

# Integrating an Automated Theorem Prover into Agda

Simon Foster and Georg Struth

Department of Computer Science  
University of Sheffield  
`{s.foster,g.struth}@dcs.shef.ac.uk`

**Abstract.** Agda is a dependently typed functional programming language *and* a proof assistant in which developing programs and proving their correctness is one activity. We show how this process can be enhanced by integrating external automated theorem provers, provide a prototypical integration of the equational theorem prover Waldmeister, and give examples of how this proof automation works in practice.

## 1 Introduction

The ideal that programs and their correctness proofs should be developed hand-in-hand has influenced decades of research on formal methods. Specification languages and formalisms such as Hoare logics, dynamics logics and temporal logics have been developed for analysing programs, protocols, and other computing systems. They have been integrated into tools such as interactive and automated theorem provers, SMT solvers and model checkers and successfully applied in the industry. Most of these formalisms do not analyse programs directly on the code, but use external tools and techniques with their own notations and semantics. This usually leaves a formalisation gap and the question remains whether the underlying program semantics has been faithfully captured.

But there are, in fact, programming languages in which the development of a program and its correctness proof can truly be carried out as one and the same activity within the language itself. An example are functional programming languages such as Adga [18, 7] or Epigram [15], which are based on dependent constructive type theory. In these languages, programs development requires proof à la Curry-Howard in the type system — they are therefore, in ingenious ways, programming languages *and* interactive theorem provers — and the type system is expressive enough for specifications. Program development in Agda or Epigram can therefore be based on traditional transformation techniques, on recursion patterns, or on methods from category or allegory theory. Its incrementality and modularity is supported by features such as metavariable refinement and referential transparency. In practice, however, the need of formal proof requires considerable mathematical expertise. This adds yet another layer of complexity on top of programming skills; proofs in Agda and Epigram require substantial user interaction even for trivial tasks. Increasing proof automation in these systems therefore remains a task of crucial importance.

On the one hand, interactive theorem provers such as Isabelle [17] show how such an integration could be achieved. Isabelle is currently being transformed into a versatile proof environment by integrating external automated theorem proving (ATP) systems, SMT solvers, decision procedures and counterexample generators [5, 6, 4]. Proof tasks can be delegated to these tools, and the proofs they provide are internally reconstructed to increase their trustworthiness. On the other hand, state-of-the art ATP systems and SMT solvers are all based on classical logic, whereas Agda and Epigram use constructive logic, and an ATP integration must be achieved as part of the type checking process. This makes an integration certainly more involved, but fortunately not impossible.

This paper provides an ATP integration into Agda. To avoid at least some of the complications we restrict ourselves to pure equational logic, where the distinction between classical and constructive proofs vanishes, and integrate Waldmeister [10], which is the fastest ATP system in the world for this class. In addition, Waldmeister provides detailed proof output and is one of the few ATP systems which supports sorts/types. Our main contributions are as follows.

- We implement the basic data-types for representing equational theories within Agda. Since Agda needs to manipulate these objects during the type checking process, a reflection layer is needed for the implementation.
- Since Agda provides no means for executing external programs before compile time, the reflection-layer theory data-types are complemented by a Haskell module which interfaces with Waldmeister.
- We implement equational logic (and simple term rewriting) at Agda's reflection layer together with functions that parse Waldmeister proof outputs into reflection layer proof terms. We verify this logic within Agda and link it with the level of Agda proofs. This allows us to reconstruct Waldmeister proofs step-by-step within Agda.
- Mapping Agda types into Waldmeister's simple sort system requires abstraction. However Waldmeister may perform proof steps which do not respect Agda's types. Therefore invalid proofs are caught by the reflection layer types during proof reconstruction.
- We provide a series of examples from algebra and functional programming that show the integration at work.

While part of the integration is specific to Waldmeister, most of the concepts implemented are generic enough to serve as a framework for integrating other, more expressive ATP systems. Our integration can also be used as a prototype for further optimisation, for instance, by providing more efficient data structures for terms, equations and proofs, and by improving the running time of proof reconstruction. Such issues are further addressed in the final section of this paper.

Formal program development can certainly be split into creative and routine tasks. Our integration aims at empowering programmers to perform proofs at the level of detail they desire, thus making program development a cleaner, faster and less error-prone process.

The complete code for our implementation can be found at a website<sup>1</sup>.

<sup>1</sup> <http://simon-foster.staff.shef.ac.uk/agdaatp>

## 2 Agda as a Programming Language

Agda [18] is a dependently typed programming language and proof-assistant. It is strongly inspired by the functional programming language Haskell and offers a similar syntax. In this section we briefly introduce Agda as a programming language, whereas the next section focusses on theorem proving aspects.

The data-types featured here section are taken from the standard library<sup>2</sup> and they will reappear throughout. The following inductive data-type declaration introduces the natural numbers.

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

The data-type  $\mathbb{N}$  has type `Set`, which is the set of all simple types. Its constructors `zero` and `suc` are defined as usual. In contrast to languages like Haskell, Agda also supports *dependent* data-types.

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)
```

this defines the data-type of vectors depending on their length  $n$ . In Agda syntax the parameters before the colon are *constants*, whose values cannot be changed by the constructors. Parameters after the colon are *indices*; their definition depends on the particular constructor. In this example, the element type  $A$  of a vector is fixed, whereas the size varies.

Vectors have two constructors: The empty vector `[]` has type `Vec A zero` and zero length. The operation `::` (cons) takes, for each  $n$ , an element  $x : A$  and a vector  $xs : \text{Vec } A \ n$  of length  $n$ , and yields a vector `Vec A (suc n)` of length  $n + 1$ . Instances of this data-type need not explicitly give the parameter  $n$ . It can be inferred by Agda; *hidden* parameters are indicated by braces.

We can now define inductive functions in the obvious way.

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
head : ∀ {n} {A : Set} → Vec A (1 + n) → A
head (x :: xs) = x
tail : ∀ {n} {A : Set} → Vec A (1 + n) → Vec A n
tail (x :: xs) = xs
```

Agda will only accept *total functions*, but `head` and `tail` should only be defined for non-zero length vectors. The dependent type declaration captures this constraint. It thus allows a fine control of data validity.

Predicates can also be defined as data-types:

<sup>2</sup> <http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.StandardLibrary>

```

data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n} → zero ≤ n
  s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n

```

The expressions  $z \leq n$  and  $s \leq s$  are *names* and not definitions. We will adopt this convention throughout the paper. The elements of this data-type are *inductive proofs* of the  $\leq$  relation. For instance,  $s \leq s$  ( $s \leq s \ z \leq n$ ) is a proof of  $2 \leq 3$ . In Agda, therefore, data-types can capture proofs as well as objects such as numbers or vectors.

Agda provides two main definitions of equality. The most important is *propositional equality* which hold when two values (and their types) have the same normal forms.

```

data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x

  sym  : ∀ {A : Set} (x y : A) → x ≡ y → y ≡ x
  trans : ∀ {A : Set} (x y z : A) → x ≡ y → y ≡ z → x ≡ z
  cong  : ∀ {A : Set} {B : Set} (f : A → B) {x y} → x ≡ y → f x ≡ f y

```

By this declaration reflexivity is the only constructor required (due to normalisation). Symmetry, transitivity and congruence are derivable. The elements of this type are again proofs, for example `refl` is a proof of  $1 + 1 \equiv 2$  by normalisation.

