Trinity

Andres Löh

joint work with Ralf Hinze

Utrecht University

January 11, 2008

About me

- PhD at Utrecht University, 2004: "Exploring Generic Haskell"
- 2005–2007 PostDoc at Bonn University, working with Ralf Hinze
- since August 2007: lecturer at Utrecht University
- interests:
  - functional programming (Haskell),
  - polytypic / datatype-generic programming,
  - type systems (dependent types)

What is Trinity?

- Trinity is a programming language designed by Ralf Hinze and me.
- It is called Trinity because it supports
  - functional programming (strict, impure),
  - imperative programming,
  - object-oriented programming.
- It looks a bit like ML (OCaml), but that is an accident . . .

What is Trinity?

- Trinity is a programming language designed by Ralf Hinze and me.
- It is called Trinity because it supports
  - functional programming (strict, impure),
  - imperative programming,
  - object-oriented programming.
- It looks a bit like ML (OCaml), but that is an accident . . .

Let's have a look at some programs . . .

Hello world

```
put-line "Hello world"
```

Factorial

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

Overview

History of Trinity

- In the summer of 2006, Ralf Hinze devised fragments of a programming language for a master-level course “Prinzipien von Programmiersprachen” (Principles of Programming Languages).
- It was decided at Bonn that there should be an introductory (first-year) course on PL concepts (with an expected 200 students). Ralf was supposed to teach that course, too.
- Ralf had already written a toy implementation of his language (without the typesystem) in Haskell in one weekend.
- The idea was to reuse the language for the new course, and have an implementation for the students to play with, to make the course less theoretical.

History – contd.

- I joined at that point.
- We started from Ralf's original implementation, added types and several more language features. Most language concepts were revised (and often simplified) during the implementation.
- The language was used under the name "BPL" (Bonn Programming Language or Beginner's Programming Language) in the course. Student reactions were mixed.
- Continued development toward a public release after the course in 2007. Renamed to Trinity. Stalled due to both of us moving universities, but picked up the work during the Christmas break.

Design goals of Trinity

- Different paradigms:
  - value-oriented (functional) programming
  - effect-oriented (imperative) programming
  - object-oriented programming
- Many concepts and language features. (**Not** a small language.)
- Simple, orthogonal concepts. No artificial restrictions.
- Types!
- Relatively little amount of syntactic sugar, few convenience features. (Writing large programs is not a primary goal.)
- Clearly defined syntax and semantics.
- Presentable in an incremental way.

Design goals for the implementation

- Straight-forward implementation. Easy to understand (at least for us).
- Easy to extend (at least for us).
- Convenient to use.
- Sufficiently fast to run small example programs.

Tour of Trinity

- Example concepts.
- Feature overview (not complete).
- Various degrees of detail.
- Focus on the functional part.

Booleans – syntax

Expressions:

$$\begin{aligned} e ::= {} & \ldots \\ \mid {} & \textbf{true} \\ \mid {} & \textbf{false} \\ \mid {} & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \end{aligned}$$

Types:

$$\begin{aligned} \tau ::= {} & \ldots \\ \mid {} & \mathsf{Bool} \end{aligned}$$

Fragments are presented as extensions to the syntax.
Notational convention: types in red.

Booleans – type rules

$$\frac{}{\Sigma \vdash \textbf{false} : \mathsf{Bool}} \qquad \frac{}{\Sigma \vdash \textbf{true} : \mathsf{Bool}}$$

The signature Σ is an environment mapping identifiers to types.

Booleans – type rules

$$\frac{}{\Sigma \vdash \textbf{false} : \mathsf{Bool}} \qquad \frac{}{\Sigma \vdash \textbf{true} : \mathsf{Bool}}$$

The signature Σ is an environment mapping identifiers to types.

$$\frac{\Sigma \vdash e_1 : \mathsf{Bool} \quad \Sigma \vdash e_2 : \tau \quad \Sigma \vdash e_3 : \tau}{\Sigma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

Booleans – evaluation rules

Values:

$$
\begin{aligned}
\mathsf{v} ::= \;& \ldots \\
\mid \;& \mathbf{false} \\
\mid \;& \mathbf{true}
\end{aligned}
$$

