Typed Contracts for Functional Programming

Ralf Hinze

Johan Jeuring

Andres Löh

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2006-026

www.cs.uu.nl

ISSN: 0924-3275

Typed Contracts for Functional Programming

Ralf Hinze[1], Johan Jeuring[2], and Andres Löh[1]

[1] Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
{ralf,loeh}@informatik.uni-bonn.de
[2] Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
johanj@cs.uu.nl

Abstract. A robust software component fulfills a contract: it expects data satisfying a certain property and promises to return data satisfying another property. The object-oriented community uses the design-by-contract approach extensively. Proposals for language extensions that add contracts to higher-order functional programming have appeared recently. In this paper we propose an embedded domain-specific language for typed, higher-order and first-class contracts, which is both more expressive than previous proposals, and allows for a more informative blame assignment. We take some first steps towards an algebra of contracts, and we show how to define a generic contract combinator for arbitrary algebraic data types. The contract language is implemented as a library in Haskell using the concept of generalised algebraic data types.

1 Introduction

Are you familiar with the following situation?

> You are staring at the computer screen. The run of the program you are developing unexpectedly terminated with a `Prelude.head: empty list` message. A quick `grep` yields a total of 102 calls to *head* in your program. It is all very well that the run wasn't aborted with a `core dumped` notification, but the error message provided isn't very helpful either: which of the many calls to *head* is to blame?

If this sounds familiar to you, then you might be interested in *contracts*. A contract between software components is much like a contract in business, with obligations and benefits for both parties. In our scenario, the components are simply functions: the function *head* and the function that calls *head*. Here is a possible contract between the two parties (from *head*'s perspective): if you pass me a non-empty list, then I shall return its first element. The contract implies obligations and benefits: the caller is obliged to supply a non-empty list and has the benefit of receiving the first element without further ado. The restriction on the input is a benefit for *head*: it need not deal with the case for the empty list. If it receives a non-empty list, however, *head* is obliged to return its first element.

As in business, contracts may be violated. In this case the contract specifies who is to blame: the one who falls short of its promises. Thus, if *head* is called with an empty list, then the call site is to blame. In practical terms, this means that the program execution is aborted with an error message that points to the location of the caller, just what we needed above.

The underlying design methodology [1], developing programs on the basis of contracts, was popularised by Bertrand Meyer, the designer of Eiffel [2]. In fact, contracts are an integral part of Eiffel. Findler and Felleisen [3] later adapted the approach to higher-order functional languages. Their work has been the major inspiration of the present paper, which extends and revises their approach.

In particular, we make the following contributions:

- we develop a small embedded domain-specific language for contracts with a handful of basic combinators and a number of derived ones,

- we show how to define a generic contract combinator for algebraic data types,
- we present a novel approach to blame assignment that additionally tracks the cause of contract violations,
- as a proof of concept we provide a complete implementation of the approach; the implementation makes use of *generalised algebraic data types*,
- we take the first steps towards an algebra of contracts.

The rest of the paper is structured as follows. Sec. 2 introduces the basic contract language, Sec. 3 then shows how blame is assigned in the case of a contract violation. We tackle the implementation in Sec. 4 and 5 (without and with blame assignment). Sec. 6 provides further examples and defines several derived contract combinators. The algebra of contracts is studied in Sec. 7. Finally, Sec. 8 reviews related work and Sec. 9 concludes.

We use Haskell [4] notation throughout the paper. In fact, the source of the paper constitutes a legal Haskell program that can be executed using the Glasgow Haskell Compiler [5]. For the proofs it is, however, easier to pretend that we are working in a strict setting. The subtleties of lazy evaluation are then addressed in Sec. 7. Finally, we deviate from Haskell syntax in that we typeset 'x has type τ' as $x : \tau$ and 'a is consed to the list as' as $a :: as$ (as in Standard ML).

2 Contracts

This section introduces the main building blocks of the contract language.

A contract specifies a desired property of an expression. A simple contract is, for instance, $\{\, i \mid i \geqslant 0 \,\}$ which restricts the value of an integer expression to the natural numbers. In general, if x is a variable of type σ and e is a Boolean expression, then $\{\, x \mid e \,\}$ is a contract of type $\mathit{Contract}\ \sigma$, a so-called *contract comprehension*. The variable x is bound by the construct and scopes over e.

Contracts are first-class citizens: they can be passed to functions or returned as results, and most importantly they can be given a name.

$$\begin{aligned} &\mathit{nat} \ :\ \mathit{Contract}\ \mathit{Int} \\ &\mathit{nat} = \{\, i \mid i \geqslant 0 \,\} \end{aligned}$$

As a second example, here is a contract over the list data type that admits only non-empty lists.

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

The two most extreme contracts are

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

The contract *false* is very demanding, in fact, too demanding as it cannot be satisfied by any value. By contrast, *true* is very liberal: it admits every value.

Using contract comprehensions we can define contracts for values of arbitrary types, including function types. The contract $\{\, f \mid f\ 0 == 0 \,\}$, for instance, specifies that 0 is a fixed point of a function-valued expression of type $\mathit{Int} \to \mathit{Int}$. However, sometimes contract comprehensions are not expressive enough. Since a comprehension is constrained by a Haskell Boolean expression, we *cannot* state, for example, that a function maps natural numbers to natural numbers: $\{\, f \mid \forall n : \mathit{Int}\,.\, n \geqslant 0 \Rightarrow f\ n \geqslant 0 \,\}$. We consciously restrict the formula to the right of the bar to Haskell expressions so that checking of contracts remains feasible. As a compensation, we introduce a new contract combinator that allows us to explicitly specify domain and codomain of a function: $\mathit{nat} \rightarrowtriangle \mathit{nat}$ is the desired contract that restricts functions to those that take naturals to naturals.

Unfortunately, the new combinator is still too weak. Often we want to relate the argument to the result, expressing, for instance, that the result is greater than the argument. To this end we

$$\frac{\Gamma, x : \sigma \vdash e : \mathit{Bool}}{\Gamma \vdash \{\, x \mid e \,\} : \mathit{Contract}\ \sigma} \qquad \frac{\Gamma \vdash e_1 : \mathit{Contract}\ \sigma_1 \quad \Gamma, x : \sigma_1 \vdash e_2 : \mathit{Contract}\ \sigma_2}{\Gamma \vdash (x : e_1) \rightarrowtriangle e_2 : \mathit{Contract}\ (\sigma_1 \to \sigma_2)}$$

$$\frac{\Gamma \vdash e : \mathit{Contract}\ \sigma}{\Gamma \vdash [e] : \mathit{Contract}\ [\sigma]} \qquad \frac{\Gamma \vdash e_1 : \mathit{Contract}\ \sigma_1 \quad \Gamma, x : \sigma_1 \vdash e_2 : \mathit{Contract}\ \sigma_2}{\Gamma \vdash (x : e_1) \times e_2 : \mathit{Contract}\ (\sigma_1, \sigma_2)}$$

$$\frac{\Gamma \vdash e_1 : \mathit{Contract}\ \sigma \quad \Gamma \vdash e_2 : \mathit{Contract}\ \sigma}{\Gamma \vdash e_1\ \&\ e_2 : \mathit{Contract}\ \sigma}$$

Fig. 1. Typing rules for contract combinators.

generalise $e_1 \rightarrowtriangle e_2$ to the *dependent function contract* $(x{:}e_1) \rightarrowtriangle e_2$. The idea is that x, which scopes over e_2, represents the argument to the function. The above constraint is now straightforward to express: $(n : \mathit{nat}) \rightarrowtriangle \{\, r \mid n < r \,\}$. In general, if x is a variable of type σ_1, and e_1 and e_2 are contracts of type $\mathit{Contract}\ \sigma_1$ and $\mathit{Contract}\ \sigma_2$ respectively, then $(x{:}e_1) \rightarrowtriangle e_2$ is a contract of type $\mathit{Contract}\ (\sigma_1 \to \sigma_2)$. Note that like $\{\, x \mid e \,\}$, the dependent function contract $(x : e_1) \rightarrowtriangle e_2$ is a binding construct.

