Improved handling of mixfix operators: Design and implementation

Nils Anders Danielsson
Joint work with Ulf Norell

AIM8, Göteborg, 2008-05-29

This is an outline of, and motivation for, what I intend to implement during my code sprint.

Aim for Agda's syntax

To strike a nice balance between:

- Ease of parsing for humans.
- Light-weight, elegant syntax, with the possibility to use domain-specific notations.

"Easy and fun."

Method

- Simple rules.
- Large selection of characters/symbols (Unicode).
- Mixfix operators.
- Tool support (syntax highlighting, go-to-definition).

Mixfix operators

Easy to declare:

```
_≡_ if_then_else_ ⟦_⟧ _!
```

Or are they? How should

```
if n ! ≡ 3 then v else ⟦ e ⟧ ρ
```

be parsed?

Standard solution: Use fixity declarations.

Fixity

Fixity declarations specify two things:

Precedence	Result of parsing `x + y * z`
`_+_ < _*_`	`x + (y * z)`
`_+_ > _*_`	`(x + y) * z`
Neither	Parse error

Associativity	Result of parsing `x + y + z`
Left	`(x + y) + z`
Right	`x + (y + z)`
Neither	Parse error

Wart

Currently Agda's precedence relation is a linear order.

- OK for C, with fixed set of operators.
- Barely OK for Haskell.
  Works since relatively few operators are defined (due to limited set of operator characters).
- Inadequate for Agda.

Wart

Why should _+_ and _∧_ be related?

- Fewer operators related ⇒ easier for humans to parse (assuming syntactically correct code).
- Operators should be related only if this is the intention of the programmer.

Wart

And why specify precedences using numbers?

- "_∗_ binds tighter than _+_"
  is easier to remember than
  "_∗_ has precedence 7 and _+_ precedence 6".

Partial order?

- Assume $\_\wedge\_ < \_\equiv\_$ and $\_\equiv\_ < \_+\_$.
- If we do not want $\_\wedge\_ < \_+\_$, then the precedence relation cannot be transitive.

Precedence relations

- Directed acyclic graphs.
- Front-end can restrict this further.
- Acyclicity ensures that we cannot have both `_+_ < _*_` and `_+_ > _*_`.
- One or more operators per node.
- Each operator has an associated associativity.

Semantics

Given a DAG we construct a context-free grammar.
Assume one infix, non-associative operator per node.

$$
\begin{aligned}
expr &::= atom \mid \bigvee \{\, n_i \mid n_i \text{ in graph} \,\} \\
n_i &::= n_i{\uparrow}\ op_i\ n_i{\uparrow} \\
n_i{\uparrow} &::= atom \mid \bigvee \{\, n_j \mid n_i < n_j \,\} \\
op_i &::= op_i^1\ expr\ op_i^2\ expr\ \cdots\ op_i^k \\
atom &::= \ldots
\end{aligned}
$$

Acyclic graph $\Rightarrow$ grammar not left or right recursive.

Example

- Two operators: _⊢_:_ and `_,_`.
- Intended use: `ε , x , y , z ⊢ e : τ`.
- `_,_` binds tighter than _⊢_:_.
- `_,_` is left associative.

Example

$\_\vdash\_:\_ \;<\; \_,\_$

$\_,\_$ is left associative

$$\begin{aligned}
\mathit{expr} &::= \mathit{atom} \mid \mathit{type} \mid \mathit{comma} \\
\mathit{type} &::= \mathit{type}{\uparrow}\ \mathit{type}_{op}\ \mathit{type}{\uparrow} \\
\mathit{type}{\uparrow} &::= \mathit{atom} \mid \mathit{comma} \\
\mathit{type}_{op} &::= \vdash \mathit{expr}\quad : \\
\mathit{comma} &::= (\mathit{comma}{\uparrow}\ \mathit{comma}_{op})^{+}\ \mathit{comma}{\uparrow} \\
\mathit{comma}{\uparrow} &::= \mathit{atom} \\
\mathit{comma}_{op} &::= , \\
\mathit{atom} &::= \ldots
\end{aligned}$$

Simplified

$\_\vdash\_{:}\_ \ < \ \_\,,\_$
$\_\,,\_$ is left associative

$$
\begin{aligned}
\mathit{expr} &::= \mathit{atom} \mid \mathit{type} \mid \mathit{comma} \\
\mathit{type} &::= \mathit{type}{\uparrow} \ \vdash \ \mathit{expr} \ : \ \mathit{type}{\uparrow} \\
\mathit{type}{\uparrow} &::= \mathit{atom} \mid \mathit{comma} \\
\mathit{comma} &::= (\mathit{atom}\ ,)^{+}\ \mathit{atom} \\
\mathit{atom} &::= \ldots
\end{aligned}
$$

Anything new?

