Type Theory as a Language Workbench

Jan de Muijnck-Hughes
University of Glasgow, UK

Guillaume Allais
University of St Andrews, UK

Edwin Brady
University of St Andrews, UK

Abstract

Language Workbenches offer language designers an expressive environment in which to create their Domain Specific Languages (DSLs). Similarly, research into mechanised meta-theory has shown how dependently typed languages provide expressive environments to formalise and study DSLs and their meta-theoretical properties. But can we claim that dependently typed languages qualify as language workbenches? We argue yes!

We have developed an exemplar DSL called Vélo that showcases not only dependently typed techniques to realise and manipulate Intermediate Representations (IRs), but that dependently typed languages make fine language workbenches. Vélo is a simple verified language with well-typed holes and comes with a complete compiler pipeline: parser, elaborator, REPL, evaluator, and compiler passes. Specifically, we describe our design choices for well-typed IR design that includes support for well-typed holes, how Common Sub-Expression Elimination (CSE) is achieved in a well-typed setting, and how the mechanised type-soundness proof for Vélo is the source of the evaluator.

2012 ACM Subject Classification Theory of computation → Type theory; Software and its engineering → Software verification; Software and its engineering → Functional languages; Software and its engineering → Formal language definitions; Software and its engineering → Domain specific languages; Theory of computation → Lambda calculus; Software and its engineering → Compilers; Theory of computation → Invariants

Keywords and phrases dependent types, language workbenches, idris2, dsl, edsl, intrinsically scoped, well typed, co-De Bruijn

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.9

Supplementary Material We have made Vélo's source available both as a reproducible artifact and freely available source code:
Software (Source Code): `https://github.com/jfdm/velo-lang`
Software (Artifact): `https://doi.org/10.5281/zenodo.7573031`

Funding Jan de Muijnck-Hughes is funded by EPSRC grants: Border Patrol (EP/N028201/1) and AppControl (EP/V000462/1). Guillaume Allais and Edwin Brady are funded by EPSRC grant: Programming as Conversation: Type-Driven Development in Action (EP/T007265/1).

1 Introduction

Language Workbenches, such as Spoofax [28], offer language designers an expressive environment in which to design, implement, and deploy their Domain Specific Languages (DSLs) [16]. Principally speaking a language workbench [15] is a tool that supports: description of a language's *notation* – how we present a language's concrete syntax to users; implementation of a language's *semantics* – how we realise the language's behaviour; and user interaction through an *editor*. Outside of these core criteria, various language workbenches support language validation, testing, and composition.

Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 9; pp. 9:1–9:13
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Concurrently, the mechanised meta-theory research programme [7, 1] has seen a wealth of tools and techniques being developed by the programming languages theory community. In particular, dependently typed languages such as Idris 2 [9], Agda [21], and Coq [26] have been widely used to formalise DSLs, and study their meta-theoretical properties. Dependent types allow types to depend on values – that is, types are first class – and provide an expressive environment in which to reason about, and write, our programs. Efforts using dependently typed languages range from studying specific core calculi [4, 24, 10] to building generic reasoning frameworks [25, 3]. These mechanised software verification projects, however, typically stop short of building the frontend that would let users run these verified language implementations. If our verified language implementations type check, we might as well ship them too! By becoming its own implementation language, Idris 2 has successfully demonstrated that this is not an inescapable fate [9]. But can we now claim that dependently typed languages qualify as language workbenches?

Vélo[1] is a minimal functional language that we have realised in Idris 2 to showcase dependently typed techniques to implement and manipulate Intermediate Representations (IRs). This paper introduces Vélo but, most of all, seeks to show that dependently typed languages make fine language workbenches. We address both the core criteria and some optional extensions highlighted by the language workbench challenge [15] for what constitutes a language workbench. Although not all of the optional criteria are met by dependently typed languages, we are convinced that with some additional engineering (taking advantage of existing work, for example Quickchick [18]) more optional criteria can be satisfied.

Another key tenet in language workbenches, such as Spoofax, is the *ease* with which languages can be created. To that same degree, we have developed a series of reusable modules that captures functionality common to many languages, thereby reducing the *boilerplate* required when creating Embedded Domain Specific Languages (EDSLs) in Idris 2.

