Contracts in Trinity

Andres Löh

joint work with Ralf Hinze and Andreas Schmitz

Universität Bonn

April 13, 2007

About me

- PhD at Utrecht University, 2004: "Exploring Generic Haskell"
- currently PostDoc at Bonn University, working with Ralf Hinze
- interests:
  - functional programming (Haskell),
  - polytypic / datatype-generic programming,
  - type systems

Overview

History of Trinity

- While teaching “Concepts of Programming Languages” to third- and fourth-year students, Ralf Hinze devised fragments of a language together with static and dynamic semantics.
- An idea came up at Bonn university to redesign the curriculum and have an introductory first-year course on PL concepts.
- Another idea came up that while it would be ok to reuse the work already done for the other course, it would be extremely nice to have an implementation for the students to play with, to make the course less theoretical.

History of Trinity – contd.

- I joined the project at that point. With the implementation came a redesign of most language concepts.
- By now, the course has passed, with mixed reactions from the students. The language is still in development and will probably be used for other courses and projects.
- This talk is also about one of these projects: A master student is currently working on adding contracts to Trinity.

Design goals of Trinity

- Different paradigms:
  - value-oriented (functional) programming
  - effect-oriented (imperative) programming
  - object-oriented programming

Design goals of Trinity

- Different paradigms:
  - value-oriented (functional) programming
  - effect-oriented (imperative) programming
  - object-oriented programming
- Simple, orthogonal concepts. No artificial restrictions.

Design goals of Trinity

- Different paradigms:
  - value-oriented (functional) programming
  - effect-oriented (imperative) programming
  - object-oriented programming
- Simple, orthogonal concepts. No artificial restrictions.
- Minimalistic. (Writing large programs is not a goal.)

Design goals of Trinity

- Different paradigms:
  - value-oriented (functional) programming
  - effect-oriented (imperative) programming
  - object-oriented programming
- Simple, orthogonal concepts. No artificial restrictions.
- Minimalistic. (Writing large programs is not a goal.)
- Types!

Design goals of Trinity

- Different paradigms:
  - value-oriented (functional) programming
  - effect-oriented (imperative) programming
  - object-oriented programming
- Simple, orthogonal concepts. No artificial restrictions.
- Minimalistic. (Writing large programs is not a goal.)
- Types!
- Clearly defined static and dynamic semantics.

Design goals of Trinity

- Different paradigms:
  - value-oriented (functional) programming
  - effect-oriented (imperative) programming
  - object-oriented programming
- Simple, orthogonal concepts. No artificial restrictions.
- Minimalistic. (Writing large programs is not a goal.)
- Types!
- Clearly defined static and dynamic semantics.
- Presentable in an incremental way.

The Result

- We reinvented ML . . .

The Result

- We reinvented ML . . .
- . . . with some syntactic influences from Haskell.

The Result

- We reinvented ML . . .
- . . . with some syntactic influences from Haskell.
- . . . with some variations in the type system.

The Result

- We reinvented ML . . .
- . . . with some syntactic influences from Haskell.
- . . . with some variations in the type system.
- Natural numbers are the only built-in numerical type.

Example – factorial

```
function factorial (n : Nat) : Nat =
  if n == 0 then 1
            else  n * factorial (n − 1)
```

Example – factorial

```
function factorial (n : Nat) : Nat =
  if n == 0 then 1
           else  n * factorial (n − 1)
```

- Nat, not Int . . .
- Limited type inference: type annotations required for **all** recursive functions.

Example – factorial, imperatively

```
local
  open System.Control
in
  function factorial (n : Nat) : Nat =
    let
      val result = ref 1
    in
      for (1, n) (fun i ⇒ result := ! result ∗ i);
      ! result
    end
end
```

Example – factorial, imperatively

```
local
  open System.Control
in
  function factorial (n : Nat) : Nat =
    let
      val result = ref 1
    in
      for (1, n) (fun i ⇒ result := ! result ∗ i);
      ! result
    end
end
```

- Modules and references similar to ML.
- ‘for’ is a function defined in System.Control.

