# Ivor, a Proof Engine

Edwin Brady

School of Computer Science,
University of St Andrews, St Andrews, Scotland.
`Email: eb@cs.st-andrews.ac.uk.`
`Tel: +44-1334-463253, Fax: +44-1334-463278`

**Abstract.** Dependent type theory has several practical applications in the fields of theorem proving, program verification and programming language design. Ivor is a Haskell library designed to allow easy extending and embedding of a type theory based theorem prover in a Haskell application. In this paper, I give an overview of the library and show how it can be used to embed theorem proving technology in an implementation of a simple functional programming language; by using type theory as a core representation, we can construct and evaluate terms and prove correctness properties of those terms within the *same* framework, ensuring consistency of the implementation and the theorem prover.

## 1 Introduction

Type theory based theorem provers such as Coq [7] and Agda [8] have been used as tools for verification of programs (e.g. [18, 12, 27]), extraction of correct programs from proofs (e.g. [19, 9]) and formal proofs of mathematical properties (e.g. [13, 15]). However, these tools are designed with a human operator in mind; the interface is textual which makes it difficult for an external program to interact with them. In contrast, the Ivor library is designed to provide an implementation of dependent type theory (i.e. dependently typed $\lambda$-calculus) and tactics for proof and program development to a Haskell application programmer, via a stable, well-documented and lightweight (as far as possible) API. The goal is to allow: i) easy embedding of theorem proving tools in a Haskell application; and ii) easy extension of the theorem prover with *domain specific* tactics, via an embedded domain specific language (EDSL) for tactic construction.

### 1.1 Motivating Examples

Many situations can benefit from a dependently typed proof and programming framework accessible as a library from a Haskell program. For each of these, by using an implementation of a well understood type theory, we can be confident that the underlying framework is sound.

**Programming Languages** Dependent type theory can be used as the internal representation for a functional programming language. One advantage of a pure functional language is that correctness properties of programs in the language can be proven by equational reasoning. Some tools exist to assist with this, e.g. Sparkle [10] for the Clean language [31], or Cover [1] for translating Haskell into Agda [8]. However a problem with such tools is that they separate the language implementation from the theorem prover — every language feature must be translated into the theorem prover's representation, and any time the language implementation is changed, this translation must also be changed. In section 4.2, we will see how Ivor can be used to implement a language with a built-in theorem prover, with a common representation for both.

**Verified DSL Implementation** We have previously used Ivor to demonstrate the implementation of a verified domain specific language [5]. The idea is to represent the abstract syntax tree of a DSL program as a dependent type, and write an interpreter which guarantees that invariant properties of the program are maintained. Using staging annotations [34], such an interpreter can be specialised to a translator. We are continuing to explore these techniques in the context of resource aware programming [4].

**Formal Systems** A formal system can be modelled in dependent type theory, and properties of the system and derivations within the system can be proved. A simple example is propositional logic — the connectives $\wedge$, $\vee$ and $\rightarrow$ are represented as types, and a theorem prover is used to prove logical formulae. Having an implementation of type theory and an interactive theorem prover accessible as an API makes it easy to write tools for working in a formal system, whether for educational or practical purposes. In section 4.1, I will give details of an implementation of propositional logic.

In general, the library can be used wherever formally certified code is needed — evaluation of dependently typed Ivor programs is possible from Haskell programs and the results can be inspected easily. Whatever the situation, domain specific tactics are often required. For example, an implementation of a programming language with subtyping may require a tactic for inserting coercions, or a computer arithmetic system may require an implementation of Pugh's Omega decision procedure [33]. Ivor's API is designed to make implementation of such tactics as easy as possible.

## 2 The Type Theory, TT

### 2.1 The Core Calculus

The core type theory of Ivor is a strongly normalising dependently typed $\lambda$-calculus with inductive families [11], similar to Luo's UTT [20], the Calculus of Inductive Constructions in Coq [7], or Epigram's ETT [6]. This language, which I call TT, is an enriched lambda calculus, with the usual reduction rules, and properties of subject reduction, Church Rosser, and uniqueness of types up to conversion. The strong normalisation property (i.e. that evaluation always terminates) is guaranteed by allowing only primitive recursion over strictly positive inductive datatypes. The syntax of terms in this language is as follows:

$$
\begin{array}{llll}
t ::= \star_i & \text{(type universes)} \mid x & & \text{(variable)} \\
\quad\mid b.\ t & \text{(binding)} & \mid t\ t & \text{(application)} \\
b ::= \lambda x\!:\!t & \text{(abstraction)} & \mid \underline{\text{let}}\ x \mapsto t\ :\ t & \text{(let binding)} \\
\quad\mid \forall x\!:\!t & \text{(function space)} & &
\end{array}
$$

We may also write the function space $\forall x\!:\!S.\ T$ as $(x\ :\ S) \rightarrow T$, or abbreviate it to $S \rightarrow T$ if $x$ is not free in $T$. This is both for readability and a notation more consistent with traditional functional programming languages.