In addition, Agda implements *heterogeneous equality*,  $\cong$ , which requires equality of values, but not of types. Two vectors  $xs : \text{Vec } A \ (m + n)$  and  $ys : \text{Vec } A \ (n + m)$  have different types in Agda, hence  $xs \equiv ys$  is not well typed. But  $xs \cong ys$  would hold if  $xs$  and  $ys$  have same normal form. Heterogeneous equality is, again, a congruence; it is important for the implementations in this paper.

The final important feature of Agda we mention is *existential quantification*. We illustrate this using the following *record type*.

```

record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj1 : A
    proj2 : B proj1
  ∃ : ∀ {A : Set} → (A → Set) → Set
  ∃ = Σ _

```

Existentials are introduced by the ubiquitous *sigma type*, representing a dependent product type. It is dependent because the type of the second field takes the value of the first field as a parameter. Specifically, the projection `proj2` to the second field applies  $B$  to the result of the first projection `proj1`. Using this record we can define types like  $\exists (\lambda t \rightarrow t < 3)$  representing natural numbers together with a proof that they are less than 3.

### 3 Agda as a Proof-Assistant

As a constructively typed programming language, Agda uses the Curry-Howard Isomorphism to extract programs from proofs. The data-types of the previous

section provide examples of how proofs yield programs for their inhabitants. A central tool for program development by proof is the *meta-variable*; a “hole” in a program which can be instantiated to an executable program by step-wise refinement.

```
greater : ∀ (n : ℕ) → ∃ (λ m → n < m)
greater n = ?
```

In this code, ? indicates a meta-variable of the type indicated above. Upon running the type-checker, Agda generates the inference rule

$$\frac{n : \mathbb{N}}{\exists(\lambda m \rightarrow n < m)}$$

An existential is now required to prove the goal from the hypothesis. Agda provides a variety of tools for proof support. If the user invokes the *case-split* command, Agda generates two proof obligations from the inductive definition of natural numbers:

```
greater : ∀ (n : ℕ) → ∃ (λ m → n < m)
greater zero = { } 0
greater (suc n) = { } 1
```

Each contains a meta-variable, which is indicated by the braces and number. The first one requires a value of type  $\exists(\lambda m \rightarrow 0 < m)$ . The second one requires a value of type  $\exists(\lambda m \rightarrow \text{suc } n < m)$  for the parameter `suc n`, assuming  $\exists(\lambda m \rightarrow n < m)$  for the parameter `n`. In the first case, *meta-variable refinement* further splits the goal into two meta-variables.

```
greater zero = { } 0, { } 1
```

Meta-variable 0 has type  $\mathbb{N}$ ; it can be satisfied by any natural. Meta variable 1 has type  $0 < ?0$ , where `?0` refers to the first meta-variable. To satisfy this we must prove that meta-variable 0 is greater than zero. The following code displays a value and proof for these conditions:

```
greater zero = 1, s ≤ s z ≤ n
```

This proof style lends itself naturally to incremental program construction, where writing a program and proving its correctness are one activity. To further automate it, Agda includes the proof-search tool **Agsy** [14], which can sometimes automatically construct programs and proofs. The remaining proof goal in the example above can be solved automatically by calling `Agsy`.

```
greater (suc n) = (suc (proj1 (greater n)), s ≤ s (proj2 (greater n)))
```

However, `Agsy` struggles with more complicated proof goals. Increasing the degree of proof automation in Agda seems therefore desirable to further relieve programmers from trivial proof and construction tasks.

## 4 Integration of Automated Theorem Proving

Automated theorem proving systems (ATP systems) have already significantly increased proof automation in interactive theorem provers. Isabelle [17], for instance, uses a tactic called Sledgehammer to call a number of external ATP systems as an alternative to its more traditional tactics. In some applications, almost all proofs can be automated and core Isabelle is merely needed for proof management. In addition feature, Isabelle *reconstructs* all proofs provided by the external ATPs internally in order to increase trust-worthiness. It is evident that proof construction in Agda could greatly benefit from such an integration, but all state of the art ATP systems are designed for classical predicate logic. We have therefore decided to base a first integration on *Waldmeister* [10], an ATP system for pure equational logic, where this difference disappears. The integration is nevertheless generic enough to serve as a basis for full first-order ATP systems, which could still be used on subclasses of constructive formulae, such as Harrop formulae [9].

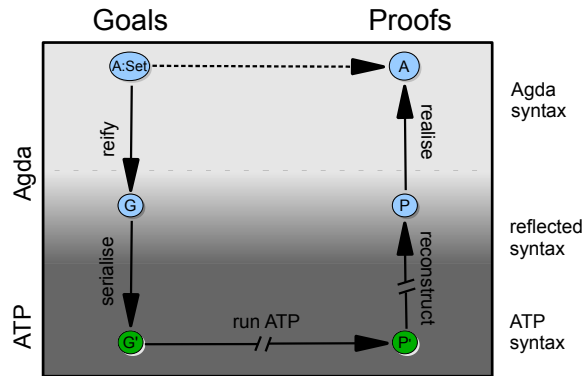


Fig. 1. Overview of automated theorem prover integration in Agda.

Nevertheless, an integration is still not straightforward, for two main reasons. Firstly, the builtin Agda normaliser tends to internally mangle hypotheses and thus counteracts the external ATP process and its reconstruction. Secondly, for proof reconstruction, Agda syntax must explicitly be manipulated within Agda itself, which requires an internal *reflection* layer. The full integration therefore requires the three layers illustrated in Figure 1.

- The *Agda layer* contains the initial proof goal and realises the final proof.
- The *reflection layer* reifies the Agda goal in a format that can be manipulated by Agda, and it reconstructs the ATP proof output.
- The *ATP layer* runs the serialised proof goal and outputs a proof in its native format.

The reification and realisation within Agda depends much on the expressivity of Agda’s reflection layer, which is still experimental. Currently we can reify and realise a large class of equational problem specifications and proofs. The serialisation of the reflected proof input into an ATP input is obtained by a Haskell module. It requires abstraction because Agda’s type system is much more powerful than the simple sorts supported by Waldmeister. In general, types can often be encoded as predicates in ATP systems. State of the art ATP systems can prove quite complex mathematical theorems but they often fail. The integration must be able to cope with this situation. The same holds for proof reconstruction. Ultimately, if Agda succeeds in realising a proof of an initial proof goal, it is guaranteed that this proof is correct in Agda.

All proof goals  $A$  reside in  $\text{Set}$ , the sort of all Agda types and their proofs will be inhabitants of  $A$ . Goals and proofs at the reflection layer have types  $G$  and  $P$ . These are matched by goal and proof types,  $G'$  and  $P'$ , of the ATP systems.