Although we have made an effort to make dependently typed programming accessible in our presentation, more introductory material is available for the interested reader [29, 8].

2 Introducing Velo

The design behind Vélo is purposefully unsurprising: it is the Simply-Typed Lambda Calculus (STLC) extended with let-bindings, booleans and their conjunction, and natural numbers and their addition. To promote the idea of interactive editing Vélo also supports well-typed holes. Below we show an example Vélo program, which contains a multiply used hole, and an extract from the REPL session that lists the current set of holes.

```
   let b = false
in let double
         = (fun x : nat => (add x x))
in let x = (double ?hole)
in         (double ?hole)
```

```
Velo> :holes
b : Bool
double : Nat -> Nat
----------
?hole : Nat
```

The featherweight language design of Vélo helps us showcase better how we can use dependently typed languages as language workbenches [17]. Regardless of language complexity, Vélo is nonetheless a complete language with a standard compiler pipeline, and REPL. A DSL captures the language's concrete syntax, and a parser turns DSL instances into raw unchecked terms. Bidirectional type checking keeps type annotations to a minimum in the

[1] A reproducible artifact, and the source code, has been provided as supplementary material.

concrete syntax, and helps to better elaborate raw un-typed terms into a set of well-typed IRs: `Holey` to support well-scoped typed holes; and `Terms` the core representation that captures our language's abstract syntax. We present interesting aspects of our IR design in Section 3. Further, elaboration performs standard desugarings that e.g. turns let-bindings into function application thus reducing the size of our core. From the core representation we also provide well-scoped Common Sub-Expression Elimination (CSE) using co-De Bruijn indexing (Section 4), and we provide a verified evaluator to reduce terms to values (Section 5).

3 Language Design

We begin our discussion by detailing the key design rationale on realising the static semantics of Vélo within Idris 2. We have opted to give Vélo an external concrete syntax (a DSL) in which users can write their programs. With dependently typed languages we can also capture the abstract syntax and its static semantics as an intrinsically scoped and typed EDSL directly within the host language [6]. That is to say that the data structure is designed in such a way that we can only represent well scoped and well typed terms and, correspondingly, that our scope- and type- checking passes are guaranteed to have rejected invalid user inputs. To keep the exposition concise, we focus on a core subset of the language. The interested reader can find the whole definition in the accompanying material.

Types are usually introduced using their context free grammar. We present it here on the left-hand side, it gives users the choice between two base types (Nat, and Bool) and a type former for function types ($\cdot \to \cdot$). On the right hand side, we give their internal representation as an inductive type in Idris 2.[2]

$$
\begin{array}{rcl}
t : \mathrm{TYPE} & ::= & \textsc{Nat} \\
 & | & \textsc{Bool} \\
 & | & t \to t
\end{array}
$$

```
data Ty = TyNat
        | TyBool
        | TyArr Ty Ty
```

Contexts can be similarly given by a context free grammar: a context is either empty (ϵ), or an existing context (Γ) extended on the right with a new type assignment ($x : t$) using a comma. In Idris 2, we will adopt a nameless representation and so we represent these contexts by using a `SnocList` of types (i.e. lists that grow on the right). Note that the Idris 2 compiler automatically supports sugar for lists and snoc lists: `[1,2,3]` represents a list counting up from `1` to `3` while `[<1,2,3]` is its snoc list pendant counting down. In particular `[<]` denotes the empty snoc list also known as `Lin`.

$$
\begin{array}{rcl}
\Gamma : \mathrm{CONTEXT} & ::= & \epsilon \\
 & | & \Gamma, x : t
\end{array}
$$

```
data SnocList a = Lin
                | (:<) (SnocList a) a
```

Typing Judgements are given by relations, and encoded in Idris 2 using inductive families, a generalisation of inductive types [14]. Each rule will become a constructor for the family, and so every proof $\Gamma \vdash t : a$ will correspond to a term t of type (`Term` Γ a). On the left hand side we present two judgements: context membership and a typing judgement, and on the right we have the corresponding inductive family declarations.

[2] Throughout this article, the Idris 2 code snippets are automatically rendered using a semantic highlighter. Keywords are typeset in **bold**, types in blue, data constructors in red, function definitions in green, bound variables in purple, and comments in *grey*.