Universe levels on types ($\star_i$) may be left implicit and inferred by the machine as in [32]. Contexts ($\Gamma$) are defined inductively; the empty context is valid, as is a context extended with a $\lambda$, $\forall$ or <u>let</u> binding:

$$
\frac{}{\vdash \underline{\text{valid}}} \qquad \frac{\Gamma \vdash \underline{\text{valid}}}{\Gamma; b \vdash \underline{\text{valid}}}
$$

The typing rules are given below. These depend on a conversion relation $\Gamma \vdash x \simeq y$, which holds if and only if $x$ and $y$ have a common redex. This requires the typechecker to perform some evaluation — in order to find a common redex, types are reduced to normal form — so it is important for decidability of typechecking that the language is strongly normalising.

$$
\frac{\Gamma \vdash \underline{\text{valid}}}{\Gamma \vdash \star_n\ :\ \star_{n+1}}\ \mathsf{Type}
$$

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

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

$$
\frac{\Gamma; \lambda x\!:\!S \vdash e\ :\ T \qquad \Gamma \vdash (x\ :\ S) \rightarrow T\ :\ \star_n}{\Gamma \vdash \lambda x\!:\!S.e\ :\ (x\ :\ S) \rightarrow T}\ \mathsf{Lam}
$$

$$
\frac{\Gamma; \forall x\!:\!S \vdash T\ :\ \star_n \qquad \Gamma \vdash S\ :\ \star_n}{\Gamma \vdash (x\ :\ S) \rightarrow T\ :\ \star_n}\ \mathsf{Forall}
$$

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

$$
\frac{\Gamma \vdash x\ :\ A \qquad \Gamma \vdash A'\ :\ \star_n \qquad \Gamma \vdash A \simeq A'}{\Gamma \vdash x\ :\ A'}\ \mathsf{Conv}
$$

## 2.2 Inductive Families

Inductive families [11] are a form of simultaneously defined collection of algebraic data types (such as Haskell `data` declarations) which can be parametrised over *values* as well as types. An inductive family is declared in a similar style to a Haskell GADT declaration [30] as follows, using the de Bruijn telescope notation, $\vec{x}$, to indicate a sequence of zero or more $x$:

$$
\underline{\text{data}}\ \mathsf{T}\,(\vec{x}\ :\ \vec{t})\ :\ t \quad \underline{\text{where}} \quad \mathsf{c}_1\ :\ t\ \mid\ \ldots\ \mid\ \mathsf{c}_n\ :\ t
$$

Constructors may take recursive arguments in the family $\mathsf{T}$. Furthermore these arguments may be indexed by another type, as long it does not involve $\mathsf{T}$ — this restriction is known as **strict positivity** and ensures that recursive arguments of a constructor are structurally smaller than the value itself.

The Peano style natural numbers can be declared as follows:

$$\underline{\text{data}}\,\mathbb{N} : \star \quad \underline{\text{where}} \quad \mathsf{0} : \mathbb{N} \mid \mathsf{s} : (k : \mathbb{N}) \to \mathbb{N}$$

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, a list is parametrised over its element type:

$$\begin{array}{l}\underline{\text{data}}\,\mathsf{List}\,(A : \star) : \star \quad \underline{\text{where}}\ \ \mathsf{nil} : \mathsf{List}\,A \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{cons} : (x : A) \to (xs : \mathsf{List}\,A) \to \mathsf{List}\,A\end{array}$$

$\mathsf{TT}$ is a dependently typed calculus, meaning that types can be parametrised 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. $\mathsf{Vect}$ is parametrised over its element type, like $\mathsf{List}$, but *indexed* over its length.

$$\begin{array}{l}\underline{\text{data}}\,\mathsf{Vect}\,(A : \star) : \mathbb{N} \to \star \quad \underline{\text{where}} \\ \qquad\quad \mathsf{vnil} : \mathsf{Vect}\,A\,\mathsf{0} \\ \qquad \mid \mathsf{vcons} : (k : \mathbb{N}) \to (x : A) \to (xs : \mathsf{Vect}\,A\,k) \to \mathsf{Vect}\,A\,(\mathsf{s}\,k)\end{array}$$

## 2.3 Elimination Rules

When we declare an inductive family $\mathsf{D}$, we give the constructors which explain how to build objects in that family. Along with this, the machine generates an **elimination operator D-Elim** (the type of which we call the **elimination rule**) and corresponding reductions. The operator implements the reduction and recursion behaviour of terms in the family — it is effectively a fold operator. The method for constructing elimination operators automatically is well documented, in particular by [11, 20, 22]. For $\mathsf{Vect}$, Ivor generates the following operator:

$$\begin{array}{ll}\mathsf{Vect}\text{-}\mathbf{Elim} : & (A : \star) \to (n : \mathbb{N}) \to (v : \mathsf{Vect}\,A\,n) \to \\ & (P : (n : \mathbb{N}) \to (v : \mathsf{Vect}\,A\,n) \to \star) \to \\ & (m_{\mathsf{vnil}} : P\,\mathsf{0}\,(\mathsf{vnil}\,A)) \to \\ & (m_{\mathsf{vcons}} : (k : \mathbb{N}) \to (x : A) \to (xs : \mathsf{Vect}\,A\,k) \to \\ & \qquad\qquad (ih : P\,k\,xs) \to P\,(\mathsf{s}\,k)\,(\mathsf{vcons}\,A\,k\,x\,xs)) \to \\ & P\,n\,v \\ \mathsf{Vect}\text{-}\mathbf{Elim}\,A\ \ \mathsf{0} \qquad\quad (\mathsf{vnil}\,A) \qquad P\,m_{\mathsf{vnil}}\,m_{\mathsf{vcons}} \leadsto m_{\mathsf{vnil}} & \\ \mathsf{Vect}\text{-}\mathbf{Elim}\,A\,(\mathsf{s}\,k)\,(\mathsf{vcons}\,A\,k\,x\,xs)\,P\,m_{\mathsf{vnil}}\,m_{\mathsf{vcons}} & \\ & \leadsto m_{\mathsf{vcons}}\,k\,x\,xs\,(\mathsf{Vect}\text{-}\mathbf{Elim}\,A\,k\,xs\,P\,m_{\mathsf{vnil}}\,m_{\mathsf{vcons}})\end{array}$$

The arguments to the elimination operator are the **indices** ($A$ and $n$ here), the **target** (the object being eliminated; $v$ here), the **motive** (a function which computes the return type of the elimination; $P$ here) and the **methods** (which describe how to achieve the motive for each constructor form).