Proof reconstruction can be obtained in different ways. Isabelle’s Sledgehammer tactic, for instance, uses the internally verified ATP system Metis [11] to perform proof search on the hypothesis provided by the external ATP systems. Experience shows that this *macro-step* verification can frequently be a bottleneck. We rather prefer *micro-step* verification, and check each ATP proof step in two stages in Agda. First, we use a Haskell module which parses the ATP output and builds an Agda term representing this unverified proof. Second, we apply an interpreter to this proof which will type-check if it is correct.

## 5 Proof Cycle Example

Assume an Agda proof requires showing the associativity of addition in the natural numbers, which requires induction.

$$\text{assoc} : \forall (x\ y\ z : \mathbb{N}) \rightarrow (x + y) + z \equiv x + (y + z)$$

We first perform a case-split on the first argument, yielding two meta-variables.

$$\begin{aligned} \text{assoc zero } y\ z &= \{ \} 0 \\ \text{assoc (suc } n) y\ z &= \{ \} 1 \end{aligned}$$

We can solve each of these goals by equational reasoning with Waldmeister. Within Agda, the proof goals must first be reified to the reflection layer, using a reflected signature  $\Sigma\text{-Nat}$  for natural numbers and their operations. The reflected axioms and the reflected proof goal are then represented as follows:

```

Nat : HypVec
Nat = HyVec  $\Sigma\text{-Nat}$  axioms
  where
    +-zero =  $\Gamma 1, '0' + \alpha \approx \alpha$ 
    +-suc  =  $\Gamma 2, 'suc\ \alpha' + \beta \approx 'suc\ (\alpha' + \beta)$ 
    axioms = (+-zero :: +-suc :: [])
assoc-zero : Nat, [],  $\Gamma 2 \vdash [] \Rightarrow ('0' + \alpha)' + \beta \approx '0' + (\alpha' + \beta)$ 
assoc-zero = ?

```

```

assoc-suc : Nat, [], Γ3 ⊢ (α' + β)' + γ ≈ α' + (β' + γ) :: []
           ⇒ ('suc α' + β)' + γ ≈ 'suc α' + (β' + γ)
assoc-suc = ?

```

The syntactical details in this encoding shall not concern us now. But they provide sufficient information for generating a Waldmeister input file for the first proof obligation:

```

NAME          agdaProof
MODE          PROOF
SORTS         Nat
SIGNATURE     zero:  -> Nat
              suc:  Nat -> Nat
              plus: Nat Nat -> Nat
              a:   -> Nat
              b:   -> Nat
ORDERING      LPO
              a > b > zero > suc > plus
VARIABLES    x: Nat
              y: Nat
EQUATIONS     plus(zero,x) = x
              plus(suc(x),y) = suc(plus(x,y))
CONCLUSION    plus(plus(zero,a),b) = plus(zero,plus(a,b))

```

In this translation abstractions have been made. Different types are enumerated as disjoint sorts. Functions and datatype constructors are enumerated as operations. Variables in the goal and inductive hypotheses are converted to constants (nullary operations).

Waldmeister instantly returns with the following equational proof:

Consider the following set of axioms:

```
Axiom 1: plus(zero,x1) = x1
```

This theorem holds true:

```
Theorem 1: plus(plus(zero,a),b) = plus(zero,plus(a,b))
```

Proof:

```
Theorem 1: plus(plus(zero,a),b) = plus(zero,plus(a,b))
```

```

plus(plus(zero,a),b)
=   by Axiom 1 LR at 1 with {x1 <- a}
   plus(a,b)
=   by Axiom 1 RL at e with {x1 <- plus(a,b)}
   plus(zero,plus(a,b))

```

Non-trivial proofs can easily have hundreds of steps. They contain sufficient information to reconstruct them step-by-step within Agda. They give the axioms

used, their orientation, the position where they are applied and the associated substitution.

For the reconstruction, this proof has now to be retranslated into a proof term at the reflection layer, where the function `reconstruct` is called. The following code represents the reflection layer proof type which is inhabited by the reconstruction proof.

```

assoc-zero : Nat, [], Γ2 ⊢ [] ⇒ ('0' + α) + β ≈ '0' + (α + β)
assoc-zero =
  fromJust (reconstruct ((inj1 (# 0)
    , true
    , eq-step (0 ::! []!) (con (# 3) ([ ]x) ::s [ ]s))
    ::! (inj1 (# 0)
    , false
    , eq-step ([ ]!) ( con (# 2) (con (# 3) ([ ]x)
      ::x con (# 4) ([ ]x) ::x [ ]x) ::s [ ]s))
    ::! [ ]!))

```

The function `reconstruct` uses reflection layer inference rules for equation logics. It returns a proof within a few seconds. Finally, to realise this proof at the Agda layer, we need to translate the reflection layer terms back into natural numbers. This is achieved by the following functions:

```

N-[[Σ]] : [[Signature]] Σ-Nat
N-[[Σ]] = SemSig (λ x → N) fs
  where fs : (i : Fin 3) → _
    fs zero = 0
    fs (suc zero) = suc
    fs (suc (suc zero)) = _+_
    fs (suc (suc (suc ()))) = _+_
N-Nat : [[HypVec]] Nat N-[[Σ]]
N-Nat = SemHypVec ((SemEq (λ ρ → refl)) ::f ((SemEq (λ ρ → refl)) ::f [ ] f))

```

The first function links the reflection layer signature to concrete Agda functions. The second grounds each of the natural number axioms. This then allows us to complete the proof cycle and finally instantiate the meta-variable into an Agda layer proof.

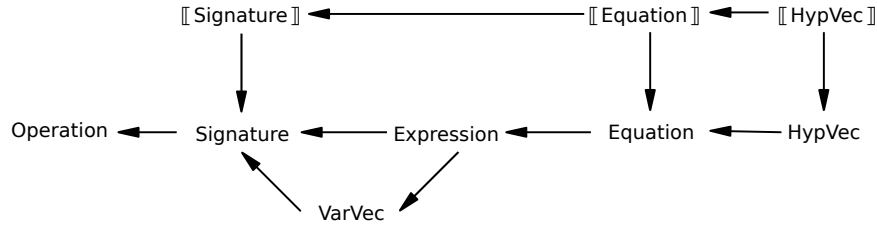
```

assoc zero y z = ≅-to-≡ (⊢≈-to-≅ _ [ ] ⊢ {[[Σ]] = add-∃-vars-[[Σ]] {Γ =
  Γ2}N-TermModel (y, z, tt)} N-Nat assoc-zero [ ] f (λ x → z))

```

## 6 Reflection Layer: Theory Representation

As discussed in Section 4, Agda data-types and proofs must be reified into a reflection layer to enable their manipulation within Agda. This section shows how data-types and theories that enable external ATP proofs can be implemented at the reflection layer. The next section presents our implementation of proof representations.



**Fig. 2.** Dependency graph of reflective components

A key part of our integration is that the representation of Waldmeister files and proofs is type-safe – a successfully reconstructed proof can be interpreted in any model. Figure 2 shows the different data-types required. They essentially provide the reflection layer syntax for the terms and equations used in the proofs and the interpretations of these objects within Agda. The reflection layer syntax is shown at the bottom of the diagram; the interpretations at the top. We will now describe each of them. For our Waldmeister integration we focus only on *equational reasoning* and do not need to represent any syntax beyond that level.

