An algebra of contracts

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
Email: ralf@informatik.uni-bonn.de
Homepage: http://www.informatik.uni-bonn.de/~ralf

August 2005

Joint work with Johan Jeuring and Andres Löh

(Pick up the slides at .../~ralf/talks.html#T43.)

Der Supergau

```
patronus > ghc --make Main.lhs
patronus > ./a.out
a.out: Prelude.head: empty list
patronus > fgrep head *lhs | wc
     102     889    7846
```

Using contracts

```
patronus > ghc --make Main.lhs
patronus > ./a.out
*** assertion "head: empty list" failed:
    the application at Stackless.lhs:182:68 is to blame.
```

Outline of the talk

- ✘ Syntax of contracts
- ✘ Contracts are not ...
- ✘ Semantics of contracts
- ✘ Examples
- ✘ Properties of contracts

Syntax: contract comprehensions

A contract specifies a desired property of an expression. For instance,

$$nat :: \mathit{Contract}\ \mathit{Int}$$
$$nat = \{\, i \mid i \geqslant 0 \,\}$$

restricts the value of an expression to the natural numbers.

☞ If x is a variable of type σ and e a Boolean expression, then $\{\, x \mid e \,\}$ is a contract of type $\mathit{Contract}\ \sigma$, a so-called contract comprehension.

$$nonempty :: \mathit{Contract}\ [\alpha]$$
$$nonempty = \{\, x \mid not\ (null\ x) \,\}$$

☞ Contracts are first-class citizens: they can be named, passed to functions etc.

Syntax: contract comprehensions—cont.

The two most extreme contracts are

$$
\begin{aligned}
&false, true :: Contract\ \alpha \\
&false = \{\, x \mid False \,\} \\
&true\ = \{\, x \mid True \,\}
\end{aligned}
$$

The contract $false$ is very demanding, in fact, too demanding, while $true$ is very liberal, in fact, some might say, too liberal.

Syntax: dependent function contracts

Using contract comprehensions we can define contracts for values of arbitrary types, including function types.

$$\{\, f \mid f\ 0 == 0 \,\} :: Contract\ (Int \longrightarrow Int)$$

However, since the formula is restricted to a Boolean expression, that is, a Haskell expression, we cannot state, for example

$$\{\, f \mid \forall n :: Int\,.\, n \geqslant 0 \Rightarrow f\ n \geqslant 0 \,\}$$ -- not valid

☞ Therefore, we introduce a new combinator that allows us to specify the domain and codomain of a function.

$$nat \mapsto nat$$

Syntax: dependent function contracts—cont.

Unfortunately, the new combinator is still not sufficient. Often we want to relate the argument to the result ("the result is greater than the argument"). To this end we generalise $e_1 \mapsto e_2$ to the dependent function contract $(x :: e_1) \mapsto e_2$.

$$(n :: nat) \mapsto \{\, r \mid n < r \,\}$$

☞ If x is a variable of type σ_1, e_1 is a contract of type $Contract\ \sigma_1$, e_2 is a contract of type $Contract\ \sigma_2$, then $(x :: e_1) \mapsto e_2$ is a contract of type $Contract\ (\sigma_1 \to \sigma_2)$.

$$(x :: nat) \mapsto \{\, r \mid r * r \leqslant x < (r+1) * (r+1) \,\}$$

Syntax: assertions

Contracts are attached to expressions using $assert$:

$$
\begin{array}{l}
head' :: [\alpha] \to \alpha \\
head' = assert\ \texttt{"head"} \\
\qquad\qquad (nonempty \mapsto true) \\
\qquad head
\end{array}
$$

The string "head" describes the expression, here $head$.

$$
\begin{array}{l}
isqrt' :: Int \to Int \\
isqrt' = assert\ \texttt{"isqrt"} \\
\qquad\qquad ((x :: nat) \mapsto \{\, r \mid r * r \leqslant x < (r+1) * (r+1) \,\}) \\
\qquad isqrt
\end{array}
$$

☞ Convention: f' is the 'contracted' version of f.

Blame assignment

$$inc' = assert\ \texttt{"inc"}\ (nat \to nat)\ (\lambda n \to n + 1)$$
$$dec' = assert\ \texttt{"dec"}\ (nat \to nat)\ (\lambda n \to n - 1)$$

Contracts may be violated. Who is to blame?

$Main\rangle \quad inc' \diamond_1\ 5$
6
$Main\rangle \quad inc' \diamond_1\ (-5)$
`*** assertion failed:  application ‘1’ is to blame.`
$Main\rangle \quad dec' \diamond_1\ 5$
4
$Main\rangle \quad dec' \diamond_1\ 0$
`*** assertion failed:  ‘dec’ is to blame.`

☞ $\diamond$ is function application made visible.

