Linear Mini-Core

J.-P. Bernardy, R. Eisenberg, M. Boespflug, R. Newton, S. Peyton Jones, and A. Spiwack

This document is the specification of Core's linear types as they are being implemented in GHC. To enable Core linting for linear types in GHC use `-dlinear-core-lint`.

This document is the authoritative specification of linear Core lint, and will remain so until linear Core lint has become stable enough and is merged into `-dcore-lint`. In which case the specification in this document will be merged with the ghc specification in the main repository.

The specification in this document is a simplified version of the whole Core, focusing on the parts which interact with linearity.

This document is hosted on the wiki at https://gitlab.haskell.org/ghc/ghc/-/wikis/linear-types/implementation.

The sources are https://github.com/tweag/linear-types/blob/master/minicore.lhs.

1 Differences between Linear Core and $\lambda^q_\to$

This section summarises the main differences between the Core specification and the $\lambda^q_\to$ calculus described in the Linear Haskell paper.

1.1 The case-binder

In $\lambda^q_\to$, the case construction has the form

$$\mathsf{case}_\pi\ t\ \mathsf{of}\ \{c_k\ x_1 ... x_{n_k} \to u_k\}_{k=1}^m$$

In Core, it has an additional binder

$$\mathsf{case}\ t\ \mathsf{of}\ z\ \{c_k\ x_1 ... x_{n_k} \to u_k\}_{k=1}^m$$

The additional variable z, called the case binder, is used in a variety of optimisation passes, and also represents variable patterns.

A proper handling of the case binder is key, in particular, to the compilation of deep pattern matching.

A difficulty is that for linear case, the case binder cannot be used at the same time as the variables from the pattern: it would violate linearity. Additionally the case binder is typically used in some, but not all branches.

1.2 Case branches

Branches of a case expression, in Core, differ from the article description of $\lambda^q_{\to}$ in two ways.

Non-exhaustive The left-hand-side of branches, in Core, need to be distinct constructors, but, contrary to $\lambda^q_{\to}$, Core doesn't require that the case expression be exhaustive: there may be missing patterns.

Wildcard The left-hand side of one of the branches can be a wildcard pattern, written _.

Non-exhaustive case expressions do not cause any additional problem: a pattern-matching failure simply raises an imprecise exception as usual. This is equivalent to having an exhaustive case expression, with `error` as the right-hand side of the wildcard pattern.

1.3 Let binders

Core-to-core passes play with let-binders in many ways (they are floated out, or in, several can be factored into one, they can be inlined in some of their sites. Join points, for instance combine many of these characteristics).

Some of these transformations are fundamentally incompatible with standard linear logic rules for let-binders. But they remain semantic persevering, hence, semantically, they preserve linearity. Therefore, these transformations must be modelled in the Core typing rules, even if this means unusual typing rules.

We'd like to stress out that these rules for typing let binders only apply to core. Let binders in the surface language behave as in the paper and in linear logic, which is much easier to reason about.

For instance consider the following

```
f (Just False) (Just False) = e1
f _            _            = e2
```

It would desugar to:

```
f = λx y →
  join fail = e2 in
    case x of x'
    { Just j → case j of j'
      { False → case y of y'
        { Just k → case k of k'
          { False → e1
          ; True  → fail }
        ; Nothing → fail }
      ; True → fail }
    ; Nothing → fail }
```

The standard typing rule for let-binders requires fail to be used linearly in every branch, but it isn't: fail is not used at all in the False → e1 branch. The standard typing rule for let-binders also enforces that the linear free variables in e2 are not used at all. But e1 necessarily has exactly the same linear free variables as e2, hence the linear free variables of e2 are all used in the False → e1 branch.

On the other hand, notice that if we'd inline fail and duplicate e2 everywhere, the term would indeed be well-typed. So, we have to teach the linter that using fail is the same thing as using e2.

Note: the typing rule for let-binders in linear mini-core can encoded in linear logic, *e.g.* with lambda-lifting, which justifies the claim that it preserves linearity. However, this encoding is *not* macro-expressible (to the best of our knowledge), therefore this typing rule strictly increases the expressiveness of linear mini-core.

Note: this only affect non-recursive let-binders. Recursive lets have all their binders at multiplicity ω (it isn't clear that a meaning could be given to a non-ω recursive definition).

2 Linear Mini-Core

2.1 Syntax

The syntax is modified to include case binders. See Fig. 1.