Many properties over data types such as the pair or the list data type can be expressed using contract comprehensions. However, it is also convenient to be able to construct contracts in a compositional manner. To this end we provide a pair combinator that takes two contracts and yields a contract on pairs: $\mathit{nat} \times \mathit{nat}$, for instance, constrains pairs to pairs of natural numbers.

We also offer a *dependent product contract* $(x{:}e_1) \times e_2$ with scoping and typing rules similar to the dependent function contract. As an example, the contract $(n : \mathit{nat}) \times (\{\, i \mid i \leqslant n \,\} \rightarrowtriangle \mathit{true})$ of type $\mathit{Contract}\ (\mathit{Int}, \mathit{Int} \to \alpha)$ constrains the domain of the function in the second component using the value of the first component. While the dependent product contract is a logically compelling counterpart of the dependent function contract, we expect the former to be less useful in practice. The reason is simply that properties of pairs that do *not* contain functions can be easily formulated using contract comprehensions. As a simple example, consider $\{\, (x_1, x_2) \mid x_1 \leqslant x_2 \,\}$.

In general, we need a contract combinator for every parametric data type. For the main bulk of the paper, we confine ourselves to the list data type: the *list contract combinator* takes a contract on elements to a contract on lists. For instance, $[\mathit{nat}]$ constrains integer lists to lists of natural numbers. Like $c_1 \times c_2$, the list combinator captures only *independent properties*; it cannot relate elements of a list. For this purpose, we have to use contract comprehensions—which, on the other hand, cannot express the contract $[\mathit{nat} \rightarrowtriangle \mathit{nat}]$.

Finally, contracts may be combined using conjunction: c_1 & c_2 holds if both c_1 and c_2 hold. However, we neither offer disjunction nor negation for reasons to be explained later (Sec. 4). Fig. 1 summarises the contract language.

3 Blame assignment

A contract is attached to an expression using *assert*:

$$\begin{aligned} &\mathit{head}' \ :\ [\alpha] \to \alpha \\ &\mathit{head}' = \mathit{assert}\ (\mathit{nonempty} \rightarrowtriangle \mathit{true})\ (\lambda x \to \mathit{head}\ x) \end{aligned}$$

The attached contract specifies that the predefined function *head* requires its argument to be non-empty and that it ensures nothing. In more conventional terms, *nonempty* is the *precondition* and *true* is the *postcondition*. Here and in what follows we adopt the convention that the 'contracted' version of the identifier x is written x'.

Attaching a contract to an expression causes the contract to be dynamically monitored at run-time. If the contract is violated, the evaluation is aborted with an informative error message. If the contract is fulfilled, then *assert* acts as the identity. Consequently, *assert* has type

$$\mathit{assert} : \mathit{Contract}\ \alpha \to (\alpha \to \alpha)$$

Contracts range from very specific to very liberal. The contract of *head*, $nonempty \rightarrowtriangle true$, is very liberal: many functions require a non-empty argument. On the other hand, a contract may uniquely determine a value. Consider in this respect the function *isqrt*, which is supposed to calculate the integer square root.

$$
\begin{aligned}
&isqrt \quad : Int \to Int \\
&isqrt\ n = loop\ 0\ 3\ 1 \\
&\quad \textbf{where}\ loop\ i\ k\ s \mid s \leqslant n \quad = loop\ (i+1)\ (k+2)\ (s+k) \\
&\qquad\qquad\quad \mid otherwise = i
\end{aligned}
$$

It is not immediately obvious that this definition actually meets its specification, so we add a contract.

$$
\begin{aligned}
&isqrt' \ : Int \to Int \\
&isqrt' = assert\ ((n : nat) \rightarrowtriangle \{\ r \mid r \geqslant 0 \wedge r^2 \leqslant n < (r+1)^2\ \})\ (\lambda n \to isqrt\ n)
\end{aligned}
$$

Here the postcondition precisely captures the intended semantics of *isqrt*.

Now that we got acquainted with the contract language, it is time to see contracts in action. When a *contract comprehension* is violated, the error message points to the expression to which the contract is attached. Let us assume for the purposes of this paper that the expression is bound to a name which we can then use for error reporting (in the implementation we refer to the source location instead). As an example, given the definitions $five = assert\ nat\ 5$ and $mfive = assert\ nat\ (-5)$, we get the following results in an interactive session.

```
Contracts⟩ five
5
Contracts⟩ mfive
*** contract failed: the expression 'mfive' is to blame.
```

The number -5 is not a natural; consequently the *nat* contract sounds alarm.

If a *dependent function contract* is violated, then either the function is applied to the wrong argument, or the function itself is wrong. In the first case, the precondition sends the alarm, in the second case the postcondition. Consider the functions *inc* and *dec*, which increase, respectively decrease, a number.

$$
\begin{aligned}
&inc, dec : Int \to Int \\
&inc = assert\ (nat \rightarrowtriangle nat)\ (\lambda n \to n+1) \\
&dec = assert\ (nat \rightarrowtriangle nat)\ (\lambda n \to n-1)
\end{aligned}
$$

Here are some example applications of these functions in an interactive session:

```
Contracts⟩ inc ⟨1⟩5
6
Contracts⟩ inc ⟨2⟩(−5)
*** contract failed: the expression labelled '2' is to blame.
Contracts⟩ dec ⟨3⟩5
4
Contracts⟩ dec ⟨4⟩0
*** contract failed: the expression dec is to blame.
```

In the session we put labels in front of the function arguments, $\langle i \rangle e$, so that we can refer to them in error messages (again, in the implementation we refer to the source location). The first contract violation is caused by passing a negative value to *inc*: its precondition is violated, hence the argument is to blame. In the last call, *dec* falls short of its promise to deliver a natural number, hence *dec* itself is to blame.

It is important to note that contract checking and detection of violations are tied to program runs: *dec* obviously does not satisfy its contract $nat \rightarrowtriangle nat$, but this is not detected until *dec* is

applied to 0. In other words, contracts do not give any static guarantees ('*dec* takes naturals to naturals'), they only make dynamic assertions about particular program runs ('*dec* always received and always delivered a natural number during this run').

