Manifest Contracts for `sqrt`

$$\texttt{val sqrt} : x : \{x{:}\texttt{int} \mid 0 \leq x\} \rightarrow \{y{:}\texttt{int} \mid y^2 \leq x < (y+1)^2\}$$

Example of Contracts

`let sqrt (x : int) = ...`

Pre-Condition: $0 \leq x$

Post-Condition: $\texttt{ret}^2 \leq x < (\texttt{ret}+1)^2$

Example of Manifest Contracts for `sqrt`

$$\texttt{val sqrt} : x : \{x{:}\texttt{int} \mid 0 \leq x\} \rightarrow \{y{:}\texttt{int} \mid y^2 \leq x < (y+1)^2\}$$

Example of Contracts

```
let sqrt (x : int) = ...
```

Pre-Condition: $0 \leq x$

Post-Condition: $\texttt{ret}^2 \leq x < (\texttt{ret}+1)^2$

Example of Manifest Contracts for `sqrt`

$$\texttt{val sqrt} : (x) : \{x{:}\texttt{int} \mid 0 \leq x\} \rightarrow \{y{:}\texttt{int} \mid y^2 \leq x < (y+1)^2\}$$

Example of Contracts

```
let sqrt (x : int) = ...
```

Pre-Condition: $0 \leq x$

Post-Condition: $\texttt{ret}^2 \leq x < (\texttt{ret}+1)^2$

Methodology for Contract Checking

Static

e.g. Liquid Types [Rondon, Kawaguchi, and Jhala PLDI 2008]

- There are some restriction to check contracts statically

Dynamic

e.g. Racket [Flatt and PLT 2010]

Hybrid

Hybrid Type Checking [Knowles and Flanagan TOPLAS 2010]

Methodology for Contract Checking

Static

e.g. Liquid Types [Rondon, Kawaguchi, and Jhala PLDI 2008]

- There are some restriction to check contracts statically

Our current extension to OCaml

Dynamic

e.g. Racket [Flatt and PLT 2010]

Hybrid

Hybrid Type Checking [Knowles and Flanagan TOPLAS 2010]

This Talk

Our ongoing work on extending OCaml to support manifest contracts system

Interaction with Our Extended OCaml

```
# let f (x :  {@ x of x > 0 @} ) = x + 1;;
val f : {@ x of x > 0 @} -> int = <fun>
# f  (1 :> {@ x of x > 0 @}) ;;
- : int = 2
# type pos = {@ x of x > 0 @};;
type pos = {@ x of x > 0 @}
# f (0 :> pos);;
Exception: Assert_failure ("_none_", 1, -1).
# f  0 ;;
Exception: Assert_failure ("_none_", 1, -1).
# let g (f : int -> pos) = f 0;;
val g : (int -> pos) -> pos = <fun>
# g (fun x -> x - 1);;
Exception: Assert_failure ("_none_", 1, -1).
```

Interaction with Our Extended OCaml

```
# let f (x : {@ x of x > 0 @}) = x + 1;;
val f : {@ x of x > 0 @} -> int = <fun>
# f  (1 :> {@ x of x > 0 @}) ;;
- : int = 2
# type pos = {@ x of x > 0 @};;
type pos = {@ x of x > 0 @}
# f (0 :> pos);;
Exception: Assert_failure ("_none_", 1, -1).
# f  0 ;;
Exception: Assert_failure ("_none_", 1, -1).
# let g (f : int -> pos) = f 0;;
val g : (int -> pos) -> pos = <fun>
# g (fun x -> x - 1);;
Exception: Assert_failure ("_none_", 1, -1).
```

Our notation of $\{x{:}\texttt{int} \mid x > 0\}$

Interaction with Our Extended OCaml

```
# let f (x : {@ x of x > 0 @}) = x + 1;;
val f : {@ x of x > 0 @} -> int = <fun>
# f  (1 :> {@ x of x > 0 @}) ;;
- : int = 2
# type pos = {@ x of x > 0 @};;
type pos = {@ x of x > 0 @}
# f (0 :> pos);;
Exception: Assert_failure ("_none_", 1, -1).
# f  0 ;;
Exception: Assert_failure ("_none_", 1, -1).
# let g (f : int -> pos) = f 0;;
val g : (int -> pos) -> pos = <fun>
# g (fun x -> x - 1);;
Exception: Assert_failure ("_none_", 1, -1).
```