The most basic objects of our reflection layer syntax are *operations* and *signatures*. Operations can either be Agda functions or data-type constructors; we need not distinguish them at this point.

```

record Operation (sn : ℕ) : Set where
  constructor Op
  field
    arity   : ℕ
    args    : Vec (Fin sn) arity
    output  : Fin sn
record Signature : Set where
  constructor Sig
  field
    sortNum : ℕ
    opNum   : ℕ
    operations : Vec (Operation sortNum) opNum
  
```

Fin is the type of finite natural numbers bounded by their size parameter. An operation of arity  $n$  is represented as a record parametrised over the number of sorts provided by the signature in which it is defined. It consists of its *arity*, the sorts of its input arguments *args* and its *output* sort. A signature is a finite number of sorts, each of which is represented by a finite natural number, together with a collection of operations over these sorts.

We also provide a data-type for variables. Their sorts are determined by the signature under which they appear. Therefore, variable vectors are parametrised by signatures. In our context, variables can be implemented as numbers.

```

record VarVec (Σ : Signature) : Set where
  constructor VVec
  open Signature Σ
  field
  
```

```

varNum : ℕ
vvec   : Vec (Fin sortNum) varNum

```

Having defined signatures, operations and variables we can now define terms which are special kinds of *expressions*. Our expression data-type consists of constructors and variables.

```

mutual
data Expr Σ (Γ : VarVec Σ) : VarSet Γ → Sort Σ → Set where
  con : (i : Fin (opNum Σ))
        → {ν : VarSet Γ} (es : ExprVec Γ ν (opArgs Σ i))
        → Expr Σ Γ ν (opOutput Σ i)
  var : (x : Var Γ) → Expr Σ Γ {x} (varSort Γ x)

data ExprVec {Σ : Signature} (Γ : Ctx Σ)
  : ∀ {n} → (vs : VarSet Γ) (ss : Vec (Sort Σ) n) → Set where
  _::x_ : ∀ {ν1} {ν2} {s} {n} {ss : Vec _ n}
        → (e : Expr Σ Γ ν1 s) (es : ExprVec Γ ν2 ss)
        → ExprVec Γ (ν1 ∪ ν2) (s :: ss)
  []x   : ExprVec Γ ∅ []

```

The expression data-type has four parameters.

1. The signature  $\Sigma$ ;
2. the variable context  $\Gamma$  which lists all variables available for building a term;
3. the set of variables  $\nu$  drawn from the context which the expressions contains, represented by shorthand  $\text{VarSet } \Gamma$ ;
4. the sort  $s$  of the expression.

The first constructor, `con`, takes the operation number  $i$ , and an expression vector parametrised over its argument sorts; it yields an expression with the output sort of  $i$ . The second constructor, `var`, takes a variable number and yields an expression with a singleton free variable  $\{x\}$  with the correct sort. The constructor `con` builds expressions from subexpressions, which are given by an expression vector. Expressions and expression vectors are mutually inductive. We also define  $\approx^x$ , which implements syntactic identity on terms.

With this infrastructure in place it is now possible to define equations and hypothesis sets which consist of vectors of equations. As usual, an equation is just a pair of terms of the same sort.

```

record Equation' (Σ : Signature) (Γ : VarVec Σ) : Set where
  constructor _≈_
  field
    {sort} : Sort Σ
    {ν1 ν2} : VarSet Γ
    lhs     : Expr Σ Γ ν1 sort
    rhs     : Expr Σ Γ ν2 sort

Equation : Signature → Set
Equation Σ = ∃ λ Γ → Equation' Σ Γ

record HypVec : Set where
  constructor HyVec

```

```

field
   $\Sigma$       : Signature
  {hypNum}  :  $\mathbb{N}$ 
  hypotheses : Vec (Equation  $\Sigma$ ) hypNum

```

Equation is a variant of Equation' where the variable context is abstracted. It is needed for technical reasons when a vector of equations with different variable contexts is required.

We now move to the upper level of Figure 2 and discuss the data-types that realise the abstract syntactic data-types of the reflection layer within Agda. First we present the Agda level signature together with the semantic functions mapping to it.

```

record [[Signature]] ( $\Sigma$  : Signature) : Set1 where
  constructor SemSig
  open Signature  $\Sigma$ 
  open Operation
  field
    types      : Fin sortNum  $\rightarrow$  Set
    functions  : (i : Fin opNum)  $\rightarrow$  opFunc  $\Sigma$  types i
mutual
  sem :  $\forall$  { $\Sigma$ } ([[ $\Sigma$ ]] : [[Signature]]  $\Sigma$ ) { $\Gamma$  : VarVec  $\Sigma$ } {s} { $\nu$ }
     $\rightarrow$  ([[ $\rho$ ]] : [[Subst]]  $\Gamma$  [[ $\Sigma$ ]]) (e : Expr  $\Sigma$   $\Gamma$   $\nu$  s)
     $\rightarrow$  [[Signature]].types [[ $\Sigma$ ]] s
  sem { $\Sigma$ } [[ $\Sigma$ ]] [[ $\rho$ ]] (con i es)
    = _$n_ {n = arity} (functions i) (sem* [[ $\Sigma$ ]] [[ $\rho$ ]] es)
  where open [[Signature]] [[ $\Sigma$ ]]
    open Operation (getOp  $\Sigma$  i)
  sem [[ $\Sigma$ ]] [[ $\rho$ ]] (var x) = [[ $\rho$ ]] x
  sem* :  $\forall$  { $\Sigma$ } ([[ $\Sigma$ ]] : [[Signature]]  $\Sigma$ ) { $\Gamma$  : VarVec  $\Sigma$ }
    {n} {ss : Vec _ n} { $\nu$ }
     $\rightarrow$  ([[ $\rho$ ]] : [[Subst]]  $\Gamma$  [[ $\Sigma$ ]])  $\rightarrow$  ExprVec  $\Gamma$   $\nu$  ss
     $\rightarrow$  HVec (map ([[Signature]].types [[ $\Sigma$ ]]) ss)
  sem* [[ $\Sigma$ ]] [[ $\rho$ ]] (e ::x es) = sem [[ $\Sigma$ ]] [[ $\rho$ ]] e, sem* [[ $\Sigma$ ]] [[ $\rho$ ]] es
  sem* [[ $\Sigma$ ]] [[ $\rho$ ]] []x = tt

```

The record [[Signature]] maps a reflection layer signature  $\Sigma$  to an Agda layer signature, associating each reflection layer operation with an Agda layer function. The function opFunc interprets the operation as an Agda function type.

The semantic map sem realises reflection layer terms within Agda, using an Agda layer signature [[ $\Sigma$ ]] and substitution [[ $\rho$ ]], the definition of which is omitted.

Each constructor is realised by looking up the corresponding function to each operation i in [[ $\Sigma$ ]], and by applying the list of semantics sem\* for each parameter to this function using the *uncurrying* combinator  $\$^n$ . Variables are interpreted by substitutions [[ $\rho$ ]]. The definitions of sem and sem\* are mutually recursive. The function sem\* produces a heterogeneous vector of expression semantics (tt corresponds to the empty HVec).