$\Gamma \ni x : a$	`data Elem : (gamma : SnocList ty) -> (a : ty) -> Type`
$\Gamma \vdash t : a$	`data Term : (gamma : SnocList Ty) -> (a : Ty) -> Type`

We leave the definition of `Elem` to the next section, focusing instead on `Term`. The most basic of typing rules are axioms, they have no premise and are mapped to constructors with no argument. We use Idris 2 comments (`--`) to format our constructor's type in such a way that they resemble the corresponding inference rule. Here we show the rule stating that 0 is a natural number and its translation as the `Zero` constructor.

$$\frac{}{\Gamma \vdash \mathsf{Zero} : \textsc{Nat}}$$

```
Zero : ------------------
         Term gamma TyNat
```

Then come typing rules with a single premise which is not a subderivation of the relation itself. They are mapped to constructors with a single argument. Here we show the typing rule for variables: given a proof that we have a variable of type a somewhere in the context, we can build a term of type a in said context.

$$\frac{\Gamma \ni x : a}{\Gamma \vdash x : a}$$

```
Var : Elem gamma a ->
      --------------
      Term gamma a
```

Next, we have typing rules with a single premise which is a subderivation. They are mapped to constructors with a single argument of the inductive family representing the subderivation. Here we show the typing rule for successor: provided that we are given a natural number in a given context, its successor is also a natural number in the same context.

$$\frac{\Gamma \vdash n : \textsc{Nat}}{\Gamma \vdash (\mathsf{Inc}\, n) : \textsc{Nat}}$$

```
Inc : Term gamma TyNat ->
      ------------------
      Term gamma TyNat
```

Similarly, rules with two premises are translated to constructors with two arguments, one for each subderivation. Here we present the typing rule for application nodes: provided that the function has a function type, and the argument has a type matching the function's domain, the application has a type corresponding to the function's codomain. Note that the context Γ is the same across the whole rule and so the same `gamma` is used everywhere.

$$\frac{\Gamma \vdash f : a \to b \qquad \Gamma \vdash t : a}{\Gamma \vdash f \,\$\, t : b}$$

```
App : Term gamma (TyArr a b) ->
      Term gamma a ->
      ------------------------
      Term gamma b
```

Finally, we have a rule where the premise's context has been extended: a function of type $(a \to b)$ is built by providing a term of type b defined in a context extended with a new variable of type a.

$$\frac{\Gamma, x : a \vdash t : b}{\Gamma \vdash (\lambda(x) \cdot t) : a \to b}$$

```
Func : Term (gamma :< a) b ->
       ------------------------
       Term gamma (TyArr a b)
```

Using this intrinsically typed representation, we can readily represent entire typing derivations. The following example[3] presents the internal representation `Plus2` of the derivation proving that $(\lambda(x) \cdot (\mathsf{Inc}\,(\mathsf{Inc}\,x)))$ can be assigned the type $(\textsc{Nat} \rightarrow \textsc{Nat})$.

$$\dfrac{\dfrac{\vdots}{\epsilon\,, x : \textsc{Nat} \vdash (\mathsf{Inc}\,(\mathsf{Inc}\,x)) : \textsc{Nat}}}{\epsilon \vdash (\lambda(x) \cdot (\mathsf{Inc}\,(\mathsf{Inc}\,x))) : \textsc{Nat} \rightarrow \textsc{Nat}}$$

```
Plus2 : Term [<] (TyArr TyNat TyNat)
Plus2 = Func (Inc (Inc (Var Here)))
```

By using `Term` as an IR in our compiler we have made entire classes of invalid programs unrepresentable: it is impossible to form an ill scoped or ill typed term. Indeed, trying to write an ill scoped or an ill typed program leads to a static error as demonstrated by the following **failing** blocks.[4] In this first example we try to refer to a variable in an empty context. Idris 2 correctly complains that this is not possible.

```
failing "Mismatch between: ?gamma :< TyNat and [<]."

  Ouch : Term [<] TyNat
  Ouch = Var Here
```

In this second example we try to type the identity function as a function from Nat to Bool. This is statically rejected as nonsensical: `TyNat` and `TyBool` are distinct constructors!

```
failing "Mismatch between: TyBool and TyNat."

  Ouch : Term [<] (TyArr TyNat TyBool)
  Ouch = Func (Var Here)
```

Using such intrinsically typed EDSLs we can statically enforce that our elaborators do check that the raw terms obtained by parsing user input are well scoped and well typed. Writing our compiler passes (model-to-model transformations) and evaluation engine (model-to-host transformation) using these invariant-rich IRs additionally ensures that each step respects the language's static semantics. In fact we will describe in Section 5 how we can use our EDSLs to both verify our static semantics whilst describing our dynamic semantics.

