JFP **23** (5): 552–593, 2013. © Cambridge University Press 2013
doi:10.1017/S095679681300018X

Idris, a general-purpose dependently typed programming language: Design and implementation

EDWIN BRADY
School of Computer Science, University of St Andrews, St Andrews KY16 9SX, UK
(*e-mail:* ecb10@st-andrews.ac.uk)

Abstract

Many components of a dependently typed programming language are by now well understood, for example, the underlying type theory, type checking, unification and evaluation. How to combine these components into a realistic and usable high-level language is, however, folklore, discovered anew by successive language implementors. In this paper, I describe the implementation of IDRIS, a new dependently typed functional programming language. IDRIS is intended to be a *general-purpose* programming language and as such provides high-level concepts such as implicit syntax, type classes and do notation. I describe the high-level language and the underlying type theory, and present a tactic-based method for *elaborating* concrete high-level syntax with implicit arguments and type classes into a fully explicit type theory. Furthermore, I show how this method facilitates the implementation of new high-level language constructs.

1 Introduction

Dependently typed programming languages, such as Agda (Norell, 2007) and Coq (Bertot & Castéran, 2004), have emerged in recent years as a promising approach to ensuring the correctness of software. Type checking ensures a program has the intended meaning; *dependent* types, where types may be predicated on values, allow a programmer to give a program a more precise type and hence have increased confidence of its correctness. The IDRIS language (Brady, 2011b) aims to take this idea further, by providing support for verification of *general-purpose* systems software. In contrast to Agda and Coq, which have arisen from the theorem proving community, IDRIS takes Haskell as its main influence. Recent Haskell extensions such as GADTs and type families have given some of the power of dependent types to Haskell programmers. In the short term this approach has a clear advantage, since it builds on a mature language infrastructure with extensive library support. Taking a longer term view, however, these extensions are inherently limited in that they are required to maintain backwards compatibility with existing Haskell implementations. IDRIS, being a new language, has no such limitation, essentially asking the question:

'What if Haskell had full *dependent types?'*

By *full* dependent types, we mean that there is no restriction on which values may appear in types. It is important for the sake of usability of a programming language to provide a notation which allows programmers to express high-level concepts in a natural way. Taking Haskell as a starting point means that Idris offers a variety of high-level structures such as type classes, `do`-notation, primitive types and monadic I/O, for example. Furthermore, a goal of Idris is to support high-level domain-specific language implementation, providing appropriate notation for *domain experts* and *systems programmers*, who should not be required to be type theorists in order to solve programming problems. Nevertheless, it is important for the sake of correctness of the language implementation to have a well-defined core language with well-understood meta-theory (Altenkirch *et al.*, 2010). How can we achieve both of these goals?

This paper describes a method for elaborating a high-level dependently typed functional programming language to a low-level core based on dependent type theory. The method involves building an elaboration monad which captures the state of incomplete programs and a collection of *tactics* used by the machine to refine programs, directed by high-level syntax. As we shall see, this method allows higher level language constructs to be elaborated in a straightforward manner, without compromising the simplicity of the underlying type theory.

1.1 Contributions

A dependently typed programming language relies on several components, many of which are now well understood. For example, we rely on a type checker for the core type theory (Chapman *et al.*, 2005; Löh *et al.*, 2010), a unification algorithm (Miller, 1992) and an evaluator. However, it is less well understood how to combine these components effectively into a practical programming language.

The primary contribution of this paper is a tactic-based method for translating programs in a high-level dependently typed programming language to a small core type theory, TT, based on UTT (Luo, 1994). The paper describes the structure of an elaboration monad capturing proof and system state, and introduces a collection of *tactics* which the machine uses to manipulate incomplete programs. Secondly, the paper gives a detailed description of the core type theory used by Idris, including a full description of the typing rules. Finally, through describing the specific tactics, the paper shows how to extend Idris with higher level features. While we apply these ideas to Idris specifically, the method for term construction is equally applicable to other typed programming languages, and indeed the tactics themselves are applicable to any high-level language which can be explained in terms of TT.

1.2 Outline

Translating an Idris source program to an executable proceeds through several phases is illustrated below:

$$\textsc{Idris} \xrightarrow{\text{(desugaring)}} \textsc{Idris}^- \xrightarrow{\text{(elaboration)}} \mathsf{TT} \xrightarrow{\text{(compilation)}} \text{Executable}$$

The main focus of this paper is the elaboration phase, which translates a desugared language Idris$^-$ into a core language TT. In order to put this into its proper context, I give an overview of the high-level language IDRIS in Section 2 and explain the core language TT and its typing rules in Section 3. Section 4 describes the elaboration process itself, beginning with a tactic-based system for constructing TT programs, then introducing the desugared language IDRIS$^-$ and showing how this is translated into TT using tactics. Section 5 describes the process for translating TT back to IDRIS and properties of the translation; and finally, Section 6 discusses related work and Section 7 concludes.

1.3 Typographical conventions

This paper presents programs in two different but related languages: a high-level language IDRIS intended for programmers, and a low-level language TT to which IDRIS is elaborated. We distinguish these languages typographically as follows:

- IDRIS programs are written in `typewriter` font, as they are written in a conventional text editor. We use $\mathtt{e}_i$ to stand for non-terminal expressions.
- TT programs are written in mathematical notation, with names arising from IDRIS expressions written in `typewriter` font. We use vector notation $\vec{e}$ to stand for sequences of expressions.

Additionally, we describe the translation from IDRIS to TT in the form of *meta-operations*, which in practice are Haskell programs. Meta-operations are operations on IDRIS and TT syntax, and are identified by their names being in SMALLCAPS.

1.4 Elaboration example

A simple example of an IDRIS program is the following, which adds corresponding elements of vectors of the same length:

```
vAdd : Num a => Vect n a -> Vect n a -> Vect n a
vAdd Nil       Nil       = Nil
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

This illustrates some basic features of IDRIS:

- Functions are defined by pattern matching, with a top-level type signature. Names which are free in the type signature (`a` and `n`) here are implicitly bound.
- The type system ensures that both input vectors are of the same length (`n`) and have the same element type (`a`), and that the element type and length are preserved in the output vector.
- Functions can be overloaded using *classes*. Here the element type of the vector `a` must be numeric and therefore supports the `+` operator.
- Unlike Haskell, a single colon is used for type signatures, and a double colon for the cons operator, emphasising the importance of types.

IDRIS programs elaborate into a small core language, TT, which is a λ-calculus with dependent types, augmented with algebraic data types and pattern matching. TT programs are fully explicitly typed, including the names `a` and `n` in the type signature, and the names bound in each pattern match clause. Classes are also made explicit. The TT translation of `vAdd` is:

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

$$
\begin{array}{l}
\mathtt{vAdd} : (a : \mathtt{Type}) \to (n : \mathtt{Nat}) \to \mathtt{Num}\,a \to \mathtt{Vect}\,n\,a \to \mathtt{Vect}\,n\,a \to \mathtt{Vect}\,n\,a \\
\underline{\mathsf{var}}\; a : \mathtt{Type},\; c : \mathtt{Num}\,a. \\
\quad \mathtt{vAdd}\,a\,\mathtt{Z}\,c\,(\mathtt{Nil}\,a)\,(\mathtt{Nil}\,a) \;\mapsto\; \mathtt{Nil}\,a \\
\underline{\mathsf{var}}\; a : \mathtt{Type},\; k : \mathtt{Nat},\; c : \mathtt{Num}\,a, \\
\quad x : a,\; xs : \mathtt{Vect}\,k\,a,\; y : a,\; ys : \mathtt{Vect}\,k\,a. \\
\quad \mathtt{vAdd}\,a\,(\mathtt{S}\,k)\,c\,((::)\,a\,k\,x\,xs)\,((::)\,a\,k\,y\,ys) \\
\qquad\qquad \mapsto\; ((::)\,a\,k\,((+)\,c\,x\,y)\,(\mathtt{vAdd}\,a\,k\,c\,xs\,ys))
\end{array}
$$

The rest of this paper describes Idris and TT, and systematically explains how to translate from one to the other.

2 Idris – the high-level language

Idris is a pure functional programming language with dependent types. It is eagerly evaluated by default, and compiled via the Epic supercombinator library (Brady, 2011a), with irrelevant values and proof terms automatically erased (Brady *et al.*, 2003; Brady, 2005). In this section I present some small examples to illustrate the features of Idris. A full tutorial is available elsewhere (Brady, 2013).

2.1 Preliminaries

Idris defines several primitive types: fixed width integers `Int`, arbitrary width integers `Integer`, `Float` for numeric operations, `Char` and `String` for text manipulation and `Ptr` which represents foreign pointers. There are also several data types declared in the library, including `Bool`, with values `True` and `False`. All of the usual arithmetic and comparison operators are defined for the primitive types.

An Idris program consists of a module declaration, followed by an optional list of imports and a collection of definitions and declarations, for example:

```
module Main

x : Int
x = 42

main : IO ()
main = putStrLn ("The answer is " ++ show x)
```

Like Haskell, the main function is called `main`, and input and output are managed with an `IO` monad. Unlike Haskell, however, *all* top-level functions must have a type signature. This is due to type inference being, in general, undecidable for languages with dependent types.

A module declaration also opens a *namespace*. The fully qualified names declared in this module are `Main.x` and `Main.main`.

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

2.2 Data types

Data types may be declared in a similar way to Haskell data types, with a similar syntax. For example, Peano natural numbers and lists are respectively declared in the library as follows:

```
data Nat    = Z   | S Nat
data List a = Nil | (::) a (List a)
```

A standard example of a *dependent* type is the type of ‘lists with length’, conventionally called ‘vectors’ in the dependently typed programming literature. In IDRIS, vectors are declared as follows:

```
data Vect : Nat -> Type -> Type where
   Nil  : Vect Z a
   (::) : a -> Vect k a -> Vect (S k) a
```

The above declaration creates a family of types. The syntax resembles a Haskell GADT declaration: It explicitly states the type of the type constructor `Vect` – it takes a `Nat` and a type as arguments, where `Type` stands for the type of types. We say that `Vect` is *indexed* over `Nat` and *parameterised* by a type. The distinction between parameters and indices is that a parameter is fixed across an entire data structure, whereas an index may vary. Note that constructor names may be overloaded (such as `Nil` and `(::)` here) provided that they are declared in separate namespaces.

Note that in the library, `List` is declared in a module `Prelude.List`, and `Vect` in a module `Prelude.Vect`, so the fully qualified names of `Nil` are `Prelude.List.Nil` and `Prelude.Vect.Nil`. They may be used qualified or unqualified; if used unqualified the elaborator will disambiguate by type.

2.3 Functions

Functions are implemented by pattern matching. For example, the following function defines concatenation of vectors, expressing in the type that the resulting vector’s length is the sum of the input lengths:

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
Nil       ++ ys = ys
(x :: xs) ++ ys = x :: xs ++ ys
```

Functions can also be defined *locally* using `where` clauses. For example, to define a function which reverses a list, we can use an auxiliary function which accumulates the new, reversed list, and does not need to be visible globally:

```
reverse : List a -> List a
reverse xs = revAcc Nil xs where
  revAcc : List a -> List a -> List a
  revAcc acc Nil = acc
  revAcc acc (x :: xs) = revAcc (x :: acc) xs
```

2.3.1 Totality checking

Internally, IDRIS programs are checked for *totality* – that they produce an answer in finite time for all possible inputs – but they are not *required* to be total by default. Totality

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

checking serves two purposes: firstly, if a program terminates for all inputs, then its type gives a strong guarantee about the properties specified by its type; secondly, we can optimise total programs more aggressively (Brady *et al.*, 2003).

Totality checking can be enforced by using the `total` keyword. For example, the following definition is accepted:

```
total vAdd : Num a => Vect n a -> Vect n a -> Vect n a
vAdd Nil       Nil       = Nil
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

The elaborator can verify that this is total by checking that it covers all possible patterns – in this case, both arguments must be of the same form as the type requires that the input vectors are of the same length – and that recursive calls are on structurally smaller values.

Idris also supports coinductive definitions, although this and further details of totality checking are beyond the scope of this paper; the totality checker is implemented independently of elaboration and type checking.

2.4 *Implicit arguments*

In order to understand how Idris elaborates to an underlying type theory, it is important to understand the notion of *implicit* arguments. To illustrate this, consider the finite sets:

```
data Fin : Nat -> Type where
      fZ : Fin (S k)
      fS : Fin k -> Fin (S k)
```

As the name suggests, these are sets with a finite number of elements. The declaration gives `fZ` as the zeroth element of a finite set with `S k` elements, and `fS n` as the `n+1`th element of a finite set with `S k` elements.

`Fin` is indexed by `Nat`, which represents the number of elements in the set. Neither constructor targets `Fin Z`, because the empty set has no elements. The following function uses an element of a finite set as a bounds-safe index into a vector:

```
index : Fin n -> Vect n a -> a
index fZ     (x :: xs) = x
index (fS k) (x :: xs) = index k xs
```

Let us take a closer look at its type. It takes two arguments, an element of the finite set of `n` elements, and a vector with `n` elements of type `a`. But there are also two names, `n` and `a`, which are not declared explicitly. These are *implicit* arguments to `index`. The type of `index` could also be written as:

```
index : {a:_} -> {n:_} -> Fin n -> Vect n a -> a
```

This gives bindings for `a` and `n` with placeholders for their types, to be inferred by the machine. These types could also be given explicitly:

```
index : {a:Type} -> {n:Nat} -> Fin n -> Vect n a -> a
```

Implicit arguments, given in braces {} in the type signature, are not given in applications of `index`; their values can be inferred from the types of the `Fin n` and `Vect n a` arguments. Any name which appears in a non-function position in a type signature, but is otherwise free, will be automatically bound as an implicit argument. Implicit arguments can still be given explicitly in applications, using the syntax {a=value} and {n=value}, for example:

```
index {a=Int} {n=2} fZ (2 :: 3 :: Nil)
```

2.5 Classes

IDRIS supports overloading in two ways. Firstly, as we have already seen with the constructors of List and Vect, names can be overloaded in an ad hoc manner and resolved according to the context in which they are used. This is mostly for convenience, to eliminate the need to decorate constructor names in similarly structured data types, and eliminate explicit qualification of ambiguous names where only one is well-typed – this is especially useful for disambiguating record field names, although records are beyond the scope of this paper.

Secondly, IDRIS implements *classes*, following Haskell's type classes. This allows a more principled approach to overloading – a class gives a collection of overloaded operations which describe the interface for *instances* of that class. IDRIS classes follow Haskell 98 type classes, except that multiple parameters are supported, and that classes can be parameterised by *any* value, not just types. Hence, we refer to them as *classes* generally, rather than *type classes* specifically.

A simple example is the Show class, which is defined in the library and provides an interface for converting values to Strings:

```
class Show a where
    show : a -> String
```

An instance of a class is defined with an instance declaration, which provides implementations of the function for a specific type. For example:

```
instance Show Nat where
    show Z = "Z"
    show (S k) = "s" ++ show k
```

Instance declarations can themselves have constraints. For example, to define a Show instance for vectors, we need to know that there is a Show instance for the element type, because we are going to use it to convert each element to a String:

```
instance Show a => Show (Vect n a) where
    show xs = "[" ++ show' xs ++ "]" where
        show' : Vect n a -> String
        show' Nil        = ""
        show' (x :: Nil) = show x
        show' (x :: xs)  = show x ++ ", " ++ show' xs
```

Classes can also be extended. A logical next step from an equality relation Eq, for example, is to define an ordering relation Ord. We can define an Ord class which inherits methods from Eq as well as defining some of its own:

```
data Ordering = LT | EQ | GT

class Eq a => Ord a where
    compare : a -> a -> Ordering
    (<) : a -> a -> Bool
    -- etc
```

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

2.6 Matching on intermediate values

2.6.1 `case` expressions

Intermediate values of *non-dependent* types can be inspected using a `case` expression. For example, `list_lookup` looks up an index in a list, returning `Nothing` if the index is out of bounds. This can be used to write `lookup_default`, which looks up an index and returns a default value if the index is out of bounds:

```
lookup_default : Nat -> List a -> a -> a
lookup_default i xs def = case list_lookup i xs of
                              Nothing => def
                              Just x => x
```

The `case` construct is intended for simple analysis of intermediate expressions to avoid the need to write auxiliary functions. It will only work if each branch *matches* a value of the same type, and *returns* a value of the same type.

2.6.2 The `with` rule

Often, matching is required on the result of an intermediate computation with a dependent type. IDRIS provides a construct for this, the `with` rule, inspired by views in EPIGRAM (McBride & McKinna, 2004), which takes account of the fact that matching on a value in a dependently typed language can affect what is known about the forms of other values. For example, a `Nat` is either even or odd. If it is even, it will be the sum of two equal `Nat`s. Otherwise, it is the sum of two equal `Nat`s plus one:

```
data Parity : Nat -> Type where
   even : Parity (n + n)
   odd  : Parity (S (n + n))
```

We say `Parity` is a *view* of `Nat`. It has a *covering function* which tests whether it is even or odd and constructs the predicate accordingly:

```
parity : (n:Nat) -> Parity n
```

Using this, a function which converts a natural number to a list of binary digits (least significant first) is written as follows, using the `with` rule:

```
natToBin : Nat -> List Bool
natToBin Z = Nil
natToBin k with (parity k)
   natToBin (j + j)     | even = False :: natToBin j
   natToBin (S (j + j)) | odd  = True  :: natToBin j
```

The value of the result of `parity k` affects the form of `k`, because the result of `parity k` depends on `k`. So, as well as the patterns for the result of the intermediate computation (`even` and `odd`) right of the |, the definition also expresses how the results affect the other patterns left of the |. Note that there is a function in the patterns (+) and repeated occurrences of `j` – this is allowed because another argument has determined the form of these patterns. In general, non-linear patterns are allowed *only* when their value is forced by some other argument.

$$
\begin{array}{rlll}
\text{Terms, } t ::= & c & \text{(constant)} \\
\mid & x & \text{(variable)} \\
\mid & b.\ t & \text{(binding)} \\
\mid & t\ t & \text{(application)} \\
\mid & \mathtt{T} & \text{(type constructor)} \\
\mid & \mathtt{D} & \text{(data constructor)} \\
\\
\text{Constants, } c ::= & \mathtt{Type}_i & \text{(type universes)} \\
\mid & i & \text{(integer literal)} \\
\mid & str & \text{(string literal)}
\end{array}
\qquad
\begin{array}{rll}
\text{Binders, } b ::= & \lambda x{:}t & \text{(abstraction)} \\
\mid & \underline{\text{let}}\ x \mapsto t\ :\ t & \text{(let binding)} \\
\mid & \forall x{:}t & \text{(function space)}
\end{array}
$$

Fig. 1. TT expression syntax.

3 The core type theory

High-level IDRIS programs, as described in Section 2, are *elaborated* to a small core language, TT, then type checked. TT is a dependently typed λ-calculus with inductive families and pattern matching definitions similar to UTT (Luo, 1994), and building on an earlier implementation, IVOR (Brady, 2006). The core language is kept as small as is reasonably possible, which has several advantages: it is easy to type check, since type checking dependent type theory is well understood (Löh *et al.*, 2010); and it is easy to transform, optimise and compile. Keeping the core language small increases confidence that these important components of the language are correct. In this section I describe TT and its semantics.

3.1 TT *syntax*

The syntax of TT expressions is given in Figure 1. This defines:

- **Terms**, t, which can be variables, bindings, applications or constants.
- **Bindings**, b, which can be lambda abstractions, let bindings or function spaces.
- **Constants**, c, which can be integer or string literals, or $\mathtt{Type}_i$, the type of types, and may be extended with other primitives.

The function space $\forall x{:}S.\,T$ may also be written as $(x\ :\ S) \to T$, or $S \to T$ if x is not free in T, to make the notation more consistent with the high-level language. Universe levels on types ($\mathtt{Type}_i$) may be left implicit and inferred by the machine (Pollack, 1990).

A TT program is a collection of **inductive family** definitions (Section 3.3) and **pattern matching** function definitions (Section 3.4), as well as primitive operators on constants. Before defining these, let us define the semantics of TT expressions.

3.2 TT *semantics*

The static and dynamic semantics of TT are defined mutually, since evaluation relies on a well-typed term, and type checking relies on evaluation. Everything is defined relative to a context, Γ. The empty context is valid, as is a context extended with a λ, $\forall$ or $\underline{\text{let}}$ binding:

$$
\frac{}{\vdash \underline{\text{valid}}}
\qquad
\frac{\Gamma \vdash S\ :\ \mathtt{Type}_i}{\Gamma; \lambda x{:}S \vdash \underline{\text{valid}}}
\qquad
\frac{\Gamma \vdash S\ :\ \mathtt{Type}_i}{\Gamma; \forall x{:}S \vdash \underline{\text{valid}}}
\qquad
\frac{\Gamma \vdash S\ :\ \mathtt{Type}_i \qquad \Gamma \vdash s\ :\ S}{\Gamma; \underline{\text{let}}\ x\ \mapsto\ s\ :\ S \vdash \underline{\text{valid}}}
$$

We record the binding form in the context, as well as the type, because type checking relies on evaluation. In particular, $\underline{\text{let}}$ bound names may reduce.

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

$$\begin{array}{ll} \beta\text{−contraction} & \dfrac{}{\Gamma \vdash (\lambda x{:}S.\,t)\ s \leadsto t[s/x]} \\[2ex] \delta\text{−contraction} & \dfrac{}{\Gamma; \underline{\text{let}}\ x \mapsto s\ :\ S; \Gamma' \vdash x \leadsto s} \end{array}$$

Fig. 2. TT contraction schemes.

3.2.1 Evaluation

There are two distinct phases in which TT evaluation occurs, namely **compile-time** and **run-time**. Compile-time evaluation is primarily to support type-checking, in which equivalence of two terms (or types) is determined by comparing their normal forms. It is important that terms are strongly normalising at compile-time, so any pattern matching definition which is not guaranteed to be total does not reduce. In this section we discuss compile-time evaluation; run-time evaluation differs in that there is no reduction under binders, types and duplicated values have been erased (Brady *et al.*, 2003), and non-normalising definitions may reduce.

Compile-time evaluation of TT is defined by contraction schemes, given in Figure 2. **Contraction**, relative to a context Γ, is given by one of the following schemes:

- β-contraction, which substitutes a value applied to a λ-binding for the bound variable.
- δ-contraction, which replaces a $\underline{\text{let}}$ bound variable with its value.

The following contextual closure rule reduces a $\underline{\text{let}}$ binding by creating a δ-reducible expression:

$$\dfrac{\Gamma; \underline{\text{let}}\ x \mapsto s\ :\ S \vdash t \leadsto u}{\Gamma \vdash \underline{\text{let}}\ x \mapsto s\ :\ S.\,t \leadsto u}$$

Reduction ($\triangleright$) is the structural closure of contraction, and evaluation ($\triangleright^*$) is the transitive closure of reduction. **Conversion** ($\simeq$) is the smallest equivalence relation closed under reduction. If $\Gamma \vdash x \simeq y$ then y can be obtained from x by a finite, possibly empty, sequence of contractions and reversed contractions. Terms which are convertible are also said to be definitionally equal. The evaluator can also be extended by defining pattern matching functions, which will be described in more detail in Section 3.4. In principle, pattern matching functions can be understood as extending the core language with high-level reduction rules.

3.2.2 Typing rules

The type inference rules for TT expressions are given in Figure 3. These rules use the **cumulativity** relation, defined in Figure 4. The type of types, $\texttt{Type}_i$ is parameterised by a universe level, to prevent Girard's paradox (Coquand, 1986). There is an infinite hierarchy of predicative universes. Cumulativity allows programs at lower universe levels to inhabit higher universe levels. In practice, universe levels can be left implicit (and will be left implicit in the rest of this paper) – the type checker generates a graph of constraints between universe levels (such as that produced by the Forall typing rule) and checks that there are no cycles. This happens as the last stage of type checking, waiting until we have all uses of a family to establish how many levels are needed in the universe hierarchy. Otherwise,

$$\mathsf{Type}\ \frac{\Gamma\vdash\underline{\text{valid}}}{\Gamma\vdash\mathtt{Type}_n\ :\ \mathtt{Type}_{n+1}}$$

$$\mathsf{Const}_1\ \frac{\Gamma\vdash\underline{\text{valid}}}{\Gamma\vdash i\ :\ \mathtt{Int}}\qquad\mathsf{Const}_2\ \frac{\Gamma\vdash\underline{\text{valid}}}{\Gamma\vdash \mathit{str}\ :\ \mathtt{String}}$$

$$\mathsf{Const}_3\ \frac{\Gamma\vdash\underline{\text{valid}}}{\Gamma\vdash\mathtt{Int}\ :\ \mathtt{Type}_0}\qquad\mathsf{Const}_4\ \frac{\Gamma\vdash\underline{\text{valid}}}{\Gamma\vdash\mathtt{String}\ :\ \mathtt{Type}_0}$$

$$\mathsf{Var}_1\ \frac{(\lambda x\!:\!S)\in\Gamma}{\Gamma\vdash x\ :\ S}\qquad\mathsf{Var}_2\ \frac{(\forall x\!:\!S)\in\Gamma}{\Gamma\vdash x\ :\ S}\qquad\mathsf{Val}\ \frac{(\underline{\text{let}}\,x\mapsto s\ :\ S)\in\Gamma}{\Gamma\vdash x\ :\ S}$$

$$\mathsf{App}\ \frac{\Gamma\vdash f\ :\ (x\ :\ S)\to T\qquad\Gamma\vdash s\ :\ S}{\Gamma\vdash f\ s\ :\ T[s/x]}$$

$$\mathsf{Lam}\ \frac{\Gamma;\lambda x\!:\!S\vdash e\ :\ T\qquad\Gamma\vdash(x\ :\ S)\to T\ :\ \mathtt{Type}_n}{\Gamma\vdash\lambda x\!:\!S.e\ :\ (x\ :\ S)\to T}$$

$$\mathsf{Forall}\ \frac{\Gamma;\forall x\!:\!S\vdash T\ :\ \mathtt{Type}_m\qquad\Gamma\vdash S\ :\ \mathtt{Type}_n}{\Gamma\vdash(x\ :\ S)\to T\ :\ \mathtt{Type}_p}\ (\exists p.m\leq p,\ n\leq p)$$

$$\mathsf{Let}\ \frac{\begin{array}{cc}\Gamma\vdash e_1\ :\ S & \Gamma;\underline{\text{let}}\,x\mapsto e_1\ :\ S\vdash e_2\ :\ T\\ \Gamma\vdash S\ :\ \mathtt{Type}_n & \Gamma;\underline{\text{let}}\,x\mapsto e_1\ :\ S\vdash T\ :\ \mathtt{Type}_n\end{array}}{\Gamma\vdash\underline{\text{let}}\,x\mapsto e_1\ :\ S.\ e_2\ :\ T[e_1/x]}$$

$$\mathsf{Conv}\ \frac{\Gamma\vdash x\ :\ A\qquad\Gamma\vdash A'\ :\ \mathtt{Type}_n\qquad\Gamma\vdash A\preceq A'}{\Gamma\vdash x\ :\ A'}$$

Fig. 3. Typing rules for TT.

$$\frac{\Gamma\vdash S\simeq T}{\Gamma\vdash S\preceq T}\qquad\frac{}{\Gamma\vdash\mathtt{Type}_n\preceq\mathtt{Type}_{n+1}}$$

$$\frac{\Gamma\vdash R\preceq S\qquad\Gamma\vdash S\preceq T}{\Gamma\vdash R\preceq T}$$

$$\frac{\Gamma\vdash S_1\simeq S_2\qquad\Gamma;x\ :\ S_1\vdash T_1\preceq T_2}{\Gamma\vdash\forall x\!:\!S_1.T_1\preceq\forall x\!:\!S_2T_2}$$

Fig. 4. Cumulativity.

the typing rules are standard and type inference can be implemented in the usual way (Löh *et al.*, 2010).

3.3 Inductive families

Inductive families (Dybjer, 1994) are a form of simultaneously defined collection of algebraic data types which can be parameterised over *values* as well as types. An inductive family is declared in a similar style to a Haskell GADT declaration (Peyton Jones *et al.*, 2006) as follows:

$$\underline{\text{data}}\quad\mathtt{T}\ :\ t\quad\underline{\text{where}}\quad\mathtt{D}_1\ :\ t\ \mid\ \ldots\ \mid\ \mathtt{D}_n\ :\ t$$

A constructor `D` of a family `T` must return a value in the family `T`. It may take recursive arguments in the family `T` which may be higher order. Using vector notation, $\vec{x}$, to indicate a sequence of zero or more x (i.e. $x_1, x_2, \ldots, x_n$), recursive arguments have types of the form

$$\forall x_1\!:\!S_1.\ldots\forall x_n\!:\!S_n.\,\mathtt{T}\,\vec{y}$$

where T does not occur in any of the $\vec{S}$ or the $\vec{y}$. This restriction is known as **strict positivity** and ensures that recursive arguments of the constructor are structurally smaller than the value itself. In practice, this restriction can be relaxed when defining a family, but definitions using families which do not satisfy strict positivity will not pass totality checking.

For example, the IDRIS data type `Nat` would be declared in TT as follows:

$$\underline{\text{data}} \quad \mathtt{Nat} : \mathtt{Type} \quad \underline{\text{where}} \quad \mathtt{Z} : \mathtt{Nat} \mid \mathtt{S} : (k : \mathtt{Nat}) \rightarrow \mathtt{Nat}$$

A data type may have zero or more parameters (which are invariant across a structure) and a number of indices, given by the type. For example, the TT equivalent of `List` is parameterised over its element type:

$$\begin{array}{ll} \underline{\text{data}} & \mathtt{List} : (a : \mathtt{Type}) \rightarrow \mathtt{Type} \quad \underline{\text{where}} \\ & \;\; \mathtt{Nil} : \mathtt{List}\; a \\ & \mid (::) : (x : a) \rightarrow (xs : \mathtt{List}\; a) \rightarrow \mathtt{List}\; a \end{array}$$

Types can be parameterised or indexed over values. Using this, we can declare the type of vectors (lists with length), where the empty list is statically known to have length zero, and the non-empty list is statically known to have a non-zero length. The TT equivalent of `Vect` is parameterised over its element type, like `List`, but *indexed* over its length. Note also that the length index k is given *explicitly*,

$$\begin{array}{ll} \underline{\text{data}} & \mathtt{Vect} : \mathtt{Nat} \rightarrow (a : \mathtt{Type}) \rightarrow \mathtt{Type} \quad \underline{\text{where}} \\ & \;\; \mathtt{Nil} : \mathtt{Vect}\; \mathtt{Z}\; a \\ & \mid (::) : (k : \mathtt{Nat}) \rightarrow (x : a) \rightarrow (xs : \mathtt{Vect}\; k\; a) \rightarrow \mathtt{Vect}\; (\mathtt{S}\; k)\; a \end{array}$$

3.4 Pattern matching definitions

A pattern matching definition for a function named `f` takes the following form, consisting of a type declaration followed by one or more pattern clauses:

$$\begin{array}{l} \mathtt{f} : t \\ \underline{\text{var}}\, \vec{x}_1 : \vec{t}_1 .\, \mathtt{f}\; \vec{t}_1 \;=\; t_1 \\ \ldots \\ \underline{\text{var}}\, \vec{x}_n : \vec{t}_n .\, \mathtt{f}\; \vec{t}_n \;=\; t_n \end{array}$$

A pattern clause consists of a list of pattern variable bindings, introduced by <u>var</u>, and left- and right-hand sides, both of which are TT expressions. Each side is type-checked relative to the variable bindings, and the types of each side must be convertible. Additionally, the left-hand side must take the form of `f` applied to a number of TT expressions, and the number of arguments must be the same in each clause. The right-hand side may include applications of `f`, i.e. pattern matching definitions may be recursive. The validity of a pattern clause is defined by the following rule:

$$\frac{\Gamma; \lambda \vec{x} : \vec{U} \vdash \mathtt{f}\; \vec{s} : S \qquad \Gamma; \lambda \vec{x} : \vec{U} \vdash e : T \qquad \Gamma \vdash S \simeq T}{\Gamma \vdash \underline{\text{var}}\, \vec{x} : \vec{U} .\, \mathtt{f}\; \vec{s} \;=\; e \;\underline{\text{valid}}}$$

Patterns are separated into the accessible patterns (variables and constructor forms which may be inspected) and inaccessible patterns, following Agda (Norell, 2007) then implemented by compilation into case trees (Augustsson, 1985). Non-terminating definitions are

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

Church–Rosser	If $\Gamma \vdash s \simeq t$ then there is a common reduct r such that $\Gamma \vdash s \rhd^* r \qquad \Gamma \vdash t \rhd^* r$
Subject Reduction	If $\Gamma \vdash s : S \qquad \Gamma \vdash s \rhd t$ then $\Gamma \vdash t : S$
Cut	If $\Gamma_0, \underline{\mathrm{let}}\, x \mapsto s : S, \Gamma_1 \vdash t : T$ then $\Gamma_0, \Gamma_1[s/x] \vdash t[s/x] : T[s/x]$
Strengthening	If $\Gamma_0, \mathscr{B}(x,T), \Gamma_1 \vdash s : S \qquad x \notin \Gamma_1, s, S$ then $\Gamma_0, \Gamma_1 \vdash s : S$ (where $\mathscr{B}(x,T) \vdash x : T$)
Strong normalisation	If $\Gamma \vdash t : T$ then t is strongly normalising

Fig. 5. Metatheoretic properties of TT.

permitted, but do not reduce at compile-time, which means in particular that they cannot be used to construct proofs. Termination analysis is implemented separately.

A valid pattern matching definition effectively extends TT with a new constant, with the given type (extending the initial typing rules given in Section 3.2.2) and reduction behaviour (extending the initial reduction rules given in Section 3.2.1).

3.5 Metatheory

We conjecture that TT respects the usual metatheoretic properties, as shown in Figure 5. Specifically: the **Church–Rosser** property (i.e. that distinct reduction sequences lead to the same normal form); **subject reduction** (i.e. that computation preserves type); the **cut** property (i.e. that $\underline{\mathrm{let}}$-bound terms may be substituted into their scope) and **strengthening** (i.e. that removing unused definitions from scope does not affect type checking). **Strong normalisation** holds at compile-time because pattern matching definitions which cannot be shown to be terminating do not have compile-time reduction behaviour.

3.6 Totality checking

In order to ensure termination of type checking we must distinguish terms for which evaluation definitely terminates, and those which may diverge. TT takes a simple but pragmatic and effective approach to termination checking: any functions which do not satisfy a syntactic constraint on recursive calls are marked as *partial*. Additionally, any function which calls a partial function or uses a data type which is not strictly positive is also marked as partial. We use the size change principle (Lee *et al.*, 2001) to determine whether (possibly mutually defined) recursive functions are guaranteed to terminate. Higher order function arguments are assumed to be total, although if a higher order function is called with a partial function as an argument, the calling function will be marked partial. TT also marks functions which do not cover all possible inputs as partial. This totality checking is independent of the rest of the type theory, and can be extended.

This approach, separating the termination requirement from the type theory, means that an IDRIS programmer makes the decision about the importance of totality for each function rather than having the totality requirement imposed by the type theory.

3.7 From IDRIS *to* TT

TT is a very small language, consisting only of data declarations and pattern matching function definitions. There is no significant innovation in the design of TT, and this is a deliberate choice – it is a combination of small, well-understood components. The kernel of the TT implementation, consisting of a type checker, evaluator and pattern match compiler, is less than 1,000 lines of Haskell code. If we are confident in the correctness of the kernel of TT, and any higher level language feature can be translated into TT, we can be more confident of the correctness of the high-level language implementation than if it were implemented directly.

The process of elaborating a high-level language into a core type theory like TT is, however, less well understood, and presents several challenges depending on the features of the high-level language.

3.7.1 Example elaboration

Recall the following IDRIS function:

```
vAdd : Num a => Vect n a -> Vect n a -> Vect n a
vAdd Nil       Nil       = Nil
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

In order to elaborate this to TT, we must resolve the implicit arguments, and make the type class explicit. The first step is to make the implicit arguments explicit, using a placeholder to stand for the arguments we have not yet resolved. The type class argument is also treated as an implicit argument, to which we give the name c:

```
vAdd : (a : _) -> (n : _) ->
       Num a -> Vect n a -> Vect n a -> Vect n a
vAdd _ _ c (Nil _)         (Nil _)         = (Nil _)
vAdd _ _ c ((::) _ _ x xs) ((::) _ _ y ys)
                 = (::) _ _ ((+) _ x y) (vAdd _ _ _ xs ys)
```

Next, we resolve the implicit arguments. Each implicit argument can only take one value for this program to be type correct – these are solved by a unification algorithm:

```
vAdd : (a : Type) -> (n : Nat) ->
       Num a -> Vect n a -> Vect n a -> Vect n a
vAdd a 0     c (Nil a)         (Nil a)         = Nil a
vAdd a (S k) c ((::) a k x xs) ((::) a k y ys)
                 = (::) a k ((+) c x y) (vAdd a k c xs ys)
```

Finally, to build the TT definition, we need to find the type of each pattern variable and state it explicitly. This leads to the following TT definition, switching to TT notation from the ASCII IDRIS syntax:

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

```
vAdd : (a : Type) → (n : Nat) → Num a → Vect n a → Vect n a → Vect n a
var a : Type, c : Num a.
   vAdd a Z c (Nil a) (Nil a)  ↦  Nil a
var a : Type, k : Nat, c : Num a,
   x : a, xs : Vect k a, y : a, ys : Vect k a.
   vAdd a (S k) c ((::) a k x xs) ((::) a k y ys)
         ↦  ((::) a k ((+) c x y) (vAdd a k c xs ys))
```

3.7.2 An observation: programming vs theorem proving

IDRIS programs may contain several high-level constructs not present in TT, such as implicit arguments, type classes, `where` clauses, pattern matching `let` and `case` constructs. We would like the high-level language to be as expressive as possible, while remaining possible to translate to TT.

Before considering how to achieve this, we make an observation about the distinction between programming and theorem proving with dependent types, and appropriate mechanisms for constructing programs and theorems:

- *Pattern matching* is a convenient abstraction for humans to write programs, in that it allows a programmer to express exactly the computational behaviour of a function.
- *Tactics*, such as those used in the Coq theorem prover (Bertot & Castéran, 2004), are a convenient abstraction for building proofs and programs by *refinement*.

The idea behind the IDRIS elaborator, therefore, is to use the high-level program to direct **tactics** to build TT programs by refinement. The elaborator is implemented as a Haskell monad capturing proof state, with a collection of tactics for updating and refining the proof state. The remainder of this paper describes this elaborator and demonstrates how it is used to implement the high-level features of IDRIS.

4 Elaborating IDRIS

An IDRIS program consists of a series of declarations – data types, functions, classes and instances. In this section, we describe how these high-level declarations are translated into a TT program consisting of inductive families and pattern matching function definitions. We will need to work at the *declaration* level, and at the *expression* level, defining the following meta-operations.

- $\mathscr{E}[\![\cdot]\!]$, which builds a TT expression from an IDRIS expression.
- ELAB, which processes a top-level IDRIS declaration by generating one or more TT declarations.
- TTDECL, which adds a top-level TT declaration.

4.1 The development calculus TT$_{dev}$

TT expressions are built by using high-level IDRIS expressions to direct a tactic-based theorem prover, which builds the TT expressions step by step, by refinement. In order

$$b ::= \ldots \mid ?x{:}t \ \text{(hole binding)} \quad \mid ?x{\approx}t{:}t \ \text{(guess)}$$

$$\mathsf{HoleVar}\ \frac{(?x{:}S) \in \Gamma}{\Gamma \vdash x \ : \ S} \qquad \mathsf{GuessVar}\ \frac{(?x{\approx}s{:}S) \in \Gamma}{\Gamma \vdash x \ : \ S}$$

$$\mathsf{Hole}\ \frac{\Gamma; ?x{:}S \vdash e \ : \ T}{\Gamma \vdash ?x{:}S.\,e \ : \ T} \qquad \mathsf{Guess}\ \frac{\Gamma; ?x{\approx}e_1{:}S \vdash e_2 \ : \ T}{\Gamma \vdash ?x{\approx}e_1{:}S.\,e_2 \ : \ T}$$

Fig. 6. TT_{dev} extensions.

to build expressions in this way, the type theory needs to support *incomplete* terms, and a method for term construction. To achieve this, we extend TT with *holes*, calling the extended calculus TT_{dev}. Holes stand for the parts of programs which have not yet been instantiated; this largely follows the Oleg development calculus (McBride, 1999).

The idea is to extend the syntax for binders with a *hole* binding and a *guess* binding. These extensions are given in Figure 6. The *guess* binding is similar to a let binding, but without any computational force, i.e. there are no reduction rules for guess bindings. Using binders to represent holes is useful in a dependently typed setting since one value may determine another. Attaching a guess to a binder ensures that instantiating one such variable also instantiates all of its dependencies.

4.2 Proof state

We will build expressions in TT from high-level Idris expressions by creating a top-level hole binding with the appropriate type, and refining it according to the structure of the Idris expression. Noting the Curry–Howard correspondence between programs and proofs, and the relationship between our method for elaboration and tactic-based theorem proving, we refer to a development as a *proof*. A proof state is a tuple, (C, Δ, e, P, Q), containing:

- a global context, C, containing pattern matching definitions and their types,
- a local context, Δ, containing pattern bindings,
- a proof term, e, in TT_{dev},
- unsolved unification problems, P
  - these take the form (Γ, e_1, e_2), i.e. the context in which the problem is to be solved, and the expressions to be unified,
- a hole queue, Q.

The *hole queue* is a priority queue of names of hole and guess binders $\langle x_1, x_2, \ldots, x_n \rangle$ in the proof term – ensuring that each bound name is unique. Holes refer to *sub-goals* in the proof. As a proof develops, unification problems arising from unsolved holes may be introduced, and may be solved when those holes are instantiated. When the hole queue is empty, and there are no remaining unification problems, the proof term is complete.

Creating a TT expression from an Idris expression involves creating a new proof state, with an empty proof term, and using the high-level definition to direct the building of a final proof state, with a complete proof term.

In the implementation, the proof state is captured in an elaboration monad, `Elab`, which is simply a state monad with exceptions. Exceptions are captured by the option type Result which either indicates success, or reports an error Err:

$\underline{\text{data}}\ \textsc{Result}\ t \ = \ \textsc{Success}\ t \mid \textsc{Failure}\ \textsc{Err}$

If any operation fails during a proof (e.g. a sub-term fails to type check) then the entire proof fails (i.e. returns FAILURE e with some error message e) unless otherwise handled.

The `Elab` monad supports various operations for querying and updating the proof state, manipulating terms, generating fresh names etc. However, we will describe IDRIS elaboration in terms of meta-operations on the proof state in order to capture the essence of the elaboration process without being distracted by implementation details. These meta-operations include:

- **Queries**, which retrieve values from the proof state, without modifying the state. For example, we can:
  - get the type of the current sub-goal (TYPE),
  - get the current proof term (TERM),
  - retrieve the local context Γ at the current sub-goal (CONTEXT),
  - type check (CHECK) or normalise (NORMALISE) a term relative to Γ.
- **Unification**, which unifies two terms (potentially solving sub-goals) relative to Γ. This may introduce new unification problems to the proof state.
- **Tactics**, which update the proof term. Tactics operate on the sub-term at the binder specified by the head of the hole queue Q.
- **Focussing** on a specific sub-goal, which brings a different sub-goal to the head of the hole queue.

Elaboration of an IDRIS expression involves creating a new proof state, running a series of tactics to build a complete proof term, then retrieving and *rechecking* the final proof term, which must be a TT program (i.e. does not contain any of the TT_{dev} extensions). We call a sub-term which contains no hole or guess bindings **pure**. Although a pure term does not contain hole or guess bindings, it may nevertheless *refer* to hole- or guess-bound variables.

The proof state is initialised with the NEWPROOF operation. Given a global context C, NEWPROOF t sets up the proof state as:

$(C, \cdot, ?x\!:\!t.\,x, \langle\rangle, \langle x\rangle)$

The local context is initially empty, and the initial hole queue is the x standing for the entire expression. The proof term is reset with the NEWTERM operation. In an existing proof state (C, Δ, e, P, Q), NEWTERM t discards the proof term and hole queue, and updates the proof state to:

$(C, \Delta, ?x\!:\!t.\,x, \langle\rangle, \langle x\rangle)$

Resetting the term and queue but retaining pattern bindings allows the elaborator to use pattern bindings from the left-hand side of a pattern matching definition in the term on the right-hand side.

4.3 Tactics

In order to build TT expressions from IDRIS programs, we define a collection of meta-operations for querying and modifying the proof state. Meta-operations may have side

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

effects, including failure, or updating the proof state, and may fail. We have the following primitive meta-operations:

- Get and Put which allow direct access to the proof state.
- Check$_\Gamma$ e, which type checks an expression e relative to a context Γ, returning its type. Check will fail if the expression is not well-typed.
- Convert$_\Gamma$ e_1 e_2, which checks that the well-typed expressions e_1 and e_2 respect the cumulativity relation (i.e. $\Gamma \vdash e_1 \preceq e_2$), and fails otherwise.
- Normalise$_\Gamma$ e, which evaluates a well-typed expression e relative to a context Γ, returning its normal form. Normalise reduces under binders, but does not reduce definitions which are not total.
- PrimUnify$_\Gamma$ e_1 e_2, which attempts to unify e_1 and e_2 by finding the values with which holes must be instantiated for e_1 and e_2 to be convertible relative to Γ (i.e. for $\Gamma \vdash e_1 \simeq e_2$ to hold).
- Subst x e, which instantiates a hole x directly with a term e, typically arising from unification and removes x from the hole queue.

