Linear Mini-Core

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

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

The goal of this note is to document the differences between $\lambda^q_\to$, as described in the Linear Haskell paper, and Core, the intermediate language of GHC.

We shall omit, for the time being, the minor differences, such as the absence of polymorphism on types ($\lambda^q_\to$ focuses on polymorphism of multiplicities), as we don't anticipate that they cause additional issues.

Maybe we should give a typing rule for (mutally) recursive lets. Despite it probably being messy.

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 with different multiplicities in different branches. And all these rules must also handle the case where π is chosen to be a variable p.

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 are fundamentally incompatible with standard linear logic rules. 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, 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.

consider breaking the let syntax in two (let and letrec) with a single entry in the let, and multiple in the letrec

2.2 Static semantics

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

Typing case alternatives The meaning of a case expression with multiplicity π is that consuming the resulting value of the case expression exactly once, will consume the scrutinee with multiplicity π (that is: exactly once if $\pi = 1$ and without any restriction if $\pi = \omega$). This is the π in $\vdash^{\sigma}_{\pi}$ in the alternative typing judgement.

To consume the scrutiny with multiplicity π, we must, by definition, consume every field x, whose multiplicity, as a field, is μ, with multiplicity $\pi\mu$.

This is where the story ends in $\lambda^{q}_{\to}$. But, in Linear Core, we can also use the case binder. Every time the case binder z (which stands for the scrutinee) is consumed once, we consume, implicitly, x with multiplicity μ. Therefore the multiplicity of x plus μ times the multiplicity of z must equal $\pi\mu$. Which is what $\rho + \nu\mu = \pi\mu$ stands for in the rule.

There is one such constraint per field. And, since μ can be parametric, a substitution σ is applied.

Note: if the constructor c has no field, then we're always good; the tag of the constructor is forced, and thus it does not matter how many times we use z.

We may try and make the argument in this note clearer, but I don't have an idea for the moment

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.

$$
\begin{array}{l}
\textbf{case}\ \mathsf{y}\ \textbf{of}\ \mathsf{y'} \\
\{\mathsf{A} \to \mathsf{p} - \mathsf{q} \\
;\mathsf{B} \to \mathsf{p} + \mathsf{q}
\end{array}
$$

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

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

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

$$\begin{array}{rll} b ::= & c\ x_1 ... x_n \to u & \text{data constructor} \\ \mid & \_ \to u & \text{wildcard} \end{array}$$

Terms

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

Figure 1: Syntax of $\lambda^{q}_{\to}$

```
; C → p + q
; D → p * q}
```

After the join point $\mathsf{p} + \mathsf{q}$ is identified, are p and q still used linearly? We want to answer affirmatively so that this transformation is still valid for linear bindings.

```
join j = p + q in
  case y of y'
  {A → p − q
  ; B → j
  ; C → j
  ; D → p * q}
```

Therefore, the join variable j is not given an explicit multiplicity. When we see an occurence of j we instead record the multiplicities of j's right-hand side. We then type check call sites of j as if we inlined j and replaced it with its right-hand side. In this example, as p and q are both used linearly in j, we record $p :_1 Int, q :_1 Int$ (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.

TODO: explain how the variable rule uses context ordering rather than sum. And why it's just a more general definition.

3 Examples

3.1 Equations

Take, as an example, 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
```

Explain: 0 is not a multiplicity in the formalism, so $0 \leqslant \pi$ must be understood formally, rather than a statement about multiplicities.

Add rules for 0 in equations for + and *.

Explain wildcard rule in English in Sec 2.2. And adapt example explanation.

This 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)
```

$$\frac{x :_1 A \leqslant \Gamma}{\Gamma \vdash x : A}\text{var} \qquad \frac{\Delta \leqslant \Gamma}{\Gamma, x :_\Delta A \vdash x : A}\text{var.alias} \qquad \frac{\Gamma, x :_\pi A \vdash t : B}{\Gamma \vdash \lambda(x :_\pi A).t : A \to_\pi B}\text{abs}$$