For languages equipped with more advanced type systems, that cannot be as easily enforced statically, we can retain some of these guarantees by using a well scoped core language rather than a well typed one. This is the approach used in Idris 2 and it has already helped eliminate an entire class of bugs arising when attempting to solve a metavariable with a term that was defined in a different context [9].

3.1 Efficient De Bruijn Representation

A common strategy for implementing well-scoped terms is to use typed *De Bruijn* indices [13], which are easily realised as an inductive family [14] indicating where in the type-level context the variable is bound.

Concretely, we index the `Elem` family by a context (once again represented as a `SnocList` of kinds) and the kind of the variable it represents.

$\Gamma \ni x : a$

```
data Elem : (gamma : SnocList ty) -> (a : ty) -> Type
```

[3] `Here` will be defined in Section 3.1 as a constructor for the `Elem` family.
[4] Idris 2 only accepts failing blocks if checking their content yields an error matching the given string.

We then match each context membership inference rule to a constructor. The `Here` constructor indicates that the variable of interest is the most local one in scope (note the non-linear occurrence of $(x : a)$ on the left hand side, and correspondingly of `ty` on the right).

$$\frac{}{\Gamma, x : a \ni x : a}$$

```
Here : ----------------------
       Elem (gamma :< ty) ty
```

The `There` constructor skips past the most local variable to look for the variable of interest deeper in the context.

$$\frac{\Gamma \ni x : a}{\Gamma, y : b \ni x : a}$$

```
There :  Elem gamma ty ->
        ---------------------
         Elem (gamma :< _) ty
```

Whilst a valid definition, this approach unfortunately does not scale to large contexts: every `Elem` proof is linear in the size of the De Bruijn index that it represents. To improve the runtime efficiency of the representation we instead opt to model De Bruijn indices as natural numbers, which Idris 2 compiles to GMP-style unbounded integers. Further, we need to additionally define an `AtIndex` family to ensure that all of the natural numbers we use correspond to valid indices. We pointedly reuse the `Elem` names because these `Here` and `There` constructors play exactly the same role.

```
data AtIndex : (ty : kind) -> (ctxt : SnocList kind) ->
               (idx : Nat) -> Type where
  Here : AtIndex ty (ctxt :< ty) 0
  There : AtIndex ty ctxt idx -> AtIndex ty (ctxt :< _) (1 + idx)
```

We then define a variable as the pairing of a natural number and an *erased* (as indicated by the `0` annotation on the binding site for `prf`) proof that the given natural number is indeed a valid De Bruijn index.

```
data IsVar : (ctxt : SnocList kind) -> (ty : kind) -> Type where
    V : (idx : Nat) -> (0 prf : AtIndex ty ctxt idx) -> IsVar ctxt ty
```

Thanks to Quantitative Type Theory [19, 5] as implemented in Idris 2, the compiler knows that it can safely erase these runtime-irrelevant proofs. we now have the best of both worlds: a well-scoped realisation of De Bruijn indices that is compiled efficiently.

Just like the naïve definition of De Bruijn indexing is not the best suited for a practical implementation, the inductive family `Term` described in Section 3 is not the most convenient to use. We will now see one of its limitations and how we remedied it in Vélo.

3.2 Compact Constant Folding

Software Foundations' *Programming Language Foundations* opens with a constant-folding transformation exercise [23, Chapter 1]. Starting from a small language of expressions (containing natural numbers, variables, addition, subtraction, and multiplication) we are to deploy the semiring properties to simplify expressions. The definition of the simplifying traversal contains much duplicated code due to the way the source language is structured: all the binary operations are separate constructors, whose subterms need to be structurally simplified before we can decide whether a rule applies. The correction proof has just as much duplication because it needs to follow the structure of the call graph of the function it wants to see reduced. The only saving grace here is that Coq's tactics language lets users write scripts that apply to many similar goals thus avoiding duplication in the source file.