Additionally, we have Focus, Unfocus, NewHole, NextHole and RemoveHole operations which manipulate and inspect the hole queue. Focus n moves a hole to the head of the hole queue:

$$\begin{aligned} \textsc{Focus}\ n \ &= \ \underline{\mathrm{do}}\ (C,\Delta,e,P,Q;n;Q') \leftarrow \textsc{Get} \\ &\qquad\quad \textsc{Put}\ (C,\Delta,e,P,n;Q;Q') \end{aligned}$$

Focus refers to a hole n by name. Hole names are either given explicitly in proofs, or chosen (as fresh names) by tactics. Correspondingly, Unfocus moves the hole currently in focus to the end of the hole queue:

$$\begin{aligned} \textsc{Unfocus} \ &= \ \underline{\mathrm{do}}\ (C,\Delta,e,P,n;Q) \leftarrow \textsc{Get} \\ &\qquad\quad \textsc{Put}\ (C,\Delta,e,P,Q;n) \end{aligned}$$

NewHole declares that a new hole has been created and is to be solved next; NextHole returns the next hole to be solved, and RemoveHole removes a hole from the head of the queue:

$$\begin{aligned} \textsc{NewHole}\ n \ &= \ \underline{\mathrm{do}}\ (C,\Delta,e,P,Q) \leftarrow \textsc{Get} \\ &\qquad\quad \textsc{Put}\ (C,\Delta,e,P,n;Q) \\ \textsc{NextHole} \ &= \ \underline{\mathrm{do}}\ (C,\Delta,e,P,n;Q) \leftarrow \textsc{Get} \\ &\qquad\quad \underline{\mathrm{return}}\ n \\ \textsc{RemoveHole} \ &= \ \underline{\mathrm{do}}\ (C,\Delta,e,P,n;Q) \leftarrow \textsc{Get} \\ &\qquad\quad \textsc{Put}\ (C,\Delta,e,P,Q) \end{aligned}$$

Tactics are meta-operations which operate on the sub-term given by the hole at the head of the hole queue in the proof state. They take the following form:

$$\begin{aligned} \textsc{Tac}_\Gamma\ \vec{args}\ t = \ &\underline{\mathrm{do}}\ \ldots \\ &\underline{\mathrm{return}}\ t' \end{aligned}$$

A tactic Tac takes a sequence of zero or more arguments $\vec{args}$ followed by the sub-term t on which it is operating. It runs relative to a context Γ, which contains all the bindings and pattern bindings in scope at that point in the term. The sub-term t will either be a hole

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

```
Tactic $tac$ = do $e \leftarrow$ Term
                  $h \leftarrow$ NextHole
                  $e' \leftarrow$ Tactic' $\langle\rangle$ $e$
                  SetTerm $e'$

 where
    Tactic' $\Gamma$ $(?x{:}T.\,e)$       $= tac_{\Gamma}\ (?x{:}T.\,e)$    if $x = h$
    Tactic' $\Gamma$ $(?x{:}T.\,e)$       $=$ do $T' \leftarrow$ Tactic' $\Gamma\ T$
                                          return $(?x{:}T'.$ Tactic' $(\Gamma; ?x{:}T')\ e)$    otherwise
    Tactic' $\Gamma$ $(?x{\approx}v{:}T.\,e)$    $= tac_{\Gamma}\ (?x{\approx}v{:}T.\,e)$    if $x = h$
    Tactic' $\Gamma$ $(?x{\approx}v{:}T.\,e)$    $=$ do $v' \leftarrow$ Tactic' $\Gamma\ v$
                                          $T' \leftarrow$ Tactic' $\Gamma\ T$
                                          return $(?x{\approx}v'{:}T'.$ Tactic' $(\Gamma; ?x{\approx}v'{:}T'\ e)$    otherwise
    Tactic' $\Gamma$ $(\lambda x{:}T.\,e)$       $=$ do $T' \leftarrow$ Tactic' $\Gamma\ T$
                                          return $(\lambda x{:}T'.$ Tactic' $(\Gamma; \lambda x{:}T')\ e)$
    Tactic' $\Gamma$ $(\forall x{:}T.\,e)$       $=$ do $T' \leftarrow$ Tactic' $\Gamma\ T$
                                          return $(\forall x{:}T'.$ Tactic' $(\Gamma; \forall x{:}T')\ e)$
    Tactic' $\Gamma$ (let $x \mapsto v : T.\,e)$ $=$ do $v' \leftarrow$ Tactic' $\Gamma\ v$
                                          $T' \leftarrow$ Tactic' $\Gamma\ T$
                                          return (let $x \mapsto v' : T'.$ Tactic' $(\Gamma;$ let $x \mapsto v' : T')\ e)$
    Tactic' $\Gamma$ $(f\ a)$             $=$ return (Tactic' $\Gamma\ f$) (Tactic' $\Gamma\ a$)
    Tactic' $\Gamma$ $t$                  $=$ return $t$
```

Fig. 7. Applying tactics to the proof state.

binding $?x{:}T.\,e$ or a guess binding $?x{\approx}v{:}T.\,e$. The tactic returns a new term t' which can take any form, provided it is well-typed, with a type convertible to the type of t. Tactics may also have the side effect of updating the proof state, therefore we will describe tactics in a pseudo-code with do notation.

Tactics are executed by a higher level meta-operation Tactic, given in Figure 7, which locates the appropriate sub-term, applies the tactic with the context local to this sub-term and replaces the sub-term with the term returned by the tactic. Note that monadic lifting is left implicit in this figure so that the details of the algorithm are not obscured.

In the remainder of this section, we will define a set of primitive tactics: Claim, Fill and Solve, which are used to create and destroy holes; and Lambda, Pi, Let and Attack, which are used to create binders. Firstly, however, we must explain how unification is applied during elaboration.

4.3.1 Unification

Unification is crucial to the success of elaboration – it is unification which finds values of implicit arguments, in particular. We do not require unification to succeed immediately, in general, as instantiating a hole may affect the solution to other unification problems. Instead, $\textsc{PrimUnify}_{\Gamma}\ e_1\ e_2$ may give one of the following results:

- Success $(\vec{x} = \vec{e}, \vec{p})$, containing the solved holes mapping x to e, and a possibly empty set of blocked unification problems $\vec{p}$. If $\vec{p}$ is empty, there is a solution unifying e_1 and e_2 in the context Γ.
- Failure e, if no solution is found due to a failure to unify sub-terms in normal form.

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

$$
\begin{array}{lll}
\textsc{Unify}_{\Gamma}\, e_1\, e_2 & = & \underline{\mathrm{do}}\ (\vec{x} = \vec{e}, \vec{p}) \leftarrow \textsc{PrimUnify}_{\Gamma}\, e_1\, e_2 \\
 & & \quad \textsc{Subst}\, \vec{x}\, \vec{e} \\
 & & \quad \textsc{SubstProblems}\, \vec{x}\, \vec{e} \\
 & & \quad (C, \Delta, \mathit{term}, \vec{P}, Q) \leftarrow \textsc{Get} \\
 & & \quad \textsc{ReUnify}\, \vec{P} \\
 & & \\
\textsc{ReUnify}\ (\Gamma, e_1, e_2) & = & \underline{\mathrm{do}}\ (\vec{x} = \vec{e}, \vec{p}) \leftarrow \textsc{PrimUnify}_{\Gamma}\, e_1\, e_2 \\
 & & \quad \textsc{Subst}\, \vec{x}\, \vec{e} \\
 & & \quad \underline{\mathrm{return}}\ \vec{p}
\end{array}
$$

Fig. 8. The UNIFY meta-operation.

UNIFY attempts to solve the given unification problem using PRIMUNIFY, then attempts to further refine existing problems in the proof state. This operation is given in Figure 8. After attempting to solve the problem with PRIMUNIFY and updating the proof term with any solutions found, it updates the existing problems (with SUBSTPROBLEMS which applies SUBST across the terms in the problems), then attempts to resolve the existing problems with REUNIFY.

4.3.2 Creating and destroying holes

The CLAIM tactic, given a name and a type, adds a new hole binding in the scope of the current goal x, adding the new binding to the hole queue, but keeping x at the head:

$$
\begin{array}{lll}
\textsc{Claim}_{\Gamma}\, (y\ :\ S)\, (?x{:}T.\, e) & = & \underline{\mathrm{do}}\ \textsc{NewHole}\ y \\
 & & \quad \textsc{Focus}\, x \\
 & & \quad \underline{\mathrm{return}}\ ?y{:}S.\, ?x{:}T.\, e
\end{array}
$$

An obvious difficulty is in ensuring that names are unique throughout a proof term. The implementation ensures that any hole created by the NEWHOLE operation has a unique name by checking against existing names in scope and modifying the name if necessary. In this paper, we will assume that all created names are fresh.

The FILL tactic, given a value v, attempts to solve the current goal with v, creating a guess binding in its place. FILL attempts to solve other holes by unifying the expected type of x with the type of v:

$$
\begin{array}{lll}
\textsc{Fill}_{\Gamma}\, v\, (?x{:}T.\, e) & = & \underline{\mathrm{do}}\ T' \leftarrow \textsc{Check}_{\Gamma}\ v \\
 & & \quad \textsc{Unify}_{\Gamma}\ T\ T' \\
 & & \quad \underline{\mathrm{return}}\ ?x{\approx}v{:}T.\, e
\end{array}
$$

For example, consider the following proof term:

$$
\begin{array}{l}
?a{:}\mathtt{Type}.\ ?k{:}\mathtt{Nat}.\ ?x{:}a.\ ?xs{:}\mathtt{Vect}\ k\ a. \\
?ys{:}\mathtt{Vect}\ (\mathtt{S}\ k)\ a.\ ys
\end{array}
$$

If x is in focus (i.e. at the head of the hole queue) and the elaborator attempts to FILL it with an `Int` value 42, we have:

- $\textsc{Check}_{\Gamma}\, 42\ =\ \mathtt{Int}$
- Unifying `Int` with A (the type of x) is only possible if $A\ =\ \mathtt{Int}$, so solve A.

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

Therefore, the resulting proof term is:

$$\begin{aligned}&?k\!:\!\mathtt{Nat}.\ ?x\!\approx\!42\!:\!\mathtt{Int}.\ ?xs\!:\!\mathtt{Vect}\ k\ \mathtt{Int}.\\ &?ys\!:\!\mathtt{Vect}\ (\mathtt{S}\ k)\ \mathtt{Int}.\ ys\end{aligned}$$

The $\textsc{Solve}$ tactic operates on a guess binding. If the guess is *pure*, i.e. it is a TT term containing no hole or guess bindings, then the value attached to the guess is substituted into its scope, and the corresponding hole is removed from the hole queue:

$$\begin{aligned}\textsc{Solve}_\Gamma\,(?x\!\approx\!v{:}T.\,e) = \underline{\text{do}}\ &\textsc{RemoveHole}\\ &\underline{\text{return}}\ e[v/x] \qquad (\text{if } \textsc{Pure}\ v)\end{aligned}$$

The two-step process, with $\textsc{Fill}$ followed by $\textsc{Solve}$, allows the elaborator to work safely with incomplete terms, since an incomplete guess binding has no computational force. Once a term is complete in a guess binding, it may be substituted into the scope of the binding safely. In each of these tactics, if any step fails, or the term in focus does not take the correct form (e.g. is not a guess in the case of $\textsc{Solve}$ or not a hole in the case of $\textsc{Claim}$ and $\textsc{Fill}$), the entire tactic fails. We can handle failure using the $\textsc{Try}$ tactic combinator:

$$\begin{aligned}\textsc{Try}_\Gamma\ t1\ t2\ t = \underline{\text{case}}\ &t1_\Gamma\ t\ \underline{\text{of}}\\ &\textsc{Success}\ t' \mapsto \textsc{Success}\ t'\\ &\textsc{Failure}\ \_ \mapsto t2_\Gamma\ t\end{aligned}$$

We have a primitive tactic $\textsc{Fail}$, which may be invoked by any tactic which encounters a failure condition (for example, an unsolvable unification problem) and is handled by $\textsc{Try}$.

4.3.3 Creating binders

We also define primitive tactics for constructing binders. Creating a λ-binding requires that the goal normalises to a function type:

$$\begin{aligned}\textsc{Lambda}_\Gamma\,n\,(?x\!:\!T.\,x) = \underline{\text{do}}\ &\forall y\!:\!S.\,T' \leftarrow \textsc{Normalise}_\Gamma\ T\\ &\underline{\text{return}}\ \lambda n\!:\!S.\ ?x\!:\!T'[n/y].\,x\end{aligned}$$

Creating a $\forall$-binding requires that the goal type converts with $\mathtt{Type}$:

$$\begin{aligned}\textsc{Pi}_\Gamma\,(n\ :\ S)\,(?x\!:\!\mathtt{Type}.\,x) = \underline{\text{do}}\ &V \leftarrow \textsc{Check}_\Gamma\ S\\ &\textsc{Convert}_\Gamma\ V\ \mathtt{Type}\\ &\underline{\text{return}}\ \forall n\!:\!S.\ ?x\!:\!\mathtt{Type}.\,x\end{aligned}$$

Creating a $\underline{\text{let}}$ binding requires a type and a value,

$$\begin{aligned}\textsc{Let}_\Gamma\,(n\ :\ S \mapsto v)\,(?x\!:\!T.\,x) = \underline{\text{do}}\ &V \leftarrow \textsc{Check}_\Gamma\ S\\ &\textsc{Convert}_\Gamma\ V\ \mathtt{Type}\\ &S' \leftarrow \textsc{Check}_\Gamma\ v\\ &\textsc{Unify}_\Gamma\ S\ S'\\ &\underline{\text{return}}\ \underline{\text{let}}\ n\ :\ S \mapsto v.\ ?x\!:\!T.\,x\end{aligned}$$

Each of these tactics requires the term in focus to be of the form $?x : T.\,x$. This is important because if the scope of the binding were an arbitrary expression e, the binder would be scoped across this *whole* expression rather than the subexpression x as intended.

The Attack tactic ensures that a hole is in the appropriate form, creating a new hole x', which is placed at the head of the queue:

$$\begin{aligned}\textsc{Attack}_{\Gamma}\,(?x\!:\!T.\,e) = \underline{\mathrm{do}}\ & \textsc{NewHole}\ x' \\ & \underline{\mathrm{return}}\ ?x \approx (?x'\!:\!T.\,x')\!:\!T.\,e\end{aligned}$$

For example, consider the following proof term:

$$?x\!:\!a \rightarrow b.\,x\ e$$

If we try to instantiate x with a λ-binding n immediately, without applying Attack first, the λ will scope over the entirety of $x\ e$, and the resulting term is not well-typed:

$$\lambda n\!:\!a.\,?x\!:\!b.\,x\ e$$

If, on the other hand, we Attack the hole, then instantiate it with a λ-binding n, we get:

$$\begin{array}{ll} ?x \approx ?x'\!:\!a \rightarrow b.\,x'\!:\!a \rightarrow b.\,x\ e & \text{(after Attack)} \\ ?x \approx \lambda n\!:\!a.\,?x'\!:\!b.\,x'\!:\!a \rightarrow b.\,x\ e & \text{(after Lambda } n\text{)} \end{array}$$

Now, once x' has been instantiated, the guess for x can safely be closed with Solve.

4.3.4 Pattern binders

Finally, we can convert a hole binding to a pattern binding by giving the pattern variable a name. This solves a hole by adding the pattern binding to the proof state, and updating the proof term with the pattern variable directly:

$$\begin{aligned}\textsc{Pat}_{\Gamma}\,n\,(?x\!:\!T.\,e) = \underline{\mathrm{do}}\ & \textsc{PatBind}\ (n\ :\ T) \\ & \underline{\mathrm{return}}\ e[n/x]\end{aligned}$$

The PatBind operation simply updates the proof state with the given pattern binding. Once we have created bindings from the left-hand side of a pattern matching definition, for example, we can retain these bindings for use when building the right-hand side.