Syntax: dependent pair contracts

We also offer a dependent pair contract:

$$(n :: nat) \times (\{\, i \mid i \leqslant n \,\} \mapsto true)$$

☞ If x is a variable of type σ_1, e_1 is a contract of type $Contract\ \sigma_1$, e_2 is a contract of type $Contract\ \sigma_2$, then $(x :: e_1) \times e_2$ is a contract of type $Contract\ (\sigma_1, \sigma_2)$.

☞ If x does not appear in e_2, we can simply write $e_1 \times e_2$. In this case, the two components are independent.

Syntax: contracts for data types

In general, we need a contract combinator for every data type—we will see later that these combinators can be defined generically.

Here, we confine ourselves to lists

$$list\ nat$$
$$list\ (nat \mapsto nat)$$

☞ Using $list$ we cannot relate elements of a list. For this purpose we have to use contract comprehensions—which, on the other hand, cannot express $list\ (nat \mapsto nat)$.

Syntax: conjunctions

Finally, contracts may be combined using conjunction.

$$nat \mathbin{\&} \{\, n \mid n \leqslant 4711 \,\}$$

☞ We do not, however, offer negation or disjunction.

Outline of the talk

- ✔ Syntax of contracts
- ✘ Contracts are not ...
- ✘ Semantics of contracts
- ✘ Examples
- ✘ Properties of contracts

Contract are not assertions

GHC already offers assertions: $assert :: Bool \to \alpha \to \alpha$.

You can write, for example,

$$
\begin{aligned}
head\ x = assert \\
&\quad (not\ (null\ x)) \\
&\quad (\mathbf{case}\ x\ \mathbf{of}\ a : \_ \to a)
\end{aligned}
$$

However,

- assertions do not assign blame; if an assertion fails, the source location of the assertion is printed.
- assertions do not have structure; they are not type-directed.

Contracts are not dependent types

Contracts look like dependent types, but, of course, contracts are not types.

- Contracts are dynamically checked and
- they dynamically change the behaviour of the program:

$$g :: (Int \hookrightarrow Int) \hookrightarrow (Int \hookrightarrow Int)$$
$$g = assert\ \texttt{"g"}\ ((nat \mapsto nat) \mapsto true)\ (\lambda f \hookrightarrow f)$$

The function g is the identity, but

$$Main\rangle \quad g \diamond_1 (\lambda x \hookrightarrow x) \diamond_2 (-7)$$
```
*** assertion failed:  application ‘2’ is to blame.
```

Contracts are not algebraic properties

- Contracts specify pre- and postconditions, but not general algebraic properties such as associativity or monotonicity.

  We may, however, specify that a function is idempotent:

$$f' = assert\ \texttt{"f"}\ (true \mapsto \{\ y \mid y == f\ y\ \})\ f$$

- Contracts are attached to program points; algebraic properties can, in general, not be attached to program points.
- Contracts complement tools such as QuickCheck.

Outline of the talk

- ✔ Syntax of contracts
- ✔ Contracts are not ...
- ✘ Semantics of contracts
- ✘ Examples
- ✘ Properties of contracts

Semantics

☞ We specify the semantics of contracts by providing a Haskell implementation.

Though the syntax of contracts is somewhat special it can be easily translated into Haskell syntax.

concrete syntax	Haskell syntax
$\{ x \mid p\ x \}$	$Prop\ (\lambda x \rightarrow p\ x)$
$c_1 \mapsto c_2$	$Function\ c_1\ (const\ c_2)$
$(x :: c_1) \mapsto c_2\ x$	$Function\ c_1\ (\lambda x \rightarrow c_2\ x)$
$c_1 \times c_2$	$Pair\ c_1\ (const\ c_2)$
$(x :: c_1) \times c_2\ x$	$Pair\ c_1\ (\lambda x \rightarrow c_2\ x)$
$list\ c$	$List\ c$
$c_1\ \&\ c_2$	$And\ c_1\ c_2$

Semantics: a generalized algebraic data type

Using generalized algebraic data types we can represent contracts directly in Haskell.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Contract}\ \alpha\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{Prop} & :: (\alpha \to \mathit{Bool}) \to \mathit{Contract}\ \alpha \\
\mathit{Function} & :: \mathit{Contract}\ \alpha \to (\alpha \to \mathit{Contract}\ \beta) \to \mathit{Contract}\ (\alpha \to \beta) \\
\mathit{Pair} & :: \mathit{Contract}\ \alpha \to (\alpha \to \mathit{Contract}\ \beta) \to \mathit{Contract}\ (\alpha, \beta) \\
\mathit{List} & :: \mathit{Contract}\ \alpha \to \mathit{Contract}\ [\alpha] \\
\mathit{And} & :: \mathit{Contract}\ \alpha \to \mathit{Contract}\ \alpha \to \mathit{Contract}\ \alpha
\end{array}
\end{array}
$$