Example – data types

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Tree}\,\langle \mathsf{a} \rangle = \mathsf{Empty} \\
&\qquad\qquad\quad\ \mid \mathsf{Node}\ (\mathsf{Tree}\,\langle \mathsf{a} \rangle, \mathsf{a}, \mathsf{Tree}\,\langle \mathsf{a} \rangle) \\
&\textbf{data}\ \mathsf{Maybe}\,\langle \mathsf{a} \rangle = \mathsf{Nothing} \\
&\qquad\qquad\qquad\ \mid \mathsf{Just}\ \mathsf{a}
\end{aligned}
$$

Example – data types

$$
\begin{aligned}
\mathbf{data}\ \mathsf{Tree}\,\langle \mathsf{a}\rangle &= \mathsf{Empty}\\
&\mid\ \mathsf{Node}\ (\mathsf{Tree}\,\langle \mathsf{a}\rangle, \mathsf{a}, \mathsf{Tree}\,\langle \mathsf{a}\rangle)\\
\mathbf{data}\ \mathsf{Maybe}\,\langle \mathsf{a}\rangle &= \mathsf{Nothing}\\
&\mid\ \mathsf{Just}\ \mathsf{a}
\end{aligned}
$$

- Haskell-inspired syntax.
- Constructors have zero or one argument.
- Type application: generally angle brackets.

Example – polymorphism

```
function leaf ⟨a : Type⟩ (elem : a) = Node (Empty, elem, Empty)
val a-bst =
  Node (Node (leaf ("Andres", "U Bonn"),
              ("John", "U Chicago"),
              leaf ("Matthias", "TTI-C")),
        ("Ralf", "U Bonn"),
        leaf  ("Robby", "U Chicago"))
```

Example – polymorphism

function leaf $\langle a : \mathrm{Type} \rangle$ (elem : a) $=$ Node (Empty, elem, Empty)

val a-bst $=$

```
Node (Node (leaf ("Andres", "U Bonn"),
            ("John", "U Chicago"),
            leaf ("Matthias", "TTI-C")),
      ("Ralf", "U Bonn"),
      leaf  ("Robby", "U Chicago"))
```

- Polymorphism is introduced explicitly via type abstractions (but constructors are implicitly polymorphic).
- leaf : $\langle a : \mathrm{Type} \rangle \to a \to \mathrm{Tree}\ \langle a \rangle$
- If a polymorphic functions is applied to a value, missing type arguments are inferred.
- Equivalent to the value restriction.

Type abbreviations

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Environment}\,\langle \mathsf{a}, \mathsf{b}\rangle &&= \mathsf{List}\,\langle (\mathsf{a}, \mathsf{b})\rangle \\
&\textbf{type}\ \mathsf{Int} &&= (\mathsf{Nat}, \mathsf{Nat})
\end{aligned}
$$

Type abbreviations

$$\begin{aligned}
&\textbf{type}\ \mathsf{Environment}\,\langle \mathsf{a}, \mathsf{b}\rangle && = \mathsf{List}\,\langle(\mathsf{a}, \mathsf{b})\rangle \\
&\textbf{type}\ \mathsf{Int} && = (\mathsf{Nat}, \mathsf{Nat})
\end{aligned}$$

- No new data types are generated.
- No recursion.

Other features

- Simple IO functions
- Records
- Arrays
- Exceptions
- Continuations
- Objects
- Modules, Signatures, Functors (not as advanced as in ML)

Overview

Motivation

An important criterion for the quality of software is **reliability**:

- **correctness:** the software does what it is supposed to do
- **robustness:** the software can deal with unexpected situations

Motivation

An important criterion for the quality of software is **reliability**:

- **correctness:** the software does what it is supposed to do
- **robustness:** the software can deal with unexpected situations

There are different approaches in order to improve the reliability of software:

- formal proof of correctness,
- type systems (static, dynamic),
- systematic testing,
- "design by contract".

Motivation

An important criterion for the quality of software is **reliability**:

- **correctness:** the software does what it is supposed to do
- **robustness:** the software can deal with unexpected situations

There are different approaches in order to improve the reliability of software:

- formal proof of correctness,
- type systems (static, dynamic),
- systematic testing,
- "design by contract".

These approaches are not competing. They can be used simultaneously.

Design space

	static checking	**dynamic checking**
simple properties	static types	dynamic types
complex properties	theorem proving	contracts

Contracts in Trinity

- Contracts are integrated into the type system.
- Types have a static and a dynamic component.
- Contract types are translated into run-time checks.
- Contracts can be applied to higher-order functions and to polymorphic functions.
- Abstractions can be defined.