This characteristic becomes even more prominent when we consider higher-order functions.

$$codom\ :\ (Int \to Int) \to [Int]$$
$$codom = assert\ ((nat \twoheadrightarrow nat) \twoheadrightarrow [nat])\ (\lambda f \to [f\ {}_{\langle 5\rangle}\, n \mid n \leftarrow [1\,..\,9]])$$

The function *codom* takes a function argument of type $Int \to Int$. We cannot expect that a contract violation is detected the very moment *codom* is applied to a function—as we cannot expect that a contract violation is detected the very moment we attach a contract to $\lambda n \to n-1$ in *dec*. Rather, violations are discovered when the function argument f is later applied in the body of *codom*. In the extreme case where the parameter does not appear in the body, we never get alarmed, unless, of course, the result is negative. Consider the following interactive session:

```
Contracts⟩ codom ⟨6⟩(λx → x − 1)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
Contracts⟩ codom ⟨7⟩(λx → x − 2)
*** contract failed: the expression labelled '7' is to blame.
```

An error is only detected in the second call, though the first call is also wrong. The error message points to the correct location: the argument is to blame.

The following example has been adapted from the paper by Blume and McAllester [6].

$$g\ :\ (Int \to Int) \to (Int \to Int)$$
$$g = assert\ ((nat \twoheadrightarrow nat) \twoheadrightarrow true)\ (\lambda f \to \lambda x \to f\ {}_{\langle 8\rangle}\, x)$$

The higher-order function g expects a function satisfying $nat \twoheadrightarrow nat$. Again, we cannot expect that the function contract is checked immediately; rather, it is tested when the function argument is applied.

```
Contracts⟩ g ⟨9⟩(λx → x − 1) ⟨10⟩1
0
Contracts⟩ g ⟨11⟩(λx → x − 1) ⟨12⟩0
*** contract failed: the expression labelled '11' is to blame.
Contracts⟩ g ⟨13⟩(λx → x) ⟨14⟩(−7)
*** contract failed: the expression 'g' is to blame (the violation was caused by the expres-
sion(s) labelled '8').
```

The last call shows that g is blamed for a contract violation even though g's postcondition is *true*. This is because g must also take care that its argument is called correctly and it obviously does not take sufficient measurements. The error message additionally points to the location within g that *caused* the contract violation. This information is not available in the Findler and Felleisen approach [3] (see also Sec. 5). Since g returns a function, the cause is not necessarily located in g's body. As a simple example, consider the η-reduced variant of g.

$$g = assert\ ((nat \twoheadrightarrow nat) \twoheadrightarrow true)\ (\lambda f \to f)$$

Now the second argument is identified as the cause of the contract violation:

```
Contracts⟩ g ⟨15⟩(λx → x) ⟨16⟩(−7)
*** contract failed: the expression 'g' is to blame (the violation was caused by the expres-
sion(s) labelled '16').
```

4 Implementing contracts

In Sec. 2 we have seen several ways to construct contracts. The syntax we have used for contracts may seem to suggest that we need an extension of Haskell to implement contracts. However, using

concrete syntax	Haskell syntax
$\{\, x \mid p\ x \,\}$	$\mathit{Prop}\ (\lambda x \to p\ x)$
$c_1 \rightarrowtriangle c_2$	$\mathit{Function}\ c_1\ (\mathit{const}\ c_2)$
$(x : c_1) \rightarrowtriangle c_2\ x$	$\mathit{Function}\ c_1\ (\lambda x \to c_2\ x)$
$c_1 \times c_2$	$\mathit{Pair}\ c_1\ (\mathit{const}\ c_2)$
$(x : c_1) \times c_2\ x$	$\mathit{Pair}\ c_1\ (\lambda x \to c_2\ x)$
$[c]$	$\mathit{List}\ c$
$c_1\ \&\ c_2$	$\mathit{And}\ c_1\ c_2$

Fig. 2. Concrete and abstract syntax of contracts.

Generalised Algebraic Data Types (GADTs) [7, 8, **?**], we can model contracts directly in Haskell. Fig. 2 shows how the concrete syntax translates to Haskell. Note that the binding constructs of the concrete syntax are realized using functional components (higher-order abstract syntax). If we translate the typing rules listed in Fig. 1 to the abstract representation of contracts, we obtain the following GADT.

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

Given this data type we can define *assert* by a simple case analysis.

$$
\begin{array}{lll}
\mathit{assert} & & : \mathit{Contract}\ \alpha \to (\alpha \to \alpha) \\
\mathit{assert}\ (\mathit{Prop}\ p) & a & = \textbf{if}\ p\ a\ \textbf{then}\ a\ \textbf{else}\ \mathit{error}\ \texttt{"contract failed"} \\
\mathit{assert}\ (\mathit{Function}\ c_1\ c_2)\ f & & = (\lambda x' \to (\mathit{assert}\ (c_2\ x') \cdot f)\ x') \cdot \mathit{assert}\ c_1 \\
\mathit{assert}\ (\mathit{Pair}\ c_1\ c_2)\ (a_1, a_2) & & = (\lambda a_1' \to (a_1', \mathit{assert}\ (c_2\ a_1')\ a_2))\ (\mathit{assert}\ c_1\ a_1) \\
\mathit{assert}\ (\mathit{List}\ c) & as & = \mathit{map}\ (\mathit{assert}\ c)\ as \\
\mathit{assert}\ (\mathit{And}\ c_1\ c_2)\ a & & = (\mathit{assert}\ c_2 \cdot \mathit{assert}\ c_1)\ a
\end{array}
$$

The definition makes explicit that only contract comprehensions are checked immediately. In the remaining cases, the contract is taken apart and its constituents are attached to the corresponding constituents of the value to be checked. Note that in the *Function* case the *checked argument* x' is propagated to the codomain contract c_2 (ditto in the *Pair* case). There is a choice here: alternatively, we could pass the original, unchecked argument. If we chose this variant, however, we would sacrifice the idempotence of '&'. Furthermore, in a lazy setting the unchecked argument could provoke a runtime error in the postcondition, consider, for instance, $(x : \mathit{nonempty}) \rightarrowtriangle \{\, y \mid y \leqslant \mathit{head}\ x \,\}$.

A moment's reflection reveals that the checking of *independent properties* boils down to an application of the *mapping function* for the type in question. In particular, we have

$$
\begin{array}{ll}
\mathit{assert}\ (\mathit{Function}\ c_1\ (\mathit{const}\ c_2))\ f & = \mathit{assert}\ c_2 \cdot f \cdot \mathit{assert}\ c_1 \\
\mathit{assert}\ (\mathit{Pair}\ c_1\ (\mathit{const}\ c_2))\ (a_1, a_2) & = (\mathit{assert}\ c_1\ a_1, \mathit{assert}\ c_2\ a_2)
\end{array}
$$

This immediately suggests how to generalise contracts and contract checking to arbitrary container types: we map the constituent contracts over the container.

$$\mathit{assert}\ (T\ c_1\ \ldots\ c_n) = \mathit{map}T\ (\mathit{assert}\ c_1)\ \ldots\ (\mathit{assert}\ c_n)$$

Note that mapping functions can be defined completely generically for arbitrary Haskell 98 data types [9]. In the next section we will show that we can do without the GADT; then the contract combinator for an algebraic data type is just its mapping function.