4.3.5 Example

To illustrate how tactics are applied, using Tactic, to build a complete term by refining a proof state, let us consider the following TT definition for the identity function:

$$\begin{array}{l} \mathtt{id}\ :\ \forall A\!:\!\mathtt{Type}.\,\forall a\!:\!A.\,A \\ \mathtt{id}\ =\ \lambda A\!:\!\mathtt{Type}.\,\lambda a\!:\!A.\,a \end{array}$$

We can build `id` either as a complete term, or by applying a sequence of tactics. To achieve this, we create a proof state initialised with the type of `id` and apply a series of Lambda and Fill operations using Tactic. Note that the types on each Lambda are taken from the goal type,

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

Tactic	**Resulting Proof Term**	**Queue**
Initial state	$\underline{?x:\forall A:\mathtt{Type}.\,\forall a:A.\,A.\,x}$	$\langle x\rangle$
ATTACK	$?x\approx\underline{?h_0:\forall A:\mathtt{Type}.\,\forall a:A.\,A.\,h_0}:\forall A:\mathtt{Type}.\,\forall a:A.\,A.\,x$	$\langle h_0,x\rangle$
LAMBDA A	$?x\approx\lambda A:\mathtt{Type}.\,\underline{?h_0:\forall a:A.\,A.\,h_0}:\forall A:\mathtt{Type}.\,\forall a:A.\,A.\,x$	$\langle h_0,x\rangle$
ATTACK	$?x\approx\lambda A:\mathtt{Type}.\,?h_0\approx\underline{?h_1:\forall a:A.\,A.\,h_1}:\forall a:A.\,A.\,h_0:\forall A:\mathtt{Type}.\,\forall a:A.\,A.\,x$	$\langle h_1,h_0,x\rangle$
LAMBDA a	$?x\approx\lambda A:\mathtt{Type}.\,?h_0\approx\lambda a:A.\,\underline{?h_1:A.\,h_1}:\forall a:A.\,A.\,h_0:\forall A:\mathtt{Type}.\,\forall a:A.\,A.\,x$	$\langle h_1,h_0,x\rangle$
FILL a	$?x\approx\lambda A:\mathtt{Type}.\,?h_0\approx\lambda a:A.\,\underline{?h_1\approx a:A.\,h_1}:\forall a:A.\,A.\,h_0:\forall A:\mathtt{Type}.\,\forall a:A.\,A.\,x$	$\langle h_1,h_0,x\rangle$
SOLVE	$?x\approx\lambda A:\mathtt{Type}.\,\underline{?h_0\approx\lambda a:A.\,a:\forall a:A.\,A.\,h_0}:\forall A:\mathtt{Type}.\,\forall a:A.\,A.\,x$	$\langle h_0,x\rangle$
SOLVE	$\underline{?x\approx\lambda A:\mathtt{Type}.\,\lambda a:A.\,a:\forall A:\mathtt{Type}.\,\forall a:A.\,A.\,x}$	$\langle x\rangle$
SOLVE	$\lambda A:\mathtt{Type}.\,\lambda a:A.\,a$	$\langle\rangle$

Fig. 9. Proof state development for MKID.

```
MKID = do NEWPROOF $\forall A:\mathtt{Type}.\,\forall a:A.\,A$
          TACTIC ATTACK
          TACTIC (LAMBDA $A$)
          TACTIC ATTACK
          TACTIC (LAMBDA $a$)
          TACTIC (FILL $a$)
          TACTIC SOLVE
          TACTIC SOLVE
          TACTIC SOLVE
          TERM
```

Figure 9 shows how each step of this sequence of tactics affects the proof state and hole queue. In each step, the underlined fragment of the term is the fragment in focus. One might observe that the ATTACK/SOLVE pairs are unnecessary in this case, as the term is already in a form suitable for introduction. However, we retain these for uniform treatment of introductions.

To aid readability, we will elide TACTIC, and use a semicolon to indicate sequencing – in the implementation we use wrapper functions for each tactic to apply TACTIC. Using this convention, we can build id's type and definition as shown in Figure 10. Note that TERM retrieves the proof term from the current proof state. Both MKIDTYPE and MKID finish by returning a completed TT term. Note in particular that each tactic which introduces a guess (FILL and ATTACK) is closed with a SOLVE.

Setting up elaboration in this way, with a proof state captured in a monad, and a primitive collection of tactics, makes it easy to derive more complex tactics for elaborating higher level language constructs, in much the same way as the Ltac language in Coq (Delahaye, 2000). As a result, the description of elaboration of a language construct (or a program such as id) bears a strong resemblance to a Coq proof script.

4.4 System state

To elaborate IDRIS programs we will build expressions from high-level declarations of functions, data types and classes, and add the resulting definitions to a global system state. The system state is a tuple, (C,A,I), containing:

- a global context, C, containing pattern matching definitions and their types,

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

```
MkIdType = do NewProof Type
              Attack; Pi (A : Type); Attack; Pi (a : A)
              Fill A
              Solve; Solve; Solve
              Term

MkId = do t ← MkIdType; NewProof t
          Attack; Lambda A; Attack; Lambda a
          Fill a
          Solve; Solve; Solve
          Term
```

Fig. 10. Building `id` with tactics.

- implicit arguments, A, recording which arguments are implicit for each global name,
- class information, I, containing method signatures and dictionaries for classes.

In the implementation, the system state is captured in a monad, `Idris`, which includes additional information such as syntax overloadings, command line options and optimisations, which do not concern us here.

For each global name, A records whether its arguments are explicit, implicit or class constraints. For example, recall the declaration of `vAdd`:

```
vAdd : Num a => Vect n a -> Vect n a -> Vect n a
```

Written in full, and giving each argument an explicit name, we get the type declaration:

```
vAdd : (a : _) -> (n : _) -> (c : Num a) ->
       (xs : Vect n a) -> (ys : Vect n a) -> Vect n a
```

For `vAdd`, the state records that `a` and `n` are implicit, `c` is a constraint, and `xs` and `ys` are explicit. When the elaborator encounters an application of `vAdd`, it knows that unless these arguments are given explicitly, the application must be expanded.

4.5 Elaborating expressions

We will use a subset of full Idris, which we call Idris$^-$, to explain elaboration of expressions. Idris$^-$ expression syntax is given in Figure 11, and declaration syntax is given in Figure 12. Idris$^-$ is a subset of Idris without syntactic sugar – that is, without `do`-notation or infix operators – and with implicit arguments in types bound explicitly (e.g. {a : _} -> {n : _} -> `Vect` n a instead of simply `Vect` n a). Note that we separate implicit and type constraint bindings to ensure that they only appear in top-level declarations. It is in general straightforward to convert full Idris to Idris$^-$ – syntactic sugar is implemented by a direct source transformation, and implicit arguments can be identified as the names which are free in a type in a non-function position. Idris$^-$ differs from TT in several important respects. It has implicit syntax and classes, and functions are applied to multiple arguments rather than one at a time.

To elaborate expressions, we define meta-operations $\mathscr{E}[\![\cdot]\!]$ and $\mathscr{P}[\![\cdot]\!]$, which run relative to a proof state (see Section 4.2). $\mathscr{P}[\![\cdot]\!]$ elaborates *patterns*, i.e. expressions on the left-hand side of a pattern matching definition, whereas $\mathscr{E}[\![\cdot]\!]$ elaborates expressions on the right-hand side. Their effect is to update the proof state so that the hole in focus contains a

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

$$
\begin{array}{rlll|lll}
e, t ::= & c & & \text{(constant)} & & x & \text{(variable)} \\
| & \backslash x => e & & \text{(lambda abstraction)} & & e\,\vec{a} & \text{(function application)} \\
| & \mathtt{let}\; x = e\; \mathtt{in}\; e & & \text{(let binding)} & & \mathtt{(}x\;\mathtt{:}\;t\mathtt{)}\;\mathtt{->}\;t & \text{(explicit function space)} \\
| & \_ & & \text{(placeholder)} & & &
\end{array}
$$

$$
\begin{array}{rll}
a ::= & e & \text{(normal argument)} \\
| & \{x\;\mathtt{=}\;e\} & \text{(implicit argument with value)} \\
| & \{\{e\}\} & \text{(explicit class instance)}
\end{array}
$$

$$
\begin{array}{rll}
\mathit{declty} ::= & e & \text{(expression)} \\
| & \mathtt{(}x\;\mathtt{:}\;t\mathtt{)}\;\mathtt{->}\;\mathit{declty} & \text{(explicit function space)} \\
| & \{x\;\mathtt{:}\;t\}\;\mathtt{->}\;\mathit{declty} & \text{(implicit function space)} \\
| & \mathit{constr}\;\mathit{declty} & \text{(constrained type)}
\end{array}
$$

$$
\begin{array}{rll}
\mathit{constr} ::= & \vec{t}\;\mathtt{=>} & \text{(class constraint)}
\end{array}
$$

Fig. 11. IDRIS⁻ expressions.

$$
\begin{array}{rll}
d ::= & x\;\mathtt{:}\;\mathit{declty} & \text{(type declaration)} \\
| & \mathit{pclause} & \text{(pattern clause)} \\
| & \mathit{ddecl} & \text{(data type declaration)} \\
| & \mathit{cdecl} & \text{(class declaration)} \\
| & \mathit{idecl} & \text{(instance declaration)}
\end{array}
$$

$$
\begin{array}{rll}
\mathit{pclause} ::= & x\,\vec{t}\,[\mathtt{|}\;\vec{e}]\;\mathtt{=}\;e & [\mathtt{where}\;\vec{d}] \\
| & x\,\vec{t}\,[\mathtt{|}\;\vec{e}]\;\mathtt{with}\;e & \\
& \quad \overrightarrow{\mathit{pclause}} &
\end{array}
\qquad
\begin{array}{rl}
\mathit{ddecl} ::= & \mathtt{data}\;x\;\mathtt{:}\;\mathit{declty}\;\mathtt{where}\;\overrightarrow{\mathit{con}} \\
\mathit{con} ::= & x\;\mathtt{:}\;\mathit{declty} \\
& \\
\mathit{cdecl} ::= & \mathtt{class}\;[\mathit{constr}]\;x\;(\vec{x}\;\mathtt{:}\;\vec{t})\;\mathtt{where}\;\vec{d} \\
\mathit{idecl} ::= & \mathtt{instance}\;[\mathit{constr}]\;x\,\vec{t}\;\mathtt{where}\;\vec{d}
\end{array}
$$

Fig. 12. IDRIS⁻ declarations.

representation of the given expression by applying tactics. We assume that the proof state has already been set up, which means that elaboration can always be *type-directed* since the proof state contains the type of the expression we are building.

In all cases except the elaboration of variables and constants, $\mathscr{P}[\![x]\!] = \mathscr{E}[\![x]\!]$.

4.5.1 Elaborating variables and constants

In the simplest cases, there is a direct translation from an IDRIS⁻ expression to a TT expression – we build TT representations of variables and constants using the FILL tactic:

$$
\begin{array}{lcl}
\mathscr{E}[\![x]\!] & = & \underline{\mathrm{do}}\;\textsc{Fill}\;x;\;\textsc{Solve} \\
\mathscr{E}[\![c]\!] & = & \underline{\mathrm{do}}\;\textsc{Fill}\;c;\;\textsc{Solve}
\end{array}
$$

Since FILL creates a guess binding, we use SOLVE to instantiate the guess. We need not concern ourselves with type checking variables or constants here – FILL will handle this, type checking x or c and unifying the result with the hole type. If there are any errors, elaboration will fail.

If we are building the left-hand side of a pattern clause, however, there is a problem, as it is the left-hand side which *defines* variables. In this context, we assume that attempting to elaborate a variable which does not type check means that the variable is a pattern variable:

$$\begin{array}{lcl} \mathscr{P}\llbracket x \rrbracket & = & \textsc{Try}\ (\underline{\mathrm{do}}\ \textsc{Fill}\ x;\ \textsc{Solve}) \\ & & \phantom{\textsc{Try}\ }(\textsc{Pat}\ x) \end{array}$$

In Haskell, for comparison, pattern variables are all lower case, so type checking can assume that a lower case name in a pattern refers to a variable. In IDRIS, however, a name in a pattern may refer to a function if forced by some dependency.

We also need to elaborate *placeholders*, which are subexpressions we expect to solve by unification. In this case, we simply move on to the next hole in the queue, moving the current hole to the end of the queue with UNFOCUS:

$$\mathscr{E}\llbracket \_ \rrbracket \ = \ \textsc{Unfocus}$$

On encountering a placeholder, our assumption is that unification will eventually solve the hole. At the end of elaboration, any holes remaining unsolved on the left-hand side become pattern variables. If there are any unsolved on the right-hand side, elaboration is incomplete and an error will be reported when the resulting term is type checked.

4.5.2 Elaborating bindings

To elaborate a λ-binding, we ATTACK the hole, then apply the LAMBDA tactic, which will fail if the goal is not a function type. We then elaborate the scope and SOLVE, which discharges the ATTACK:

$$\begin{array}{lcl} \mathscr{E}\llbracket \backslash\ x = > e \rrbracket & = & \underline{\mathrm{do}}\ \textsc{Attack};\ \textsc{Lambda}\ x \\ & & \phantom{\underline{\mathrm{do}}\ }\mathscr{E}\llbracket e \rrbracket \\ & & \phantom{\underline{\mathrm{do}}\ }\textsc{Solve} \end{array}$$

Note that there is no type on the λ-binding in IDRIS^{-}. There is no need – since elaboration is type directed, the LAMBDA tactic finds the type of the binding by looking at the type of the hole. In full IDRIS, types are allowed on bindings, and the elaborator merely checks that the given type is equivalent to the inferred type.

Elaborating a function type is more tricky, since we have to elaborate the argument type (itself an IDRIS^{-} expression), then elaborate the scope. To achieve this, we create a new goal X for the argument type t, where X is a fresh name, and introduce a function binding with argument type X. We can then focus on X, and elaborate it with the expression t. Finally, we elaborate the scope of the binding.

$$\begin{array}{lcl} \mathscr{E}\llbracket (x\ :\ t)\ \texttt{->}\ e \rrbracket & = & \underline{\mathrm{do}}\ \textsc{Attack};\ \textsc{Claim}\ (X\ :\ \texttt{Type}) \\ & & \phantom{\underline{\mathrm{do}}\ }\textsc{Pi}\ (x\ :\ X) \\ & & \phantom{\underline{\mathrm{do}}\ }\textsc{Focus}\ X \\ & & \phantom{\underline{\mathrm{do}}\ }\mathscr{E}\llbracket t \rrbracket;\ \mathscr{E}\llbracket e \rrbracket \\ & & \phantom{\underline{\mathrm{do}}\ }\textsc{Solve} \end{array}$$

Elaborating t will involve solving unification problems, which will, if the program is type correct, solve X. After focussing on X and elaborating t, there is no need to refocus on the hole representing the scope, as it was previously at the head of the hole queue before focussing on X. Elaboration of implicit and constraint argument types is exactly the same – TT makes no distinction between then.

To elaborate a `let` binding, we take a similar approach, creating new subgoals for the `let`-bound value and its type, then elaborating the scope. Again, if elaborating the scope is successful, unification will solve the claimed variables V and X.

$$\begin{array}{lcl}\mathcal{E}[\![\texttt{let}\ x = v\ \texttt{in}\ e]\!] & = & \underline{\text{do}}\ \textsc{Attack};\ \textsc{Claim}\ (X\ :\ \texttt{Type});\ \textsc{Claim}\ (V\ :\ X) \\ & & \quad\textsc{Let}\ (x\ :\ X \mapsto V) \\ & & \quad\textsc{Focus}\ V \\ & & \quad\mathcal{E}[\![v]\!];\ \mathcal{E}[\![e]\!] \\ & & \quad\textsc{Solve}\end{array}$$

4.5.3 Elaborating applications

There are two cases to consider when elaborating applications:

- Applying a global function name to arguments, some of which may be implicit.
- Applying an expression, which does not have implicit arguments, to an argument.

In the first case, the elaborator must expand the application to include implicit arguments. For example, `vAdd xs ys` is expanded to `vAdd {a=_} {n=_} {{_}} xs ys`, adding the implicit arguments `a` and `n` and a class instance argument. The meta-operation $\textsc{Expand}\ x\ \vec{a}$, given a global function name x and the arguments supplied by the programmer $\vec{a}$, returns a new argument list $\vec{a}'$ with implicit and class arguments added. Each value in $\vec{a}'$ is paired with an explicit name for the argument.