A case analysis operator **D-Case**, is obtained similarly, but without the induction hypotheses. These operators are the only means to analyse a data structure and the only operators which can make recursive calls. This, along with the restriction that data types must be strictly positive, ensures that evaluation always terminates.

### 2.4 The Development Calculus

For developing terms interactively, the type theory needs to support *incomplete* terms, and a method for term construction. We extend TT with the concept of **holes**, which stand for the parts of constructions which have not yet been instantiated; this largely follows McBride's OLEG development calculus [22].

The basic idea is to extend the syntax for binders with a *hole* binding and a *guess* binding. The *guess* binding is similar to a $\underline{\text{let}}$ binding, but without any computational force, i.e. the bound names do not reduce:

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

Using binders to represent holes as discussed in [22] is useful in a dependently typed setting since one value may determine another. Attaching a "guess" to a binder ensures that instantiating one such value also instantiates all of its dependencies. The typing rules for binders ensure that no ? bindings leak into types, and are given below.

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

## 3 The Ivor Library

The IVOR library allows the incremental, type directed development of TT terms. In this section, I will introduce the basic tactics available to the library user, along with the Haskell interface for constructing and manipulating TT terms. This section includes only the most basic operations; the API is however fully documented on the web[1].

### 3.1 Definitions and Context

The central data type is `Context` (representing $\Gamma$ in the typing rules), which is an abstract type holding information about inductive types and function definitions as well as the current proof state. All operations are defined with respect to the context. An empty context is contructed with `emptyContext :: Context`.

Terms may be represented several ways; either as concrete syntax (a `String`), an abstract internal representation (`Term`) or as a Haskell data structure (`ViewTerm`). A typeclass `IsTerm` is defined, which allows each of these to be converted into the internal representation. This typeclass has one method:

[1] `http://www.cs.st-andrews.ac.uk/~eb/Ivor/doc/`

```
class IsTerm a where
    check :: Monad m => Context -> a -> m Term
```

The `check` method parses and typechecks the given term, as appropriate, and if successful returns the internal representation. Constructing a term in this way may fail (e.g. due to a syntax or type error) so `check` is generalised over a monad `m` — it may help to read `m` as `Maybe`.

In this paper, for the sake of readability we will use the syntax described in section 2.1, and assume an instance of `IsTerm` for this syntax.

Similarly, there is a typeclass for inductive families, which may be represented either as concrete syntax or a Haskell data structure.

```
class IsData a where
    addData :: Monad m => Context -> a -> m Context
```

The `addData` method adds the constructors and elimination rules for the data type to the context. Again, we assume an instance for the syntax presented in section 2.2.

The simplest way to add new function definitions to the context is with the `addDef` function. Such definitions may not be recursive, other than via the automatically generated elimination rules:

```
addDef :: (IsTerm a, Monad m) => Context -> Name -> a -> m Context
```

However, Ivor is primarily a library for constructing proofs; the Curry-Howard correspondence identifies programs and proofs, and therefore such definitions can be viewed as proofs; to prove a theorem is to add a well-typed definition to the context. We would like to be able to construct more complex proofs (and indeed programs) interactively — and so at the heart of Ivor is a theorem proving engine.

## 3.2 Theorems

In the `emptyContext`, there is no proof in progress, so no proof state — the `theorem` function creates a proof state in a context. This will fail if there is already a proof in progress, or the goal is not well typed.

```
theorem :: (IsTerm a, Monad m) => Context -> Name -> a -> m Context
```

A proof state can be thought of as an incomplete term, i.e. a term in the development calculus. For example, calling `theorem` with the name **plus** and type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, an initial proof state would be:

$\textbf{plus} \; = ?plus : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$

Proving a theorem proceeds by applying tactics to each unsolved hole in the proof state. The system keeps track of which subgoals are still to be solved, and a default subgoal, which is the next subgoal to be solved. I will write proof states in the following form:

$$\begin{array}{l} \textit{bindings in the context of the subgoal } x_0 \\ \ldots \\ \hline \qquad\qquad ?x_0 : \textit{default subgoal type} \\ \qquad\qquad \ldots \\ \qquad\qquad ?x_i : \textit{other subgoal types} \\ \qquad\qquad \ldots \end{array}$$

Functions are available for querying the bindings in the context of any subgoal. A tactic typically works on the bindings in scope and the type of the subgoal it is solving.

When there are no remaining subgoals, a proof can be lifted into the context, to be used as a complete definition, with the `qed` function:

```
qed :: Monad m => Context -> m Context
```

This function typechecks the entire proof. In practice, this check should never fail — the development calculus itself ensures that partial constructions as well as complete terms are well-typed, so it is impossible to build ill-typed partial constructions. However, doing a final typecheck of a complete term means that the soundness of the system relies only on the soundness of the typechecker for the core language, e.g. [2]. We are then free to implement tactics in any way we like, safe in the knowledge that any ill-typed or unsound constructions will be caught by the typechecker.