Semantics: without blame assignment

Recall: using $assert$ we attach contracts to expressions.

$$
\begin{array}{lll}
assert\ ::\ Contract\ \alpha \to (\alpha \to \alpha) & & \\
assert\ (Prop\ p)\ a \mid p\ a & & = a \\
\qquad\qquad\qquad\qquad \mid otherwise & & = error\ (\texttt{"assertion failed"}) \\
assert\ (Function\ c_1\ c_2)\ f & & = \lambda x \to (assert\ (c_2\ x) \cdot f \cdot assert\ c_1)\ x \\
assert\ (Pair\ c_1\ c_2) & (a_1, a_2) & = (assert\ c_1\ a_1, assert\ (c_2\ a_1)\ a_2) \\
assert\ (List\ c) & as & = map\ (assert\ c)\ as \\
assert\ (And\ c_1\ c_2) & a & = (assert\ c_2 \cdot assert\ c_1)\ a
\end{array}
$$

☞ $assert\ c$ is the identity except when c is violated, in which case an error is raised.

☞ Contracts are a lot like projections ($\pi \sqsubseteq id$ and $\pi \cdot \pi = \pi$).

☞ $List$ is mapped to map; And to composition.

Semantics: with blame assigment

Blame assignment is easy for non-functional values: $assert$ takes an extra string which is used to point to the location of the expression in case of a contract violation.

In case of functional values, the caller is to blame if the precondition is violated and the callee is to blame if the postcondition is violated. For that reason, we arrange that every function call takes an additional argument, the location of the caller.

$$\begin{array}{l}\mathbf{infixr}\ \ \hookrightarrow \\ \mathbf{data}\ \alpha \hookrightarrow \beta = Fun\{\, apply :: Loc \to \alpha \to \beta \,\}\end{array}$$

We use $e_1 \diamond\ e_2$ as an abbreviation for $apply\ e_1\ \ell\ e_2$ where ℓ is the location of the application.

Semantics:

$$\begin{array}{l}\textbf{data}\ \mathit{Contract}\ \alpha\ \textbf{where}\\ \quad \mathit{Function} :: \mathit{Contract}\ \alpha \to (\alpha \to \mathit{Contract}\ \beta) \to \mathit{Contract}\ (\alpha \hookrightarrow \beta)\\ \quad \ldots\end{array}$$

$$\begin{array}{l}\mathit{assert} :: \mathit{Loc} \to \mathit{Contract}\ \alpha \to (\alpha \to \alpha)\\ \mathit{assert}\ \ell\ (\mathit{Prop}\ p)\ a\\ \quad \mid p\ a \qquad\qquad = a\\ \quad \mid \mathit{otherwise} = \mathit{error}\ (\texttt{"assertion failed: `"} +\!\!+\\ \qquad\qquad\qquad\qquad \ell +\!\!+ \texttt{"' is to blame."})\\ \mathit{assert}\ \ell\ (\mathit{Function}\ c_1\ c_2)\ f\\ \qquad\qquad = \mathit{Fun}\ (\lambda \ell'\ x \to (\mathit{assert}\ \ell\ (c_2\ x) \cdot \mathit{apply}\ f\ \ell' \cdot \mathit{assert}\ \ell'\ c_1)\ x)\\ \mathit{assert}\ \ell\ (\mathit{Pair}\ c_1\ c_2) \qquad (a_1, a_2) = (\mathit{assert}\ \ell\ c_1\ a_1, \mathit{assert}\ \ell\ (c_2\ a_1)\ a_2)\\ \mathit{assert}\ \ell\ (\mathit{List}\ c) \qquad\quad \mathit{as} \qquad\quad = \mathit{map}\ (\mathit{assert}\ \ell\ c)\ \mathit{as}\\ \mathit{assert}\ \ell\ (\mathit{And}\ c_1\ c_2) \qquad\quad a \qquad\quad = (\mathit{assert}\ \ell\ c_2 \cdot \mathit{assert}\ \ell\ c_1)\ a\end{array}$$

Outline of the talk

- ✔ Syntax of contracts
- ✔ Contracts are not ...
- ✔ Semantics of contracts
- ✘ Examples
- ✘ Properties of contracts