Implicit arguments are solved by unification – the type or value of another argument determines the value of an implicit argument, with the appeal to $\textsc{Unify}$ in the $\textsc{Fill}$ tactic solving as many extra holes as it can. However, unification problems can take several forms. For example, assuming f, g, x are holes, we might have unification problems of the following forms:

$$\begin{array}{l}\textsc{Unify}_{\Gamma}\, f\ \texttt{Int} \\ \textsc{Unify}_{\Gamma}\ (f\ x)\ \texttt{Int} \\ \textsc{Unify}_{\Gamma}\ (f\ x)\ (g\ x)\end{array}$$

The first problem has a solution: $f = \texttt{Int}$. The second and third problems have no solution without further information. In the second case, we cannot conclude anything about f or x just from knowing that $f\ x\ =\ \texttt{Int}$. In the third case, although we have $x\ =\ x$ in argument position, we cannot conclude that $f\ =\ g$ in general as a result.

Once f or g is solved in some other way, perhaps by being given explicitly in another argument, these unification problems can make progress. Otherwise, unification blocks. In such a case, the elaborator stores the problem (i.e. the expressions to be unified with their local context) and refines it with further information when it becomes available. Elaborating a global function application makes a $\textsc{Claim}$ for each argument, in turn. Then the function is elaborated, and applied to each of the claimed arguments. Each argument which has been given explicitly is elaborated. Finally, any class instance arguments are resolved with the built-in $\textsc{Instance}$ tactic,

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

$$\begin{aligned}
\mathscr{E}[\![x\,\vec{a}]\!] \;=\; \underline{\text{do}}\ & (\vec{n},\vec{v}) \leftarrow \textsc{Expand}\ x\,\vec{a} \\
& \overrightarrow{\textsc{Claim}}\ (\vec{n}\ :\ \vec{T}) \\
& \textsc{Fill}\ x\,\vec{n} \\
& \overrightarrow{\textsc{ElabArg}}\ \vec{n}\,\vec{v} \qquad \text{(for non-placeholder } v\text{)} \\
& \textsc{Solve} \\
& \overrightarrow{\textsc{Instance}}\ \vec{n} \qquad \text{(for class constraint argument } n\text{)}
\end{aligned}$$

$$\textsc{ElabArg}\ n\,v \;=\; \textsc{Focus}\ n;\ \mathscr{E}[\![v]\!]$$

The INSTANCE tactic focuses on the given hole and searches the context for a class instance which would solve the goal directly. Firstly, it examines the local context, then recursively searches the global context. INSTANCE is covered in detail in Section 4.6.6.

To elaborate a simple function application, of an arbitrary expression to an arbitrary argument (possibly with a dependent-type), we need not worry about implicit arguments or class constraints. Instead, the function and argument are elaborated, and the results are applied. Since elaboration is type directed, however, there must be an appropriate type for the function,

$$\begin{aligned}
\mathscr{E}[\![e\ a]\!] \;=\; \underline{\text{do}}\ & \textsc{Claim}\ (A\ :\ \mathtt{Type});\ \textsc{Claim}\ (B\ :\ \mathtt{Type}) \\
& \textsc{Claim}\ (f\ :\ A \rightarrow B);\ \textsc{Claim}\ (s\ :\ A) \\
& \textsc{Focus}\ f;\ \mathscr{E}[\![e]\!] \\
& \textsc{Focus}\ s;\ \mathscr{E}[\![a]\!]
\end{aligned}$$

4.6 Elaborating declarations

To elaborate declarations, we define a meta-operation ELAB, which runs relative to the system state (see Section 4.4) and has access to the current proof state. ELAB uses $\mathscr{E}[\![\cdot]\!]$ to help translate IDRIS^{-} type, function and class declarations into TT declarations.

4.6.1 Elaborating type declarations

Elaborating a type declaration involves creating a new proof state, translating the IDRIS^{-} type to a TT type, then adding the resulting type as a TT declaration:

$$\begin{aligned}
\textsc{Elab}\ (x\ :\ t) \;=\; \underline{\text{do}}\ & \textsc{NewProof}\ \mathtt{Type} \\
& \mathscr{E}[\![t]\!] \\
& t' \leftarrow \textsc{Term} \\
& \textsc{TTDecl}\ (x\ :\ t')
\end{aligned}$$

The final TTDECL takes the result of elaboration, type checks it and adds it to the global context if type checking succeeds. This final type check ensures that the elaboration process does not allow any ill-typed terms to creep into the context.

4.6.2 Elaborating data types

Elaborating a data declaration involves elaborating the type declaration itself, as a normal type declaration. This ensures that the type is in scope when elaborating the constructor types. Then using the results, the elaborator adds a TT data declaration,

$$\begin{array}{l}
\textsc{Elab}\ (\texttt{data}\,x\ :\ t\,\texttt{where}\,\vec{c}) \\
\quad = \underline{\text{do}}\ \textsc{NewProof}\ \texttt{Type} \\
\qquad\quad \mathscr{E}[\![t]\!] \\
\qquad\quad t' \leftarrow \textsc{Term} \\
\qquad\quad \textsc{TTDecl}\ (x\ :\ t') \\
\qquad\quad \vec{c}\,' \leftarrow \textsc{Elab}\vec{\textsc{C}}\textsc{on}\ \vec{c} \\
\qquad\quad \textsc{TTDecl}\ (\underline{\text{data}}\,x\ :\ t'\ \underline{\text{where}}\,\vec{c}\,')
\end{array}$$

$$\begin{array}{l}
\textsc{ElabCon}\ (x\ :\ t) \\
\quad = \underline{\text{do}}\ \textsc{NewProof}\ \texttt{Type} \\
\qquad\quad \mathscr{E}[\![t]\!] \\
\qquad\quad t' \leftarrow \textsc{Term} \\
\qquad\quad \underline{\text{return}}\ (x\ :\ t')
\end{array}$$

4.6.3 Elaborating pattern matching

Elaborating a pattern matching definition works clause by clause, elaborating the left- and right-hand sides in turn. $\textsc{ElabClause}$ returns the elaborated left- and right-hand sides, and may have the side effect of adding entries to the global context, such as definitions in `where` clauses. Firstly, let us consider the simplest case, with no `where` clause:

$$\textsc{ElabClause}\ (x\,\vec{t}\ =\ e)\ =\ \ldots$$

How does the left-hand side get elaborated, given that elaboration is type directed, and its type is not known until after elaboration? This can be achieved without any change to the elaborator by defining a type `Infer`:

$$\underline{\text{data}}\quad \texttt{Infer}\ :\ \texttt{Type}\quad \underline{\text{where}}\quad \texttt{MkInfer}\ :\ \forall a{:}\texttt{Type}.\,a \rightarrow \texttt{Infer}$$

Now elaboration of the left-hand side proceeds by elaborating $\texttt{MkInfer}\ \_\ (x\,\vec{t})$ and extracting the value and unified type when complete.

$$\begin{array}{l}
\textsc{ElabClause}\ (x\,\vec{t}\ =\ e)\ = \\
\qquad \underline{\text{do}}\ \textsc{NewProof}\ \texttt{Infer};\ \mathscr{P}[\![\texttt{MkInfer}\ \_\ (x\,\vec{t})]\!] \\
\qquad\quad \texttt{MkInfer}\ T\ \mathit{lhs} \leftarrow \textsc{Term} \\
\qquad\quad \vec{p} \leftarrow \textsc{Patterns} \\
\qquad\quad \textsc{NewTerm}\ T;\ \mathscr{E}[\![e]\!] \\
\qquad\quad \mathit{rhs} \leftarrow \textsc{Term} \\
\qquad\quad \underline{\text{return}}\ (\underline{\text{var}}\,\vec{p}.\,\mathit{lhs}\ =\ \mathit{rhs})
\end{array}$$

This infers a type for the left-hand side, creating pattern variable bindings. Then it elaborates the right-hand side using the inferred type of the left-hand side and the inferred pattern bindings, which are retrieved from the state with the $\textsc{Patterns}$ operation. Finally, it returns a pattern clause in TT form. Elaborating a collection of pattern clauses then proceeds by mapping $\textsc{ElabClause}$ over the clauses and adding the resulting collection to TT

$$\begin{array}{l}
\textsc{Elab}\ \overrightarrow{\mathit{pclause}}\ =\ \underline{\text{do}}\ \vec{c} \leftarrow \textsc{Elab}\vec{\textsc{C}}\textsc{lause}\ \overrightarrow{\mathit{pclause}} \\
\qquad\qquad\qquad\qquad\ \ \textsc{TT}\vec{\textsc{D}}\textsc{ecl}\ \vec{c}
\end{array}$$

Elaborating a clause is made only slightly more complex by the presence of `where` clauses. In this case, after elaborating the left-hand side, the pattern-bound variables are added as extra arguments to the declarations in the `where` block, which are then recursively elaborated before the right-hand side

```
ElabClause $(x\,\vec{t} \;=\; e$ where $\vec{d}) \;=$
    do NewProof Infer; $\mathscr{P}[\![$MkInfer _ $(x\,\vec{t})]\!]$
       MkInfer $T$ $lhs \leftarrow$ Term
       $\vec{p} \leftarrow$ Patterns
       $(\vec{d'}, e') \leftarrow$ Lift $\vec{p}\;\vec{d}\;e$
       Elab $\vec{d'}$
       NewTerm $T$; $\mathscr{E}[\![e']\!]$
       $rhs \leftarrow$ Term
       return (var $\vec{p}.\,lhs \;=\; rhs$)
```

In TT, all definitions must be at the top level. Therefore, the declarations in the `where` block are lifted out, adding the pattern-bound names as additional arguments to ensure they are in scope, using the Lift operation. This also modifies the right-hand side e to use the lifted definitions, rather than the original.

4.6.4 Elaborating the `with` rule

The `with` rule allows *dependent* pattern matching on intermediate values. Translating this into TT involves constructing an auxiliary top-level definition for the intermediate pattern match. For example, recall `natToBin`:

```
natToBin : Nat -> List Bool
natToBin Z = Nil
natToBin k with (parity k)
   natToBin (j + j)     | even = False :: natToBin j
   natToBin (S (j + j)) | odd  = True  :: natToBin j
```

This is equivalent to the following, using top-level definitions only:

```
natToBin : Nat -> List Bool
natToBin Z = Nil
natToBin k = natToBin’ k (parity k)

natToBin’ : (n : Nat) -> Parity n -> List Bool
natToBin’ (j + j)     even = False :: natToBin j
natToBin’ (S (j + j)) odd  = True  :: natToBin j
```

Additionally, the elaborator abstracts over the term being analysed in the type of the generated auxiliary definition and reorders the arguments as required. For example:

```
f : Nat -> Bool

ftest : (n : Nat) -> so (f n) -> Nat
ftest n x with (f n)
     | True  = ...
     | False = ...
```

Here we may require the knowledge that `f n` is `True` or `False` in the appropriate case. The elaborator abstracts over `f n` in the auxiliary definition, specialising the type of `x` in each case:

```
ftest : (n : Nat) -> so (f n) -> Nat
ftest n x = ftest' n (f n) x

ftest' : (n : Nat) -> (p : Bool) -> so p -> Nat
ftest' n True  x = ...
ftest' n False x = ...
```

Elaboration of `with` builds the type declaration and auxiliary function automatically. The elaborator checks the left-hand side, resulting in a set of patterns $\vec{p}$, then infers a type W for the expression w. The patterns $\vec{p}$ are then split into those necessary to type W, and the rest ($\vec{p}_w$ and $\vec{p}_n$ respectively) using $\textsc{SplitContext}$. Then it builds a new declaration, abstracting over the matched expression w in the types of $\vec{p}_n$. $\textsc{Abstract}\ e\ w\ p$ simply replaces any occurrence of w in p with the name e,

```
$\textsc{ElabClause}\ (x\ \vec{t}\ \texttt{with}\ w\ \overrightarrow{pclause})\ =$
        $\underline{\text{do}}\ \textsc{NewProof}\ \texttt{Infer};\ \mathscr{P}[\![\texttt{MkInfer}\ \_\ (x\ \vec{t})]\!]$
          $\texttt{MkInfer}\ T\ lhs \leftarrow \textsc{Term}$
          $\vec{p} \leftarrow \textsc{Patterns}$
          $\textsc{NewTerm}\ \texttt{Infer};\ \mathscr{E}[\![\texttt{MkInfer}\ \_\ w]\!]$
          $\texttt{MkInfer}\ W\ w' \leftarrow \textsc{Term}$
          $(\vec{p}_w, \vec{p}_n)\ \leftarrow\ \textsc{SplitContext}\ W\ \vec{p}$
          $\textsc{TTDecl}\ (x'\ :\ \vec{p}_w \rightarrow (e\ :\ W) \rightarrow \overrightarrow{\textsc{Abstract}}\ e\ w'\ \vec{p}_n \rightarrow T)$
          $\vec{c}' \leftarrow \overrightarrow{\textsc{MatchWith}}\ \vec{p}\ \vec{t}\ \overrightarrow{pclause}$
          $\overrightarrow{\textsc{Elab}}\ \vec{c}'$
          $\underline{\text{return}}\ (\underline{\text{var}}\ \vec{p}.\ lhs\ =\ x'\ \vec{p}_w\ w'\ \vec{p}_n)$

$\textsc{MatchWith}\ \vec{p}_w\ \vec{p}_n\ \vec{t}\ (x\ \vec{w}\ \mid\ e\ rhs)\ =$
        $\underline{\text{do}}\ \vec{p}' \leftarrow \overrightarrow{\textsc{Match}}\ \vec{w}\ \vec{t}$
          $(\vec{p}'_w, \vec{p}'_n)\ \leftarrow\ \textsc{SplitPats}\ \vec{p}_w\ \vec{p}_n\ \vec{p}'$
          $\underline{\text{return}}\ (x'\ \vec{p}'_w\ e\ \vec{p}'_n\ rhs)$
```

A `with` block in a clause for x results in an auxiliary definition x', where x' is a fresh name, which takes an extra argument corresponding to the intermediate result which is to be matched. $\textsc{MatchWith}$ matches the left-hand side of the top-level definition against each *pclause*, using the result of the match to build each pattern clause for x'. The clauses in the block must be of a specific form: they must have an initial set of arguments $\vec{w}$ which matches the outer clause arguments $\vec{t}$, and an extra argument e which is the same type as the scrutinee of the `with`, W. The right-hand side, *rhs*, may either return a value directly, containing `where` clauses, or be a nested `with` block. In any case, it remains unchanged. $\textsc{Match}\ w\ t$ returns a list of pattern bindings containing the variables in t and their matches in w. The resulting matches are split, using $\textsc{SplitPats}$, into matches for those variables necessary to type W and the rest as before.