## 3.3 Basic Tactics

A tactic is an operation on a goal in the current system state; we define a type synonym `Tactic` for functions which operate as tactics. Tactics modify system state and may fail, hence a tactic function returns a monad:

```
type Tactic = forall m . Monad m => Goal -> Context -> m Context
```

A tactic operates on a hole binding, specified by the `Goal` argument. This can be a named binding, `goal :: Name -> Goal`, or the default goal `defaultGoal :: Goal`. The default goal is the first goal generated by the most recent tactic application.

**Hole Manipulations** There are three basic operations on holes, **claim**, **fill**, and **abandon**; these are given the following types:

```
claim :: IsTerm a => Name -> a -> Tactic
fill :: IsTerm a => a -> Tactic
abandon :: Tactic
```

The `claim` function takes a name and a type and creates a new hole. The `fill` function takes a guess to attach to the current goal. In addition, `fill` attempts to solve other goals by unification. Attaching a guess does not necessarily solve the goal completely; if the guess contains further hole bindings, it cannot yet have any computational force. A guess can be removed from a goal with the `abandon` tactic.

**Introductions** A basic operation on terms is to introduce $\lambda$ bindings into the context. The `intro` and `introName` tactics operate on a goal of the form $(x : S) \to T$, introducing $\lambda x : S$ into the context and updating the goal to $T$. `introName` allows a user specified name choice, otherwise IVOR chooses the name.

```
intro :: Tactic
introName :: Name -> Tactic
```

For example, to define a unary addition function, we might begin with

$$\frac{}{?plus : \mathbb{N} \to \mathbb{N} \to \mathbb{N}}$$

Applying `introName` twice with the names $x$ and $y$ gives the following proof state, with $x$ and $y$ introduced into the local context:

$$\frac{\begin{array}{l}\lambda x : \mathbb{N} \\ \lambda y : \mathbb{N}\end{array}}{?plus\_H : \mathbb{N}}$$

**Refinement** The `refine` tactic solves a goal by an application of a function to arguments. Refining attempts to solve a goal of type $T$, when given a term $t : (\vec{x} : \vec{S}) \to T$. The tactic creates a subgoal for each argument $x_i$, attempting to solve it by unification.

```
refine :: IsTerm a => a -> Tactic
```

For example, given a goal

$$\frac{}{?v : \mathsf{Vect}\ \mathbb{N}\ (\mathsf{s}\ n)}$$

Refining by $\mathsf{vcons}$ creates subgoals for all four arguments, and attaches a guess to $v$:

$$\frac{}{\begin{array}{l} ?A : \star \\ ?k : \mathbb{N} \\ ?x : A \\ ?xs : \mathsf{Vect}\ A\ k \\ ?v : \mathsf{Vect}\ \mathbb{N}\ (\mathsf{s}\ n) \approx \mathsf{vcons}\ A\ k\ x\ xs \end{array}}$$

However, for $\mathsf{vcons}\,A\,k\,x\,xs$ to have type $\mathsf{Vect}\,\mathbb{N}\,(\mathsf{s}\,n)$ requires that $A = \mathbb{N}$ and $k = n$. Refinement unifies these, leaving the following goals:

$$\begin{array}{l}
\hline
?x : \mathbb{N} \\
?xs : \mathsf{Vect}\ \mathbb{N}\ n \\
?v : \mathsf{Vect}\ \mathbb{N}\ (\mathsf{s}\ n) \approx \mathsf{vcons}\ \mathbb{N}\ n\ x\ xs
\end{array}$$

**Elimination** Refinement solves goals by constructing new values; we may also solve goals by deconstructing values in the context, using an elimination operator as described in section 2.3. The `induction` and `cases` tactics apply the **D-Elim** and **D-Case** operators respectively to the given target:

```
induction, cases :: IsTerm a => a -> Tactic
```

These tactics proceed by refinement by the appropriate elimination operator. The motive for the elimination is calculated automatically from the goal to be solved. Each tactic generates subgoals for each method of the appropriate elimination rule.

An example of `induction` is in continuing the definition of our addition function. This can be defined by induction over the first argument. We have the proof state

$$\begin{array}{c}
\lambda x : \mathbb{N} \\
\lambda y : \mathbb{N} \\
\hline
?plus\_H : \mathbb{N}
\end{array}$$

Applying `induction` to $x$ leaves two subgoals, one for the case where $x$ is zero, and one for the inductive case:

$$\begin{array}{l}
\qquad\qquad\lambda x : \mathbb{N} \\
\qquad\qquad\lambda y : \mathbb{N} \\
\hline
?plus\_O : \mathbb{N} \\
?plus\_S : (k\ :\ \mathbb{N}) \to (k\_H\ :\ \mathbb{N}) \to \mathbb{N}
\end{array}$$

By default, the next goal to solve is $plus\_O$. However, the `focus` tactic can be used to change the default goal. The $k\_H$ argument to the $plus\_S$ goal is the result of a recursive call on $k$.