Syntax: predicate contracts

A contract specifies a desired property. For example:

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Pos} &&= \{\, \mathsf{i} : \mathsf{Nat} \mid \mathsf{i} \gneq 0 \,\} \\
&\textbf{type}\ \mathsf{True}\,\langle \mathsf{a} \rangle &&= \{\, \_ : \mathsf{a} \mid \mathsf{true} \,\} \\
&\textbf{type}\ \mathsf{Nonempty}\,\langle \mathsf{a} \rangle &&= \{\, \mathsf{x} : \mathsf{List}\,\langle \mathsf{a} \rangle \mid \mathsf{length}\ \mathsf{x} \not= 0 \,\}
\end{aligned}
$$

Syntax: predicate contracts

A contract specifies a desired property. For example:

$$\begin{aligned}
&\textbf{type}\ \mathsf{Pos} &&= \{\, \mathsf{i} : \mathsf{Nat} \mid \mathsf{i} \gneq 0 \,\} \\
&\textbf{type}\ \mathsf{True}\,\langle \mathsf{a} \rangle &&= \{\, _ : \mathsf{a} \mid \mathsf{true} \,\} \\
&\textbf{type}\ \mathsf{Nonempty}\,\langle \mathsf{a} \rangle &&= \{\, \mathsf{x} : \mathsf{List}\,\langle \mathsf{a} \rangle \mid \mathsf{length}\ \mathsf{x} \neq 0 \,\}
\end{aligned}$$

Formation rule for **predicate contracts**:

$$\frac{\Sigma \vdash \tau : \mathsf{Type} \qquad \Sigma, \mathsf{x} : \tau \vdash \mathsf{e} : \mathsf{Bool}}{\Sigma \vdash \{\, \mathsf{x} : \tau \mid \mathsf{e} \,\} : \mathsf{Type}}$$

Parameterized contracts

Type synonyms now also be parameterized over values.

$$\mathbf{type}\ \mathsf{Between}\ (\mathsf{m} : \mathsf{Nat})\ (\mathsf{n} : \mathsf{Nat}) = \{\, \mathsf{x} : \mathsf{Nat} \mid \mathsf{m} \leqslant \mathsf{x}\ \&\&\ \mathsf{x} \leqslant \mathsf{n} \,\}$$

Recall: we always use angle brackets for **type application**, and no brackets for **expression application**.

Syntax: assigning contracts

We can assert a contract by annotating an expression:

$$\begin{array}{l}
\textbf{function}\ \mathsf{factors}\ \mathsf{n} = \mathsf{filter}\ (\textbf{fun}\ \mathsf{i} \Rightarrow \mathsf{n}\ \%\ \mathsf{i} == 0)\ (\mathsf{between}\ (1, \mathsf{n})) \\
\textbf{type}\ \mathsf{Prime} = \{\, \mathsf{n} : \mathsf{Nat} \mid \mathsf{eqList}\ (\textbf{fun}\ \mathsf{x}\ \mathsf{y} \Rightarrow \mathsf{x} == \mathsf{y}) \\
\qquad\qquad\qquad\qquad\qquad (\mathsf{factors}\ \mathsf{n})\ (\mathsf{Cons}\ (1, \mathsf{Cons}\ (\mathsf{n}, \mathsf{Nil})))\,\} \\
\textbf{val}\ \mathsf{mersenne} = \mathsf{power}\ (2, 30402457) - 1 : \mathsf{Prime}
\end{array}$$

Static and dynamic checking

Each type has a static and a dynamic part. For a predicate contract such as

$$\textbf{type}\ \mathsf{Prime} = \{\, \mathsf{n} : \mathsf{Nat} \mid \mathsf{eqList}\ (\textbf{fun}\ \mathsf{x}\ \mathsf{y} \Rightarrow \mathsf{x} == \mathsf{y})\ (\mathsf{factors}\ \mathsf{n})\ (\mathsf{Cons}\ (1, \mathsf{Cons}\ (\mathsf{n}, \mathsf{Nil})))\,\}$$

the static part is Nat.

Static and dynamic checking

Each type has a static and a dynamic part. For a predicate contract such as

```
type Prime = { n : Nat | eqList (fun x y ⇒ x == y)
                                (factors n) (Cons (1, Cons (n, Nil))) }
```

the static part is Nat.

The dynamic part is a **code transformation** that wraps the expression in a run-time test:

```
power (2, 30402457) − 1
```

is transformed into

```
(fun n ⇒ if eqList (fun x y ⇒ x == y)
                   (factors n) (Cons (1, Cons (n, Nil))))
         then n
         else throw Contract)
  (power (2, 30402457) − 1)
```

Syntax: contracts on functions

Contracts can be embedded into type expressions, for example into function types:

> **type** $\mathsf{F}\ \langle \mathsf{a} \rangle = \mathsf{Nonempty}\ \langle \mathsf{a} \rangle \rightarrow \mathsf{Pos}$

A function with type $\mathsf{F}\ \langle \mathsf{a} \rangle$ requires its argument to be a non-empty list with element of type a and ensures that its result is a positive number; Nonempty is the **precondition**, Pos the **postcondition**.

Syntax: contracts on functions – contd.