Examples: factorization

$$
\begin{array}{l}
factors' :: Int \hookrightarrow [Int] \\
factors' = assert\ \texttt{"factors"} \\
\qquad\qquad ((n :: pos) \mapsto \{\, r \mid product\ r == n \,\}) \\
\qquad\quad (\lambda n \hookrightarrow factors\ n)
\end{array}
$$

☞ *factors* is the functional inverse of *product*. We can capture this idiom as a contract combinator:

$$
\begin{array}{l}
inverse \quad :: (Eq\ \beta) \Rightarrow (\alpha \to \beta) \to Contract\ (\beta \hookrightarrow \alpha) \\
inverse\ f = (x :: true) \mapsto \{\, y \mid f\ y == x \,\} \\
\\
factors' :: Int \hookrightarrow [Int] \\
factors' = assert\ \texttt{"factors"} \\
\qquad\qquad ((pos \mapsto true)\ \&\ inverse\ product) \\
\qquad\quad (\lambda n \hookrightarrow factors\ n)
\end{array}
$$

Examples: sorting

The contract ord constrains lists to ordered lists.

$$
\begin{aligned}
&ord :: (Ord\ \alpha) \Rightarrow Contract\ [\alpha] \\
&ord = \{\, x \mid ordered\ x \,\} \\
\\
&ordered :: (Ord\ \alpha) \Rightarrow [\alpha] \rightarrow Bool \\
&ordered\ [\,] &&= True \\
&ordered\ [a] &&= True \\
&ordered\ (a_1 : as@(a_2 : \_)) &&= a_1 \leqslant a_2 \wedge ordered\ as
\end{aligned}
$$

Examples: sorting—cont.

Insertion sort:

$$
\begin{aligned}
&insert' :: (Ord\ \alpha) \Rightarrow \alpha \hookrightarrow [\alpha] \hookrightarrow [\alpha] \\
&insert' = assert\ \texttt{"insert"} \\
&\qquad\qquad (true \mapsto ord \mapsto ord) \\
&\qquad\quad (\lambda a \hookrightarrow \lambda x \hookrightarrow insert\ a\ x)
\end{aligned}
$$

The function $insertionSort$ returns an ordered list:

$$
\begin{aligned}
&insertionSort' :: (Ord\ \alpha) \Rightarrow [\alpha] \hookrightarrow [\alpha] \\
&insertionSort' = assert\ \texttt{"insertionSort"} \\
&\qquad\qquad\quad (true \mapsto ord) \\
&\qquad\qquad (\lambda x \hookrightarrow insertionSort\ x)
\end{aligned}
$$

☞ We did not specify that the output list is a permutation of the input list.

Examples: sorting—cont.

Assuming a function $bag :: [\alpha] \to Bag\ \alpha$ that turns a list into a bag, we can fully specifiy $insertionSort$.

$$
\begin{array}{l}
insertionSort' :: (Ord\ \alpha) \Rightarrow [\alpha] \hookrightarrow [\alpha] \\
insertionSort' = assert\ \texttt{"insertionSort"} \\
\qquad\qquad\qquad (true \mapsto ord\ \&\ (x :: true) \mapsto \{\, s \mid bag\ x == bag\ s \,\}) \\
\qquad\qquad\quad (\lambda x \hookrightarrow insertionSort\ x)
\end{array}
$$

☞ In a sense, $insertionSort$ preserves the 'baginess' of lists. Again, we can single out this idiom as a combinator:

$$
\begin{array}{l}
preserve :: (Eq\ \beta) \Rightarrow (\alpha \to \beta) \to Contract\ (\alpha \hookrightarrow \alpha) \\
preserve\ f = (x :: true) \mapsto \{\, r \mid f\ x == f\ r \,\} \\
\\
insertionSort' = assert\ \texttt{"insertionSort"} \\
\qquad\qquad\qquad (true \mapsto ord\ \&\ preserve\ bag) \quad \text{-- weaker: } preserve\ len \\
\qquad\qquad\quad (\lambda x \hookrightarrow insertionSort\ x)
\end{array}
$$

Examples: sorting—cont.