**Rewriting** It is often desirable to rewrite a goal given an equality proof, to perform equational reasoning. The `replace` tactic replaces occurrences of the left hand side of an equality with the right hand side. To do this, it requires four arguments:

1. The equality type; for example $\mathsf{Eq}\ :\ (A\ :\ \star) \to A \to A \to \star$.
2. A replacement lemma, which explains how to substitute one term for another; for example
   $\mathbf{repl}\ :\ (A\ :\ \star) \to (a, b\ :\ A) \to \mathsf{Eq}\ \_\ a\ b \to (P\ :\ A \to \star) \to P\ a \to P\ b$

3. A symmetry lemma, proving that equality is symmetric; for example **sym** $: (A : \star) \to (a, b : A) \to \mathsf{Eq}\ \_\ a\ b \to \mathsf{Eq}\ \_\ b\ a$
4. An equality proof.

The Ivor distribution contains a library of TT code with the appropriate definitions and lemmas. Requiring the lemmas to be supplied as arguments makes the library more flexible — for example, heterogeneous equality [22] may be preferred. The type of the `replace` tactic is:

```
replace :: (IsTerm a, IsTerm b, IsTerm c, IsTerm d) =>
              a -> b -> c -> d -> Bool -> Tactic
```

The `Bool` argument determines whether to apply the symmetry lemma to the equality proof first, which allows rewriting from right to left. This `replace` tactic is similar to Lego's `Qrepl` tactic [21].

For example, consider the following fragment of proof state:

$$\frac{\begin{array}{l}\ldots\\ \lambda x : \mathsf{Vect}\ A\ (\mathbf{plus}\ x\ y)\end{array}}{?\mathit{vect\_H} : \mathsf{Vect}\ A\ (\mathbf{plus}\ y\ x)}$$

Since **plus** is commutative, $x$ ought to be a vector of the correct length. However, the type of $x$ is not convertible to the type of *vect_H*. Given a lemma **plus_commutes** $: (n, m : \mathbb{N}) \to \mathsf{Eq}\ \_\ (\mathbf{plus}\ n\ m)\ (\mathbf{plus}\ m\ n)$, we can use the `replace` tactic to rewrite the goal to the correct form. Applying `replace` to `Eq`, **repl**, **sym** and **plus_commutes** $y\ x$ yields the following proof state:

$$\frac{\begin{array}{l}\ldots\\ \lambda x : \mathsf{Vect}\ A\ (\mathbf{plus}\ x\ y)\end{array}}{?\mathit{vect\_H} : \mathsf{Vect}\ A\ (\mathbf{plus}\ x\ y)}$$

This is easy to solve using the `fill` tactic with $x$.

## 3.4 Tactic Combinators

Ivor provides an embedded domain specific language for building tactics, in the form of a number of combinators for building more complex tactics from the basic tactics previously described. By providing an API for basic tactics and a collection of combinators, it becomes easy to extend the library with more complex domain specific tactics. We will see examples in sections 4.1 and 4.2.

**Sequencing Tactics** There are three basic operators for combining two tactics to create a new tactic:

```
(>->), (>+>), (>=>) :: Tactic -> Tactic -> Tactic
```

1. The `>->` operator constructs a new tactic by sequencing two tactic applications to the *same* goal.

2. The `>+>` operator constructs a new tactic by applying the first, then applying the second to the next *default* goal.
3. The `>=>` operator constructs a new tactic by applying the first tactic, then applying the second to every subgoal generated by the first.

Finally, the `tacs` function takes a list of tactics and applies them in sequence to the default goal:

```
tacs :: Monad m => [Goal -> Context -> m Context] ->
                   Goal -> Context -> m Context
```

Note that the type of this is better understood as `[Tactic] -> Tactic`, but the Haskell typechecker requires that the same monad be abstracted over all of the combined tactics.

**Handling Failure** Tactics may fail (for example a refinement may be ill-typed). Recovering gracefully from a failure may be needed, for example to try a number of possible ways of rewriting a term. The `try` combinator is an exception handling combinator which tries a tactic, and chooses a second tactic to apply to the same goal if the first tactic succeeds, or an alternative tactic if the first tactic fails. The identity tactic, `idTac`, is often appropriate on success.

```
try :: Tactic -> -- apply this tactic
       Tactic -> -- apply if the tactic succeeds
       Tactic -> -- apply if the tactic fails
       Tactic
```

## 4 Examples

In this section we show two examples of embedding Ivor in a Haskell program. The first shows an embedding of a simple theorem prover for propositional logic. The second example extends this theorem prover by using the same logic as a basis for showing properties of a functional language.

### 4.1 A Propositional Logic Theorem Prover

Propositional logic is straightforward to model in dependent type theory; here we show how Ivor can be used to implement a theorem prover for propositional logic. The full implementation is available from `http://www.cs.st-andrews.ac.uk/~eb/Ivor/`. The language of propositional logic is defined as follows, where $x$ stands for an arbitrary free variable:

$L ::= x \mid L \wedge L \mid L \vee L \mid L \rightarrow L \mid \neg L$

There is a simple mapping from this language to dependent type theory — the $\wedge$ and $\vee$ connectives can be declared as inductive families, where the automatically derived elimination rules give the correct elimination behaviour, and the $\rightarrow$ connective follows the same rules as the function arrow. Negation can be defined with the empty type.