In Vélo, we structure our core language's representation in an algebraic manner so that this duplication is never needed. All builtin operators (from primitive operations on builtin types to function application itself) are represented using a single `Call` constructor which takes an operation and a type-indexed list of subterms.

```
data Term : (ctxt : SnocList Ty) -> Ty -> Type where
  Var : IsVar ctxt ty -> Term ctxt ty
  Fun : Term (ctxt :< a) b -> Term ctxt (TyArr a b)
  Call : {tys : _} -> (operator : Prim       tys  ty)
                   -> (operands : Terms ctxt tys)
                   -> Term                 ctxt      ty
```

Here `Terms` is the pointwise lifting of `Term` to lists of types. In practice we use the generic `All` quantifier, but this is morally equivalent to the specialised version presented below:

```
data Terms : (ctxt : SnocList Ty) -> List Ty -> Type where
  Nil : Terms ctxt Nil
  (::) : Term ctxt ty -> Terms ctxt tys -> Terms ctxt (ty :: tys)
```

The primitive operations can now be enumerated in a single datatype `Prim` which lists the primitive operation's arguments and the associated return type.

```
data Prim : (args : List Ty) -> (ret : Ty) -> Type where
    Zero : Prim []                     TyNat
    Inc  : Prim [TyNat]                TyNat
    App  : Prim [TyArr dom cod, dom] cod
```

Using `Prim`, structural operations can now be implemented by handling recursive calls on the subterms of `Call` nodes uniformly before dispatching on the operator to see whether additional simplifications can be deployed. Similarly, all of the duplication in the correction proofs is factored out in a single place where the induction hypotheses can be invoked.

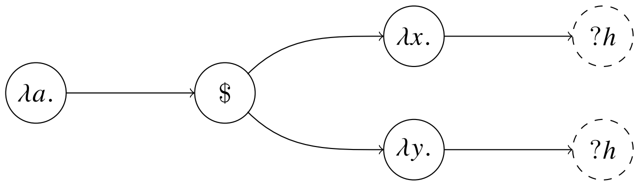
3.3 Well-Typed Holes

Holes are a special kind of placeholder that programmers can use for parts of the program they have not yet written. In a typed language, each hole will be assigned a type based on the context it is used in.

Type-Driven Development [22] is a practice by which the user enters into a dialogue *with* the compiler to interactively build the program. We can enable type-driven programming in part by providing special language support for holes and operations on them. Such operations will include the ability to inspect, refine, compute with, and instantiate (with an adequately typed term) holes. We believe that bare-bones language support for type-driven development should at least include the ability to: (1) inspect the type of a hole and the local context it appears in; (2) instantiate a hole with an adequately typed term; and as well (3) safely evaluate programs that still contain holes. Vélo provides all three.

Idris 2 elaborates holes as it encounters them by turning them into global declarations with no associated definition. Because of this design choice users cannot mention the same hole explicitly in different places to state their intention that these yet unwritten terms ought to be the same. Users can refer to the hole's solution by its name, but that hole is placed in one specific position and it is from that position that Idris 2 infers its context.

In Vélo, however, we allow holes to be mentioned arbitrarily many times in arbitrarily different local contexts. In the following example, the hole `?h` occurs in two distinct contexts: ϵ, a, x and ϵ, a, y.



As a consequence, a term will only fit in that hole if it happens to live in the shared common prefix of these two contexts (ϵ, a). Indeed, references to x will not make sense in ϵ, a, y and vice-versa for y.