Our notation of $\{x{:}\texttt{int} \mid x > 0\}$

Interaction with Our Extended OCaml

```
# let f (x : {@ x of x > 0 @}) = x + 1;;
val f : {@ x of x > 0 @} -> int = <fun>
# f (1 :> {@ x of x > 0 @});;
- : int = 2
# type pos = {@ x of x > 0 @};;
type pos = {@ x of x > 0 @}
# f (0 :> pos);;
Exception: Assert_failure ("_none_", 1, -1).
# f  0 ;;
Exception: Assert_failure ("_none_", 1, -1).
# let g (f : int -> pos) = f 0;;
val g : (int -> pos) -> pos = <fun>
# g (fun x -> x - 1);;
Exception: Assert_failure ("_none_", 1, -1).
```

Our notation of $\{x{:}\texttt{int} \mid x > 0\}$

A contract is checked with a coercion

Interaction with Our Extended OCaml

```
# let f (x : {@ x of x > 0 @}) = x + 1;;
val f : {@ x of x > 0 @} -> int = <fun>
# f (1 :> {@ x of x > 0 @});;
- : int = 2
# type pos = {@ x of x > 0 @};;
type pos = {@ x of x > 0 @}
# f (0 :> pos);;
Exception: Assert_failure ("_none_", 1, -1).
# f  0 ;;
Exception: Assert_failure ("_none_", 1, -1).
# let g (f : int -> pos) = f 0;;
val g : (int -> pos) -> pos = <fun>
# g (fun x -> x - 1);;
Exception: Assert_failure ("_none_", 1, -1).
```

Our notation of $\{x{:}\texttt{int} \mid x > 0\}$

A contract is checked with a coercion

Interaction with Our Extended OCaml

```
# let f (x : {@ x of x > 0 @}) = x + 1;;
val f : {@ x of x > 0 @} -> int = <fun>
# f (1 :> {@ x of x > 0 @});;
- : int = 2
# type pos = {@ x of x > 0 @};;
type pos = {@ x of x > 0 @}
# f (0 :> pos);;
Exception: Assert_failure ("_none_", 1, -1).
# f  0 ;;
Exception: Assert_failure ("_none_", 1, -1).
# let g (f : int -> pos) = f 0;;
val g : (int -> pos) -> pos = <fun>
# g (fun x -> x - 1);;
Exception: Assert_failure ("_none_", 1, -1).
```

Our notation of $\{x{:}\texttt{int} \mid x > 0\}$

A contract is checked with a coercion

Interaction with Our Extended OCaml

```
# let f (x : {@ x of x > 0 @}) = x + 1;;
val f : {@ x of x > 0 @} -> int = <fun>
# f (1 :> {@ x of x > 0 @});;
- : int = 2
# type pos = {@ x of x > 0 @};;
type pos = {@ x of x > 0 @}
# f (0 :> pos);;
Exception: Assert_failure ("_none_", 1, -1).
# f  0 ;;
Exception: Assert_failure ("_none_", 1, -1).
# let g (f : int -> pos) = f 0;;
val g : (int -> pos) -> pos = <fun>
# g (fun x -> x - 1);;
Exception: Assert_failure ("_none_", 1, -1).
```

Our notation of $\{x{:}\texttt{int} \mid x > 0\}$

A contract is checked with a coercion

Interaction with Our Extended OCaml

```
# let f (x : {@ x of x > 0 @}) = x + 1;;
val f : {@ x of x > 0 @} -> int = <fun>
# f (1 :> {@ x of x > 0 @});;
- : int = 2
# type pos = {@ x of x > 0 @};;
type pos = {@ x of x > 0 @}
# f (0 :> pos);;
Exception: Assert_failure ("_none_", 1, -1).
# f 0;;
Exception: Assert_failure ("_none_", 1, -1).
# let g (f : int -> pos) = f 0;;
val g : (int -> pos) -> pos = <fun>
# g (fun x -> x - 1);;
Exception: Assert_failure ("_none_", 1, -1).
```