The $\wedge$ connective is declared as an inductive family, where an instance of the family gives a proof of the connective. The $\mathsf{and\_intro}$ constructor builds a proof of $A \wedge B$, given a proof of $A$ and a proof of $B$:

$$\begin{array}{l} \underline{\text{data}}\ \mathsf{And}\ (A, B\ :\ \star)\ :\ \star \quad \underline{\text{where}} \\ \qquad \mathsf{and\_intro}\ :\ (a\ :\ A) \rightarrow (b\ :\ B) \rightarrow \mathsf{And}\ A\ B \end{array}$$

Similarly, $\vee$ is declared as an inductive family; an instance of $A \vee B$ is built either from a proof of $A$ ($\mathsf{inl}$) or a proof of $B$ ($\mathsf{inr}$):

$$\begin{array}{l} \underline{\text{data}}\ \mathsf{Or}\ (A, B\ :\ \star)\ :\ \star \quad \underline{\text{where}} \\ \qquad \quad \mathsf{inl}\ :\ (a\ :\ A) \rightarrow \mathsf{Or}\ A\ B \\ \qquad \mid \mathsf{inr}\ :\ (b\ :\ B) \rightarrow \mathsf{Or}\ A\ B \end{array}$$

I will write $[\![e]\!]$ to denote the translate from an expression $e \in L$ to an implementation in $\mathsf{TT}$; in the implementation, this is a parser from strings to `ViewTerm`s:

$$\begin{array}{lcl} [\![e_1 \wedge e_2]\!] & = & \mathsf{And}\ [\![e_1]\!]\ [\![e_2]\!] \\ [\![e_1 \vee e_2]\!] & = & \mathsf{Or}\ [\![e_1]\!]\ [\![e_2]\!] \\ [\![e_1 \rightarrow e_2]\!] & = & [\![e_1]\!] \rightarrow [\![e_2]\!] \end{array}$$

To implement negation, we declare the empty type:

$$\underline{\text{data}}\ \mathsf{False}\ :\ \star \quad \underline{\text{where}}$$

Then $[\![\neg e]\!]\ =\ [\![e]\!] \rightarrow \mathsf{False}$. The automatically derived elimination rule has the following type, showing that a value of *any* type can be created from a proof of the empty type:

$$\mathsf{False}\text{-}\mathbf{Elim}\ :\ (x\ :\ \mathsf{False}) \rightarrow (P\ :\ \mathsf{False} \rightarrow \star) \rightarrow P\ x$$

In the implementation, we initialise the `Context` with these types (using `addData`) and propositional variables $A \ldots Z$ (using `addAxiom`[2]).

**Domain Specific Tactics** Mostly, the implementation of a propositional logic theorem prover consists of a parser and pretty printer for the language $L$, and a top level loop for applying introduction and elimination tactics. However, some domain specific tactics are needed, in particular to deal with negation and proof by contradiction.

To prove a negation $\neg A$, we assume $A$ and attempt to prove $\mathsf{False}$. This is achieved with an `assumeFalse` tactic which assumes the negation of the goal. Negation is defined with a function **not**; the `assumeFalse` tactic then unfolds this name so that a goal (in $\mathsf{TT}$ syntax) **not** $A$ is transformed to $A\ \rightarrow\ \mathsf{False}$, then $A$ can be introduced.

[2] This adds a name with a type but no definition to the context.

```
assumeFalse :: Tactic
assumeFalse = unfold (name "not") >+> intro
```

The proof by contradiction tactic is implemented as follows:

```
contradiction :: String -> String -> Tactic
contradiction x y = claim (name "false") "False" >+>
                    induction "false" >+>
                    (try (fill $ x ++ " " ++ y)
                          idTac
                          (fill $ y ++ " " ++ x))
```

This tactic takes the names of the two contradiction premises. One is of type $A \rightarrow \mathsf{False}$ for some $A$, the other is of type $A$. The tactic works by claiming there is a contradiction and solving the goal by induction over that assumed contradiction (which gives no subgoals, since False-**Elim** has no methods). Finally, using `>+>` to solve the next subgoal (and discharge the assumption of the contradiction), it looks for a value of type False by first applying $y$ to $x$ then, if that fails, applying $x$ to $y$.

### 4.2 Funl, a Functional Language with a Built-in Theorem Prover

Propositional logic is an example of a simple formal system which can be embedded in a Haskell program using Ivor; however, more complex languages can be implemented. Funl is a simple functional language, with primitive recursion over integers and higher order functions. It is implemented on top of Ivor as a framework for both language representation and correctness proofs in that language. By using the same framework for both, it is a small step from implementing the language to implementing a theorem prover for showing properties of programs in the language, making use of the theorem prover developed in sec. 4.1. An implementation is available from `http://www.cs.st-andrews.ac.uk/~eb/Funl/`; in this section I will sketch some of the important details of this implementation. Like the propositional logic theorem prover, much of the detail is in the parsing and pretty printing of terms and propositions.

**Programs and Properties** The Funl language allows programs and statements of the properties those programs should satisfy to be expressed within the same input file. Functions are defined as in the following examples, using `rec` to mark a primitive recursive definition:

```
fac : Int -> Int =
     lam x . rec x 1 (lam k. lam recv. (k+1)*recv);
myplus : Int -> Int -> Int =
     lam x. lam y. rec x y (lam k. lam recv. 1+recv);
```

The `myplus` function above defines addition by primitive recursion over its input. To show that this really is a definition of addition, we may wish to show that it satisfies some appropriate properties of addition. In the FUNL syntax, we declare the properties we wish to show as follows:

```
myplusn0 proves
     forall x:Int. myplus x 0 = x;
myplusnm1 proves
     forall x:Int. forall y:Int. myplus x (1+y) = 1+(myplus x y);
myplus_commutes proves
     forall x:Int. forall y:Int. myplus x y = myplus y x;
```

On compiling a program, the compiler requires that proofs are provided for the stated properties. Once built, the proofs can be saved as proof terms, so that properties need only be proved once.