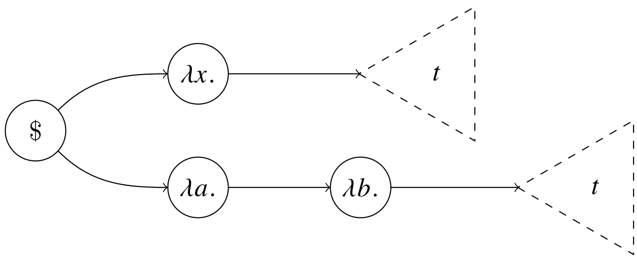
Our elaborator proceeds in two steps. First, a bottom-up pass records holes as they are found and, in nodes with multiple subterms, reconciles conflicting hole occurrences by computing the appropriate local context restrictions. This process produces a list of holes, their types, and local contexts, together with a `Holey` term that contains invariants ensuring these collected holes do fit in the term. Second, a top-down pass produces a core `Term` indexed by the list of `Meta` (a simple record type containing the hole's name, the context it lives in, and its type). Hole occurrences end up being assigned a thinning that embeds the metavariable's actual context into the context it appears in. We discuss thinnings and their use in Vélo in Section 4.

Although these intermediate representations are Vélo-specific, the technique and invariants are general and can be reused by anyone wanting to implement well-scoped holes in their functional DSL.

4 Compiler Passes

Now that our core language is well-scoped by construction, our compiler passes must also be shown to be scope-preserving. This is not a new requirement, merely it makes concrete a constraint that used to be enforced informally. More importantly we show, with our compiler passes, that model-to-same-model transformation of our EDSL is possible, and that the infrastructure required is not bespoke to Vélo.

The purpose of CSE is to identify subterms that appear multiple times in the syntax tree, and to abstract over them to avoid needless recomputations at runtime. In the following example for instance, we would like to let-bind t before the application node (denoted \$) so that t may be shared by both subtrees.



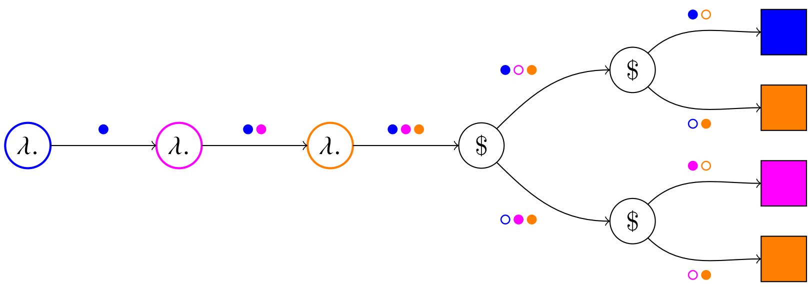
One of the challenges of CSE, as exemplified above, is that the term of interest may be buried deep inside separate contexts. In our intrinsically scoped representation, t in scope $\Gamma,\ x:\sigma$ is potentially not actually syntactically equal to a copy living in $\Gamma,\ a:\tau,\ b:\nu$. Indeed a variable v bound in Γ will, for instance, be represented by the De Bruijn index $(1+v)$ in $\Gamma,\ x:\sigma$ but by the index $(2+v)$ in $\Gamma,\ a:\tau,\ b:\nu$.

If only terms were indexed by their exact *support* (i.e. a context restricted to the variables actually used in the term)! We would not care about additional yet irrelevant variables that happen to be in scope. The principled solution here is to switch to a different representation when performing CSE. The co-De Bruijn representation [20] provides exactly this guarantee.

In the co-De Bruijn representation, every term is precisely indexed by its exact support. That is to say that every subterm explicitly throws away the bound variables that are not mentioned in it. By the time we reach a variable node, a single bound variable remains in scope: precisely the one being referred to.

This process of throwing unused variables away is reified using thinnings i.e. renamings that are injective, and order preserving. We can think of thinnings as sequences of 0/1 bits, stating whether each variable is kept or dropped.

Below, we give a graphical presentation (taken from [2]) of the S combinator (the lambda term $\lambda g.\lambda f.\lambda x.gx(fx)$) in co-De Bruijn notation. In it we represent thinnings (i.e. lists of bits) as lists of either $\bullet$ (1) or $\circ$ (0).



The first three λ abstractions only use $\bullet$ in their thinnings because all of g, f, and x do appear in the body of the combinator. The first application node then splits the context into two: the first subterm (gx) drops f while the second (fx) gets rid of g. Further application nodes select the one variable still in scope for each leaf subterm: g, x, f, and x.