It remains to explain the equation for *And*: the conjunction *And* c_1 c_2 is tested by first checking c_1 and then checking c_2, that is, conjunction is implemented by functional composition. This seems odd at first sight: we expect conjunction to be commutative; composition is, however, not commutative in general. We shall return to this issue in Sec. 7. Also, note that we offer conjunction but neither disjunction nor negation. To implement disjunction we would need some kind of exception handling: if the first contract fails, then the second is tried. Exception handling is, however, not available in Haskell (at least not in the pure, non-*IO* part). For similar reasons, we shy away from negation.

Although *assert* implements the main ideas behind contracts, the fact that it returns an uninformative error message makes this implementation rather useless for practical purposes. In the following section we will show how to return the precise location of a contract violation.

Nonetheless, we can use the simple definition of *assert* to *optimise* contracted functions. Reconsider the definition of *inc* repeated below.

$$inc = assert\ (nat \rightarrowtriangle nat)\ (\lambda n \rightarrow n + 1)$$

Intuitively, *inc* satisfies its contract, so we can optimize the definition by leaving out the postcondition. Formally, we have to prove that

$$assert\ (nat \rightarrowtriangle nat)\ (\lambda n \rightarrow n + 1) = assert\ (nat \rightarrowtriangle true)\ (\lambda n \rightarrow n + 1)$$

Note that we must keep the precondition to ensure that *inc* is called correctly: the equation $assert\ (nat \rightarrowtriangle nat)\ (\lambda n \rightarrow n + 1) = \lambda n \rightarrow n + 1$ does not hold. Now, unfolding the definition of *assert* the equation above rewrites to

$$assert\ nat \cdot (\lambda n \rightarrow n + 1) \cdot assert\ nat = (\lambda n \rightarrow n + 1) \cdot assert\ nat$$

which can be proved using a simple case analysis.

In general, we say that *f satisfies the contract c* iff

$$assert\ c\ f = assert\ c^+\ f$$

where c^+ is obtained from c by replacing all sub-contracts at positive positions by *true*:

$$\begin{aligned}
&(\cdot)^+ : Contract\ \alpha \rightarrow Contract\ \alpha \\
&(Prop\ p)^+ &&= true \\
&(Function\ c_1\ c_2)^+ &&= Function\ c_1^-\ (\lambda x \rightarrow (c_2\ x)^+) \\
&(\cdot)^- : Contract\ \alpha \rightarrow Contract\ \alpha \\
&(Prop\ p)^- &&= Prop\ p \\
&(Function\ c_1\ c_2)^- &&= Function\ c_1^+\ (\lambda x \rightarrow (c_2\ x)^-)
\end{aligned}$$

In the remaining cases, $(\cdot)^+$ and $(\cdot)^-$ are just propagated to the components. As an example, $\lambda n \rightarrow n + 1$ satisfies $nat \rightarrowtriangle nat$, whereas $\lambda n \rightarrow n - 1$ does not. The higher-order function g of Sec. 3 also does not satisfy its contract $(nat \rightarrowtriangle nat) \rightarrowtriangle nat$. As an aside, note that $(\cdot)^+$ and $(\cdot)^-$ are executable Haskell functions. Here, the GADT proves its worth: contracts are data that can be as easily manipulated as, say, lists.

5 Implementing blame assignment

To correctly assign blame in the case of contract violations, we pass program locations to both *assert* and to the contracted functions themselves. For the purposes of this paper, we keep the type *Loc* of source locations abstract. We have seen in Sec. 3 that blame assignment involves at least one location. In the case of function contracts two locations are involved: if the precondition fails, then the argument is to blame; if the postcondition fails, then the function itself is to blame. For the former case, we need to get hold of the location of the argument. To this end, we extend the function by an extra parameter, which is the location of the 'ordinary' parameter.

$$\textbf{infixr} \twoheadrightarrow$$
$$\textbf{newtype}\ \alpha \twoheadrightarrow \beta = \mathit{Fun}\{\,\mathit{app} : \mathit{Locs} \to \alpha \to \beta\,\}$$

In fact, we take a slightly more general approach: we allow to pass a data structure of type *Locs* containing one or more locations. We shall provide two implementations of *Locs*, one that realizes blame assignment in the style of Findler & Felleisen and one that additionally provides information about the causers of a contract violation. We postpone the details until the end of this section and remark that *Locs* records at least the locations of the parties involved in a contract.

The type $\alpha \twoheadrightarrow \beta$ is the type of *contracted functions*: abstractions of this type, $\mathit{Fun}\ (\lambda \ell s \to \lambda x \to e)$, additionally take locations; applications, $\mathit{app}\ e_1\ \ell s\ e_2$, additionally pass locations. We abbreviate $\mathit{Fun}\ (\lambda \ell s \to \lambda x \to e)$ by $\lambda x \twoheadrightarrow e$ if ℓs does not appear free in e (which is the norm for user-defined functions). Furthermore, $\mathit{app}\ e_1\ \ell s\ e_2$ is written $e_1\ {}_{\ell s}e_2$. In the actual program source, the arguments of *assert* and of the contracted functions are always single locations, written $\langle \ell \rangle$, which explains the notation used in Sec. 3.

Since contracted functions have a distinguished type, we must adapt the type of the *Function* constructor.

$$\mathit{Function} : \mathit{Contract}\ \alpha \to (\alpha \to \mathit{Contract}\ \beta) \to \mathit{Contract}\ (\alpha \twoheadrightarrow \beta)$$

Given these prerequisites, we can finally implement contract checking with proper blame assignment.

$$\begin{aligned}
&\mathit{assert} : \mathit{Contract}\ \alpha \to (\mathit{Locs} \to \alpha \to \alpha)\\
&\mathit{assert}\ (\mathit{Prop}\ p)\quad \ell s\ a\\
&\quad = \textbf{if}\ p\ a\ \textbf{then}\ a\ \textbf{else}\ \mathit{error}\ (\texttt{"contract failed: "} +\!\!+ \mathit{blame}\ \ell s)\\
&\mathit{assert}\ (\mathit{Function}\ c_1\ c_2)\ \ell s_f\ f\\
&\quad = \mathit{Fun}\ (\lambda \ell_x \to (\lambda x' \to (\mathit{assert}\ (c_2\ x')\ \ell s_f \cdot \mathit{app}\ f\ \ell_x)\ x') \cdot \mathit{assert}\ c_1\ (\ell s_f \rhd \ell_x))\\
&\mathit{assert}\ (\mathit{Pair}\ c_1\ c_2)\ \ell s\ (a_1, a_2) = (\lambda a_1' \to (a_1', \mathit{assert}\ (c_2\ a_1')\ \ell s\ a_2))\ (\mathit{assert}\ c_1\ \ell s\ a_1)\\
&\mathit{assert}\ (\mathit{List}\ c)\quad \ell s\ as\quad = \mathit{map}\ (\mathit{assert}\ c\ \ell s)\ as\\
&\mathit{assert}\ (\mathit{And}\ c_1\ c_2)\ \ell s\ a\quad = (\mathit{assert}\ c_2\ \ell s \cdot \mathit{assert}\ c_1\ \ell s)\ a
\end{aligned}$$

The *Function* case merits careful study. Note that ℓs_f are the locations involved in f's contract and that ℓ_x is *the* location of its argument (ℓ_x has type *Locs* but it is always a single location of the form $\langle \ell \rangle$). First, the precondition c_1 is checked possibly blaming ℓs_f or ℓ_x. The single location ℓ_x is then passed to f, whose evaluation may involve further checking. Finally, the postcondition $c_2\ x'$ is checked possibly blaming a location in ℓs_f. Note that c_2 receives the checked argument, not the unchecked one.