The postcondition may depend on the function argument:

$$\textbf{type}\ \mathsf{Inc} = \textbf{forall}\ (\mathsf{n} : \mathsf{Nat}) \Rightarrow \{\, \mathsf{r} : \mathsf{Nat} \mid \mathsf{n} \lneq \mathsf{r} \,\}$$

The variable n is bound in the construct and may be used in predicate contracts to the right.

The postcondition may depend on the function argument:

$\textbf{type}\ \mathsf{Inc} = \textbf{forall}\ (\mathsf{n} : \mathsf{Nat}) \Rightarrow \{\, \mathsf{r} : \mathsf{Nat} \mid \mathsf{n} \lneq \mathsf{r} \,\}$

The variable n is bound in the construct and may be used in predicate contracts to the right.

Formation rule for **dependent function contracts**:

$$\frac{\Sigma \vdash \tau : \mathsf{Type} \qquad \Sigma, \mathsf{x} : \tau \vdash \tau' : \mathsf{Type}}{\Sigma \vdash \textbf{forall}\ (\mathsf{x} : \tau) \Rightarrow \tau' : \mathsf{Type}}$$

Contracts: obligations, benefits, violations

A function contract $\tau_1 \longrightarrow \tau_2$ is like a business contract, with obligations and benefits for both parties.

party	obligations	benefits
client	ensure precondition τ_1	require postcondition τ_2
supplier	ensure postcondition τ_2	require precondition τ_1

The obligations of one party are the benefits of the other.

Contracts: obligations, benefits, violations

A function contract $\tau_1 \rightarrow \tau_2$ is like a business contract, with obligations and benefits for both parties.

party	obligations	benefits
client	ensure precondition τ_1	require postcondition τ_2
supplier	ensure postcondition τ_2	require precondition τ_1

The obligations of one party are the benefits of the other.

If a contract is violated at runtime, the software is erroneous.

If the **precondition** is violated, the **client is to blame**.
If the **postcondition** is violated, the **supplier is to blame**.

Contract violations: first-order functions

$\textbf{type}\ \mathsf{PosInc} = \textbf{forall}\ (\mathsf{n} : \mathsf{Pos}) \Rightarrow \{\ \mathsf{r} : \mathsf{Pos} \mid \mathsf{n} \lneq \mathsf{r}\ \}$

$\textbf{val}\ \mathsf{inc}\ = (\textbf{fun}\ \mathsf{n} \Rightarrow \mathsf{n} + 1) : \mathsf{PosInc}$
$\textbf{val}\ \mathsf{dec} = (\textbf{fun}\ \mathsf{n} \Rightarrow \mathsf{n} - 1) : \mathsf{PosInc}$

Contract violations: first-order functions

$\textbf{type}\ \text{PosInc} = \textbf{forall}\ (\text{n} : \text{Pos}) \Rightarrow \{\ \text{r} : \text{Pos} \mid \text{n} \lneq \text{r}\ \}$

$\textbf{val}\ \text{inc}\ = (\textbf{fun}\ \text{n} \Rightarrow \text{n} + 1) : \text{PosInc}$
$\textbf{val}\ \text{dec} = (\textbf{fun}\ \text{n} \Rightarrow \text{n} - 1) : \text{PosInc}$

Another possibility to define inc is

$\textbf{function}\ \text{inc}\ (\text{n} : \text{Pos}) : \{\ \text{r} : \text{Pos} \mid \text{n} \lneq \text{r}\ \} = \text{n} + 1$

Contract violations: first-order functions

$$\textbf{type}\ \text{PosInc} = \textbf{forall}\ (\text{n} : \text{Pos}) \Rightarrow \{\, \text{r} : \text{Pos} \mid \text{n} \lnsim \text{r} \,\}$$

$$\begin{aligned}
&\textbf{val}\ \text{inc}\ = (\textbf{fun}\ \text{n} \Rightarrow \text{n} + 1) : \text{PosInc} \\
&\textbf{val}\ \text{dec} = (\textbf{fun}\ \text{n} \Rightarrow \text{n} - 1) : \text{PosInc}
\end{aligned}$$

Another possibility to define inc is

$$\textbf{function}\ \text{inc}\ (\text{n} : \text{Pos}) : \{\, \text{r} : \text{Pos} \mid \text{n} \lnsim \text{r} \,\} = \text{n} + 1$$

Note: Contract violations are only detected if a value is **used** outside of its specification.

Function contracts versus flat contracts

It is possible to define flat function contracts:

$$\textbf{type}\ \mathsf{PreserveZero} = \{\mathsf{f} : \mathsf{Nat} \rightarrow \mathsf{Nat} \mid \mathsf{f}\ 0 == 0\}$$

Syntax: contracts in datatypes

In principle, contract types can be embedded arbitrarily in other types:

> List $\langle$Pos$\rangle$

describes a list of positive numbers. In general, this requires ‘mapping’ the assertion over the elements of arbitrary data structures (polytypic programming).