Our notation of $\{x{:}\texttt{int} \mid x > 0\}$

A contract is checked with a coercion

A coercion can be omitted. It's inserted internally

Interaction with Our Extended OCaml

```
# let f (x : {@ x of x > 0 @}) = x + 1;;
val f : {@ x of x > 0 @} -> int = <fun>
# f (1 :> {@ x of x > 0 @});;
- : int = 2
# type pos = {@ x of x > 0 @};;
type pos = {@ x of x > 0 @}
# f (0 :> pos);;
Exception: Assert_failure ("_none_", 1, -1).
# f 0;;
Exception: Assert_failure ("_none_", 1, -1).
# let g (f : int -> pos) = f 0;;
val g : (int -> pos) -> pos = <fun>
# g (fun x -> x - 1);;
Exception: Assert_failure ("_none_", 1, -1).
```

Our notation of $\{x{:}\texttt{int} \mid x > 0\}$

A contract is checked with a coercion

A coercion can be omitted. It's inserted internally

Interaction with Our Extended OCaml

```
# let f (x : {@ x of x > 0 @}) = x + 1;;
val f : {@ x of x > 0 @} -> int = <fun>
# f (1 :> {@ x of x > 0 @});;
- : int = 2
# type pos = {@ x of x > 0 @};;
type pos = {@ x of x > 0 @}
# f (0 :> pos);;
Exception: Assert_failure ("_none_", 1, -1).
# f 0;;
Exception: Assert_failure ("_none_", 1, -1).
# let g (f : int -> pos) = f 0;;
val g : (int -> pos) -> pos = <fun>
# g (fun x -> x - 1);;
Exception: Assert_failure ("_none_", 1, -1).
```

Our notation of $\{x{:}\texttt{int} \mid x > 0\}$

A contract is checked with a coercion

A coercion can be omitted. It's inserted internally

Features of Our Extended OCaml

- Refinement types are integrated as a part of OCaml types
- Type inference deals with refinement types though predicates are not inferred
- Coercions (run-time checking) are inserted automatically

Still ongoing

better inference, dependent function types, investigation towards rich OCaml features, etc.

Refinement Types

Type Inference & Coercion Insertion

Coercions

Refinement Types

Syntax `{@ pattern of expression @}`

- A pattern can be used at parameter position
  `{@ (x, y) of x = y @}`
- An underlying type is inferred from use of pattern variables in a predicate
  `{@ x : int of x > 0 @}`
- A predicate can be an arbitrary boolean OCaml expression. However, we will need some purity checks

Refinement Types

Syntax {@ *pattern* of *expression* @}

- A pattern can be used at parameter position
  {@ (x, y) of x = y @}
- An underlying type is inferred from use of pattern variables in a predicate
  {@ x : int of x > 0 @}
- A predicate can be an arbitrary boolean OCaml expression. However, we will need some purity checks