Values are the results of evaluation.

$$\frac{}{\mathbf{false} \Downarrow \mathbf{false}} \qquad \frac{}{\mathbf{true} \Downarrow \mathbf{true}}$$

$$\frac{\mathsf{e}_1 \Downarrow \mathbf{true} \quad \mathsf{e}_2 \Downarrow \mathsf{v}}{\mathbf{if}\ \mathsf{e}_1\ \mathbf{then}\ \mathsf{e}_2\ \mathbf{else}\ \mathsf{e}_3 \Downarrow \mathsf{v}} \qquad \frac{\mathsf{e}_1 \Downarrow \mathbf{false} \quad \mathsf{e}_3 \Downarrow \mathsf{v}}{\mathbf{if}\ \mathsf{e}_1\ \mathbf{then}\ \mathsf{e}_2\ \mathbf{else}\ \mathsf{e}_3 \Downarrow \mathsf{v}}$$

Big-step semantics . . .

Excursion: Evolution of the evaluation rules

$$\mathsf{e} \Downarrow \mathsf{v}$$

Starting point.

$$\sigma_1 \parallel \mathsf{e} \Downarrow \mathsf{v} \parallel \sigma_2$$

Evaluation with a store: for mutable state (references).

$$\mathsf{e} \Downarrow_{\mathsf{t}} \mathsf{v}$$

Evaluation with an external effect: for input/output.

$$\kappa \mid \mathsf{e} \Downarrow \mathsf{v}$$

Stack-based evaluation: for exceptions, continuations.

Booleans – derived forms

Boolean operators are introduced as syntactic sugar:

$$
\begin{aligned}
e_1 \mathbin{\&\&} e_2 &\equiv \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else false} \\
e_1 \mathbin{||} e_2 &\equiv \textbf{if}\ e_1\ \textbf{then true else}\ e_2 \\
\mathrm{not}\ e &\equiv \textbf{if}\ e\ \textbf{then false else true}
\end{aligned}
$$

In truth, 'not' is just a predefined function.

Natural numbers

Natural numbers are the only numeric type:

- Many standard functions work on natural numbers, not integers.
- Floating-point computation is rarely required while teaching programming languages.
- Implementing other numeric types such as integers or rationals using natural numbers actually serves as a nice exercise for datatypes.

$$\begin{array}{lll} \mathsf{e} ::= & \ldots & \\ \quad \mid & 0, 1, 2, \ldots & \text{(numeric literal)} \end{array}$$

Operators

There is a fixed set of operators on natural numbers:

$$
\begin{aligned}
e ::= \; & \ldots \\
\mid \; & e_1 + e_2 \\
\mid \; & e_1 - e_2 \\
\mid \; & e_1 * e_2 \\
\mid \; & e_1 \div e_2 \\
\mid \; & e_1 \,\%\, e_2 \\
\mid \; & e_1 \lneq e_2 \text{ (concrete syntax: \#<)} \\
\mid \; & e_1 \leqslant e_2 \text{ (concrete syntax: =<)} \\
\mid \; & e_1 == e_2 \text{ (concrete syntax: ==)} \\
\mid \; & e_1 \neq e_2 \text{ (concrete syntax: /=)} \\
\mid \; & e_2 \gneq e_2 \text{ (concrete syntax: >\#)} \\
\mid \; & e_2 \geqslant e_2 \text{ (concrete syntax: >=)}
\end{aligned}
$$

Operator syntax is chosen such that arrow symbols and angle brackets remain free.

Declarations

Declarations:

$$
\begin{aligned}
d ::= {} & \ldots \\
\mid {} & \textbf{val}\ x = e \\
\mid {} & d_1\ d_2 && \text{(sequencing)} \\
\mid {} & \textbf{local}\ d_1\ \textbf{in}\ d_2\ \textbf{end}
\end{aligned}
$$

Expressions:

$$
\begin{aligned}
e ::= {} & \ldots \\
\mid {} & x && \text{(identifier)} \\
\mid {} & \textbf{let}\ d\ \textbf{in}\ e\ \textbf{end}
\end{aligned}
$$

Declarations and expressions are kept separate ...