Equations at the Agda layer are implemented next.

```

record [[Equation]]'  $\Sigma$   $\Gamma$  [[ $\Sigma$ ]] (hyp : Equation'  $\Sigma$   $\Gamma$ ) : Set1 where
  constructor SemEq
  open Equation' hyp
  field
    valid :  $\forall$  [[ $\rho$ ]]  $\rightarrow$  sem [[ $\Sigma$ ]] [[ $\rho$ ]] lhs  $\equiv$  sem [[ $\Sigma$ ]] [[ $\rho$ ]] rhs

```

This data-type corresponds to a proof that the interpretation of the left-hand side of an equation is propositionally equal to the interpretation of the right-hand side. This means that the equation is valid in Agda. Finally we define hypothesis sets at the Agda layer:

```

record [[HypVec]] A ([[ $\Sigma$ ]] : [[Signature]] (HypVec. $\Sigma$  A)) : Set1 where
  constructor SemHypVec
  open HypVec A
  field
    formulae : [[Equation]]Vec [[ $\Sigma$ ]] axioms

```

This implementation enables us to represent all elements of Waldmeister input files in a well-typed way at the reflection layer, and to realise these elements within Agda. Since Agda cannot run external program before compile-time, we have written a Haskell module which interfaces with Waldmeister. This module contains corresponding data-types for each of the reflection layer data-types described. These are used within a function which serialises a Waldmeister input file, as shown in Section 5, executes the prover and parses the resulting Waldmeister proof output back into Agda. These proofs can then be reconstructed.

## 7 Reflection Layer: Proof Representation

In order to reconstruct Waldmeister proofs within Agda, we must provide data-types for equational proofs at the reflection layer. First, the parsed proof output provided by Waldmeister must be translated into an inhabitant of a proof data-type. Second, it must be proved that all inhabitants of this data-type are correct with respect to propositional equality. This section describes this infrastructure.

At the core of proof verification is the implementation of the basic steps of equational reasoning, as performed by Waldmeister. This requires modelling the application of an equation for rewriting terms into terms, using substitutions.

We need some standard notation and concepts from term rewriting (cf. [13]). As usual, a *substitution* is a map  $\rho$  from variables to terms, which extends to an endofunction on terms. We say that a term  $t$  *matches* a term  $s$  (or  $s$  *subsumes*  $t$ ) if  $s\rho = t$  for some substitution  $\rho$ . We write  $t \sqsubseteq s$  if  $s$  subsumes  $t$ . It is well known that subsumption is a preorder. More specifically, to denote the ternary relation  $s\rho = t$  between  $s$ ,  $\rho$  and  $t$ , we write  $t \sqsubseteq_{\rho} s$ .

Based on subsumption we can now model the primitive steps of equational logic. Let  $E$  be a set of equations  $l_i \approx r_i$ . We write  $E \vdash s = t$  if there exists a substitution  $\rho$ , a context  $C$  and an equation  $l \approx r \in E$  such that  $s = C[l\rho]$  and  $t = C[r\rho]$ . Hence

$$E \vdash s =^1 t \iff s \sqsubseteq_{\rho} C[l] \wedge t \sqsubseteq_{\rho} C[r] \quad (1)$$

for some substitution  $\rho$ , context  $C$  and  $l \approx r \in E$ . Finally, we can extend this one-step equational rewriting to equational proofs with more than one step by inductively defining the relation  $=$  as the transitive closure of  $=^1$ .

To implement these concepts, we first provide a data-type for substitutions.

```
Subst : ∀ {Σ} (Γ1 Γ2 : VarVec Σ) → VarSet Γ2 → Set
Subst Γ1 Γ2 ν = SubstVec Γ1 ν vvec
where open Ctx Γ2
```

Substitutions are partial maps represented as vectors. They are parametrised over two contexts that contain the variables in the target and the source type. Additionally, they carry the set of variables  $\nu$  in their domains, that is where they are defined.

This now allows us to implement the relation  $\sqsubseteq_\rho$ .

```
mutual
data _[_]_ {Σ} {Γ1 Γ2} {ν} (ρ : Subst Γ1 Γ2 ν)
  : ∀ {ν1} {ν2} {s1 s2}
  → Expr Σ Γ1 ν1 s1 → Expr Σ Γ2 ν2 s2 → Set where
var-sbs    : ∀ {x} {ν1} {e : Expr Σ Γ1 ν1 (varSort Γ2 x)}
  → (x ∈ ν : x ∈ ν) (ρ(x) ≈ e : ρ ⟨ x, x ∈ ν ⟩ o ≈ o (ν1, e))
  → ρ [e ⊆ var x]
con-sbs    : ∀ {i j} {ν1 ν2} {es : ExprVec Γ1 ν1 (opArgs Σ i)}
  {fs : ExprVec Γ2 ν2 (opArgs Σ j)}
  → (i ≡ j : i ≡ j) (es ⊆* fs : ρ [es ⊆* fs])
  → ρ [con i es ⊆ con j fs]
data _[_]_*_ -- Code omitted
```

This proof data-type distinguishes a variable case and a constructor case. In the first case, to show that a term  $e$  is subsumed by variable  $x$  under substitution  $\rho$ , we need a proof that  $x \in \nu$ , that is,  $x$  is in the domain of  $\rho$ , and another proof that  $\rho(x) \approx o e$ . The relation  $\approx o$  holds if two terms are identical without considering their sorts at the type level. In the second case, to show that a constructor term subsumes another one under a given substitution, the subsumption relation needs to be established between its subterms, as defined by  $\sqsubseteq^*$ . In term rewriting, the algorithm implementing this proof is known as *matching*.

**Lemma 1.** *Subsumption implies propositional equality.*

$$s \sqsubseteq_\rho t \implies \forall \sigma. \llbracket s \rrbracket \sigma \equiv \llbracket t \rrbracket (\sigma \circ \llbracket \rho \rrbracket).$$

*Proof.* The proof has been automated in Agda, and we only sketch an outline. First, we explain the notation.  $s$ ,  $\rho$  and  $t$  and the entire left hand side are defined at the reflection layer, whereas  $\sigma$  and the entire right hand side are defined at the Agda layer. Since the reflection layer is part of Agda, there is no problem in writing an implication between the two. The substitution  $\sigma$  is used to interpret the free variables in Agda. The proof itself is given by the following function (definition omitted):

```
_[_]-to-≅ : ∀ {Σ} {Γ1 Γ2} {[[Σ]]} {ν1 ν2} {s1 s2}
  → (f : Expr Σ Γ2 ν2 s2) (e : Expr Σ Γ1 ν1 s1)
```

$$\begin{aligned} &\rightarrow (\rho : \text{Subst } \Gamma_1 \Gamma_2 \text{ full}) \rightarrow \rho [e \sqsubseteq f] \\ &\rightarrow (\forall [\sigma] \rightarrow \text{sem } [\Sigma] [\sigma] e \cong \\ &\quad \text{sem } [\Sigma] (\text{sem-subst } \{[\Sigma] = [\Sigma]\} \rho [\sigma]) f) \end{aligned}$$