Alternatively, we can specify $insertionSort$ in terms of a trusted sorting routine.

$$
\begin{aligned}
insertionSort' = {} & assert\ \texttt{"insertionSort"} \\
& \quad ((x :: true) \mapsto \{\, s \mid s == trustedSort\ x \,\}) \\
& (\lambda x \hookrightarrow insertionSort\ x)
\end{aligned}
$$

Again, we can capture this idiom:

$$
\begin{aligned}
& is :: (Eq\ \beta) \Rightarrow (\alpha \to \beta) \to Contract\ (\alpha \hookrightarrow \beta) \\
& is\ f = (x :: true) \mapsto \{\, y \mid y == f\ x \,\} \\
& \\
& insertionSort' = assert\ \texttt{"insertionSort"} \\
& \quad (is\ trustedSort) \\
& (\lambda x \hookrightarrow insertionSort\ x)
\end{aligned}
$$

Examples: while

Control constructs do not need a special (extra-linguistic) treatment.

$$
\begin{array}{l}
\mathit{while} :: (\alpha \to \mathit{Bool}) \to (\alpha \to \alpha) \to (\alpha \to \alpha) \\
\mathit{while}\ p\ f\ a = \textbf{if}\ p\ a\ \textbf{then}\ \mathit{while}\ p\ f\ (f\ a)\ \textbf{else}\ a \\
\\
\mathit{while}' :: \mathit{Contract}\ \alpha \to (\alpha \to \mathit{Bool}) \hookrightarrow (\alpha \hookrightarrow \alpha) \hookrightarrow (\alpha \hookrightarrow \alpha) \\
\mathit{while}'\ \mathit{inv} = \mathit{assert}\ \texttt{"while"} \\
\qquad\qquad (\mathit{true} \mapsto (\mathit{inv} \mapsto \mathit{inv}) \mapsto (\mathit{inv} \mapsto \mathit{inv})) \\
\qquad\quad (\lambda p \hookrightarrow \lambda f \hookrightarrow \lambda a \hookrightarrow \mathit{while}\ p\ (\lambda x \to f \diamond x)\ a)
\end{array}
$$

☞ We pass a contract to while'.

Examples: foldr

The technique of passing invariants is also applicable to $foldr$.

$$
\begin{array}{l}
foldr' :: Contract\ \beta \to (\alpha \hookrightarrow \beta \hookrightarrow \beta) \hookrightarrow \beta \hookrightarrow [\alpha] \hookrightarrow \beta \\
foldr'\ inv = assert\ \texttt{"foldr"} \\
\qquad\qquad ((true \mapsto inv \mapsto inv) \mapsto inv \mapsto true \mapsto inv) \\
\qquad\qquad (\lambda f \hookrightarrow \lambda e \hookrightarrow \lambda x \hookrightarrow foldr\ (\lambda a \to \lambda b \to f \diamond\ a \diamond\ b)\ e\ x) \\
\\
insertionSort' = foldr'\ ord \diamond\ (\lambda a \hookrightarrow \lambda x \hookrightarrow insert\ a\ x) \diamond\ [\,]
\end{array}
$$

Outline of the talk

- ✔ Syntax of contracts
- ✔ Contracts are not ...
- ✔ Semantics of contracts
- ✔ Examples
- ✘ Properties of contracts

Properties

$$
\begin{array}{lcl}
\mathit{false} & = & \mathit{prop}\ (\lambda a \rightarrow \mathit{False}) \\
\mathit{true} & = & \mathit{prop}\ (\lambda a \rightarrow \mathit{True}) \\
\mathit{prop}\ p_1\ \&\ \mathit{prop}\ p_2 & = & \mathit{prop}\ (\lambda a \rightarrow p_1\ a \wedge p_2\ a) \\
 & & \\
\mathit{false}\ \&\ c & = & \mathit{false} \\
c\ \&\ \mathit{false} & = & \mathit{false} \\
\mathit{true}\ \&\ c & = & c \\
c\ \&\ \mathit{true} & = & c \\
c_1\ \&\ c_2 & = & c_2\ \&\ c_1
\end{array}
$$

☞ The last property, commutativity of '$\&$', only holds in a strict setting where every contract satisfies: $c\ x \in \{\bot, x\}$.

Properties

Since $list$, '$\mapsto$' and '$\times$' are essentially maps, we have the familar functor laws :

$$
\begin{array}{lcl}
list\ true & = & true \\
list\ (c_1\ \&\ c_2) & = & list\ c_1\ \&\ list\ c_2 \\
 & & \\
true \mapsto true & = & true \\
(pre_1 \mapsto post_1)\ \&\ (pre_2 \mapsto post_2) & = & (pre_2\ \&\ pre_1) \mapsto (post_1\ \&\ post_2) \\
 & & \\
true \times true & = & true \\
(pre_1 \times post_1)\ \&\ (pre_2 \times post_2) & = & (pre_1\ \&\ pre_2) \times (post_1\ \&\ post_2)
\end{array}
$$

Outline of the talk

- ✔ Syntax of contracts
- ✔ Contracts are not ...
- ✔ Semantics of contracts
- ✔ Examples
- ✔ Properties of contracts