Using a co-De Bruijn representation, we can identify shared subterms: they need to not be mentioning any of the most local variables and be syntactically equal. Our pass successfully transforms the program on the left-hand side to the one on the right-hand side where the repeated expressions `(add m n)` and `(add n m)` have been let-bound.

```
let m = zero in
let n = (inc zero)
in (add (add (add m n) (add n m))
        (add (add n m) (add m n)))
```

```
let m = zero       in
let n = (inc zero) in
let p = (add n m)  in
let q = (add m n)
in (add (add q p) (add p q))
```

This pass relies on the ability to have a compact representation of thinnings (as the co-De Bruijn representation makes heavy use of them), and additionally the existence of a cheap equality test for them. This is not the case in the implementation we include in Vélo but it is a solved problem [2].

5 Execution

The Vélo REPL lets users reduce terms down to head-normal forms. We can realise Vélo's dynamic semantics either through definitional interpreters [4, 6], or by providing a more traditional syntactic proof of type-soundness [30] but mechanised [29, Part 2: Properties] using inductive families.

We chose the latter approach: by using inductive families, we can make explicit our language's operational semantics. This enables us to study its meta-theoretical properties and in particular prove a progress result: every term is either a value or can take a reduction step. By repeatedly applying the progress result, until we either reach a value or the end user runs out of patience and kills the process, this proof freely gives us an evaluator that is guaranteed to be correct with respect to Vélo's operational semantics.

Following existing approaches [29, Part 2: Properties], we have defined inductive families describing how terms reduce.

```
data Redux : (this,that : Term ctxt type) -> Type where
  SimplifyCall : (op   : Prim tys ty)
              -> (step : Reduxes these those)
                      -> Redux (Call p these) (Call p those)

  ReduceFuncApp : {body  : Term (ctxt :< type) return}
               -> {arg   : Term ctxt type}
               -> (value : Value arg)
                       -> Redux (Call App [Fun body, arg])
                                (subst arg body)
```

As can be seen above, our setting enforces call-by-value: as described by the rule `ReduceFuncApp` $((\lambda(x) \cdot b)\,\$\,t)$ only reduces to $(b\,\{x \leftarrow t\})$ if t is already known to be a value. Furthermore, our algebraic design (Section 3.2) allows us to easily enforce a left-to-right evaluation order by having a generic family describing how primitive operations' arguments reduce. As can be seen below: when considering a type-aligned list of arguments, either the `hd` takes a step and the `rest` is unchanged, or the `hd` is already known to be a value and a further argument is therefore allowed to take a step.

```
data Reduxes : (these, those : Terms ctxt tys) -> Type where
  (!:) : (hd   : Redux this that)
      -> (rest : Terms ctxt tys)
              -> Reduxes (this :: rest) (that :: rest)

  (::) : (value : Value hd)
      -> (tl    : Reduxes these those)
               -> Reduxes (hd :: these) (hd :: those)
```

We differ, however, from standard approaches by making our proofs of progress generic such that the boilerplate for computing the reflexive transitive closure when reducing terms is tidied away in a shareable module. Our top-level progress definition is thus parameterised by reduction and value definitions:

```
data Progress : (0 value : Pred a) -> (0 redux : Rel a) -> (tm : a) -> Type
  where Done : {0 tm : a} -> (val : value tm) -> Progress value redux tm

        Step : {this, that : a}
            -> (step : redux this that) -> Progress value redux this
```

and the result of execution, which is similarly parameterised, is as follows (where `RTList` is the type taking a relation and returning its reflexive-transitive closure):

```
data Result : (0 value : Pred a) -> (0 redux : Rel a) -> (this : a) -> Type
  where R : (that : a) -> (val : value that)
         -> (steps : RTList redux this that) -> Result value redux this
```

The benefit of our approach is that language designers need only provide details of what reductions are, and how to compute a single reduction, the rest comes for free. Moreover, with the result of evaluation we also get the list of reduction steps made that can, optionally, be printed to show a trace of execution.

6 Conclusion

We have shown that dependently typed languages satisfy the core requirements from the *Language Workbench Challenge* [15]. Vélo's *notation* as a DSL is, by design, textual, and the internal core bounded by Idris 2's own notation requirements. More importantly the *semantics* (statics and dynamics) of Vélo are verified as part of the implementation thanks to the dependently typed setting. The weakest supported core criteria, unfortunately, is that for *editor* support. Languages created through Idris 2 do not get an editor, they are free form languages which require their parsers and elaborators be hand written. This can change with future investigation. Idris 2 has support for elaborator reflection [11] which provides a vehicle through which deriving parsers and elaborators can happen.