4.6.5 Elaborating class and instance declarations

Classes are implemented as dictionaries of functions for each instance, following the implementation of first class type classes in Coq (Sozeau & Oury, 2008). Therefore, a class declaration elaborates to a record type containing all of the functions in the class, and an instance is simply an instance of the record. The methods are translated to top-level functions which extract the relevant function from the dictionary. For example, for the `Show` class, the `class` declaration is translated (in the elaborated code, using the surface IDRIS syntax for readability) to:

```
data Show : Type -> Type where
    InstanceShow : (show : a -> String) -> Show a

show : Show a -> a -> String
show (InstanceShow show') = show'
```

An instance for a type `a` is then a value of type `Show a`; for example, for Nat:

```
showNat : Show Nat
showNat = InstanceShow show' where
    show' : Nat -> String
    show' Z = "Z"
    show' (S k) = "s" ++ show k
```

The call to `show` in the body of the instance uses the top-level `show` function, with the instance resolved by the elaborator. Where instances have constraints, these constraints are passed on to the function declaration for the instance. For example, for `Show (List a)`:

```
showList : Show a => Show (List a)
showList = InstanceShow show' where
    show' : List a -> String
    show' []        = "[]"
    show' (x :: xs) = show x ++ " :: " ++ show xs
```

Where classes have constraints, the constraints are again passed on to the data type declaration, and resolved by the elaborator. For example, for `Ord`:

```
class Eq a => Ord a where
    compare : a -> a -> Ordering
    max : a -> a -> a
    -- remaining methods elided
```

This translates to:

```
data Ord : Type -> Type where
    InstanceOrd : Eq a => (compare : a -> a -> Ordering) ->
                          (max : a -> a -> a) ->  Ord a

compare : Ord a -> a -> a -> Ordering
compare (InstanceOrd compare' max') = compare'

max : Ord a -> a -> a -> a
max (InstanceOrd compare' max') = max'
```

The elaborator also adds functions to retrieve parent classes from a dictionary, to assist with class resolution. In this case:

```
OrdEq : Ord a -> Eq a
OrdEq (InstanceOrd {{eq}} compare' max') = eq
```

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

In general, `class` declarations are elaborated as follows:

$$
\begin{array}{l}
\textsc{Elab}\ (\texttt{class}\,\vec{c}\ \texttt{=>}\ \texttt{C}\,(\vec{a}\ :\ \vec{t})\ \texttt{where}\,\vec{d})\\
\qquad = \underline{\text{do}}\ \textsc{Elab}\ (\underline{\text{data}}\ \texttt{C}\ :\ (\vec{a}\ :\ \vec{t}) \to \texttt{Type}\ \texttt{where}\\
\qquad\qquad\qquad\qquad \texttt{InstanceC}\ :\ \vec{c}\ \texttt{=>}\ \vec{d} \to \texttt{C}\,\vec{a})\\
\qquad\qquad\quad \textsc{Elab}\vec{\textsc{M}}\textsc{eth}\,\vec{d}\\
\qquad\qquad\quad \textsc{Elab}\vec{\textsc{P}}\textsc{arent}\,\vec{c}
\end{array}
$$

$$
\begin{array}{l}
\textsc{ElabMeth}\ (f\ :\ t)\\
\qquad = \underline{\text{do}}\ \textsc{Elab}\ (f\ :\ \texttt{C}\,\vec{a} \to t)\\
\qquad\qquad\quad \textsc{Elab}\ (f\ (\texttt{InstanceC}\,\overrightarrow{meth})\ =\ \overrightarrow{meth}[f])
\end{array}
$$

$$
\begin{array}{l}
\textsc{ElabParent}\ c\\
\qquad = \underline{\text{do}}\ \textsc{Elab}\ (cC\ :\ \texttt{C}\,\vec{a} \to c)\\
\qquad\qquad\quad \textsc{Elab}\ (cC\ (\texttt{InstanceC}\,\{\{c'\}\}\,\overrightarrow{meth})\ =\ c')
\end{array}
$$

Then `instance` declarations are elaborated as follows. One complication is that the function definitions in an `instance` declaration do not have explicit types, so we must retrieve the method types from the system state and substitute $\texttt{C}\,\vec{t}$, using $\textsc{ExpandTypes}$. The name `instanceC` is a fresh name generated for the newly defined instance:

$$
\begin{array}{l}
\textsc{Elab}\ (\texttt{instance}\,\vec{c}\ \texttt{=>}\ \texttt{C}\,\vec{t}\,\texttt{where}\,\vec{d})\\
\qquad = \underline{\text{do}}\ (\overrightarrow{meth}, \overrightarrow{ty}) \leftarrow \textsc{ExpandTypes}\,\vec{d}\\
\qquad\qquad\quad \overrightarrow{ty'}\ =\ \overrightarrow{ty}[\vec{t}/\vec{a}]\\
\qquad\qquad\quad \textsc{Elab}\ (\texttt{instanceC}\ :\ \vec{c}\ \texttt{=>}\ \texttt{C}\,\vec{t})\\
\qquad\qquad\quad \textsc{Elab}\ (\texttt{instanceC}\ =\ \texttt{InstanceC}\,\overrightarrow{meth'}\ \texttt{where}\\
\qquad\qquad\qquad\qquad \overrightarrow{meth'}\ :\ \overrightarrow{ty'}\\
\qquad\qquad\qquad\qquad \vec{d})
\end{array}
$$

Adding default definitions is straightforward: simply insert the default definition where there is a method missing in an `instance` declaration.

Here we have added a higher level language construct (classes) simply by elaborating in terms of lower level language constructs (data types and functions). We achieve type class resolution by implementing a tactic, Instance. We can build several extensions this way because we have effectively built, bottom up, an Embedded Domain Specific Language for constructing programs in TT.

4.6.6 The Instance *tactic*

When elaborating applications, recall that implicit arguments are filled in by unification as described in Section 4.5.3, and class constraints by an Instance tactic, which searches the context for an instance of the required class.

The Instance tactic begins by trying to find a solution in the local context, using the Trivial tactic, which attempts to solve the current goal with each local variable in turn. It is used as a very basic primitive for proof search, and is defined as follows:

$$\begin{aligned}\textsc{Trivial} \;=\; & \underline{\mathrm{do}}\;(\vec{v} : \vec{t}) \leftarrow \textsc{Context} \\ & \quad \textsc{TryAll}\;\vec{v}\end{aligned}$$

$$\begin{aligned}\textsc{TryAll}\;(v_1, v_2 \ldots) \;&=\; \textsc{Try}\;(\textsc{Fill}\;v_1)\;(\textsc{TryAll}\;v_2 \ldots) \\ \textsc{TryAll}\;\langle\rangle \;&=\; \textsc{Fail}\end{aligned}$$

For example, in the following function, the class constraint required for the > operator is resolved by finding an instance for `Ord a` in the local context:

```
max : Ord a => a -> a -> a
max x y = if (x > y) then x else y
```

If TRIVIAL does not find a solution, INSTANCE begins a global search, applying every class instance and attempting to resolve the instance's parameters recursively. Given an instance function `instanceC` with parameters of type $\vec{t}$:

$$\begin{aligned}\textsc{TryInstance}\;\texttt{instanceC} \;=\; & \underline{\mathrm{do}}\;\overrightarrow{\textsc{Claim}}\;(\vec{x} : \vec{t}) \\ & \quad \mathscr{E}[\![\texttt{instanceC};\vec{x}]\!] \\ & \quad \overrightarrow{\textsc{Focus}}\;\vec{x};\;\overrightarrow{\textsc{Instance}}\end{aligned}$$

We define INSTANCE as follows, using ALLINSTANCES to retrieve all functions which define instances:

$$\begin{aligned}\textsc{Instance} \;=\; & \textsc{Try}\;\textsc{Trivial} \\ & \quad (\underline{\mathrm{do}}\;\vec{i} \leftarrow \textsc{AllInstances} \\ & \quad\quad \overrightarrow{\textsc{TryInstance}}\;\vec{i})\end{aligned}$$

In practice, for efficiency and to ensure that resolution terminates, we constrain the search by recording which instances are defined for each class, and by ensuring that a recursive call of INSTANCE is either searching for a different class, or a structurally smaller instance of the current class.

For example, in the following function, the constraints required by the `show` function are resolved by a global search which locates a `Show` instance for `List a`, which in turn requires a `Show` instance for `a`. In this case `a` is instantiated by `Int`, so resolution finishes by locating a `Show` instance for `Int`:

```
main : IO ()
main = putStrLn (show [1,2,3,4])
```

4.7 Syntax extensions

We now have a complete elaborator for IDRIS$^-$, covering dependent pattern matching definitions and expansion of implicit arguments and classes. In order to make the language truly general purpose, however, we will need to add higher level extensions. Full IDRIS is IDRIS$^-$ extended with `do`-notation, idiom brackets (McBride & Paterson, 2008), `case` expressions, pattern matching `let`, metavariables and tactic-based theorem proving. The majority of these extensions are straightforward transformations of high-level IDRIS programs – for example, `do`-notation can be reduced directly to IDRIS$^-$ function applications. Some, however, require further support. In this section, we complete the presentation of IDRIS$^-$ by extending it with metavariables and `case` expressions:

$e ::= \ldots$

$\mid\ ?x$ (metavariable) $\mid$ `case` e `of` $\vec{alt}$ (case expression)

$alt ::= e$ `=>` e (case alternative)

4.7.1 Metavariables

Metavariables are terms which stand for incomplete programs. A metavariable serves a similar purpose to a hole binding in TT, but gives rise to a global rather than a local variable. For example, the following is an incomplete definition of the vector append function, containing a metavariable append_rec:

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
Nil       ++ ys = ys
(x :: xs) ++ ys = x :: ?append_rec
```

This generates a new (as yet undefined) function append_rec, which takes all of the variables in scope at the point it is used, and returns a value of the required type. We can inspect the type of append_rec at the Idris prompt:

```
*vec> :t append_rec
append_rec : (a : Type) -> (m : Nat) -> (n : Nat) -> a ->
             Vect n a -> Vect m a -> Vect (n + m) a
```

Metavariables serve two purposes: Firstly, they aid type-directed program development by allowing a programmer direct access to the inferred types of local variables, and the types of subexpressions. Secondly, they allow a separation of program structure from proof details. Metavariables can be solved either by directly editing program source, or by providing a definition elsewhere in the file. For example, we can later say:

```
append_rec a m n x xs ys = app xs ys
```

Elaborating a metavariable involves adding a new top-level definition which is applied to all of the variables in local scope.

$$\begin{aligned}
\mathscr{E}[\![?x]\!] \;=\; & \underline{\mathrm{do}}\ (\vec{v} : \vec{t}) \leftarrow \textsc{Context} \\
& T \leftarrow \textsc{Type} \\
& \mathrm{TT}\textsc{Decl}\ (x\ :\ (\vec{v} : \vec{t}) \rightarrow T) \\
& \mathscr{E}[\![x\ \vec{v}]\!]
\end{aligned}$$

4.7.2 `case` expressions

A `case` expression allows pattern matching on intermediate values. The difficulty in elaborating `case` expressions is that TT allows matching only on *top-level* values. Elaborating a `case` expression, therefore, involves creating a new top-level function standing for the expression, and applying it. The natural way to implement this is to use a metavariable. For example, we have already seen lookup_default:

```
lookup_default : Nat -> List a -> a -> a
lookup_default i xs def = case list_lookup i xs of
                               Nothing => def
                               Just x => x
```

This elaborates as follows:

```
lookup_default : Nat -> List a -> a -> a
lookup_default i xs def =
   let scrutinee = list_lookup i xs in ?lookup_default_case