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

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

$$\frac{\begin{array}{c}\Gamma \vdash t : D\ \pi_1\ ...\ \pi_n \\ \sigma = \pi_1/p_1, ..., \pi_n/p_n \qquad \Delta; z; D\ p_1 ... p_n \vdash^\sigma_\pi b_k : C \text{ for each } 1 \leqslant k \leqslant m\end{array}}{\pi\Gamma + \Delta \vdash \mathsf{case}\ t\ \mathsf{of}\ z :_\pi D\ \pi_1\ ...\ \pi_n\ \{b_k\}^m_{k=1} : C}\text{case}$$

$$\frac{\Gamma_i, x_1 :_\omega A_1 ... x_n :_\omega A_n \vdash t_i : A_i \qquad \Delta, x_1 :_\omega A_1 ... x_n :_\omega A_n \vdash u : C}{\Delta + \omega \sum_i \Gamma_i \vdash \mathsf{let}\ x_1 : A_1 = t_1\ ...\ x_n : A_n = t_n\ \mathsf{in}\ u : C}\text{letrec}$$

$$\frac{\Delta \vdash u : A \qquad \Gamma, x :_\Delta A \vdash t : B}{\Gamma \vdash \mathsf{let}\ x : A = u\ \mathsf{in}\ t : B}\text{let} \qquad \frac{\Gamma \vdash t : A \qquad p \text{ fresh for } \Gamma}{\Gamma \vdash \lambda p.t : \forall p.A}\text{m.abs}$$

$$\frac{\Gamma \vdash t : \forall p.A}{\Gamma \vdash t\ \pi : A[\pi/p]}\text{m.app}$$

$$\frac{\begin{array}{l}c : A_1 \to_{\mu_1} ... \to_{\mu_{r-1}} A_n \to_{\mu_n} D\ p_1\ ...\ p_r \text{ constructor} \\ \Delta, z :_\nu (D\ p_1 ... p_n)[\sigma], x_1 :_{\rho_1} A_i, ..., x_n :_{\rho_n} A_n \vdash u : C \\ \rho_1 + \nu\mu_1[\sigma] = \pi\mu_1[\sigma] \quad ... \quad \rho_n + \nu\mu_n[\sigma] = \pi\mu_n[\sigma]\end{array}}{\Gamma; z; D\ p_1 ... p_r \vdash^\sigma_\pi c\ x_1 ... x_n \to u : C}\text{alt.constructor}$$

$$\frac{\Delta, z :_\pi (D\ p_1 ... p_n)[\sigma] \vdash u : C}{\Gamma; z; D\ p_1 ... p_n \vdash^\sigma_\pi \_ \to u : C}\text{alt.wildcard}$$

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

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

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

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

Context ordering

$$\frac{}{- \leqslant -}\text{sub.ctx.empty} \qquad \frac{\Gamma \leqslant \Delta \qquad 0 \leqslant \pi}{\Gamma \leqslant \Delta, x :_\pi A}\text{sub.ctx.zero}$$

$$\frac{\Gamma \leqslant \Delta \qquad \pi \leqslant \rho}{\Gamma, x :_\pi A \leqslant \Delta, x :_\rho A}\text{sub.ctx.cons}$$

Figure 3: Equality and ordering rules

```
    { Green → p2
    ; _     →
        case p2 of (p3 :: (' One) Colour) { Blue → q2 }
}}
```

This is well typed because (focusing on the case of `p2`)

- In the `Red` branch, no variables are introduced by the constructor.
- In the `WILDCARD` branch, we see `WILDCARD` as a variable which can't be referenced, from the rules we get that the multiplicity of `WILDCARD` (necessarily 0) plus the multiplicity of `p2` must be 1. Which is the case as $p2$ is used linearly in each branch.

This example illustrates that, even in a multi-argument equation setting, the compiled code is linear when all the equations, individually, are linear.