2.2 Static semantics

See Fig. 2. The typing rules depend on an equality on multiplicities as well as an ordering, and operations on context, which are defined in Figure 3.

Typing let-binders A program which starts its life as linear may be transformed by the optimiser to use a join point (a special form of let-binder). In this example, both p and q are used linearly.

```
case y of y'
{A → p − q
; B → p + q
; C → p + q
; D → p ∗ q
}
```

Let's say that we want to transform p + q into a join-point call, we need the following to be linear in p and q.

```
join j = p + q in
  case y of y'
  {A → p − q
```

Multiplicities

$$\pi, \mu ::= 1 \mid \omega \mid p \mid \pi + \mu \mid \pi \cdot \mu$$

Types

$$A, B ::= A \to_\pi B \mid \forall p.A \mid D\ p_1\ ...\ p_n$$

Contexts

$$\Gamma, \Delta ::= (x : A), \Gamma \mid (x :_\Delta A), \Gamma \mid -$$

Usage

$$U, V ::= x \mapsto \mu \mid -$$

Datatype declaration

$$\mathsf{data}\ D\ p_1\ ...\ p_n\ \mathsf{where}\left(c_k : A_1 \to_{\pi_1} ... \ A_{n_k} \to_{\pi_{n_k}} D\right)_{k=1}^{m}$$

Case alternatives

$b ::= c\ x_1 ... x_n \to u$	data constructor
$\mid \_ \to u$	wildcard

Terms

$e, s, t, u ::= x$	variable
$\mid \lambda(x :_\pi A).t$	abstraction
$\mid t\ s$	application
$\mid \lambda p.t$	multiplicity abstraction
$\mid t\ \pi$	multiplicity application
$\mid c\ t_1 ... t_n$	data construction
$\mid \mathsf{case}\ t\ \mathsf{of}\ z :_\pi A\ \{b_k\}_{k=1}^m$	case
$\mid \mathsf{let}\ x :_U A = t\ \mathsf{in}\ u$	let
$\mid \mathsf{let\ rec}\ x_1 :_{U_1} A_1 = t_1 ... x_n :_{U_n} A_n = t_n\ \mathsf{in}\ u$	letrec

Judgements

$\Gamma \vdash t : A \rightsquigarrow \{U\}$	typing judgement
$\Gamma; z; D\ \pi_1 ... \pi_n \vdash_\pi b : C \rightsquigarrow \{U\}$	case-alternative typing judgement
$\pi = \mu$	multiplicity equality
$\pi \leqslant \mu$	sub-multiplicity judgement
$0 \leqslant \mu$	nullable multiplicity judgement
$U \leqslant V$	sub-usage-environment judgement

Macros

$U + V$	Usage environment sum
πV	Usage environment scaling

```
; B → j
; C → j
; D → p ∗ q
}
```

To this effect, the join variable j is not annotated with a multiplicity, instead, it is annotated with the usage of variables in its right-hand side (we call this annotation a *usage environment*). We then type check call sites of j as if we inlined j and replaced it with its right-hand side: the computed usage of call to j is its usage annotation. In this example, as p and q are both used linearly in j, we record $p \mapsto 1, q \mapsto 1$ (in the rule *let*). Then when j is used in the branches we use these multiplicities to check the linearity of p and q as necessary. This is the role of the extra variable typing rule *var.alias*.

Similar examples can be built with float-out, common-subexpression elimination, and inlining. At least.

Typing case alternatives In a constructor-pattern alternative $c\ x_1 ... x_n \to u$, the case binder z really is an alias for $c\ x_1 ... x_n \to u$, it is therefore type-checked as if it were $\mathsf{let}\ z = c\ x_1 ... x_n\ \mathsf{in}\ u$.

Note: as a consequence, if the constructor c has no field (*i.e.* $n = 0$), z is unrestricted.