There are, however, more criteria from the language workbench feature model to consider: semantic & syntactic services for editors; testing & debugging; and composability.

With the rise of the Language Server Protocol (LSP) it would be a good idea to look at how we can derive LSP compatible language servers generically, thus addressing the missing provision of the optional semantic and syntactic services. Idris 2 itself provides an *IDE-Protocol*, and there is support for the LSP in Idris 2.

Our languages also do not come with the ability to test and debug their implementation. Some of the features we have presented are fully formalised (e.g. execution), others are only known to be scope-and-type preserving (e.g. CSE). Therefore the dependently typed setting does not mean we do not need testing anymore. Prior work on generators for inductive families [18] should allow us to bring property-based testing [12] to our core passes.

Finally there is language composability. It would be advantageous to support the reuse of existing languages, and their type-systems when designing new ones. This is a hard problem: One has to not only combine their semantics but also the remainder of the workbench tooling. The *language fragments* approach [27] provides a solution to language composability for intrinsically typed definitional interpreters, but this does not extend to workbench tooling. Extending this approach to our definition of semantics based on inductive families and to creating composable workbench tooling is an open problem.

We strongly believe that with future engineering we can satisfy these missing criteria, and make dependently typed languages a mighty fine language workbench.

References

1 Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.*, 29:e19, 2019. doi:10.1017/S0956796819000170.

2 Guillaume Allais. Builtin types viewed as inductive families. *CoRR*, abs/2301.02194, 2023. doi:10.48550/arXiv.2301.02194.

3 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *J. Funct. Program.*, 31:e22, 2021. doi:10.1017/S0956796820000076.

4 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. *SIGPLAN Not.*, 52(1):666–679, January 2017. doi:10.1145/3093333.3009866.

5 Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. doi:10.1145/3209108.3209189.

6 Lennart Augustsson and Magnus Carlsson. An Exercise in Dependent Types: A Well-Typed Interpreter. In *Workshop on Dependent Types in Programming, Gothenburg*, 1999.

7 Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005. doi:10.1007/11541868_4.

8 Edwin C. Brady. *Type-Driven Development with Idris.* Manning Publications, 2017.

9 Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.9.

10 James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in Agda, for fun and profit. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 255–297. Springer, 2019. doi:10.1007/978-3-030-33636-3_10.

11 David R. Christiansen and Edwin C. Brady. Elaborator reflection: extending idris in idris. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 284–297. ACM, 2016. doi:10.1145/2951913.2951932.

12 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000. doi:10.1145/351240.351266.

13 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

14 Peter Dybjer. Inductive families. *Formal Aspects Comput.*, 6(4):440–465, 1994. doi:10.1007/BF01211308.

15 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches - conclusions from the language workbench challenge. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013. doi:10.1007/978-3-319-02654-1_11.

16 Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

17 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

18 Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL):45:1–45:30, 2018. doi:10.1145/3158133.

19 Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.

20 Conor McBride. Everybody's got to be somewhere. In Robert Atkey and Sam Lindley, editors, *Proceedings of the 7th Workshop on Mathematically Structured Functional Programming, MSFP@FSCD 2018, Oxford, UK, 8th July 2018*, volume 275 of *EPTCS*, pages 53–69, 2018. doi:10.4204/EPTCS.275.6.

21 Ulf Norell. Dependently typed programming in Agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008. doi:10.1007/978-3-642-04652-0_5.

22 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *Proc. ACM Program. Lang.*, 3(POPL):14:1–14:32, 2019. doi:10.1145/3290327.

23 Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2020. Version 5.8, http://softwarefoundations.cis.upenn.edu.

24 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 284–298. ACM, 2020. doi:10.1145/3372885.3373818.

25 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180. ACM, 2019. doi:10.1145/3293880.3294101.

26 The Coq Development Team. The coq proof assistant, January 2022. doi:10.5281/zenodo.5846982.

27 Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563355.

28 Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. Language design with the spoofax language workbench. *IEEE Softw.*, 31(5):35–43, 2014. doi:10.1109/MS.2014.100.

29 Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*, August 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

30 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.