It may seem surprising at first that *assert* c_1 adds ℓs_f to its file of suspects: can f be blamed if the precondition fails? If f is a first-order function, then this is impossible. However, if f takes a function as an argument, then f must take care that this argument is called correctly (see the discussion about g at the end of Sec. 3). If f does not to ensure this, then f is to blame.

In essence, *assert* turns a contract of type *Contract* α into a contracted function of type $\alpha \twoheadrightarrow \alpha$. If we re-phrase *assert* in terms of this type, we obtain the implementation listed in Fig. 3. Note that the elements of $\alpha \twoheadrightarrow \beta$ form the arrows of a category, the Kleisli category of a comonad, with $\lambda x \twoheadrightarrow x$ as the identity and '$\diamond$' acting as composition. Furthermore, *list* is the mapping function of the list functor. The implementation also makes clear that we can do without the GADT provided *assert* is the only operation on the data type *Contract*: the combinators of the contract library can be implemented directly in terms of *prop*, *fun*, *pair*, *list* and '$\diamond$'. Then *assert* is just the identity.

It remains to implement the data type *Locs* and the associated functions. Let us start with a simple version that supports blame assignment in the style of Findler & Felleisen. A contract either involves one or two parties.

$$\textbf{data}\ \mathit{Locs} = \mathit{Pos}\{\,\mathit{pos} : \mathit{Loc}\,\} \mid \mathit{NegPos}\{\,\mathit{neg} : \mathit{Loc}, \mathit{pos} : \mathit{Loc}\,\}$$

We distinguish between positive and negative locations corresponding to function and argument locations. Blame is always assigned to the positive location.

$$
\begin{array}{l}
\mathit{assert} \qquad\qquad\qquad\quad : \mathit{Contract}\ \alpha \to (\alpha \mathrel{+\!\!\!\to} \alpha) \\
\mathit{assert}\ (\mathit{Prop}\ p) \qquad\quad = \mathit{prop}\ p \\
\mathit{assert}\ (\mathit{Function}\ c_1\ c_2) = \mathit{fun}\ (\mathit{assert}\ c_1)\ (\mathit{assert} \cdot c_2) \\
\mathit{assert}\ (\mathit{Pair}\ c_1\ c_2) \quad = \mathit{pair}\ (\mathit{assert}\ c_1)\ (\mathit{assert} \cdot c_2) \\
\mathit{assert}\ (\mathit{List}\ c) \qquad\quad = \mathit{list}\ (\mathit{assert}\ c) \\
\mathit{assert}\ (\mathit{And}\ c_1\ c_2) \quad = \mathit{assert}\ c_2 \diamond \mathit{assert}\ c_1 \\
\\
\mathit{prop} \quad : (\alpha \to \mathit{Bool}) \to (\alpha \mathrel{+\!\!\!\to} \alpha) \\
\mathit{prop}\ p \;= \mathit{Fun}\ (\lambda \ell s\ a \to \mathbf{if}\ p\ a\ \mathbf{then}\ a\ \mathbf{else}\ \mathit{error}\ (\texttt{"contract failed: "} +\!\!+ \mathit{blame}\ \ell s)) \\
\\
\mathit{fun} \quad : (\alpha_1 \mathrel{+\!\!\!\to} \beta_1) \to (\beta_1 \to \alpha_2 \mathrel{+\!\!\!\to} \beta_2) \to ((\beta_1 \mathrel{+\!\!\!\to} \alpha_2) \mathrel{+\!\!\!\to} (\alpha_1 \mathrel{+\!\!\!\to} \beta_2)) \\
\mathit{fun}\ g\ h \;= \mathit{Fun}\ (\lambda \ell s_f\ f \to \mathit{Fun}\ (\lambda \ell_x \to \\
\qquad\qquad (\lambda x' \to (\mathit{app}\ (h\ x')\ \ell s_f \cdot \mathit{app}\ f\ \ell_x)\ x') \cdot \mathit{app}\ g\ (\ell s_f \rhd \ell_x))) \\
\\
\mathit{pair} \quad : (\alpha_1 \mathrel{+\!\!\!\to} \beta_1) \to (\beta_1 \to \alpha_2 \mathrel{+\!\!\!\to} \beta_2) \to ((\alpha_1, \alpha_2) \mathrel{+\!\!\!\to} (\beta_1, \beta_2)) \\
\mathit{pair}\ g\ h = \mathit{Fun}\ (\lambda \ell s\ (a_1, a_2) \to (\lambda a_1' \to (a_1', \mathit{app}\ (h\ a_1')\ \ell s\ a_2))\ (\mathit{app}\ g\ \ell s\ a_1)) \\
\\
\mathit{list} \quad : (\alpha \mathrel{+\!\!\!\to} \beta) \to ([\alpha] \mathrel{+\!\!\!\to} [\beta]) \\
\mathit{list}\ g \;= \mathit{Fun}\ (\lambda \ell s \to \mathit{map}\ (\mathit{app}\ g\ \ell s)) \\
\\
(\diamond) \quad : (\beta \mathrel{+\!\!\!\to} \gamma) \to (\alpha \mathrel{+\!\!\!\to} \beta) \to (\alpha \mathrel{+\!\!\!\to} \gamma) \\
g \diamond h \;= \mathit{Fun}\ (\lambda \ell s \to \mathit{app}\ g\ \ell s \cdot \mathit{app}\ h\ \ell s)
\end{array}
$$

Fig. 3. Contract checking with proper blame assignment.

$$
\begin{array}{l}
\mathit{blame} \quad : \mathit{Locs} \to \mathit{String} \\
\mathit{blame}\ \ell s = \texttt{"the expression "} +\!\!+ \mathit{show}\ (\mathit{pos}\ \ell s) +\!\!+ \texttt{" is to blame."}
\end{array}
$$

The actual locations in the source are positive.

$$\langle \ell \rangle = \mathit{Pos}\ \ell$$

The magic lies in the implementation of '$\rhd$', which combines two elements of type *Locs*.

$$
\begin{array}{l}
(\rhd) : \mathit{Locs} \to \mathit{Locs} \to \mathit{Locs} \\
\mathit{Pos}\ \ell \qquad \rhd \mathit{Pos}\ \ell' = \mathit{NegPos}\ \ell\ \ell' \\
\mathit{NegPos}\ \ell'\ \ell \rhd \_ \qquad = \mathit{NegPos}\ \ell\ \ell'
\end{array}
$$

Two single locations are merged into a double location; if the first argument is already a double location, then the second argument is ignored. Furthermore, positive and negative occurrences are interchanged in the second case. This is vital for functions of order 2 or higher. Re-consider the function g of Sec. 3.

$$
\begin{array}{l}
g = \mathit{assert}\ ((\mathit{nat} \mathrel{+\!\!\!\to} \mathit{nat}) \mathrel{+\!\!\!\to} \mathit{true})\ {}_{\langle 0 \rangle}(\lambda f \mathrel{+\!\!\!\to} \lambda x \mathrel{+\!\!\!\to} f\ {}_{\langle 2 \rangle}x) \\
\ldots\ g\ {}_{\langle 1 \rangle}(\lambda x \mathrel{+\!\!\!\to} x)\ {}_{\langle 3 \rangle}(-7)\ \ldots
\end{array}
$$

The precondition of g, $\mathit{nat} \mathrel{+\!\!\!\to} \mathit{nat}$, and the postcondition of g's argument f, *nat*, are checked using $\mathit{Pos}\ 0 \rhd \mathit{Pos}\ 1 = \mathit{NegPos}\ 0\ 1$. The precondition of f, however, is checked using $\mathit{NegPos}\ 0\ 1 \rhd \mathit{Pos}\ 2 = \mathit{NegPos}\ 1\ 0$. Thus, if f's precondition fails, g itself is blamed.