3 Examples

3.1 Equations

The following Linear Haskell function:

```
data Colour = {Red; Green; Blue}
f :: Colour ⊸ Colour ⊸ Colour
f Red  q     = q
f p    Green = p
f Blue q     = q
```

is compiled in Core as

```
f = λ(p :: (' One) Colour) (q :: (' One) Colour) →
  case p of (p2 :: (' One) Colour)
  {Red → q
  ;_   →
    case q of (q2 :: (' One) Colour)
    {Green → p2
    ;_     →
      case p2 of (p3 :: (' One) Colour) {Blue → q2}
  }}
```

$$\frac{x \in \Gamma}{\Gamma \vdash x : A \rightsquigarrow \{x \mapsto 1\}}\text{var} \qquad \frac{\Delta \leqslant \Gamma}{\Gamma, x :_U A \vdash x : A \rightsquigarrow \{U\}}\text{var.alias}$$

$$\frac{\Gamma, x : A \vdash t : B \rightsquigarrow \{x \mapsto \mu, U\} \qquad \mu \leqslant \pi}{\Gamma \vdash \lambda(x :_\pi A).t : A \to_\pi B \rightsquigarrow \{U\}}\text{abs}$$

$$\frac{\Gamma \vdash t : A \to_\pi B \rightsquigarrow \{U\} \qquad \Gamma \vdash u : A \rightsquigarrow \{V\}}{\Gamma \vdash t\ u : B \rightsquigarrow \{U + \pi V\}}\text{app}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_i : A_i \rightsquigarrow \{U_i\} \\ c : A_1 \to_{\mu_1} ... \to_{\mu_{n-1}} A_n \to_{\mu_n} D\ p_1\ ...\ p_n \text{ constructor} \qquad \sigma = \pi_1/p_1, ..., \pi_n/p_n\end{array}}{\Gamma \vdash c\ t_1 ... t_n : D\ \pi_1\ ...\ \pi_n \rightsquigarrow \{\sum_i \mu_i[\sigma]U_i\}}\text{con}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : D\ \pi_1\ ...\ \pi_n \rightsquigarrow \{U\} \\ \Gamma; z; D\ p_1 ... p_n \vdash_\pi b_k : C \rightsquigarrow \{V_k\} \text{ for each } 1 \leqslant k \leqslant m \\ V_k \leqslant V \text{ for each } 1 \leqslant k \leqslant m\end{array}}{\pi\Gamma + \Delta \vdash \mathsf{case}\ t\ \mathsf{of}\ z :_\pi D\ \pi_1\ ...\ \pi_n\ \{b_k\}_{k=1}^m \rightsquigarrow \{\pi U + V\}}\text{case}$$

$$\frac{\begin{array}{c}\Gamma, x_1 :_{U_1} A_1 ... x_n :_{U_n} A_n \vdash t_i : A_i \rightsquigarrow \{U_i\} \\ \Gamma, x_1 :_{U_1} A_1 ... x_n :_{U_n} A_n \vdash u : C \rightsquigarrow \{V\}\end{array}}{\Gamma \vdash \mathsf{let}\ x_1 :_{U_1} A_1 = t_1\ ...\ x_n :_{U_n} A_n = t_n\ \mathsf{in}\ u : C \rightsquigarrow \{V\}}\text{letrec}$$

$$\frac{\Gamma \vdash u : A \rightsquigarrow \{U\} \qquad \Gamma, x :_U A \vdash t : B \rightsquigarrow \{V\}}{\Gamma \vdash \mathsf{let}\ x :_U A = u\ \mathsf{in}\ t : B \rightsquigarrow \{V\}}\text{let}$$

$$\frac{\Gamma \vdash t : A \rightsquigarrow \{U\} \qquad p \text{ fresh for } \Gamma}{\Gamma \vdash \lambda p.t : \forall p.A \rightsquigarrow \{U\}}\text{m.abs} \qquad \frac{\Gamma \vdash t : \forall p.A \rightsquigarrow \{U\}}{\Gamma \vdash t\ \pi : A[\pi/p] \rightsquigarrow \{U\}}\text{m.app}$$

$$\frac{\begin{array}{c}c : A_1 \to_{\mu_1} ... \to_{\mu_{r-1}} A_n \to_{\mu_n} D\ p_1\ ...\ p_r \text{ constructor} \\ \sigma = \pi_1/p_1, ..., \pi_n/p_n \qquad V = x_1 \mapsto \pi\mu_1[\sigma], ..., x_n \mapsto \mu_n[\sigma] \\ \Delta, z :_V D\ \pi_1 ... \pi_n, x_1 : A_i, ..., x_n : A_n \vdash u : C \rightsquigarrow \{x_1 \mapsto \rho_1, ..., x_n \mapsto \rho_n, U\} \\ \rho_1 \leqslant \pi\mu_1[\sigma] \quad ... \quad \rho_n \leqslant \pi\mu_n[\sigma]\end{array}}{\Gamma; z; D\ \pi_1 ... \pi_r \vdash^\sigma_\pi c\ x_1 ... x_n \to u : C \rightsquigarrow \{U\}}\text{alt.constructor}$$

$$\frac{\Delta, z : D\ \pi_1 ... \pi_n \vdash u : C \rightsquigarrow \{z \mapsto \mu, U\} \qquad \mu \leqslant \pi}{\Gamma; z; D\ \pi_1 ... \pi_n \vdash_\pi \_ \to u : C \rightsquigarrow \{U\}}\text{alt.wildcard}$$

Figure 2: Typing rules.