Aasa also considers parsing of mixfix operators, with similar approach.

- Her approach is more restricted (e.g. linear order), but ensures non-ambiguity.
- More importantly, she seems to try to maximise the number of syntactically correct expressions.
- Our approach: fewer parse correct expressions, but arguably easier to understand.

Example

Assume $\neg\_ < \_\land\_$. What about `a` $\land$ $\neg$ `b`?

- Our approach: No parse since $\_\land\_ \not< \neg\_$.
- Aasa: `a` $\land$ `(`$\neg$ `b).`

New feature: sections

- Sections will also be supported.
- Examples:
  - `2 +_` $\mapsto$ `\x -> 2 + x`.
  - `_+ 2` $\mapsto$ `\x -> x + 2`.
  - `if b then_else_`
    $\mapsto$ `\x y -> if b then x else y`.
- Straightforward to add to grammar; distinguish initial, internal and final _ in lexical syntax to avoid combinatory explosion.

Implementation

1. Parse program, treating expressions as flat token lists.
2. Scope checking, fixity declarations.
3. Parse expressions using dynamically generated context-free grammar.

Implementation

- Parser combinators obvious choice.
- The grammar is heavily non-left-factorised $\Rightarrow$ cannot use arbitrary parser combinator library.
- Memoised backtracking parser combinators provide sufficient efficiency.
- Memoisation fits nicely into the parser combinator interface.

Fixity declarations

- How do we specify the precedence DAG?
- Most tentative part of the design.
- Feedback extra welcome.

Fixity declarations

Important criteria:

1. The relationship between operators must be fixed once they are both defined. Later declarations may not change this relationship.
2. Order of declarations must not matter. If two declarations can be reordered, then this should not affect the resulting DAG. (Consider reordering import statements, for instance.)

Suggested scheme

`infix` [`left`|`right`] *ops* = *op*

The operators *ops*, which are (left|right|non-) associative, have the same precedence as *op*.

Suggested scheme

$\texttt{infix}$ [$\texttt{left}$|$\texttt{right}$] *ops* [$\texttt{<}$ $\texttt{(}$$ops_{<}$$\texttt{)}$] [$\texttt{>}$ $\texttt{(}$$ops_{>}$$\texttt{)}$]

The operators *ops* bind looser than the operators $ops_{<}$, and tighter than $ops_{>}$. (Unless this introduces a cycle.)

Lack of transitivity might lead to huge fixity declarations. Can this be avoided (elegantly)?

- $\texttt{infix (}ops_1\texttt{) < (}ops_2\texttt{) < (}ops_3\texttt{) <}\ldots$
- Module names in operator lists.

Suggested scheme

$\texttt{infix}$ $[\texttt{left}|\texttt{right}]$ ops $[\texttt{<}\ (\mathit{ops}_<)]$ $[\texttt{>}\ (\mathit{ops}_>)]$

The operators *ops* bind looser than the operators $\mathit{ops}_<$, and tighter than $\mathit{ops}_>$. (Unless this introduces a cycle.)

Lack of transitivity might lead to huge fixity declarations. Can this be avoided (elegantly)?

- $\texttt{infix}\ (\mathit{ops}_1)\ \texttt{<}\ (\mathit{ops}_2)\ \texttt{<}\ (\mathit{ops}_3)\ \texttt{<}\dots$
- Module names in operator lists.

Some notes

- More parentheses or more fixity declarations will be necessary.
  - But that is the point of this exercise.
- No least precedence level $\Rightarrow$ _$_ is harder to define.

Summary

- New semantics of precedences.
- New method for specifying precedences.
- No need to specify syntactic relation between semantically unrelated operators.
- Hopefully this new approach makes it (slightly) easier to read programs.

Feedback?