It is apparent that '$\rhd$' throws away information: location 2, which possibly causes the contract violation, is ignored. We can provide a more informative error message if we keep track of all the locations involved. To this end we turn *Locs* into a pair of stacks, see Fig. 4. Blame is assigned to the top-most element of the stack of positive locations; the remaining entries if any detail the cause of the contract violation. The new version of '$\rhd$' simply concatenates the stacks after swapping the two stacks of its first argument. Just in case you wonder: the total length of the stacks is equal to the order of the contracted function plus one. Thus, the stacks seldom contain more than 2 or 3 elements.

6 Examples

In this section we give further examples of the use of contracts. Besides, we shall introduce a number of derived contract combinators.

```
data Locs = NegPos{ neg : [Loc], pos : [Loc] }
blame    : Locs → String
blame ℓs = "the expression " ++ show (head (pos ℓs)) ++ " is to blame"
         ++ (case tail (pos ℓs) of
               [] → "."
               ℓs' → " (the violation was caused by the expression(s) " ++
                     concat (interleave ", " (map show ℓs')) ++ ").")
⟨·⟩ : Loc → Locs
⟨ℓ⟩ = NegPos [] [ℓ]
(⊳) : Locs → Locs → Locs
NegPos ns ps ⊳ NegPos ns' ps' = NegPos (ps ++ ns') (ns ++ ps')
```

Fig. 4. Extended blame assignment.

6.1 Sorting

An *invariant* is a property that appears both as a pre- and postcondition. To illustrate the use of invariants, consider the implementation of insertion sort:

```
insertion-sort : (Ord α) ⇒ [α] → [α]
insertion-sort = foldr insert []
insert : (Ord α) ⇒ α → [α] → [α]
insert a  []                    = [a]
insert a1 (a2 :: as) | a1 ⩽ a2  = a1 :: a2 :: as
                     | otherwise = a2 :: insert a1 as
```

The helper function *insert* takes an element a and an ordered list, and inserts the element at the right, according to the order, position in the list. In other words, *insert a* takes an ordered list to an ordered list.

```
insert' : (Ord α) ⇒ α ↣ [α] ↣ [α]
insert' = assert (true ↣ ord ↣ ord) (λa ↣ λx ↣ insert a x)
```

The contract *ord* for ordered lists is defined as follows:

```
ord : (Ord α) ⇒ Contract [α]
ord = { x | ordered x }
ordered                : (Ord α) ⇒ [α] → Bool
ordered []             = True
ordered [a]            = True
ordered (a1 :: a2 :: as) = a1 ⩽ a2 ∧ ordered (a2 :: as)
```

The type 'ordered list' can be seen as an abstract data type (it is concrete here, but it could easily be made abstract), whose invariant is given by *ord*. Other ADTs such as heaps, search trees etc can be handled in an analogous manner.

For completeness, here is the contracted version of *insertion-sort*:

```
insertion-sort' : (Ord α) ⇒ [α] ↣ [α]
insertion-sort' = assert (true ↣ ord) (λx ↣ insertion-sort x)
```

Note that we did not specify that the output list is a permutation of the input list. Assuming a function $bag : (Ord\ \alpha) \Rightarrow [\alpha] \to \wr\alpha\int$ that turns a list into a bag, we can fully specify sorting: $(x : true) \rightarrowtriangle ord\ \&\ \{\ s \mid bag\ x == bag\ s\ \}$. Loosely speaking, sorting preserves the 'baginess' of the input list. Formally, $g : \sigma \to \sigma$ preserves the function $f : \sigma \to \tau$ iff $f\ x == f\ (g\ x)$ for all x. Again, we can single out this idiom as a contract combinator.

$$\begin{aligned}&\mathit{preserves} \;\; : (\mathit{Eq}\ \beta) \Rightarrow (\alpha \to \beta) \to \mathit{Contract}\ (\alpha \twoheadrightarrow \alpha)\\&\mathit{preserves}\ f = (x : \mathit{true}) \twoheadrightarrow \{\ y \mid f\ x == f\ y\ \}\end{aligned}$$

Using this combinator the sort contract now reads $(\mathit{true} \twoheadrightarrow \mathit{ord})$ & $\mathit{preserves\ bag}$. Of course, either *bag* or the equality test for bags is an expensive operation (it almost certainly involves sorting), so we may content ourselves with a weaker property, for instance, that *insertion-sort* preserves the length of the input list: $(\mathit{true} \twoheadrightarrow \mathit{ord})$ & $\mathit{preserves\ length}$.

The example of sorting shows that the programmer or library writer has a choice as to how precise contracts are. The fact that contracts are first-class citizens renders it possible to abstract out common idioms. As a final twist on this topic, assume that you already have a trusted sorting function at hand. Then you could simply specify that your new sorting routine is extensionally equal to the trusted one. We introduce the *is* contract combinator for this purpose.

$$\begin{aligned}&\mathit{is} \;\; : (\mathit{Eq}\ \beta) \Rightarrow (\alpha \to \beta) \to \mathit{Contract}\ (\alpha \twoheadrightarrow \beta)\\&\mathit{is}\ f = (x : \mathit{true}) \twoheadrightarrow \{\ y \mid y == f\ x\ \}\\&\mathit{insertion\text{-}sort}'' = \mathit{assert}\ (\mathit{is}\ \mathit{sort})\ (\lambda x \twoheadrightarrow \mathit{insertion\text{-}sort}\ x)\end{aligned}$$

6.2 Recursion schemes

The function *insertion-sort* is defined in terms of *foldr*, the catamorphism of the list data type. An intriguing question is whether we can also attach a contract to *foldr* itself?

$$\begin{aligned}&\mathit{foldr} &&: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta\\&\mathit{foldr}\ f\ e\ [\,] &&= e\\&\mathit{foldr}\ f\ e\ (a :: \mathit{as}) &&= f\ a\ (\mathit{foldr}\ f\ e\ \mathit{as})\end{aligned}$$

The application to sorting gives $(\mathit{true} \twoheadrightarrow \mathit{ord} \twoheadrightarrow \mathit{ord}) \twoheadrightarrow \mathit{ord} \twoheadrightarrow \mathit{true} \twoheadrightarrow \mathit{ord}$ as a contract, but this one is, of course, way too specific. The idea suggests itself to abstract from the invariant, that is, to pass the invariant as an argument.