Syntax: contracts in datatypes

In principle, contract types can be embedded arbitrarily in other types:

> List ⟨Pos⟩

describes a list of positive numbers. In general, this requires 'mapping' the assertion over the elements of arbitrary data structures (polytypic programming).

Formation rule for **contract application**:

$$\frac{\Sigma \vdash \tau : \mathsf{Type} \to \kappa \qquad \Sigma \vdash \tau' : \mathsf{Type}}{\Sigma \vdash \tau\ \langle \tau' \rangle : \kappa}$$

Syntax: composing contracts

Contracts can be combined using "and":

$$\mathsf{Pos}\ \&\ \{\, \mathsf{n} : \mathsf{Nat} \mid \mathsf{n} \leqslant 4711 \,\}$$

Formation rule for **contract composition**:

$$\frac{\Sigma \vdash \tau : \mathsf{Type} \qquad \Sigma \vdash \tau' : \mathsf{Type}}{\Sigma \vdash \tau \,\&\, \tau' : \mathsf{Type}}$$

Overview

Example: factorization

Let f′ be the 'contracted' variant of f.

$$\textbf{val}\ \text{prime-factors}' = \text{prime-factors}$$
$$\quad : \textbf{forall}\ (\mathsf{n} : \mathsf{Pos}) \Rightarrow \mathsf{List}\ \langle \mathsf{Prime} \rangle\ \&\ \{\ \mathsf{fs} : \mathsf{List}\ \langle \mathsf{Nat} \rangle \mid \text{product fs} == \mathsf{n}\ \}$$

Example: factorization

Let f' be the 'contracted' variant of f.

$$\begin{aligned}&\textbf{val}\ \text{prime-factors}' = \text{prime-factors}\\&\quad : \textbf{forall}\ (n : \mathrm{Pos}) \Rightarrow \mathrm{List}\,\langle \mathrm{Prime}\rangle\ \&\ \{\, fs : \mathrm{List}\,\langle \mathrm{Nat}\rangle \mid \text{product}\ fs == n \,\}\end{aligned}$$

The function prime-factors is an inverse of product. This idiom can be captured using a 'higher-order' function:

$$\begin{aligned}&\textbf{type}\ \mathrm{Inverse}\,\langle a, b\rangle\ (f : a \rightarrow b)\ (eq : b \rightarrow b \rightarrow b) =\\&\quad \textbf{forall}\ (x : b) \Rightarrow \{\, y : a \mid eq\ (f\ y)\ x \,\}\\&\textbf{val}\ \text{prime-factors}' = \text{prime-factors}\\&\quad : \mathrm{Pos} \rightarrow (\mathrm{List}\,\langle \mathrm{Prime}\rangle\ \&\ \mathrm{Inverse}\ \text{product}\ (\textbf{fun}\ x\ y \Rightarrow x == y))\end{aligned}$$

Example: until

Polymorphic functions such as until do not need to be treated in any special way:

function until $\langle a \rangle$ (p : $a \rightarrow$ Bool) (f : $a \rightarrow a$) (a : a) : a =
  if p a **then** a **else** until p f (f a)

Type arguments can be inferred, but can also be explicitly supplied.
A polymorphic function can therefore be instantiated with a contract type (an invariant).

Example: until

Polymorphic functions such as until do not need to be treated in any special way:

$\textbf{function}\ \mathsf{until}\ \langle a \rangle\ (p : a \rightarrow \mathsf{Bool})\ (f : a \rightarrow a)\ (a : a) : a =$
$\quad \textbf{if}\ p\ a\ \textbf{then}\ a\ \textbf{else}\ \mathsf{until}\ p\ f\ (f\ a)$

Type arguments can be inferred, but can also be explicitly supplied.
A polymorphic function can therefore be instantiated with a contract type (an invariant).

The expression

$\mathsf{until}\ \langle \mathsf{Pos} \rangle$

is equivalent to

$\mathsf{until}\ \langle \mathsf{Nat} \rangle : (\mathsf{Pos} \rightarrow \mathsf{Bool}) \rightarrow (\mathsf{Pos} \rightarrow \mathsf{Pos}) \rightarrow \mathsf{Pos} \rightarrow \mathsf{Pos}$

Type rules for contracts

Type-checking introduces run-time contract checks, therefore type rules are of the form:

$$\Sigma \vdash e : \sigma \rightsquigarrow e'$$

where σ is a **static type**, i.e., it does not contain any contract constructs.