**Building Terms** Terms are parsed into a data type `Raw`; the name `Raw` reflects the fact that these are raw, untyped terms; note in particular that `Rec` is an operator for primitive recursion on arbitrary types, like the **D-Elim** operators in TT — it would be fairly simple to write a first pass which translated recursive calls into such an operator using techniques similar to McBride and McKinna's labelled types [26], which are implemented in IVOR. Using this, we could easily extend the language with more primitive types (e.g. lists) or even user defined data types. The representation is as follows:

```
data Raw = Var String | Lam String Ty Raw | App Raw Raw
         | Num Int | Boolval Bool  | InfixOp Op Raw Raw
         | If Raw Raw Raw | Rec Raw [Raw]
```

Building a FUNL function consists of creating a `theorem` with a goal representing the function's type, then using the `buildTerm` tactic to traverse the structure of the raw term, constructing a proof of the theorem:

```
buildTerm :: Raw -> Tactic
buildTerm (Var x) = refine x
buildTerm (Lam x ty sc) = introName (name x) >+> buildTerm sc
buildTerm (Language.App f a) = buildTerm f >+> buildTerm a
buildTerm (Num x) = fill (mkNat x)
buildTerm (If a t e) =
    cases (mkTerm a) >+> buildTerm t >+> buildTerm e
buildTerm (Rec t alts) =
    induction (mkTerm t) >+> tacs (map buildTerm alts)
buildTerm (InfixOp Plus x y) =
    refine "plus" >+> buildTerm x >+> buildTerm y
buildTerm (InfixOp Times x y) = ...
```

A helper function, `mkTerm`, is used to translate simple expressions into `ViewTerm`s. This is used for the scrutinees of `if` and `rec` expressions, although if more complex expressions are desired here, it would be possible to use `buildTerm` instead.

```
mkTerm :: Raw -> ViewTerm
mkTerm (Var x) = (Name Unknown (name x))
mkTerm (Lam x ty sc) = Lambda (name x) (mkType ty) (mkTerm sc)
mkTerm (Apply f a) = App (mkTerm f) (mkTerm a)
mkTerm (Num x) = mkNat x
mkTerm (InfixOp Plus x y) =
    App (App (Name Free (name "plus")) (mkTerm x)) (mkTerm y)
mkTerm (InfixOp Times x y) = ...
```

IVOR handles the typechecking and any issues with renaming, using techniques from [25]; if there are any type errors in the `Raw` term, this tactic will fail (although some extra work is required to produce readable error messages). By using IVOR to handle typechecking and evaluation, we are in no danger of constructing or evaluating an ill-typed term.

**Building Proofs** We also define a language of propositions over terms in FUNL. This uses propositional logic, just like the theorem prover in section 4.1, but extended with equational reasoning. For the equational reasoning, we use a library of equality proofs to create tactics for applying commutativity and associativity of addition and simplification of expressions.

A basic language of propositions with the obvious translation to TT is:

```
data Prop = Eq Raw Raw
          | And Prop Prop | Or Prop Prop
          | All String Ty Prop | FalseProp
```

This allows equational reasoning over FUNL programs, quantification over variables and conjunction and disjunction of propositions. A more full featured prover may require relations other than `Eq` or even user defined relations.

## 5 Related Work

The ability to extend a theorem prover with user defined tactics has its roots in Robin Milner's LCF [28]. This introduced the programming language ML to allow users to write tactics; we follow the LCF approach in exposing the tactic engine as an API. However, unlike other systems, we have not treated the theorem prover as an end in itself, but intend to expose the technology to any Haskell application which may need it. The implementation of IVOR is based on the presentation of OLEG in Conor McBride's thesis [22]; this technology also forms the basis for the implementation of EPIGRAM [26]. The core language of EPIGRAM [6] is similar to TT, with extensions for observational equality. We use implementation techniques from [25] for dealing with variables and renaming.

Other theorem provers such as Coq [7], Agda [8] and Isabelle [29] have varying degrees of extensibility. The interface design largely follows that of Coq. Coq includes a high level domain specific language for combining tactics and creating new tactics, along the lines of the tactic combinators presented in section 3.4. This language is ideal for many purposes, such as our `contradiction` tactic, but more complex examples such as `buildTerm` would require extending Coq itself. One advantage of using an embedded DSL [17] as provided by Ivor is that it gives complete flexibility in the construction of tactics, and allows a close relationship between the tactics and the Haskell-implemented structures on which they operate (e.g. `Raw`).

Isabelle [29] is a generic theorem prover, in that it includes a large body of object logics and a meta-language for defining new logics. It includes a typed, extensible tactic language, and can be called from ML programs, but unlike Ivor is not based on a dependent type theory. There is therefore no *proof term* associated with an Isabelle proof — the proof term gives a derivation tree for the proof, allowing easy and independent rechecking without referring to the tactics used to build the proof.

The implementation of Funl allows a theorem prover to be attached to the language in a straightforward way, using Ivor's tactics directly. This would be a possible method of attaching a theorem prover to a more full featured programming language such as the Sparkle [10] prover for Clean [31]. Implementing a full language in this way would require some extra work to deal with general recursion and partial definitions (in particular, dealing with $\perp$ as a possible value), but the general method remains the same.