The function `sem-subst` applies  $\sigma$  to each term instantiated by  $\rho$ . We use heterogeneous equality  $\cong$ , rather than propositional equality because the sorts of expressions on either side of a rewrite may differ. The main step in the proof is the `var-sbs` case which proves that  $e$  is propositionally equal to  $x\rho$  and well-typed. The case for `con-sbs` is schematic.  $\square$

In the next step we implement the relation  $E \vdash s = t$  without considering contexts, that is  $s \sqsubseteq_\rho l \wedge t \sqsubseteq_\rho r$ .

```

data _ $\sqsubseteq$ _  $\sqsupseteq$ _ { $\Sigma$ } { $\Gamma$ } { $\nu$ } { $s_1 s_2$ } (e : Expr  $\Sigma$   $\Gamma$   $\nu$   $s_1$ )
  (u $\approx$ v : Equation  $\Sigma$ ) (f : Expr  $\Sigma$   $\Gamma$   $\nu$   $s_2$ ) : Set where
  - [_ $\times$ _] : let  $\Gamma' = \text{Equation.}\Gamma$  u $\approx$ v;
    u = Equation.lhs u $\approx$ v; v = Equation.rhs u $\approx$ v
    in  $\forall$  ( $\rho$  : Subst  $\Gamma$   $\Gamma'$  full)
       $\rightarrow$  (e $\sqsubseteq$ u :  $\rho$  [e  $\sqsubseteq$  u]) (f $\sqsubseteq$ v :  $\rho$  [f  $\sqsubseteq$  v])
       $\rightarrow$  e  $\sqsubseteq$  u $\approx$ v  $\sqsupseteq$  f

```

**Lemma 2.** *Suppose that  $s \sqsubseteq_\rho l$ ,  $t \sqsubseteq_\rho r$  and  $[[l]]\sigma \equiv [[r]]\sigma$  for all substitutions  $\sigma$ . Then  $[[s]]\sigma \equiv [[t]]\sigma$  for all substitutions  $\sigma$ .*

*Proof.* The proof is given by the following Agda function.

```

 $\sqsubseteq$   $\sqsupseteq$  -to- $\cong$  :  $\forall$  { $\Sigma$ } { $\Gamma$ } { $\nu$ } { $s_1 s_2$ }
  {e : Expr  $\Sigma$   $\Gamma$   $\nu$   $s_1$ } {f : Expr  $\Sigma$   $\Gamma$   $\nu$   $s_2$ }
  {[[ $\Sigma$ ]]} u $\approx$ v ([[u $\approx$ v]] : [[Equation]]  $\Sigma$  [[ $\Sigma$ ]] u $\approx$ v)
   $\rightarrow$  e  $\sqsubseteq$  u $\approx$ v  $\sqsupseteq$  f
   $\rightarrow$  ( $\forall$  {[[ $\rho$ ]]}  $\rightarrow$  sem [[ $\Sigma$ ]] [[ $\rho$ ]] e  $\cong$  sem [[ $\Sigma$ ]] [[ $\rho$ ]] f)

```

Applications of transitivity then connect  $e \cong u\rho \equiv v\rho \cong f$ .  $\square$

Finally, we add contexts to capture  $E \vdash s =^1 t$  within propositional equality.

```

data _ $\vdash$ _  $\approx^1$ _ { $\Sigma$ } { $\Gamma$ } { $\nu_1 \nu_2$ } { $s_1 s_2$ } (u $\approx$ v : Equation  $\Sigma$ )
  (e : Expr  $\Sigma$   $\Gamma$   $\nu_1$   $s_1$ ) (f : Expr  $\Sigma$   $\Gamma$   $\nu_2$   $s_2$ ) : Set where
   $\vdash \approx^1$  -cons :  $\forall$  { $\nu e'$ } { $\nu f'$ } { $\nu g$ } { $s$ } { $sg$ } (g : Expr  $\Sigma$  (freshen- $\Gamma$  s  $\Gamma$ )  $\nu g$   $sg$ )
    (e' : Expr  $\Sigma$   $\Gamma$   $\nu e'$  s) (f' : Expr  $\Sigma$   $\Gamma$   $\nu f'$  s)
     $\rightarrow$  (e $\sqsubseteq$ g : singleton- $\rho$  e' [e  $\sqsubseteq$  g]) (f $\sqsubseteq$ g : singleton- $\rho$  f' [f  $\sqsubseteq$  g])
     $\rightarrow$  (e' $\sqsubseteq$ f' : e'  $\sqsubseteq$  u $\approx$ v  $\sqsupseteq$  f')
     $\rightarrow$  u $\approx$ v  $\vdash$  e  $\approx^1$  f

```

This data-type defines that two terms  $e$  and  $f$  under equation  $u \approx v$  can be proved equal if there exists a common context  $g[X]$  of both, with  $e = g[e']$  and  $f = g[f']$ , such that  $e' \sqsubseteq_\rho u$  and  $f' \sqsubseteq_\rho v$ . The variable of  $g$ , and variables of  $e$  and  $f$  are the same, up to an additional fresh variable in  $g$  added by `freshen- $\Gamma$`  to represent the context's hole. A proof step then requires that both  $e$  and  $f$  are subsumed by  $g$  under the substitution which replaces its hole with  $e'$  and  $f'$  respectively. This substitution is implemented by function `singleton- $\rho$`  which maps the first variable to  $e'$  or  $f'$  and leaves the remainder identical.

The corresponding proof of validity is similar to those in Lemma 1 and Lemma 2.

**Proposition 1.** *One-step rewriting implies propositional equality*

$$u \approx v \vdash s = t \implies \forall \rho. \llbracket s \rrbracket \sigma \equiv \llbracket t \rrbracket \sigma.$$

*Proof.* The proof is given by the following Agda function.

$$\begin{aligned} \Rightarrow \approx^1\text{-to-}\cong & : \forall \{ \Sigma \} \{ \Gamma \} \{ \nu_1 \nu_2 \} \{ s_1 s_2 \} \\ & \{ e : \text{Expr } \Sigma \Gamma \nu_1 s_1 \} \{ f : \text{Expr } \Sigma \Gamma \nu_2 s_2 \} \\ & \{ \llbracket \Sigma \rrbracket \} (u \approx v : \text{Equation } \Sigma) (\llbracket u \approx v \rrbracket : \llbracket \text{Equation} \rrbracket \Sigma \llbracket \Sigma \rrbracket u \approx v) \\ & \rightarrow u \approx v \Rightarrow e \approx^1 f \\ & \rightarrow (\forall \llbracket \rho \rrbracket \rightarrow \text{sem } \llbracket \Sigma \rrbracket \llbracket \rho \rrbracket e \cong \text{sem } \llbracket \Sigma \rrbracket \llbracket \rho \rrbracket f) \end{aligned}$$

It uses the previous lemmas and, approximately, the pseudo code steps

$$\llbracket e \rrbracket \rho \cong \llbracket g \rrbracket (\rho \cup \{x \mapsto e'\}) \cong \llbracket g \rrbracket (\rho \cup \{x \mapsto f'\}) \cong \llbracket f \rrbracket \rho.$$

□

Finally, we implement the n-step equality  $E \vdash s = t$ .