Refinement Types

Type Inference & Coercion Insertion

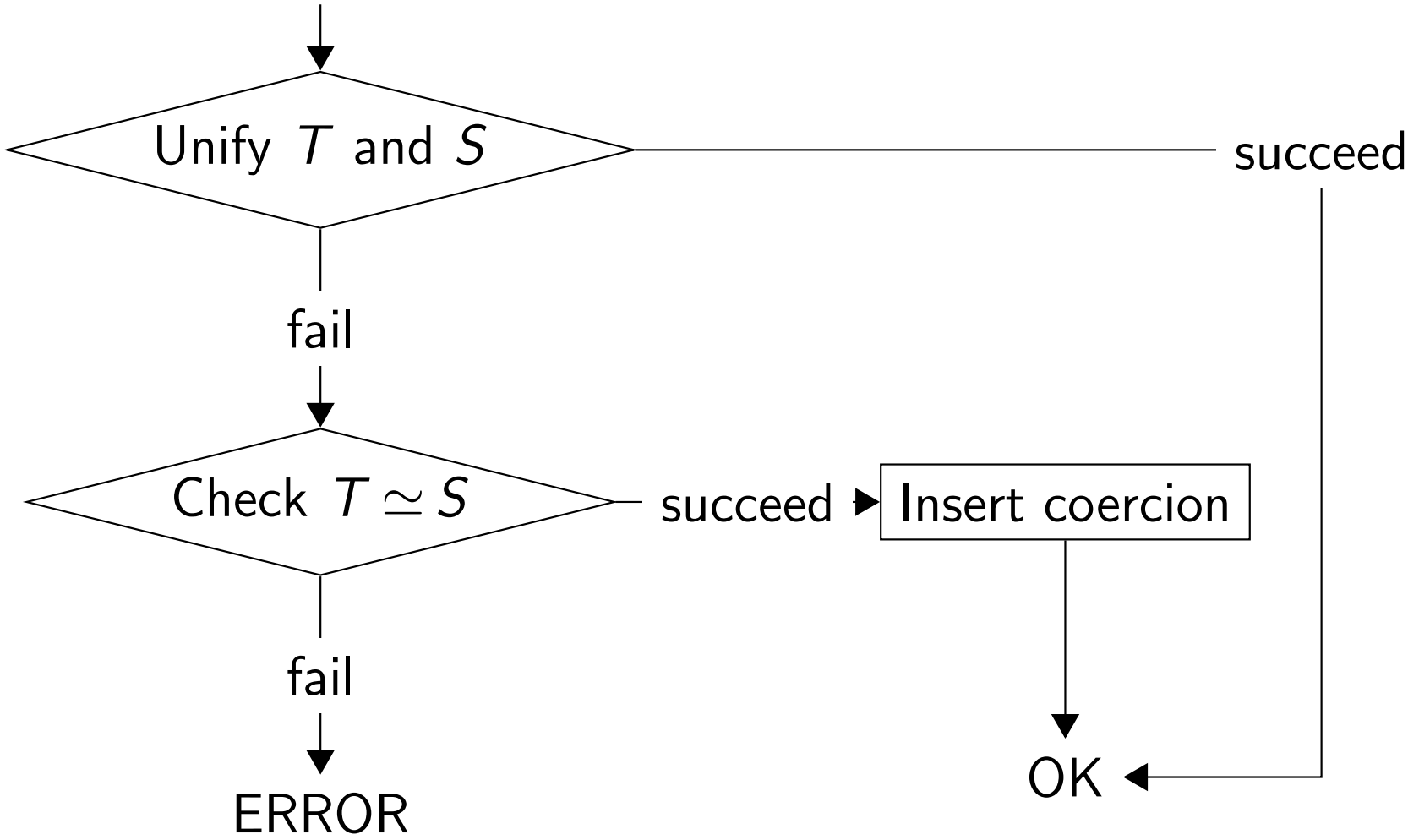
Coercions

Type Inference & Coercion Insertion

```
# ...
val f : {@ x of x > 0 @} -> int = <fun>
# let g x = f x;;
val g : {@ x of x > 0 @} -> int = <fun>
# g 0;;
(* ↓ Compiler inserts a coercion as follows *)
> g (0 :> {@ x of x > 0 @});;
Exception: Assert_failure ("_none_", 1, -1).
```