Type rules for contracts

Type-checking introduces run-time contract checks, therefore type rules are of the form:

$$\Sigma \vdash \mathsf{e} : \sigma \leadsto \mathsf{e}'$$

where σ is a **static type**, i.e., it does not contain any contract constructs.

We use two built-in functions:

- **static** computes the "static part" of a type
- **assert** computes an expression that asserts a contract

Type rules for contract – contd.

$$\frac{\textbf{static}\ \langle\tau\rangle = \sigma \qquad \Sigma \vdash \mathsf{e} : \sigma \leadsto \mathsf{e}'}{\Sigma \vdash (\mathsf{e} : \tau) : \sigma \leadsto \textbf{assert}\ \langle\tau\rangle\ \mathsf{e}'}$$

Type rules for contract – contd.

$$\frac{\textbf{static}\ \langle \tau \rangle = \sigma \qquad \Sigma \vdash \mathsf{e} : \sigma \leadsto \mathsf{e}'}{\Sigma \vdash (\mathsf{e} : \tau) : \sigma \leadsto \textbf{assert}\ \langle \tau \rangle\ \mathsf{e}'}$$

$$\frac{\textbf{static}\ \langle \tau \rangle = \sigma \qquad \Sigma \vdash \mathsf{e} : \langle \mathsf{a} : \mathsf{Type} \rangle \rightarrow \sigma' \leadsto \mathsf{e}'}{\Sigma \vdash \mathsf{e}\,\langle \tau \rangle : \sigma'\ [\mathsf{a} \mapsto \sigma] \leadsto \textbf{assert}\ \langle \sigma'\ [\mathsf{a} \mapsto \tau] \rangle\ (\mathsf{e}'\,\langle \sigma \rangle)}$$

Type rules for contract – contd.

$$\frac{\textbf{static}\ \langle\tau\rangle = \sigma \qquad \Sigma \vdash \mathsf{e} : \sigma \leadsto \mathsf{e}'}{\Sigma \vdash (\mathsf{e} : \tau) : \sigma \leadsto \textbf{assert}\ \langle\tau\rangle\ \mathsf{e}'}$$

$$\frac{\textbf{static}\ \langle\tau\rangle = \sigma \qquad \Sigma \vdash \mathsf{e} : \langle \mathsf{a} : \mathsf{Type}\rangle \rightarrow \sigma' \leadsto \mathsf{e}'}{\Sigma \vdash \mathsf{e}\,\langle\tau\rangle : \sigma'\ [\mathsf{a} \mapsto \sigma] \leadsto \textbf{assert}\ \langle\sigma'\ [\mathsf{a} \mapsto \tau]\rangle\ (\mathsf{e}'\,\langle\sigma\rangle)}$$

$$\begin{aligned}
\textbf{static}\ &\langle \mathsf{Nat}\rangle &&= \mathsf{Nat} \\
\textbf{static}\ &\langle \{\, \mathsf{x} : \tau \mid \mathsf{e}\,\}\rangle &&= \tau \\
\textbf{static}\ &\langle \textbf{forall}\ (\mathsf{x} : \tau) \Rightarrow \tau'\rangle &&= \textbf{static}\ \langle\tau\rangle \rightarrow \textbf{static}\ \langle\tau'\rangle
\end{aligned}$$

Type rules for contract – contd.

$$\frac{\textbf{static}\ \langle\tau\rangle = \sigma \qquad \Sigma \vdash \mathsf{e} : \sigma \leadsto \mathsf{e}'}{\Sigma \vdash (\mathsf{e} : \tau) : \sigma \leadsto \textbf{assert}\ \langle\tau\rangle\ \mathsf{e}'}$$

$$\frac{\textbf{static}\ \langle\tau\rangle = \sigma \qquad \Sigma \vdash \mathsf{e} : \langle \mathsf{a} : \mathsf{Type}\rangle \to \sigma' \leadsto \mathsf{e}'}{\Sigma \vdash \mathsf{e}\ \langle\tau\rangle : \sigma'\ [\mathsf{a} \mapsto \sigma] \leadsto \textbf{assert}\ \langle\sigma'\ [\mathsf{a} \mapsto \tau]\rangle\ (\mathsf{e}'\ \langle\sigma\rangle)}$$

$$\begin{aligned}
&\textbf{static}\ \langle \mathsf{Nat} \rangle && = \mathsf{Nat} \\
&\textbf{static}\ \langle \{\, \mathsf{x} : \tau \mid \mathsf{e} \,\} \rangle && = \tau \\
&\textbf{static}\ \langle \textbf{forall}\ (\mathsf{x} : \tau) \Rightarrow \tau' \rangle && = \textbf{static}\ \langle\tau\rangle \to \textbf{static}\ \langle\tau'\rangle
\end{aligned}$$