$$\begin{aligned}&\mathit{foldr}' : \mathit{Contract}\ \beta \to (\alpha \twoheadrightarrow \beta \twoheadrightarrow \beta) \twoheadrightarrow \beta \twoheadrightarrow [\alpha] \twoheadrightarrow \beta\\&\mathit{foldr}'\ \mathit{inv} = \mathit{assert}\ ((\mathit{true} \twoheadrightarrow \mathit{inv} \twoheadrightarrow \mathit{inv}) \twoheadrightarrow \mathit{inv} \twoheadrightarrow \mathit{true} \twoheadrightarrow \mathit{inv})\\&\qquad\qquad (\lambda f \twoheadrightarrow \lambda e \twoheadrightarrow \lambda x \twoheadrightarrow \mathit{foldr}\ (\lambda a \to \lambda b \to f\ {}_{\langle 17\rangle}\, a\ {}_{\langle 18\rangle}\, b)\ e\ x)\end{aligned}$$

Again, the fact that contracts are first-class citizens proves its worth. Higher-order functions that implement general recursion schemes or control constructs typically take contracts as arguments.

Interestingly, we can optimize *foldr*′ as it satisfies its contract:

$$\begin{aligned}&\mathit{assert}\ ((\mathit{true} \twoheadrightarrow \mathit{inv} \twoheadrightarrow \mathit{inv}) \twoheadrightarrow \mathit{inv} \twoheadrightarrow \mathit{true} \twoheadrightarrow \mathit{inv})\ \overline{\mathit{foldr}}\\&\quad = \mathit{assert}\ ((\mathit{true} \twoheadrightarrow \mathit{true} \twoheadrightarrow \mathit{inv}) \twoheadrightarrow \mathit{inv} \twoheadrightarrow \mathit{true} \twoheadrightarrow \mathit{true})\ \overline{\mathit{foldr}}\end{aligned}$$

where $\overline{\mathit{foldr}} = \lambda f \twoheadrightarrow \lambda e \twoheadrightarrow \lambda x \twoheadrightarrow \mathit{foldr}\ (\lambda a \to \lambda b \to f\ {}_{\langle 17\rangle}\, a\ {}_{\langle 18\rangle}\, b)\ e\ x$ is the contracted version of *foldr*. If we unfold the definition of *assert*, the equation simplifies to

$$\mathit{assert}\ \mathit{inv} \cdot \mathit{foldr}\ \bar{f}\ \bar{e} = \mathit{foldr}\ \hat{f}\ \bar{e} \tag{1}$$

where $\bar{f} = \mathit{assert}\ (\mathit{true} \twoheadrightarrow \mathit{inv} \twoheadrightarrow \mathit{inv})\ f$, $\hat{f} = \mathit{assert}\ (\mathit{true} \twoheadrightarrow \mathit{true} \twoheadrightarrow \mathit{inv})\ f$, and $\bar{e} = \mathit{assert}\ \mathit{inv}\ e$. Equation (1) can be shown either by a simple appeal to *foldr*'s fusion law [10] or using parametricity [11]. In both cases, it remains to prove that

$$\begin{aligned}&\mathit{assert}\ \mathit{inv}\ \bar{e} &&= \bar{e}\\&\mathit{assert}\ \mathit{inv}\ (\bar{f}\ a\ \mathit{as}) &&= \hat{f}\ a\ (\mathit{assert}\ \mathit{inv}\ \mathit{as})\end{aligned}$$

Both parts follow immediately from the idempotence of conjunction: $c\ \&\ c = c$ or more verbosely $\mathit{assert}\ c \cdot \mathit{assert}\ c = \mathit{assert}\ c$, see Sec. 7.

$$
\begin{array}{ll}
\mathit{false}\ \&\ c = \mathit{false} & c_1\ \&\ (c_2\ \&\ c_3) = (c_1\ \&\ c_2)\ \&\ c_3 \\
c\ \&\ \mathit{false} = \mathit{false} & c_1\ \&\ c_2 = c_2\ \&\ c_1 \quad (\dagger) \\
\mathit{true}\ \&\ c = c & c\ \&\ c = c \quad (\dagger) \\
c\ \&\ \mathit{true} = c & \{\, x \mid p_1 \,\}\ \&\ \{\, x \mid p_2 \,\} = \{\, x \mid p_1\ x \wedge p_2\ x \,\}
\end{array}
$$

$$
\begin{array}{c}
\mathit{true} \rightarrowtriangle \mathit{true} = \mathit{true} \\
(c_1 \rightarrowtriangle d_1)\ \&\ (c_2 \rightarrowtriangle d_2) = (c_2\ \&\ c_1) \rightarrowtriangle (d_1\ \&\ d_2) \\
\mathit{true} \times \mathit{true} = \mathit{true} \\
(c_1 \times d_1)\ \&\ (c_2 \times d_2) = (c_1\ \&\ c_2) \times (d_1\ \&\ d_2) \\
[\mathit{true}] = \mathit{true} \\
[c_1\ \&\ c_2] = [c_1]\ \&\ [c_2]
\end{array}
$$

Fig. 5. Properties of contracts.

7 Properties of contracts

In this section we study the algebra of contracts. The algebraic properties can be used, for instance, to optimize contracts: we shall see that $[c_1]\ \&\ [c_2]$ is the same as $[c_1\ \&\ c_2]$, but the latter contract is more efficient. The properties are also helpful for showing that a function satisfies its contract: we have seen that the 'correctness' of *foldr′* relies on $c\ \&\ c = c$.

Up to now we have pretended to work in a strict language: we did not consider bottom in the proofs in the previous section. Let us now switch back to Haskell's non-strict semantics in order to study the algebra of contracts in a more general setting.

It is easy to show that *assert c* is less than or equal to *assert true*:

$$c \preccurlyeq \mathit{true}$$

where '$\preccurlyeq$' denotes the standard information ordering. This property implies, in particular, that *assert c* is *strict*. Note that, for brevity, we abbreviate the law $\mathit{assert}\ c_1 \preccurlyeq \mathit{assert}\ c_2$ by $c_1 \preccurlyeq c_2$ (ditto for equations).

Now, what happens if we apply the same contract twice; is the result the same as applying it once? In other words, is '&' idempotent? One can show that idempotence holds if '&' is commutative (the other cases go through easily). Since '&' is implemented by function composition, commutativity is somewhat doubtful and, indeed, it does not hold in general as the following example shows: let $c_1 = \{\, x \mid \mathit{sum}\ x == 0 \,\}$ and $c_2 = [\mathit{false}]$, then

```
Contracts⟩ length (assert (c1 & c2) [−2, 2])
2
Contracts⟩ length (assert (c2 & c1) [−2, 2])
*** contract failed: the expression '[−2, 2]' is to blame.
Contracts⟩ length (assert ((c1 & c2) & (c1 & c2)) [−2, 2])
*** contract failed: the expression '[−2, 2]' is to blame.
```

The reason is that $[\mathit{false}]$ is not the same as *false* in a lazy setting: the first contract returns a lazy list of contract violations, the second is a contract violation. In a strict setting, commutativity holds trivially as $\mathit{assert}\ c\ x \in \{\, \bot, x \,\}$. The first and the last call demonstrate that idempotence of '&' does not hold for contracts that involve conjunctions, that is, these contracts are not *projections*.