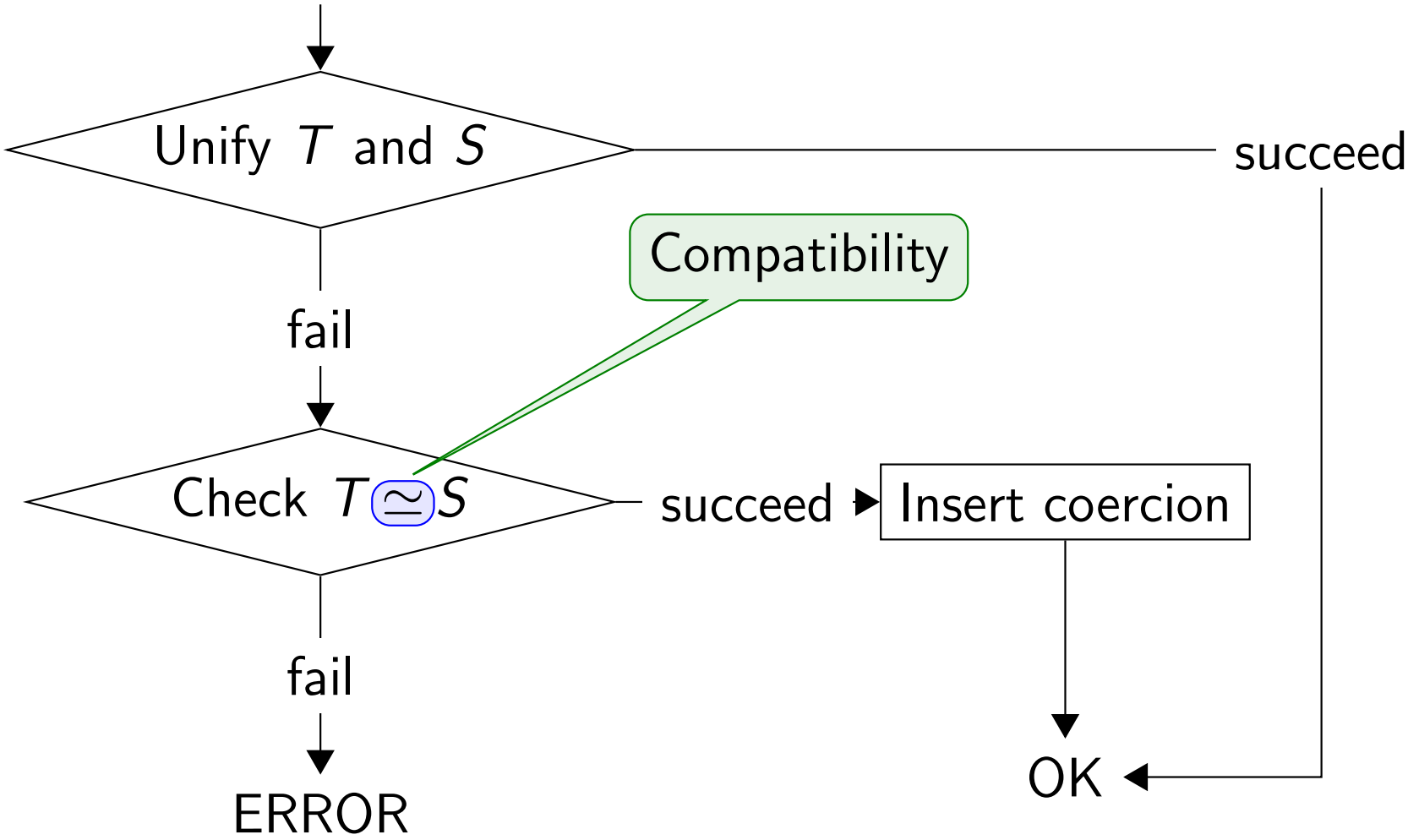
Type Checking for Function Application

T is the parameter type of a function. S is an actual argument type.



Type Checking for Function Application

T is the parameter type of a function. S is an actual argument type.



Compatibility

Two types are compatible if they are identical if we drop predicates

Examples of compatible types

- $\texttt{int} \simeq \texttt{int}$
- $\texttt{int} \simeq \{x{:}\texttt{int} \mid x > 0\}$
- $\{x{:}\texttt{int} \mid x < 0\} \simeq \{x{:}\texttt{int} \mid x > 0\}$
- $\{x{:}\texttt{int} \mid x \neq 0\} \to \texttt{bool} \quad \simeq \quad \texttt{int} \to \texttt{bool}$