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) }
```

It is well typed because

- a is a linear field, hence imposes that the multiplicity of a (here 0) and the multiplicity of the case binder z (here 1) sum to 1, which holds
- b is an unrestricted field, hence imposes that the multiplicity of b (1) plus ω times the multiplicity of z (1) equals ω (times 1 since this is a linear case). That is $1 + \omega 1 = \omega$ which holds.

3.3 Wildcard

The following is ill-typed

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

Because the multiplicity of `WILDCARD` (necessarily 0) plus the multiplicity of the case binder z (0) does not equal 1.

This follows intuition as x really isn't being consumed (x is forced to head normal form, but if it has subfield they will never get normalised, hence this program is rightly rejected).

This also follows our intended semantics, as f amounts to duplicating a value of an arbitrary type, which is not possible in general.

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)}
```

Because both z and a are used in the branch, hence their multiplicities sum to ω, but it should be 1.

4 Typechecking linear Mini-Core

It may appear that typechecking the case rule requires guessing multiplicities ν and ρ_i so that they verify the appropriate constraint given from the context. But it is in fact not the case as the multiplicity will be an output of the type-checker.

In this section we shall sketch how type-checking can be performed on Linear Core.

4.1 Representation

Core, in GHC, attaches its type to every variable x (let's call it $\mathsf{type}(x)$). Similarly, in Linear Core, variables come with a multiplicity ($\mathsf{mult}(x)$).

- $\lambda x :_\pi A.u$ is represented as $\lambda x.u$ such that $\mathsf{type}(x) = A$ and $\mathsf{mult}(x) = \pi$
- $\mathsf{case}\ u\ \mathsf{of}\ z ::_\pi A\ \{...\}_{k=1}^m$ is represented as $\mathsf{case}\ u\ \mathsf{of}\ z\ \{...\}_{k=1}^m$ such that $\mathsf{mult}(z) = \pi$

Contrary to $\mathsf{type}(x)$, which is used both at binding and call sites, $\mathsf{mult}(x)$ will only be used at binding site.

4.2 Terminology & notations

A mapping is a finite-support partial function.

Explain sum, scaling, and join for mapping.

- We write $k \mapsto v$ for the mapping defined only on k, with value v.
- For two mapping m_1 and m_2 *with disjoint supports*, we write m_1, m_2 for the mapping defined the obvious way on the union of their supports.

4.3 Algorithm sketch

The typechecking algorithm, $\mathsf{lint}(t)$, takes as an input a Linear Core term t, and returns a pair of

- The type of the term

- A mapping of every variable to its number of usages (ρ). Later on we check that usages are compatible with the declared multiplicity π. ($\rho \leqslant \pi$)

We assume that the variables are properly α-renamed, so that there is no variable shadowing.

The algorithm is as follows (main cases only):

Explain multiplicity ordering. Explain how zero-usage is handled. Explain how empty cases are handled.

- $\mathsf{lint}(x) = (\mathsf{type}(x), x \mapsto 1)$
- $\mathsf{lint}(u\ v) = (B, m_u + \pi m_v)$ where $(A \to_\pi B, m_u) = \mathsf{lint}(u)$ and $(A, m_v) = \mathsf{lint}(v)$
- $\mathsf{lint}(\lambda_\pi x : A.u) = (A \to_\pi B, m)$ where $(B, (x \mapsto \rho, m)) = \mathsf{lint}(u)$ and $\rho \leqslant \pi$.
- $\mathsf{lint}(\mathsf{case}_\pi\ u\ \mathsf{of}\ z\ \{c_k\ x_1 ... x_{n_k} \to v_k\}_{k=1}^m) = (A, \pi m_u + \bigvee_{k=1}^m m_k)$, where the $c_k : B_1^k \to_{\mu_1^k} ... \to_{\mu_{n_k}^k} B_{n_k} \to D$ are constructors of the data type D, $(m_u, D) = \mathsf{lint}(u)$, $(A, (z \mapsto \nu^k, x_1 \mapsto \rho_1^k, ..., \rho_{n_k}^k, m_k)) = \mathsf{lint}(v_k)$ and $\rho_i^k + \nu^k \mu_i^k \leqslant \pi \mu_i^k$ for all i and k.

Expand using branch type checking. Also explain that we need to check the multiplicity of x_i.