Contracts and Types

Andres Löh[1]

joint work with Johan Jeuring[2], Ralf Hinze[1], Andreas Schmitz[1]

[1]Universität Bonn

[2]Universiteit Utrecht

March 2, 2007

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

Overview

"Design by contract" is a widely used method to design software in the object-oriented world.

Overview

"Design by contract" is a widely used method to design software in the object-oriented world.

In this talk: Apply the idea to functional programming, while paying attention to

- higher-order functions,
- algebraic data types,
- parametric (type-)polymorphism.

Overview

"Design by contract" is a widely used method to design software in the object-oriented world.

In this talk: Apply the idea to functional programming, while paying attention to

- higher-order functions,
- algebraic data types,
- parametric (type-)polymorphism.

Idea: we design a type system that includes contracts, but types have a static and a dynamic component.

Overview

"Design by contract" is a widely used method to design software in the object-oriented world.

In this talk: Apply the idea to functional programming, while paying attention to

- higher-order functions,
- algebraic data types,
- parametric (type-)polymorphism.

Idea: we design a type system that includes contracts, but types have a static and a dynamic component.
Note: in a sufficiently expressive functional language, contracts can also be implemented purely as a library.

Structure

Structure

1. Quick intro to BPL
2. Syntax of contracts
3. Examples
4. (Semantics)
5. Conclusions

Syntax: predicate contracts

A contract specifies a desired property. For example:

> **type** Pos = $\{\, \mathsf{i} : \mathsf{Nat} \mid \mathsf{i} \gneq 0 \,\}$

If e is a boolean expression (in which x of type τ may occur free), then $\{\, \mathsf{x} : \tau \mid \mathsf{e} \,\}$ is a contract, a so-called **predicate contract** or **flat contract**.

> **type** True $\langle \mathsf{a} \rangle = \{\, \_ : \mathsf{a} \mid \mathsf{true} \,\}$
> **type** Nonempty $\langle \mathsf{a} \rangle = \{\, \mathsf{x} : \mathsf{List}\ \langle \mathsf{a} \rangle \mid \mathsf{length}\ \mathsf{x} \not= 0 \,\}$

Parameterized contracts

Contracts can not also be parameterized over values.

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

Syntax: assigning contracts

We can assert a contract by annotating an expression:

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

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

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

> **type** $\mathrm{F}\ \langle \mathrm{a} \rangle = \mathrm{Nonempty}\ \langle \mathrm{a} \rangle \to \mathrm{Pos}$

A function with type $\mathrm{F}\ \langle \mathrm{a} \rangle$ 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**.

The postcondition may depend on the function argument:

> **type** $\mathrm{Inc} = \textbf{fun}\ (\mathrm{n} : \mathrm{Nat}) \Rightarrow \{\ \mathrm{r} : \mathrm{Nat} \mid \mathrm{n} \lneq \mathrm{r}\ \}$

Syntax: contracts on functions

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

$$\textbf{type}\ \mathsf{F}\ \langle \mathsf{a} \rangle = \mathsf{Nonempty}\ \langle \mathsf{a} \rangle \to \mathsf{Pos}$$

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**.

The postcondition may depend on the function argument:

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

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

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

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

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

Demo.

Contract violations: first-order functions

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

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

Demo.

Another possibility to define inc is

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

Contract violations: first-order functions

$$\textbf{type}\ \textcolor{red}{\mathsf{PosInc}} = \textbf{fun}\ (\mathsf{n} : \textcolor{red}{\mathsf{Pos}}) \Rightarrow \{\, \mathsf{r} : \textcolor{red}{\mathsf{Pos}} \mid \mathsf{n} \lneq \mathsf{r} \,\}$$

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

Demo.

Another possibility to define inc is

$$\textbf{function}\ \mathsf{inc}\ (\mathsf{n} : \textcolor{red}{\mathsf{Pos}}) : \{\, \mathsf{r} : \textcolor{red}{\mathsf{Pos}}\ (\mathsf{n} \lneq \mathsf{r}) \mid \ \} = \mathsf{n} + \textcolor{blue}{1}$$

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:

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

Syntax: other contracts

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

> List $\langle$Pos$\rangle$

describes a list of positive numbers.

Syntax: other contracts

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

$\mathsf{List}\,\langle \mathsf{Pos} \rangle$

describes a list of positive numbers.

Contracts can be combined using "and":

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

Note: We do not offer negation or disjunction.

Overview

Example: factorization

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

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

Example: factorization

Let f′ be the ‘contracted’ variant of f.

$$
\begin{array}{l}
\textbf{val}\ \text{prime-factors}' = \\
\quad \text{prime-factors} : \textbf{fun}\ (\mathsf{n} : \mathsf{Pos}) \Rightarrow \ (\ \mathsf{List}\ \langle \mathsf{Prime} \rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \&\ \{\, \mathsf{fs} : \mathsf{List}\ \langle \mathsf{Nat} \rangle \mid \text{product fs} == \mathsf{n} \,\})
\end{array}
$$

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

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

Example: sorting

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

The contract Sorted restricts lists to sorted lists.

Example: sorting

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

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

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

Example: sorting, continued

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

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

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

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

Example: sorting, continued

Let $\text{bag} : \text{List}\,\langle a\rangle \to \text{Bag}\,\langle a\rangle$ be a function that turns a list into a bag.

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

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

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

A weaker assertion: $\text{Preserve}\ (\text{cmp2eq cmp})\ \text{length}$.

Example: sorting, continued

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

$$
\begin{aligned}
&\textbf{function}\ \text{fast-sort}'\ \langle \text{a} \rangle (\text{cmp} : \text{a} \rightarrow \text{a} \rightarrow \text{Ordering}) \\
&\quad :\ \textbf{fun}\ (\text{x} : \text{List}\ \langle \text{a} \rangle) \Rightarrow \\
&\qquad\quad \{\, \text{s} : \text{List}\ \langle \text{a} \rangle \mid \text{eqList}\ (\text{cmp2eq cmp})\ \text{s}\ (\text{trusted-sort}\ \text{x}) \,\} \\
&\quad = \text{fast-sort cmp}
\end{aligned}
$$

Example: sorting, continued

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

$$
\begin{array}{l}
\textbf{function}\ \mathsf{fast\text{-}sort}'\ \langle \mathsf{a}\rangle(\mathsf{cmp} : \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Ordering}) \\
\quad :\ \textbf{fun}\ (\mathsf{x} : \mathsf{List}\ \langle \mathsf{a}\rangle) \Rightarrow \\
\qquad \{\, \mathsf{s} : \mathsf{List}\ \langle \mathsf{a}\rangle \mid \mathsf{eqList}\ (\mathsf{cmp2eq}\ \mathsf{cmp})\ \mathsf{s}\ (\mathsf{trusted\text{-}sort}\ \mathsf{x})\,\} \\
\quad = \mathsf{fast\text{-}sort}\ \mathsf{cmp}
\end{array}
$$

Another idiom:

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

Example: until

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

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

The function until can be instantiated with a contract type (an invariant).

Demo.

Overview

Overview

Conclusions

We have introduced a type system for contracts.

- contracts are an integral part of the programming language (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,
- it might be possible to perform some contract checks statically and thereby optimize the contracts (also see the paper on the Haskell library),
- open problems: control effects in contracts, implement disjunction.