Compatibility

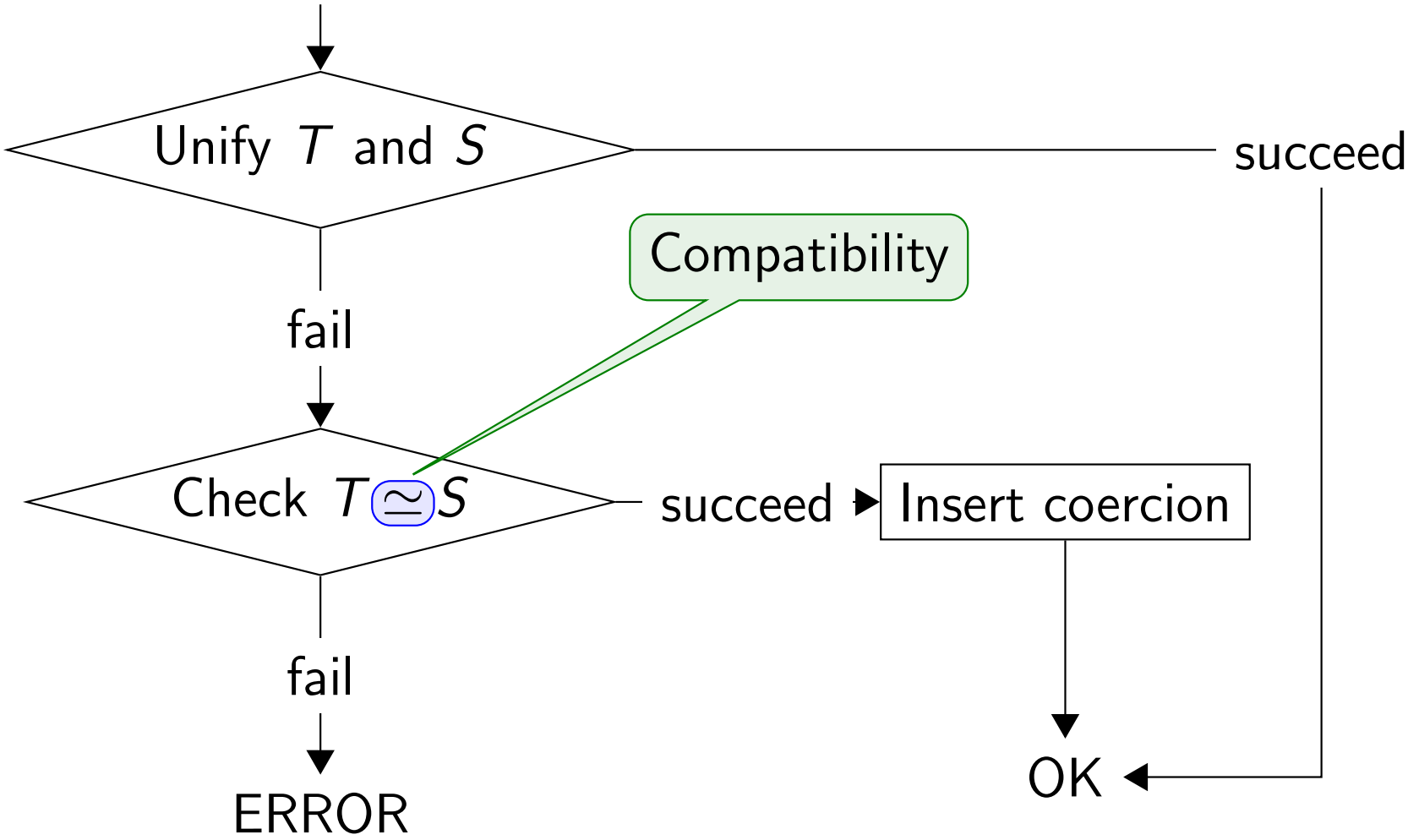
Two types are compatible if they are identical if we drop predicates

Examples of compatible types

- $\texttt{int} \simeq \texttt{int}$
- $\texttt{int} \simeq \texttt{int}$
- $\texttt{int} \simeq \texttt{int}$
- $\texttt{int} \to \texttt{bool} \simeq \texttt{int} \to \texttt{bool}$

Type Checking for Function Application

T is the parameter type of a function. S is an actual argument type.



Inference Example

```
let f (x : {@ x of x > 0 @}) = x;;
let g (x : {@ x of x > 1 @}) = x;;
let h  y  = ( f y , g y);;
```

1. Assign a fresh type variable α
2. Unify α and {@ x of x > 0 @}
   - Succeed & α becomes {@ x of x > 0 @}
3. Unify {@ x of x > 0 @} and {@ x of x > 1 @}
   - Fail
4. Check {@ x of x > 0 @} $\simeq$ {@ x of x > 1 @}
   - Succeed & Insert a coercion into g y

Inference Example

```
let f (x : {@ x of x > 0 @}) = x;;
let g (x : {@ x of x > 1 @}) = x;;
let h y = ( f y , g y);;
```

1. Assign a fresh type variable α
2. Unify α and {@ x of x > 0 @}
   - Succeed & α becomes {@ x of x > 0 @}
3. Unify {@ x of x > 0 @} and {@ x of x > 1 @}
   - Fail
4. Check {@ x of x > 0 @} $\simeq$ {@ x of x > 1 @}
   - Succeed & Insert a coercion into g y

Inference Example

```
let f (x : {@ x of x > 0 @}) = x;;
let g (x : {@ x of x > 1 @}) = x;;
let h  y  = (f y, g y);;
```

1. Assign a fresh type variable α
2. Unify α and {@ x of x > 0 @}
   - Succeed & α becomes {@ x of x > 0 @}
3. Unify {@ x of x > 0 @} and {@ x of x > 1 @}
   - Fail
4. Check {@ x of x > 0 @} $\simeq$ {@ x of x > 1 @}
   - Succeed & Insert a coercion into g y

Inference Example

```
let f (x : {@ x of x > 0 @}) = x;;
let g (x : {@ x of x > 1 @}) = x;;
let h (y : {@ x of x > 0 @}) = (f y,  g y );;
```