Fig. 5 summarises the properties of conjunctions. Equations that are marked with a (†) only hold in a strict setting. The list combinator and the *independent* variants of '$\rightarrowtriangle$' and '$\times$' are implemented in terms of mapping functions. The remaining laws listed in Fig. 5 are immediate consequences of the well-known functor laws for these maps (bearing in mind that *true* corresponds to *id* and '&' to composition).

8 Related work

Contracts are widely used in procedural and object-oriented (first-order) programming languages [2]. The work on higher-order contracts by Findler and Felleisen [12, 3] has been the main inspiration for this paper. Blume and McAllester [6, **?**] describe a sound and complete model for F&F contracts, which proves that the contract checker discovers all violations, and always assigns blame properly. They show how by restricting the predicate contracts in the F&F language mixing semantics and soundness is avoided, and they show how to regain the expressiveness of the original F&F language by adding general recursive contracts. Furthermore, Findler, Blume, and Felleisen [13] prove many properties about contracts, for example, that contracts are a special kind of *projections* (which have been used to give a meaning to types), and that contracts only modify the behaviour of a program to assign blame. We have implemented contracts as a library in Haskell, using generalised algebraic data types, giving a strongly typed approach to contracts. Our approach allows for a more informative blame assignment. We provide contract constructors for pairs, lists and algebraic data types and a combinator for conjunction. Conjunctions greatly increase the usability of the contract language: they allow the programmer to specify independent properties separately. However, conjunctions also have a disturbing effect on the algebra: in a lazy setting, contracts that include conjunctions are not necessarily projections.

Stating and verifying properties of software is one of the central themes in computer science. The properties of interest range from simple properties like 'this function takes an integer and returns an integer' to complex properties that precisely describe the behaviour of a function like the contract for *insertion-sort* given in Sec. 6.1. Relatively simple properties like Hindley-Milner types can be statically checked by a compiler. To statically prove a complex property for a function it is usually necessary to resort to theorem provers or interactive type-checking tools. Contracts also allow the specification of complex properties; their checking, however, is relegated to run-time. The design space is summarised in the table below.

	static checking	dynamic checking
simple properties	static type checking	dynamic type checking
complex properties	theorem proving	contract checking

Contracts look a bit like types, but they are not. Contracts are dynamic instead of static, and they dynamically change the program. Contracts also differ from dependent types [14]. A dependent type may depend on a value, and may take a different form depending on a value. A contract refines a type (besides changing the behaviour as explained above). Dependently typed programs contain a proof of the fact that the program satisfies the property specified in the type. A contract is only checked, and might fail.

As a characteristic property, contracts are attached to program points, which suggests that they cannot capture general *algebraic properties* such as associativity or distributivity. These properties typically involve several functions or several calls to the same function, which makes it hard to attach them to *one* program point. Furthermore, they do not follow the type structure as required by contracts. As a borderline example, an algebraic property that can be formulated as a contract, since it can be written in a type-directed fashion, is idempotence of a function:

$$f' = \mathit{assert}\ (\mathit{true} \rightarrowtriangle \{\ y \mid y == f\ y\ \})\ (\lambda x \rightarrowtriangle f\ x)$$

In general, however, algebraic properties differ from properties that can be expressed using contracts. In practice, we expect that contract checking is largely complementary to tools that support expressing and testing general algebraic properties such as Quickcheck [15]. We may even observe a synergy: Quickcheck can possibly be a lot more effective in a program that has good contracts.

GHC [5], one of the larger compilers for Haskell, provides *assertions* for expressions: *assert* x returns x only if p evaluates to *True*. The function *assert* is a strict function. Chitil et al. [16] show how to define *assert* lazily. In contrast to contracts, assertions do not assign blame: if the precondition of a function is not satisfied, the function is blamed. Furthermore, contracts are type directed, whereas an assertion roughly corresponds to a contract comprehension.

9 Conclusion

We have introduced an embedded domain-specific language for typed, higher-order and first-class contracts, which is both more expressive than previous proposals, and allows for a more informative blame assignment. The contract language is implemented as a library in Haskell using the concept of generalised algebraic data types. We have taken some first steps towards an algebra of contracts, and we have shown how to define a generic contract combinator for arbitrary algebraic data types.

We left a couple of topics for future work. We intend to take an existing debugger or tracer for Haskell, and use the available information about source locations to let blaming point to real source locations, instead of user-supplied locations as supported by the implementation described in this paper. Furthermore, we want to turn the algebra for contracts into a more or less complete set of laws for contracts.

Acknowledgements We are grateful to Matthias Blume, Matthias Felleisen, Robby Findler and the five anonymous referees for valuable suggestions regarding content and presentation. Special thanks go to Matthias Blume and referee #5 for pointing out infelicities in the previous implementation of blame assignment.

References

1. Meyer, B.: Applying 'design by contract'. IEEE Computer **25** (1992) 40–51
2. Meyer, B.: Eiffel: The Language. Prentice Hall (1992)
3. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. ACM SIGPLAN Notices **37** (2002) 48–59
4. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press (2003)
5. The GHC Team: The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4.1. (2005) Available from `http://www.haskell.org/ghc/`.
6. Blume, M., McAllester, D.: A sound (and complete) model of contracts. ACM SIGPLAN Notices **39** (2004) 189–200
7. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: POPL '03, ACM Press (2003) 224–235
8. Hinze, R.: Fun with phantom types. In Gibbons, J., de Moor, O., eds.: The Fun of Programming. Palgrave Macmillan (2003) 245–262 ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.
9. Hinze, R.: Polytypic values possess polykinded types. Science of Computer Programming **43** (2002) 129–159
10. Hutton, G.: A tutorial on the universality and expressiveness of fold. Journal of Functional Programming **9** (1999) 355–372
11. Wadler, P.: Theorems for free! In: The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, UK, Addison-Wesley Publishing Company (1989) 347–359
12. Findler, R.B.: Behavioral software contracts (dissertation). Technical Report TR02-402, Department of Computer Science, Rice University (2002)
13. Findler, R.B., Blume, M., Felleisen, M.: An investigation of contracts as projections. Technical Report TR-2004-02, The University of Chicago (2004)
14. Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's Type Theory. Oxford University Press (1990)
15. Claessen, K., Runciman, C., Chitil, O., Hughes, J., Wallace, M.: Testing and tracing lazy functional programs using Quickcheck and Hat. In Jeuring, J., Peyton Jones, S., eds.: Advanced Functional programming. Volume 2638 of Lecture Notes in Computer Science., Springer-Verlag (2003)
16. Chitil, O., McNeill, D., Runciman, C.: Lazy assertions. In Trinder, P., Michaelson, G., Peña, R., eds.: Implementation of Functional Languages: 15th International Workshop, IFL 2003, Edinburgh, UK, September 8–11, 2003. Volume 3145 of Lecture Notes in Computer Science., Springer-Verlag (2004) 1–19