Multiplicity equality

$$\frac{}{\pi = \pi}\text{eq.refl} \qquad \frac{\pi = \rho}{\rho = \pi}\text{eq.sym} \qquad \frac{\pi = \rho \qquad \rho = \mu}{\pi = \mu}\text{eq.trans} \qquad \frac{}{1 + 1 = \omega}$$

$$\frac{}{1 + \omega = \omega} \qquad \frac{}{\omega + \omega = \omega} \qquad \frac{}{\pi + \rho = \rho + \pi} \qquad \frac{}{\pi + (\rho + \mu) = (\pi + \rho) + \mu}$$

$$\frac{}{\pi\rho = \rho\pi} \qquad \frac{}{\pi(\rho\mu) = (\pi\rho)\mu} \qquad \frac{}{1\pi = \pi} \qquad \frac{}{(\pi + \rho)\mu = \pi\mu + \rho\mu}$$

$$\frac{\pi = \pi' \qquad \rho = \rho'}{\pi + \rho = \pi' + \rho'}\text{eq.plus.compat} \qquad \frac{\pi = \pi' \qquad \rho = \rho'}{\pi\rho = \pi'\rho'}\text{eq.mult.compat}$$

Multiplicity ordering

$$\frac{}{\pi \leqslant \pi}\text{sub.sym} \qquad \frac{\pi \leqslant \rho \qquad \rho \leqslant \mu}{\pi \leqslant \mu}\text{sub.trans} \qquad \frac{}{1 \leqslant \omega} \qquad \frac{}{0 \leqslant \omega}$$

$$\frac{\pi \leqslant \pi' \qquad \rho \leqslant \rho'}{\pi + \rho \leqslant \pi' + \rho'}\text{sub.plus.compat} \qquad \frac{\pi \leqslant \pi' \qquad \rho \leqslant \rho'}{\pi\rho \leqslant \pi'\rho'}\text{sub.mult.compat}$$

$$\frac{\pi = \pi' \qquad \rho = \rho' \qquad \pi \leqslant \rho}{\pi' \leqslant \rho'}\text{sub.eq.compat}$$

Usage environment ordering

$$\frac{}{- \leqslant -}\text{sub.ctx.empty} \qquad \frac{U \leqslant V \qquad 0 \leqslant \pi}{U \leqslant V, x \mapsto \pi}\text{sub.ctx.zero}$$

$$\frac{U \leqslant V \qquad \pi \leqslant \rho}{\Gamma, x \mapsto \pi \leqslant \Delta, x \mapsto \rho}\text{sub.ctx.cons}$$

Usage environment operations

$$\begin{cases} U & + & (x \mapsto \pi, V) & = x \mapsto \pi, (U + V) & \text{if } x \notin U \\ (x \mapsto \pi, U) & + & V & = x \mapsto \pi, (U + V) & \text{if } x \notin V \\ (x \mapsto \pi, U) & + & (x \mapsto \mu, V) & = x \mapsto (\pi + \mu), (U + V) & \\ - & + & - & = - & \end{cases}$$

$$\begin{cases} \pi(x \mapsto \mu, U) & = x \mapsto \pi\mu, \pi U \\ \pi - & = - \end{cases}$$

Figure 3: Operations on multiplicity

3.2 Unrestricted fields

The following is well-typed:

```
data Foo where
  Foo :: A ⊸ B → C
f = λ(x :: (' One) Foo) →
  case x of (z :: (' One) Foo)
  {Foo a b → (z, b)}
```

3.3 Wildcard

The following is ill-typed

```
f = λ(x :: (' One) Foo) →
  case x of (z :: (' One) Foo)
  {_ → True}
```

3.4 Duplication

The following is ill-typed

```
data Foo = Foo A
f = λ(x :: (' One) Foo) →
  case (1) x of z
  {Foo a → (z, a)}
```