Expressions vs. declarations

$$\Sigma \vdash \mathsf{e} : \tau \qquad \Sigma_1 \vdash \mathsf{d} : \Sigma_2$$

The “types” of declarations are signatures.

Expressions vs. declarations

$$\Sigma \vdash \mathsf{e} : \tau \qquad \Sigma_1 \vdash \mathsf{d} : \Sigma_2$$

The "types" of declarations are signatures.

$$\mathsf{e} \Downarrow \mathsf{v} \qquad \mathsf{d} \Downarrow \delta$$

The evaluation results of declarations are environments mapping identifiers to values.

Declarations – types and evaluation examples

Declaration of an identifier:

$$\frac{\Sigma \vdash \mathsf{e} : \tau}{\Sigma \vdash \textbf{val}\ \mathsf{x} = \mathsf{e} : \{\mathsf{x} \mapsto \tau\}} \qquad \frac{\mathsf{e} \Downarrow \mathsf{v}}{\textbf{val}\ \mathsf{x} = \mathsf{e} \Downarrow \{\mathsf{x} \mapsto \mathsf{v}\}}$$

Declarations – types and evaluation examples

Declaration of an identifier:

$$\frac{\Sigma \vdash \mathsf{e} : \tau}{\Sigma \vdash \textbf{val}\ \mathsf{x} = \mathsf{e} : \{\mathsf{x} \mapsto \tau\}} \qquad \frac{\mathsf{e} \Downarrow \mathsf{v}}{\textbf{val}\ \mathsf{x} = \mathsf{e} \Downarrow \{\mathsf{x} \mapsto \mathsf{v}\}}$$

Sequence of declarations:

$$\frac{\Sigma_1 \vdash \mathsf{d}_1 : \Sigma_2 \quad \Sigma_1, \Sigma_2 \vdash \mathsf{d}_2 : \Sigma_3}{\Sigma_1 \vdash \mathsf{d}_1\ \mathsf{d}_2 : \Sigma_2, \Sigma_3} \qquad \frac{\mathsf{d}_1 \Downarrow \delta_1 \quad \mathsf{d}_2\delta_1 \Downarrow \delta_2}{\mathsf{d}_1\ \mathsf{d}_2 \Downarrow \delta_1, \delta_2}$$

Later declarations can refer to earlier ones.

Declarations – types and evaluation examples

Declaration of an identifier:

$$\frac{\Sigma \vdash \mathsf{e} : \tau}{\Sigma \vdash \textbf{val}\ \mathsf{x} = \mathsf{e} : \{\mathsf{x} \mapsto \tau\}} \qquad \frac{\mathsf{e} \Downarrow \mathsf{v}}{\textbf{val}\ \mathsf{x} = \mathsf{e} \Downarrow \{\mathsf{x} \mapsto \mathsf{v}\}}$$

Sequence of declarations:

$$\frac{\Sigma_1 \vdash \mathsf{d}_1 : \Sigma_2 \quad \Sigma_1, \Sigma_2 \vdash \mathsf{d}_2 : \Sigma_3}{\Sigma_1 \vdash \mathsf{d}_1\ \mathsf{d}_2 : \Sigma_2, \Sigma_3} \qquad \frac{\mathsf{d}_1 \Downarrow \delta_1 \quad \mathsf{d}_2\delta_1 \Downarrow \delta_2}{\mathsf{d}_1\ \mathsf{d}_2 \Downarrow \delta_1, \delta_2}$$

Later declarations can refer to earlier ones.

Let:

$$\frac{\Sigma_1 \vdash \mathsf{d} : \Sigma_2 \quad \Sigma_1, \Sigma_2 \vdash \mathsf{e} : \tau}{\Sigma_1 \vdash \textbf{let}\ \mathsf{d}\ \textbf{in}\ \mathsf{e}\ \textbf{end} : \tau} \qquad \frac{\mathsf{d} \Downarrow \delta \quad \mathsf{e}\delta \Downarrow \mathsf{v}}{\textbf{let}\ \mathsf{d}\ \textbf{in}\ \mathsf{e}\ \textbf{end} \Downarrow \mathsf{v}}$$