$$\begin{aligned} \mathbf{data} \ \_ \approx \_ \vdash \_ \approx \_ & \text{ E } \{ n \} (L : \text{EquationVec } (\Sigma \text{ E}) n) \Gamma \\ & : \forall \{ s_1 s_2 \} \{ \nu_1 \nu_2 \} \\ & \rightarrow \text{Expr } (\Sigma \text{ E}) \Gamma \nu_1 s_1 \\ & \rightarrow \text{Expr } (\Sigma \text{ E}) \Gamma \nu_2 s_2 \rightarrow \text{Set}_1 \mathbf{where} \\ \vdash \approx \text{-app} & : \forall \{ \nu_1 \nu_2 \nu_3 \} \{ s \} \{ e : \text{Expr } (\Sigma \text{ E}) \Gamma \nu_1 s \} \\ & \{ e' : \text{Expr } (\Sigma \text{ E}) \Gamma \nu_2 s \} \{ f : \text{Expr } (\Sigma \text{ E}) \Gamma \nu_3 s \} \\ & \rightarrow (\text{ap} : \text{Fin } (\text{hypNum } \text{E}) \uplus \text{Fin } n) \\ & \rightarrow (\text{lr} : \text{Bool}) \\ & \rightarrow (e \approx^1 e' : \text{orientEq } \text{lr } (\text{getEq } \text{E } \text{L } \text{ap}) \vdash e \approx^1 e') \\ & \rightarrow (e' \approx f : \text{E, L, } \Gamma \vdash e' \approx f) \\ & \rightarrow \text{E, L, } \Gamma \vdash e \approx f \\ \vdash \approx \text{-refl} & : \forall \{ \nu \} \{ s \} \{ e : \text{Expr } (\Sigma \text{ E}) \Gamma \nu s \} \\ & \rightarrow \text{E, L, } \Gamma \vdash e \approx e \end{aligned}$$

In fact, the implementation is slightly more complex because we implement the relation  $E, L \vdash s = t$ , where  $L$  is a set of hypothesis different from the axioms which we usually place in  $E$ . We now discuss the constructors  $\vdash \approx \text{-app}$  and  $\vdash \approx \text{-refl}$  separately.

The main constructor,  $\vdash \approx \text{-app}$ , represents an application of an axiom or lemma. The equation  $l \approx r$ , numbered by  $\text{ap}$ , can be drawn either from the main axiom set  $E$  or the lemma set  $L$ . It can be applied left-right,  $l \rightarrow r$  or right-left  $r \rightarrow l$ , indicated by boolean  $\text{lr}$ . The data-type then requires a proof that  $e$  rewrites to  $e'$  in one step by using  $l \approx r$  in the way indicated.

The constructor  $\vdash \approx \text{-refl}$  implements reflexivity, that is the base case  $E, L \vdash s =^0 t$  which holds only when  $s$  and  $t$  are identical.

**Theorem 1.** *n-step rewriting implies propositional equality.*

$$E, L \vdash s = t \implies \llbracket E \rrbracket, \llbracket L \rrbracket \models \llbracket s \rrbracket \equiv \llbracket t \rrbracket$$

*Proof.* In the statement of the theorem, the previous quantification over a substitution  $\sigma$  has been absorbed into the  $\models$  relation. The proof is represented by the following function:

$$\begin{aligned} \vdash \approx\text{-to-}\cong & : \forall \ E \ \{n\} \ \{L : \text{Vec} \ (\text{Equation} \ (\Sigma \ E)) \ n\} \ \{\llbracket \Sigma \rrbracket\} \ \{\Gamma\} \ \{\nu_1 \ \nu_2\} \ \{s_1 \ s_2\} \\ & \ \{e : \text{Expr} \ (\Sigma \ E) \ \Gamma \ \nu_1 \ s_1\} \ \{f : \text{Expr} \ (\Sigma \ E) \ \Gamma \ \nu_2 \ s_2\} \\ & \rightarrow E, L, \Gamma \vdash e \approx f \rightarrow \llbracket \text{HypVec} \rrbracket \ E \ \llbracket \Sigma \rrbracket \rightarrow \llbracket \text{Equation} \rrbracket \text{Vec} \ \llbracket \Sigma \rrbracket \ L \\ & \rightarrow (\forall \ \rho \rightarrow \llbracket \Sigma \rrbracket \models \rho \llbracket e \rrbracket e \cong \llbracket \Sigma \rrbracket \models \rho \llbracket f \rrbracket e) \end{aligned}$$

Proving that  $e$  is equal to  $f$  modulo the given rewrite proof requires an Agda layer grounding of the hypothesis set. Any additional lemmas also need corresponding equation groundings. We can then obtain a proof that the  $e \cong f$  under any valid substitution  $\rho$  by recursive application of Proposition 1.  $\square$

The theorems in this section provide the formal underpinning for proof reconstruction.

## 8 Proof Reconstruction

Proof reconstruction is the process of taking a proof output from an external ATP system, in this case Waldmeister, and verifying it internally in the proof environment, in this case Agda. In our context it is mandatory as part of the type-checking process. In this section, we discuss the implementation of an Agda function `reconstruct` which executes a Waldmeister proof and builds a reflection layer proof if successful. By the results of the previous section, this proof is *a fortiori* correct within Agda. Proof reconstruction, therefore, completes the proof cycle indicated in Figure 1.

Proof reconstruction can fail as we are relying on Waldmeister providing a proof in our expected domain. If, for instance, Waldmeister introduces a new constant which has not been accounted for in Agda, reconstruction cannot proceed. Moreover, due to abstracting Agda types into Waldmeister sorts, Waldmeister proofs may not always be type correct in Agda. In most cases though, reconstruction does succeed as equational reasoning is sound with respect to constructive type-theory.

A Waldmeister proof consists of a list of equational steps, each augmented by an axiom number, a term position at which the axiom is applied, the orientation of the axiom (left-right or right-left), and the substitution used. All these features have been implemented at the reflection layer in Section 6. Execution of an individual rewrite from term  $e$  to term  $f$  proceeds in two stages, which correspond to the definition in Equation (1):

Assume the equational proof step  $e \stackrel{1}{=} f$  is obtained by applying the equation  $u \approx v$  under the context  $g$  of  $e$ , and using substitution  $\rho$ .

1. Use the term position to split  $e = g[e']$ , supported by subsumption.
2. Check whether  $e' \sqsubseteq_{\rho} u$ . If so, replace  $e'$  by  $v\rho$  in context  $g$ .

The first stage is executed by the function `build-split`.

```

data Split {Σ} {Γ} {νe} {s} (e : Expr Σ Γ νe s) : Set where
  split : ∀ {νg} {νe'} {se'}
    (g : Expr Σ (freshen-Γ se' Γ) νg s) (e' : Expr Σ Γ νe' se')
    (e ⊆ g : singleton-ρ e' [e ⊆ g]) → Split e
  Pos : Set
  Pos = List ℕ
  build-split : ∀ {Σ} {Γ} {ν} {s} (e : Expr Σ Γ ν s) → Pos → Maybe (Split e)

```

A `Split` consists of a context term  $g[x]$  together with a subterm  $e'$  and a proof that  $g[x]$  subsumes  $e$  under the singleton substitution  $x \mapsto e'$ . The function `build-split` takes a term  $e$  and a subterm position; it attempts to extract this triple. The `Maybe` type represents this partiality; it is populated either by `just x`, where  $x$  has the type of its parameter, or else `nothing`.