1. Assign a fresh type variable α
2. Unify α and {@ x of x > 0 @}
   - Succeed & α becomes {@ x of x > 0 @}
3. Unify {@ x of x > 0 @} and {@ x of x > 1 @}
   - Fail
4. Check {@ x of x > 0 @} $\simeq$ {@ x of x > 1 @}
   - Succeed & Insert a coercion into g y

Inference Example

```
let f (x : {@ x of x > 0 @}) = x;;
let g (x : {@ x of x > 1 @}) = x;;
let h (y : {@ x of x > 0 @}) = (f y, g y);;
```

1. Assign a fresh type variable α
2. Unify α and {@ x of x > 0 @}
   - Succeed & α becomes {@ x of x > 0 @}
3. Unify {@ x of x > 0 @} and {@ x of x > 1 @}
   - Fail
4. Check {@ x of x > 0 @} $\simeq$ {@ x of x > 1 @}
   - Succeed & Insert a coercion into g y

Inference Example

```
let f (x : {@ x of x > 0 @}) = x;;
let g (x : {@ x of x > 1 @}) = x;;
let h (y : {@ x of x > 0 @}) = (f y, g y);;
```

1. Assign a fresh type variable α
2. Unify α and {@ x of x > 0 @}
   - Succeed & α becomes {@ x of x > 0 @}
3. Unify {@ x of x > 0 @} and {@ x of x > 1 @}
   - Fail
4. Check {@ x of x > 0 @} $\simeq$ {@ x of x > 1 @}
   - Succeed & Insert a coercion into g y

Inference Example

```
let f (x : {@ x of x > 0 @}) = x;;
let g (x : {@ x of x > 1 @}) = x;;
let h (y : {@ x of x > 0 @}) = (f y, g y);;
```

1. Assign a fresh type variable α
2. Unify α and {@ x of x > 0 @}
   - Succeed & α becomes {@ x of x > 0 @}
3. Unify {@ x of x > 0 @} and {@ x of x > 1 @}
   - Fail
4. Check {@ x of x > 0 @} $\simeq$ {@ x of x > 1 @}
   - Succeed & Insert a coercion into g y

Inference Example

```
let f (x : {@ x of x > 0 @}) = x;;
let g (x : {@ x of x > 1 @}) = x;;
let h (y : {@ x of x > 0 @}) =
 (f y, g (y :> {@ x of x > 1 @}));;
```

1. Assign a fresh type variable α
2. Unify α and {@ x of x > 0 @}
   - Succeed & α becomes {@ x of x > 0 @}
3. Unify {@ x of x > 0 @} and {@ x of x > 1 @}
   - Fail
4. Check {@ x of x > 0 @} $\simeq$ {@ x of x > 1 @}
   - Succeed & Insert a coercion into g y

Refinement Types

Type Inference & Coercion Insertion

Coercions

Coercions

A coercion checks contracts at run-time

Some coercions are rejected at compile-time

`(0 :> bool)`, `(true :> int -> int)`

Compilation of a Coercion

A coercion is compiled by following the operational semantics of contract calculus [Findler and Felleisen ICFP 2002]

A compilation example

```
(0 :> {@ x of x > 0 @} )
```

↓ compiled into

```
( fun x -> if x > 0 then x else assert false ) 0
```

Coercion into a function type creates a wrapped function

(e : S_1 -> S_2 :> T_1 -> T_2)

↓ compiled into

(fun f x -> (f (x : T_1 :> S_1) : S_2 :> T_2)) e

Compilation of a Coercion

A coercion is compiled by following the operational semantics of contract calculus [Findler and Felleisen ICFP 2002]

A compilation example

```
(0 :> {@ x of x > 0 @})
```

↓ compiled into

```
(fun x -> if x > 0 then x else assert false) 0
```

Coercion into a function type creates a wrapped function

```
(e : S1 -> S2 :> T1 -> T2)
```

↓ compiled into

```
(fun f x -> (f (x : T1 :> S1) : S2 :> T2)) e
```

Conclusion

We implement a manifest system extension to OCaml

- few restrictions on predicates for expressiveness
- type inference and coercion insertion
- simple implementation: modified 500 LOC for now

Future work

- Better type inference
- Dependent function types
- Formal correctness of our implementation
- Other OCaml features