Excursion: Implementation

$$\text{Trinity} \xrightarrow{\text{lex}} \text{tokens} \xrightarrow{\text{parse}} \text{abstract syntax} \xrightarrow{\text{desugar}} \text{core} \xrightarrow{\text{evaluate}} \text{value}$$

Excursion: Implementation

$$\text{Trinity} \xrightarrow{\text{lex}} \text{tokens} \xrightarrow{\text{parse}} \text{abstract syntax} \xrightarrow{\text{desugar}} \text{core} \xrightarrow{\text{evaluate}} \text{value}$$

- Typechecking is an **optional** phase that operates on the abstract syntax – it does not change the program.
- The core language has no syntactic sugar, is untyped, and is more uniform than the surface language – for example, declarations are expressions, consequently first-class environments.
- Evaluation as an abstract machine (bonus: tracing of evaluation).

Excursion: syntax

- Keyword- rather than symbol oriented.
- All declarations start with a keyword.
- Most constructs have a closing keyword.
- No braces, no separator between declarations required, neither is a layout rule.
- The whole Trinity grammar is LR without any twisting.

Functions

Function application:

$$\texttt{f@x} \quad \text{or} \quad \mathsf{f}\ \mathsf{x}$$

Anonymous function:

$$\textbf{fun}\ \mathsf{x} \Rightarrow \mathsf{e}$$

(General) recursion:

$$\textbf{rec}\ \mathsf{x} \Rightarrow \mathsf{e}$$

Syntactic sugar: recursive function:

$$(\textbf{function}\ \mathsf{f}\ \mathsf{x} = \mathsf{e}) \equiv (\textbf{val}\ \mathsf{f} = \textbf{rec}\ \mathsf{f} \Rightarrow \textbf{fun}\ \mathsf{x} \Rightarrow \mathsf{e})$$

More types

- strings
  `"Hello world"` : String
- tuples, records (non-extensible, no first-class labels)
  (1, `"c"`) : (Nat, String)
  (x = 1, y = `"c"`) : (x = Nat, y = String)
- type definitions
  type coordinates = (Nat, Nat)
- datatypes
  data Bool = False | True

Type definitions and datatypes are declarations. There is no such concept as top-level declarations.

Datatypes

$\textbf{data}\ \mathsf{List}\ \langle \mathsf{a} \rangle = \mathsf{Nil} \mid \mathsf{Cons}\ (\mathsf{a}, \mathsf{List}\ \langle \mathsf{a} \rangle)$

- Constructors must be fully applied, and can only have zero or one argument.
- Datatypes can be recursive.
- Datatypes can be parameterized.
- Type application is written using angle brackets.
- Experimental: Currently all datatypes are open.

Datatypes – contd.

(Parameterized) Datatypes lead naturally to pattern matching and polymorphism.

$$
\begin{array}{l}
\textbf{function}\ \mathsf{append}\ \langle \mathsf{a} \rangle\ (\mathsf{xs} : \mathsf{List}\ \langle \mathsf{a} \rangle, \mathsf{ys} : \mathsf{List}\ \langle \mathsf{a} \rangle) = \\
\quad \textbf{case}\ \mathsf{xs}\ \textbf{of} \\
\quad \quad \mathsf{Nil} \qquad\qquad\quad\ \Rightarrow \mathsf{ys} \\
\quad \mid \mathsf{Cons}\ (\mathsf{z}, \mathsf{zs}) \Rightarrow \mathsf{Cons}\ (\mathsf{z}, \mathsf{append}\ (\mathsf{zs}, \mathsf{ys})) \\
\quad \textbf{end}
\end{array}
$$

- Type abstraction is explicit.
- Type application can be omitted in most situations, but can also be given explicitly.
- Thus higher-ranked (even impredicative) polymorphism.