lookup_default_case i xs def Nothing  = def
lookup_default_case i xs def (Just x) = x
```

To elaborate a case expression involves let binding the scrutinee of the case expression, introducing a new name *scrutinee*, and creating a metavariable *xcase* to implement the pattern matching. Then the elaborator builds a new pattern matching definition for *xcase* and elaborates it:

$$\begin{aligned}
\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ \vec{alt}]\!] &= \underline{\mathrm{do}}\ \mathcal{E}[\![\texttt{let}\ scrutinee = e\ \texttt{in}\ ?xcase]\!] \\
&\qquad (\vec{v}\ :\ \vec{t}) \leftarrow \textsc{Context} \\
&\qquad \textsc{Mk}\vec{\textsc{C}}\textsc{ase}\ \vec{v}\ alt \\
\textsc{MkCase}\ \vec{v}\ (l \;\texttt{=>}\; r) &= \textsc{Elab}\ (xcase\ \vec{v}\ l\ =\ r)
\end{aligned}$$

Any nested case expressions will be handled by the recursive call to ELAB. Again, because of the way we have set up an Embedded Domain Specific Language for describing elaboration, we have been able to implement a new higher level language feature in terms of elaboration of lower level language features.

5 Reversing elaboration

As well as translating from IDRIS to TT so that programs can be type checked and evaluated, it is valuable to define the reverse transformation. This serves two principal purposes:

- To assist the user, it is preferable that the results of evaluation, and any error messages produced by the elaborator, are presented in IDRIS syntax rather than TT.
- For correctness, we would like to ensure as far as possible that the result of elaboration is equivalent to the original program. Informally, we can achieve this by checking that reversing the elaboration process yields the original program (with implicit arguments expanded).

In this section, we describe the process for reversing elaboration and the required properties of the elaboration process as a whole. Fortunately, translating from TT to IDRIS is significantly easier than IDRIS to TT, because it is primarily *erasing* information.

5.1 From TT *to* IDRIS

We define a meta-operation $\mathcal{T}[\![t]\!]$, which converts a TT expression t to an IDRIS expression which would elaborate to t:

$$
\begin{aligned}
\mathscr{T}\llbracket x \rrbracket &= x \\
\mathscr{T}\llbracket c \rrbracket &= c \\
\mathscr{T}\llbracket x\,\vec{a} \rrbracket &= x\,(\overrightarrow{\textsc{Impl}}\,x\,\vec{a}) \\
\mathscr{T}\llbracket f\,a \rrbracket &= \mathscr{T}\llbracket f \rrbracket\,\mathscr{T}\llbracket a \rrbracket \\
\mathscr{T}\llbracket \lambda x{:}T.\,e \rrbracket &= \backslash\, x \mathrel{=>} \mathscr{T}\llbracket e \rrbracket \\
\mathscr{T}\llbracket \forall x{:}T.\,e \rrbracket &= \texttt{(}x\ \texttt{:}\ \mathscr{T}\llbracket T \rrbracket\texttt{)}\ \texttt{->}\ \mathscr{T}\llbracket e \rrbracket \\
\mathscr{T}\llbracket \underline{\mathrm{let}}\,x \mapsto t\ :\ T.\,e \rrbracket &= \texttt{let}\,x = \mathscr{T}\llbracket t \rrbracket\ \texttt{in}\,\mathscr{T}\llbracket e \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
\textsc{Impl}\,x\,a_i = {} & \{n\ \texttt{=}\ \mathscr{E}\llbracket a_i \rrbracket\} && \text{(if the } i\text{th argument to } x \text{ is implicit argument } n\text{)} \\
& \{\{\mathscr{E}\llbracket a_i \rrbracket\}\} && \text{(if the } i\text{th argument to } x \text{ is a constraint argument)} \\
& \mathscr{E}\llbracket a_i \rrbracket && \text{(otherwise)}
\end{aligned}
$$

This is mostly a straightforward traversal of the TT expression, translating directly to an Idris equivalent. The interesting case is for applications of named functions, $\mathscr{T}\llbracket x\,\vec{a} \rrbracket$, where the arguments are translated to either implicit, constraint or explicit arguments according to the definition of x. Since only type declarations are allowed to have implicit or constraint arguments, and $\mathscr{T}\llbracket\cdot\rrbracket$ translates *expressions*, all function types are assumed to take explicit arguments.

We also define an operation Unelab, which translates TT declarations to corresponding Idris declarations. This generates data declarations and pattern matching definitions only – it makes no attempt to reconstruct `class` or `instance` declarations, or rebuild `case` expressions. Firstly, we define the reverse elaboration of type declarations, which must reconstruct which arguments are implicit or constraint arguments:

$$
\textsc{UnelabType}\,(x\ :\ t) = x\ :\ \textsc{UnelabTyDecl}\,0\,t
$$

$$
\begin{aligned}
\textsc{UnelabTyDecl}\,i\,(\forall y{:}T.\,e) = {} & \{y\ :\ \mathscr{T}\llbracket T \rrbracket\}\ \texttt{->}\ (\textsc{UnelabTyDecl}\,(i+1)\,e) \\
& \quad\text{(if the } i\text{th argument to } x \text{ is an implicit argument)} \\
& \mathscr{T}\llbracket T \rrbracket\ \texttt{=>}\ (\textsc{UnelabTyDecl}\,(i+1)\,e) \\
& \quad\text{(if the } i\text{th argument to } x \text{ is a constraint argument)} \\
& \texttt{(}y\ :\ \mathscr{T}\llbracket T \rrbracket\texttt{)}\ \texttt{->}\ (\textsc{UnelabTyDecl}\,(i+1)\,e) \\
& \quad\text{(otherwise)}
\end{aligned}
$$

Using this, we define Unelab for top-level declarations. For pattern matching clauses reversing elaboration proceeds as follows, discarding the explicit pattern variable bindings and applying UnelabType to reconstruct the type declaration:

$$
\begin{aligned}
&\textsc{Unelab}\,(x\ :\ t)\ =\ \textsc{UnelabType}\,(x\ :\ t) \\
&\textsc{Unelab}\,(\underline{\mathrm{var}}\,\vec{x}{:}\vec{U}.\,\texttt{f}\,\vec{s} = e)\ =\ \mathscr{T}\llbracket \texttt{f}\,\vec{s} \rrbracket\ =\ \mathscr{T}\llbracket e \rrbracket
\end{aligned}
$$

For data type declarations, reverse elaboration proceeds as follows, applying UnelabType for each of the top-level type declarations:

$$
\begin{aligned}
&\textsc{Unelab}\,(\underline{\mathrm{data}}\ \texttt{T}\,(\vec{x}\ :\ \vec{t})\ :\ T\ \underline{\mathrm{where}}\ \vec{cons}) \\
&\qquad = \texttt{data}\,\textsc{UnelabType}\,(\texttt{T}\ :\ \forall \vec{x}{:}\vec{t}.\,T)\,\texttt{where}\,(\overrightarrow{\textsc{UnelabType}}\,\vec{cons})
\end{aligned}
$$

5.2 Elaboration properties

Elaboration satisfies two important properties. We limit our discussion of these properties to Idris^{-} without type classes, i.e. elaboration of type declarations, functions and data types. This is primarily because there is not enough information in a TT program alone to reverse elaboration fully. However, since elaboration of the higher level IDRIS constructs is implemented in terms of the lower level IDRIS$^-$ constructs, we should not consider this a serious limitation.

Informally stated, the properties that elaboration satisfies are that (i) if elaboration is successful, the resulting program is a well-typed TT program; and (ii) elaboration preserves the meaning of the original IDRIS program. The first property is true by the definition of elaboration – elaboration fails if it attempts to construct an ill-typed term at any point. Furthermore, the development calculus TT_{dev} ensures that partial constructions are well-typed. The second property can be stated as the following conjecture:

Conjecture: Preservation of meaning
Given an IDRIS$^-$ declaration d, which is either a type declaration, a pattern match clause, or a data type declaration, if ELAB $d = t$, and UNELAB $t = d'$, then MATCH $d\,d'$ produces a valid match.

The output of UNELAB is not necessarily equal to the input of ELAB because elaboration fills in placeholder subexpressions. Therefore, it suffices for the input to match the output.

We have not yet proved this conjecture. Its truth depends on the implementation of ELAB faithfully translating each construct, and we observe that the present description of ELAB elaborates each non-placeholder subexpression according to the structure of the expression. However, since the truth of this conjecture is crucial to the correctness of the implementation, the elaborator checks *dynamically* that meaning is preserved by evaluating UNELAB t and matching the input against the result. In practice, this does not have a significant impact on performance.

6 Related works

Dependently typed programming languages have become more prominent in recent years as tools for verifying software correctness, and several experimental languages are being developed, in particular Agda (Norell, 2007), Epigram (McBride & McKinna, 2004; Chapman *et al.*, 2010) and Trellys (Kimmell *et al.*, 2012). Furthermore, recent extensions to Haskell (Vytiniotis *et al.*, 2011), implemented in the Glasgow Haskell Compiler, are bringing more of the power of dependent types to Haskell. The problem of refining high-level syntax to a core type theory also applies to theorem provers based on dependent types such as Coq (Bertot & Castéran, 2004) and Matita (Asperti *et al.*, 2011).

Checking advanced type system features in Haskell involves a type system parameterised over an underlying constraint system X which captures constraints such as type classes, constrained data types and type families. Types are checked using an inference algorithm OUTSIDEIN(X) (Vytiniotis *et al.*, 2011), which is stratified into an inference engine independent of the constraint system, and a constraint solver for X. An additional difficulty faced by Haskell, and hence any extensions, is the desire to support type *inference*, in which principal types may be inferred for top-level functions. We have avoided such difficulties since, in general, type inference is undecidable for full dependent types. Indeed, it is not clear that type inference is even desirable in many cases, as programmers

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

can use dependent types to state their intentions (hence, a program specification) more precisely. However, in future work we may consider adapting the OUTSIDEIN approach to provide limited type inference.

An earlier implementation of IDRIS was built on the IVOR proof engine (Brady, 2006). This implementation differed in one important way – unlike the present implementation, there was limited separation between the type theory and the high-level language. The type theory itself supported implicit syntax and unification, with high-level constructs such as the `with` rule implemented directly. Two important disadvantages were found with this approach, however: firstly, the type checker is much more complicated when combined with unification, making it harder to maintain; secondly, adding new high-level features requires the type checker to support those features directly. In contrast, elaboration by tactics gives a clean separation between the low-level and high-level languages, and results in programs in a core type theory which are separately checkable, perhaps even by an independently written checker which implements the TT rules directly.

Matita uses a bi-directional refinement algorithm (Asperti *et al.*, 2012). This is a type-directed approach, maintaining a set of yet to be solved unification problems and relying on a small kernel, similar to the approach we now take with IDRIS, but using refinement rules rather than tactics. This leads to good error messages, although it is not clear how easy it would be to extend to additional high-level language features, unlike the tactic-based approach.

The Agda implementation is based on a type theory with implicit syntax and pattern matching – Norell gives an algorithm for type checking a dependently typed language with pattern matching and metavariables (Norell, 2007). Unlike the present IDRIS implementation, metavariables and implicit arguments are part of the type theory. This has the advantage that implicit arguments can be used more freely (for example, in higher order function arguments) at the expense of complicating the type system.

Epigram (McBride & McKinna, 2004) and Oleg (McBride, 1999) have provided much of the inspiration for the IDRIS elaborator. Indeed, the hole and guess bindings of TT_{dev} are taken directly from Oleg. EPIGRAM does not implement pattern matching directly, but rather translates pattern matching into elimination rules (McBride, 2000). This has the advantage that elimination rules provide termination and coverage proofs *by construction*. Furthermore, they simplify implementation of the evaluator and provide easy optimisation opportunities (Brady *et al.*, 2003). However, it requires the implementation of extra machinery for constructor manipulation (McBride *et al.*, 2006) and so we have avoided it in the present implementation.

7 Conclusions

In this paper, I have given an overview of the programming language IDRIS, and its core type theory TT, giving a detailed algorithm for translating high-level programs into TT. TT itself is deliberately small and simple, and the design has deliberately resisted innovation so that we can rely on existing metatheoretic properties being preserved. The kernel of the IDRIS implementation consists of a type checker and evaluator for TT along with a pattern match compiler, which are implemented in under 1,000 lines of Haskell code. It is important that this kernel remains small – the correctness of the language implementation

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

relies to a large extent on the correctness of the underlying type system, and keeping the implementation small reduces the possibility of errors.

The approach we have taken to implementing the high-level language, implementing an elaboration monad with tactics for program construction, allows us to build programs on top of a small and unchanging kernel, rather than extending the core language to deal with implicit syntax, unification and type classes. High-level IDRIS features are implemented by describing the corresponding sequence of tactics to build an equivalent program in TT, via a development calculus of incomplete terms, TT_{dev}. A significant advantage we have found with this approach is that higher level features can easily be implemented in terms of existing elaborator components. For example, once we have implemented elaboration for data types and functions, it is easy to add several features:

- **Type classes**: A dictionary is merely a record containing the functions which implement a type class instance. Since we have a tactic-based refinement engine, we can implement type class resolution as a tactic.
- `where` **clauses**: We have access to local variables and their types, so we can elaborate `where` clauses at the point of definition simply by lifting them to the top level.
- `case` **expressions**: Similar to `where` clauses, these are implemented by lifting the branches out to a top-level function.

We do not need to make any changes to the core language type system in order to implement these high-level features. Other high-level features, such as dependent records, tuples and monad comprehensions, can be added equally easily – and indeed have been added in the full implementation. Furthermore, since we have taken a tactic-based approach to elaborating IDRIS to TT, it is possible to expose tactics to the programmer. This opens up the possibility of implementing domain-specific decision procedures, or implementing user-defined tactics in a style similar to Coq's `Ltac` language (Delahaye, 2000). Although TT is primarily intended as a core language for IDRIS, its rich type system also means that it could be used as a core language for other high-level languages, especially when augmented with primitive operators, and used to express additional properties of those languages.

We have not discussed the performance of the elaboration algorithm, or described how IDRIS compiles to executable code. In practice, we have found performance to be acceptable – for example, the IDRIS library (31 files, 3,718 lines of code in total at the time of writing) elaborates in around 12 seconds.[1] Profiling suggests that the main bottleneck is locating holes in a proof term, which can be improved by choosing a better representation for proof terms, perhaps based on a zipper (Huet, 1997). Compilation is made straightforward by the Epic library (Brady, 2011a), with I/O and foreign functions handled using command–response interaction trees (Hancock & Setzer, 2000). Although I have not yet run detailed benchmarks of the performance of compiled code, I believe that rich type information and guaranteed termination will allow aggressive optimisations (Brady *et al.*, 2003; Brady, 2005). I will investigate this in future work.

The objective of this implementation of IDRIS is to provide a platform for experimenting with realistic, general-purpose programming with dependent types, by implementing a

[1] On a MacBook Pro, 2.8 GHz Intel Core 2 Duo, 4Gb RAM.

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

Haskell-like language augmented with *full* dependent types. In this paper, we have seen how such a high-level language can be implemented by building on top of a small, well-understood, easy to reason about type theory. However, a programming language implementation is not an end in itself. Programming languages exist to support research and practice in many different domains. In future work, therefore, I plan to apply domain-specific language-based techniques to realistic problems in important safety critical domains such as security and network protocol design and implementation. In order to be successful, this will require a language which is expressive enough to describe protocol specifications at a high-level, and robust enough to guarantee correct implementation of those protocols. IDRIS, I believe, is the right tool for this work.

Acknowledgments

This work was funded by the Scottish Informatics and Computer Science Alliance (SICSA) and by EU Framework 7 Project No. 248828 (ADVANCE). My thanks to Philip Hölzenspies, Kevin Hammond, Vilhelm Sjöberg, Falko Spiller and Nathan Collins for their comments on an earlier draft of this paper, and to the anonymous referees for their many insightful and constructive comments.

References

Altenkirch, T., Danielsson, N. A., Löh, A. & Oury, N. (2010) Dependent types without the sugar. In *Tenth International Symposium on Functional and Logic Programming (FLOPS 2010)*, Blume, M., Kobayashi, N. & Vidal, G. (eds), Lecture Notes in Computer Science 6009. Berlin, Germany: Springer, pp. 40–55.

Asperti, A., Ricciotti, W., Coen, C. S. & Tassi, E. (2011) The Matita Interactive theorem prover. In *Automated Deduction – CADE-23,* Lecture Notes in Computer Science, 6803. Berlin, Germany: Springer, pp. 64–69.

Asperti, A., Ricciotti, W., Coen, C. S. & Tassi, E. (2012) A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods Comput. Sci.* **8**, 1–49.

Augustsson, L. (1985) Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, Jouannaud, J.-P. (ed), Lecture Notes in Computer Science, vol. 201. Berlin, Germany: Springer, pp. 368–381.

Bertot, Y. & Castéran, P. (2004) *Interactive Theorem Proving and Program Development: Coq'Art: the Calculus of Inductive Constructions*. Berlin, Germany: Springer.

Brady, E. (2005) *Practical Implementation of a Dependently Typed Functional Programming Language*, PhD thesis, University of Durham, Durham, NC.

Brady, E. (2006, September) Ivor, a proof engine. In *Implementation and Application of Functional Languages (IFL'06)*, Budapest, Hungary, pp. 145–162.

Brady, E. (2011a) Epic – A library for generating compilers. In *Proceedings of the International Symposium on Trends in Functional Programming (TFP11)*, Madrid, Spain.

Brady, E. (2011b) Idris – Systems programming meets full dependent types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification (PLPV '11)*. New York, NY: ACM Press.

Brady, E. (2013). *Programming in Idris : A Tutorial*. Available at http://www.idris-lang.org/documentation/.

Brady, E., McBride, C. & McKinna, J. (2003) Inductive families need not store their indices. In *Types for Proofs and Programs (TYPES 2003)*, Torino, Italy. Berlin, Germany: Springer.

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press

Chapman, J., Altenkirch, T. & McBride, C. (2005). Epigram reloaded: A standalone typechecker for ETT. In *Sixth Symposium on Trends in Functional Programming*, Tallinn, Estonia.

Chapman, J., Dagand, P.-E., McBride, C. & Morris, P. (2010, September) The gentle art of levitation. In *Proceedings of 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*, Baltimore, MD, vol. 45.

Coquand, T. (1986) An analysis of Girard's paradox. In *Proceedings of the First IEEE Symposium on Logic in Computer Science (LICS'86)*. Washington, DC: IEEE Comp. Soc. Press, pp. 227–246.

Delahaye, D. (2000) A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning (LPAR)*. New York, NY: Springer, pp. 85–95.

Dybjer, P. (1994) Inductive families. *Form. Asp. Comput.* **6**(4), 440–465.

Hancock, P. & Setzer, A. (2000, August) Interactive programs in dependent type theory. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic,* Oxford, UK, pp. 317–331.

Huet, G. (1997) The zipper. *J. Funct. Program.* **7**(5), 549–554.

Kimmell, G., Stump, A., Eades III, H. D., Fu, P., Sheard, T., Weirich, S., Casinghino, C., Sjöberg, V., Collins, N. & Ahn, K. Y. (2012) Equational reasoning about programs with general recursion and call-by-value semantics. In *Proceedings of the 6th ACM workshop on Programming Languages Meets Program Verification (PLPV '12)*, Philadelphia, PA.

Lee, C. S., Jones, N. D., & Ben-Amram, A. M. (2001) The size-change principle for program termination. *ACM SIGPLAN Not.* **36**(3), 81–92.

Löh, A., McBride, C. & Swierstra, W. (2010) A tutorial implementation of a dependently typed lambda calculus. *Fundamenta Informaticae* **102**(2), 177–207.

Luo, Z. (1994) *Computation and Reasoning: A Type Theory for Computer Science*. Oxford, UK: Oxford University Press.

McBride, C. (1999) *Dependently Typed Functional Programs and their Proofs*, PhD thesis, University of Edinburgh, Edinburgh, UK.

McBride, C. (2000) Elimination with a motive. In *Types for Proofs and Programs (TYPES 2000)*, Durham, UK.

McBride, C., Goguen, H. & McKinna, J. (2006). A few constructions on constructors. In *Types for Proofs and Programs (TYPES 2006)*, Nottingham, UK.

McBride, C. & McKinna, J. (2004) The view from the left. *J. Funct. Program.* **14**(1), 69–111.

McBride, C. & Paterson, R. (2008) Applicative programming with effects. *J. Funct. Program.* **18**(1), 1–13.

Miller, D. (1992). Unification under a mixed prefix. *J. Symb. Comput.* **14**, 321–358.

Norell, U. (2007). *Towards a Practical Programming Language Based on Dependent Type Theory*, PhD thesis, Chalmers University of Technology, Sweden.

Peyton Jones, S., Vytiniotis, D., Weirich, S. & Washburn, G. (2006) Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP '06)*, vol. 41. New York, NY: ACM, pp. 50–61.

Pollack, R. (1990). Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks, Antibes*. Citeseer.

Sozeau, M. & Oury, N. (2008) First-class type classes. In *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, Montreal, Canada, pp. 278–293.

Vytiniotis, D., Peyton Jones, S., Schrijvers, T. & Sulzmann, M. (2011) OutsideIn(X) modular type inference with local assumptions. *J. Funct. Program.* **21**(4–5), 333–412.

https://doi.org/10.1017/S095679681300018X Published online by Cambridge University Press