## 6 Conclusions and Further Work

We have seen an overview of the Ivor library, including basic tactics for building proofs similar to the tactics available in other proof assistants such as Coq. By exposing the tactic API and providing an interface for term construction and evaluation, we are able to embed theorem proving technology in a Haskell application. This in itself is not a new idea, having first been seen as far back as the LCF [28] prover — however, the theorem proving technology is not an end in itself, but a mechanism for constructing domain specific tools such as the propositional logic theorem prover in section 4.1 and the programming language with built in equational reasoning support in section 4.2.

The library includes several features we have not been able to discuss here. There is experimental support for multi-stage programming with dependent types, exploited in [5]. The term language can be extended with primitive types and operations, e.g. integers and strings with associated arithmetic and string manipulation operators. Such features would be essential in a representation of a real programming language. In this paper, we have stated that $\mathsf{TT}$ is strongly normalising, with no general recursion allowed, but again in the representation of a real programming language general recursion may be desirable — however, this means that correctness proofs can no longer be total. The library can optionally allow general recursive definitions, but such definitions cannot be reduced by the typechecker. Finally, a command driven interface is available, which can be accessed as a Haskell API or used from a command line driver program, and allows user directed proof scripts in the style of other proof assistants. These and other features are fully documented on the web site[3].

Development of the library has been driven by the requirements of our research into Hume [16], a resource aware functional language. We are investigating the use of dependent types in representing and verifying resource bounded functional programs [4]. For this, automatic generation of injectivity and disjointness lemmas for constructors will be essential [24], as well as an elimination with a motive [23] tactic. Future versions will include optimisations from [3] and some support for compiling TT terms; this would not only improve the efficiency of the library (and in particular its use for evaluating certified code) but also facilitate the use of Ivor in a real language implementation. Finally, an implementation of coinductive types [14] is likely to be very useful; currently it can be achieved by implementing recursive functions which do not reduce at the type level, but a complete implementation with criteria for checking productivity would be valuable for modelling streams in Hume.

## Acknowledgements

This work is generously supported by EPSRC grant EP/C001346/1.

## References

1. Cover translator. `http://coverproject.org/CoverTranslator/`.
2. B. Barras and B. Werner. Coq in Coq, 1997.
3. E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.
4. E. Brady and K. Hammond. A dependently typed framework for static analysis of program execution costs. In *Proc. Implementation of Functional Languages (IFL 2005)*, volume 4015 of *LNCS*, pages 74–90. Springer, 2006.
5. E. Brady and K. Hammond. A verified staged interpreter is a verified compiler. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '06)*, 2006.
6. J. Chapman, T. Altenkirch, and C. McBride. Epigram reloaded: A standalone typechecker for ETT. In *Trends in Functional Programming 2005*, 2006. To appear.
7. Coq Development Team. The Coq proof assistant — reference manual. `http://coq.inria.fr/`, 2001.
8. C. Coquand. Agda. `http://agda.sourceforge.net/`, 2005.
9. L. Cruz-Filipe and B. Spitters. Program extraction from large proof developments, 2003.
10. M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers, 2002.
11. P. Dybjer. Inductive families. *Formal Aspects Of Computing*, 6:440–465, 1994.

[3] `http://www.cs.st-andrews.ac.uk/~eb/Ivor/`

12. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
13. H. Geuvers, F. Wiedijk, and J. Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In *TYPES 2000*, pages 96–111, 2000.
14. E. Giménez. An application of co-inductive types in coq: Verification of the alternating bit protocol. In *Proceedings of the 1995 Workshop on Types for Proofs and Programs*, volume 1158 of *LNCS*, pages 135–152. Springer-Verlag, 1995.
15. G. Gonthier. A computer-checked proof of the Four Colour Theorem. `http://research.microsoft.com/~gonthier/4colproof.pdf`, 2005.
16. K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
17. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28A(4), December 1996.
18. X. Leroy. Formal certification of a compiler back-end. In *Principles of Programming Languages 2006*, pages 42–54. ACM Press, 2006.
19. P. Letouzey. A new extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for proofs and programs*, LNCS. Springer, 2002.
20. Z. Luo. *Computation and Reasoning – A Type Theory for Computer Science*. Intl. Series of Monographs on Comp. Sci. OUP, 1994.
21. Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical report, Department of Computer Science, University of Edinburgh, 1992.
22. C. McBride. *Dependently Typed Functional Programs and their proofs*. PhD thesis, University of Edinburgh, May 2000.
23. C. McBride. Elimination with a motive. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs*, pages 197–216. Springer, 2000.
24. C. McBride, H. Goguen, and J. McKinna. Some constructions on constructors. In *TYPES 2004*, volume 3839 of *LNCS*. Springer, 2005.
25. C. McBride and J. McKinna. I am not a number, I am a free variable. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, 2004.
26. C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
27. J. McKinna and J. Wright. A type-correct, stack-safe, provably correct, expression compiler in epigram. Submitted for publication, 2006.
28. R. Milner. LCF: A way of doing proofs with a machine. In *Mathematical Foundations of Computer Science 1978*, volume 64 of *LNCS*, pages 146–159, 1979.
29. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A proof assistant for higher order logic*, volume 2283 of *LNCS*. Springer-Verlag, March 2002.
30. S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06*, 2006.
31. R. Plasmeijer and M. van Eekelen. The Concurrent CLEAN language report (draft). Available from `http://www.cs.kun.nl/~clean/`, 2003.
32. R. Pollack. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May 1990.
33. W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, pages 102–114, 1992.
34. W. Taha. A gentle introduction to multi-stage programming, 2003. Available from `http://www.cs.rice.edu/~taha/publications/journal/dspg04a.pdf`.