Pattern language

Patterns form their own syntactic category.
General and- and or-patterns:

> $p_1 \mathbin{\&} p_2$ (generalization of Haskell as-patterns)
> $p_1 \mid p_2$ (both patterns have to bind the same variables)

Example:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Maybe}\ \langle \mathsf{a} \rangle = \mathsf{Nothing} \mid \mathsf{Just}\ \mathsf{a} \\
\textbf{function}\ \mathsf{plus}\ (\mathsf{x} : \mathsf{Maybe}\ \langle \mathsf{Nat} \rangle, \mathsf{y} : \mathsf{Maybe}\ \langle \mathsf{Nat} \rangle) : \mathsf{Maybe}\ \langle \mathsf{Nat} \rangle = \\
\quad \textbf{case}\ (\mathsf{x}, \mathsf{y})\ \textbf{of} \\
\qquad (\mathsf{Nothing}, \mathsf{z}) \mid (\mathsf{z}, \mathsf{Nothing}) \Rightarrow \mathsf{z} \\
\quad\ \mid (\mathsf{Just}\ \mathsf{x}, \mathsf{Just}\ \mathsf{y}) \qquad\qquad\quad \Rightarrow \mathsf{Just}\ (\mathsf{x} + \mathsf{y}) \\
\quad \textbf{end}
\end{array}
$$

Datatypes – Subtleties

Treating datatypes as "normal" declarations leads to subtle semantics:

```
let data X = C Bool
  function f (x : X) : Bool = case x of C y ⇒ not y end
  data X = C Nat
in
  f (C 42)
end
```

Arrays and References

Parameterized built-in types.

$[1, 2, 3] : \mathsf{Array}\ \langle \mathsf{Nat} \rangle$
$\textbf{array}\ [10]\ i \Rightarrow i * i : \mathsf{Array}\ \langle \mathsf{Nat} \rangle$

```
let
   val x : Ref ⟨Nat⟩ = ref 2
in
   x := !x + 1; !x
end
```

- References introduce impurity.
- The sequencing operator is syntactic sugar:
  $e_1; e_2 \equiv \textbf{let val}\ \_ = e_1\ \textbf{in}\ e_2\ \textbf{end}$

Subtlety: effects and polymorphism

$$\textbf{val}\ \mathsf{r} : \textbf{forall}\ \langle \mathsf{a} \rangle \Rightarrow \mathsf{Ref}\ \langle \mathsf{Maybe}\ \langle \mathsf{a} \rangle \rangle =$$
$$\quad \textbf{fun}\ \langle \mathsf{a} \rangle \Rightarrow \textbf{ref}\ \mathsf{Nothing}$$

ML has the value restriction to prevent polymorphic values like this.

Subtlety: effects and polymorphism

val r : **forall** ⟨a⟩ ⇒ Ref ⟨Maybe ⟨a⟩⟩ =
  fun ⟨a⟩ ⇒ **ref** Nothing

ML has the value restriction to prevent polymorphic values like this.
In Trinity, the above is a function, i.e., it is delayed even though it only depends on a type argument. Type application for polymorphic values is explicit and triggers the effect.

r := 0 is a type error

r ⟨Nat⟩ := 0 works, but creates a new reference cell

More effectful features

- built-in input/output functions
- exceptions

Yet more features

- continuations
- objects
- subtyping for objects and records
- modules
- experimental: delimited continuations, contracts, run-time typing, functors, ...

Some of these features are still somewhat experimental.

End of the tour – the implementation

The implementation:

- abstract machine interpreters are reasonably fast and extremely easy to implement in Haskell
- is still relatively small: 12500 kloc including comments
- is very straight-forward (except maybe for the type-checker)
- comes with a test suite of currently about 200 tests, some of them being medium-sized example programs

In spirit, Trinity is a type-checked language, but the implementation is too liberal at the moment and can infer quite a lot

Plans

- Fix a few remaining design decisions.
- Clearly identify a core set of features.
- Extract documentation from lecture notes. Completely document the core set of features.
- Implement more convenience features for the interpreter (and a GUI version?).
- Make implementation more systematic. Ideally, prove correctness of the implementation by using Coq or a similar system.
- Experiment with additional language concepts.
- Make it easier and more systematic to add more primitive (read: foreign) functions.
- Use it in other courses (Stefan Holdermans is currently using a Trinity-subset in the "Implementation of Programming Languages" course at Utrecht as the language that is implemented).

Conclusions

- Every Haskell programmer should write an interpreter for his or her favourite programming language.
- If you like Trinity, you can play with it. Just send me a mail or wait for the official release (hopefully soon) – it's GPL.
- For Utrecht students: there are definitely experimentation projects, and possibly master projects available on the topic of Trinity.