$$\begin{aligned}
&\textbf{assert}\ \langle \mathsf{Nat} \rangle && = \mathsf{id} \\
&\textbf{assert}\ \langle \{\, \mathsf{x} : \tau \mid \mathsf{e} \,\} \rangle && = \textbf{fun}\ \mathsf{x} \Rightarrow \textbf{if}\ \mathsf{e}\ \textbf{then assert}\ \langle\tau\rangle\ \mathsf{x} \\
& && \qquad\qquad\qquad\quad \textbf{else}\ \ \textbf{throw}\ \mathsf{Contract} \\
&\textbf{assert}\ \langle \textbf{forall}\ (\mathsf{x} : \tau) \Rightarrow \tau' \rangle && = \textbf{fun}\ \mathsf{f}\ \mathsf{x} \Rightarrow \textbf{assert}\ \langle\tau'\rangle\ (\mathsf{f}\ (\textbf{assert}\ \langle\tau\rangle\ \mathsf{x}))
\end{aligned}$$

Overview

1. Trinity
   - Background
   - Examples
2. Contracts
   - Motivation
   - Syntax
   - Examples
   - Semantics
3. Conclusions

Conclusions

We have introduced a type system for contracts.

- Trinity is a very beautiful language,
- contracts are an integral part of Trinity (contracts have a much better status than for example in Eiffel),
- implemented (still ongoing work, but available on request),
- we can define our own abstractions,
- higher-order functions are handled in a natural way,
- polymorphic functions can be instantiated to invariants,
- data types can be treated generically,
- future work: perform some contract checks statically and thereby optimize the contracts,
- future work: formalize the metatheory of Trinity,
- future work: control effects in contracts

The End

$\{\, \mathsf{x} : () \mid \textbf{let function}\ \mathsf{r}\ ()\ : \mathsf{Bool} = \mathsf{put\text{-}line}\ \texttt{"Thank you"}; \mathsf{r}\ ()\ \textbf{in}\ \mathsf{r}\ ()\ \textbf{end} \,\}$

Example: sorting

$$\begin{aligned}&\textbf{function}\ \text{fast-sort}'\ \langle \text{a}\rangle\ (\text{cmp} : \text{a} \to \text{a} \to \text{Ordering})\\ &\qquad\qquad\qquad\quad : \text{List}\ \langle \text{a}\rangle \to \text{Sorted}\ \langle \text{a}\rangle\ \text{cmp} =\\ &\quad \text{fast-sort cmp}\end{aligned}$$

The contract Sorted restricts lists to sorted lists.

Example: sorting

$$\begin{aligned}
&\textbf{function}\ \text{fast-sort}'\ \langle \text{a} \rangle\ (\text{cmp} : \text{a} \rightarrow \text{a} \rightarrow \text{Ordering}) \\
&\qquad\qquad\qquad\quad : \text{List}\ \langle \text{a} \rangle \rightarrow \text{Sorted}\ \langle \text{a} \rangle\ \text{cmp} = \\
&\quad \text{fast-sort cmp}
\end{aligned}$$

The contract Sorted restricts lists to sorted lists.

We have not (yet) specified that the output list is a permutation of the input list.

Example: sorting, continued

Let bag : List $\langle a \rangle \rightarrow$ Bag $\langle a \rangle$ be a function that turns a list into a bag.

$$
\begin{array}{l}
\textbf{function}\ \text{fast-sort}'\ \langle \mathsf{a} \rangle\ (\mathsf{cmp} : \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Ordering}) \\
\quad :\ \textbf{forall}\ (\mathsf{x} : \mathsf{List}\ \langle \mathsf{a} \rangle) \Rightarrow \\
\qquad\quad (\ \ \mathsf{Sorted}\ \langle \mathsf{a} \rangle\ \mathsf{cmp} \\
\qquad\quad \&\ \{\ \mathsf{s} : \mathsf{List}\ \langle \mathsf{a} \rangle \mid \mathsf{eqBag}\ (\mathsf{cmp2eq}\ \mathsf{cmp})\ (\mathsf{bag}\ \mathsf{x})\ (\mathsf{bag}\ \mathsf{s})\ \}) \\
\quad = \text{fast-sort}\ \mathsf{cmp}
\end{array}
$$

Example: sorting, continued

Let bag : $\mathsf{List}\ \langle a \rangle \rightarrow \mathsf{Bag}\ \langle a \rangle$ be a function that turns a list into a bag.

$$
\begin{array}{l}
\textbf{function}\ \text{fast-sort}'\ \langle a \rangle\ (\text{cmp} : a \rightarrow a \rightarrow \text{Ordering}) \\
\quad :\ \textbf{forall}\ (x : \text{List}\ \langle a \rangle) \Rightarrow \\
\qquad\quad (\ \ \text{Sorted}\ \langle a \rangle\ \text{cmp} \\
\qquad\quad \&\ \{\ s : \text{List}\ \langle a \rangle \mid \text{eqBag}\ (\text{cmp2eq cmp})\ (\text{bag}\ x)\ (\text{bag}\ s)\ \}) \\
\quad = \text{fast-sort cmp}
\end{array}
$$

The function fast-sort does not change the number of occurrences of the elements. This idiom can again be captured by a 'higher-order' contract:

$$
\begin{array}{l}
\textbf{type}\ \text{Preserve}\ \langle a, b \rangle\ (\text{eq} : b \rightarrow b \rightarrow \text{Bool})\ (f : a \rightarrow b) = \\
\quad \textbf{forall}\ (x : a) \Rightarrow \{\ y : a \mid \text{eq}\ (f\ x)\ (f\ y)\ \} \\
\textbf{function}\ \text{fast-sort}'\ \langle a \rangle\ (\text{cmp} : a \rightarrow a \rightarrow \text{Ordering}) \\
\quad :\ \ (\text{List}\ \langle a \rangle \rightarrow \text{Sorted}\ \langle a \rangle)\ \&\ \text{Preserve}\ (\text{cmp2eq cmp})\ \text{bag} \\
\quad = \text{fast-sort cmp}
\end{array}
$$

Example: sorting, continued

Let bag : List ⟨a⟩ → Bag ⟨a⟩ be a function that turns a list into a bag.

$$
\begin{aligned}
&\textbf{function}\ \text{fast-sort}'\ \langle a\rangle\ (\text{cmp} : a \to a \to \text{Ordering})\\
&\quad :\ \textbf{forall}\ (x : \text{List}\ \langle a\rangle) \Rightarrow\\
&\qquad\quad (\ \ \text{Sorted}\ \langle a\rangle\ \text{cmp}\\
&\qquad\quad \&\ \{\, s : \text{List}\ \langle a\rangle \mid \text{eqBag}\ (\text{cmp2eq cmp})\ (\text{bag}\ x)\ (\text{bag}\ s)\,\})\\
&\quad = \text{fast-sort cmp}
\end{aligned}
$$

The function fast-sort does not change the number of occurrences of the elements. This idiom can again be captured by a 'higher-order' contract:

$$
\begin{aligned}
&\textbf{type}\ \text{Preserve}\ \langle a, b\rangle\ (\text{eq} : b \to b \to \text{Bool})\ (f : a \to b) =\\
&\quad \textbf{forall}\ (x : a) \Rightarrow \{\, y : a \mid \text{eq}\ (f\ x)\ (f\ y)\,\}\\
&\textbf{function}\ \text{fast-sort}'\ \langle a\rangle\ (\text{cmp} : a \to a \to \text{Ordering})\\
&\quad :\ (\text{List}\ \langle a\rangle \to \text{Sorted}\ \langle a\rangle)\ \&\ \text{Preserve}\ (\text{cmp2eq cmp})\ \text{bag}\\
&\quad = \text{fast-sort cmp}
\end{aligned}
$$

A weaker assertion: Preserve (cmp2eq cmp) length.

Example: sorting, continued

Alternatively, we can specify fast-sort using a trusted sorting function:

$$\begin{array}{l}
\textbf{type}\ \mathsf{Is}\ \langle \mathsf{a}, \mathsf{b} \rangle\ (\mathsf{eq} : \mathsf{b} \rightarrow \mathsf{b} \rightarrow \mathsf{Bool}) = \\
\quad \textbf{fun}\ (\mathsf{x} : \mathsf{a}) \Rightarrow \{\ \mathsf{y} : \mathsf{b} \mid \mathsf{eq}\ \mathsf{y}\ (\mathsf{f}\ \mathsf{x})\ \} \\
\textbf{function}\ \mathsf{fast\text{-}sort}'\ \langle \mathsf{a} \rangle\ (\mathsf{cmp} : \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Ordering}) \\
\qquad :\ \mathsf{Is}\ (\mathsf{cmp2eq}\ \mathsf{cmp})\ (\mathsf{trusted\text{-}sort}\ \langle \mathsf{a} \rangle) \\
\qquad = \mathsf{fast\text{-}sort}\ \mathsf{cmp}
\end{array}$$