The second stage is executed by the following function.

```

build-⊢≈1 : ∀ {Σ} {Γ} {ν} {s} (e : Expr Σ Γ ν s) u≈v
  (ρ : Subst Γ (proj1 u≈v) full) → Split e
  → Maybe (∃ (λ ν' → ∃ (λ (f : Expr Σ Γ ν' s) → u≈v ⊢ e ≈1 f))

```

This function takes an equation, substitution and split term, and it uses them to perform the rewriting to  $f$ . The decision procedure for  $e' \sqsubseteq_{\rho} u$  is provided by

```

dec-⊆ : ∀ {Σ} {Γ1 Γ2} {ν1 ν2 ν} {s1 s2}
  → (ρ : Subst Γ1 Γ2 ν) (e : Expr Σ Γ1 ν1 s1) (f : Expr Σ Γ2 ν2 s2)
  → ν2 ⊆ ν → Maybe (ρ [e ⊆ f])

```

Both functions are then used together in the main function `reconstruct`:

```

reconstruct : ∀ {E : HypVec} {Γ} {ν1 ν2} {s}
  {n} {L : Vec (Equation (Σ E)) n}
  → {e : Expr (Σ E) Γ ν1 s} {f : Expr (Σ E) Γ ν2 s}
  → EqProof E Γ L
  → Maybe (E, L, Γ ⊢ e ≈ f)

```

The type `EqProof` represents the raw input from Waldmeister which has been reformatted by the Haskell module. Again, it is a list of quadruples consisting of the axiom/lemma number to be applied, the orientation, the subterm position and the substitution. A Waldmeister proof output may have been subdivided into a number of lemmas and these are currently flattened out to produced a single sequence of rewrites. The `reconstruct` function simply applies the one-step reconstruction functions iteratively to yield the complete reflection layer proof.

## 9 Examples

This section shows the Waldmeister integration at work. Apart from some proofs about natural number, an example of which has been given in Section 5, we have considered basic proofs from group theory and Boolean algebras, and some simple proofs about lists.

The following code implements the axioms of group theory. The signature definition should be clear from the natural number example and the code in Section 5. It is therefore omitted.

```

Group : HypVec
Group = HyVec  $\Sigma$ -Group axioms
  where
    assoc =  $\Gamma 3, (\alpha \cdot \beta) \cdot \gamma \approx \alpha \cdot (\beta \cdot \gamma)$ 
    ident =  $\Gamma 1, e \cdot \alpha \approx \alpha$ 
    inv   =  $\Gamma 1, \alpha^{-1} \cdot \alpha \approx e$ 
    axioms = (assoc :: ident :: inv :: [])

```

We can now automatically prove some basic facts, e.g., that the left identity is also a right identity, that the right inverse is also left inverse and that the inverse is involutive and contravariant. The first fact is an auxiliary lemma.

```

ident-var : Group, [],  $\Gamma 2 \vdash \alpha^{-1} \cdot (\alpha \cdot \beta) \approx \beta$ 
ident-var = fromJust (reconstruct ((inj1 (# 0), false, eq-step ([ ]1)
  (con (# 2) (var (# 0) ::x [ ]x) ::s var (# 0) ::s var (# 1)
  ::s [ ]s)) ::! (inj1 (# 2), true, eq-step (0 ::! [ ]!) (var
  (# 0) ::s [ ]s)) ::! (inj1 (# 1), true, eq-step ([ ]!) (var
  (# 1) ::s [ ]s)) ::! [ ]!))

```

The first line states that a certain equation follows from the group axioms, with no additional hypotheses and a two variable context. The second line shows how the Waldmeister proof output, parsed into Agda, is reconstructed. The function `fromJust` lifts a `Maybe A` type to an `A` type in the case that the proof is successfully reconstructed, otherwise the proof does not type-check.

This lemma can now be used to proof the other facts mentioned.

```

rident : Group, ( $\vdash \approx$ -eq ident-var :: []),  $\Gamma 1 \vdash \alpha \cdot e \approx \alpha$ 
rident = -- proof omitted

rinv : Group, ( $\vdash \approx$ -eq rident ::  $\vdash \approx$ -eq ident-var :: []),  $\Gamma 1 \vdash \alpha \cdot \alpha^{-1} \approx e$ 
rinv = -- proof omitted

invol : Group, ( $\vdash \approx$ -eq rinv ::  $\vdash \approx$ -eq rident ::  $\vdash \approx$ -eq ident-var :: []),
   $\Gamma 1 \vdash (\alpha^{-1})^{-1} \approx \alpha$ 
invol = -- proof omitted

contrav : Group, ( $\vdash \approx$ -eq invol ::  $\vdash \approx$ -eq rinv ::  $\vdash \approx$ -eq rident ::  $\vdash \approx$ -eq ident-var :: []),
   $\Gamma 2 \vdash (\alpha \cdot \beta)^{-1} \approx \beta^{-1} \cdot \alpha^{-1}$ 
contrav = -- proof omitted

```

On these very simple examples, Waldmeister returned almost instantaneously. Proof reconstruction required several seconds. All these proofs are at the reflection layer.

Proofs in Boolean algebra are more complex, and proof-search is more involved.

```

BA : HypVec
BA = HyVec  $\Sigma$ -BA axioms
  where
    +-assoc =  $\Gamma 3, \alpha + (\beta + \gamma) \approx (\alpha + \beta) + \gamma$ 
    --assoc =  $\Gamma 3, \alpha \cdot (\beta \cdot \gamma) \approx (\alpha \cdot \beta) \cdot \gamma$ 
    +-comm =  $\Gamma 2, \alpha + \beta \approx \beta + \alpha$ 

```

```

--comm  =  $\Gamma 2, \alpha \cdot \beta \approx \beta \cdot \alpha$ 
distr1  =  $\Gamma 3, \alpha + (\beta \cdot \gamma) \approx (\alpha + \beta) \cdot (\alpha + \gamma)$ 
distr2  =  $\Gamma 3, \alpha \cdot (\beta + \gamma) \approx (\alpha \cdot \beta) + (\alpha \cdot \gamma)$ 
+-id    =  $\Gamma 1, \alpha + \perp \approx \alpha$ 
-id     =  $\Gamma 1, \alpha \cdot \top \approx \alpha$ 
comp1   =  $\Gamma 1, \alpha + \alpha' \approx \top$ 
comp2   =  $\Gamma 1, \alpha \cdot \alpha' \approx \perp$ 
axioms  = (+-assoc :: -assoc :: +-comm :: --comm :: distr1
           :: distr2 :: +-id :: -id :: comp1 :: comp2 :: [])

```

The following proofs could all be reconstructed.

```

idem1 : BA, [],  $\Gamma 1 \vdash \alpha + \alpha \approx \alpha$ 
idem1 = -- proof omitted
 $\top 1$  : BA, [],  $\Gamma 1 \vdash \alpha + \top \approx \top$ 
 $\top 1$  = -- proof omitted
 $\perp 1$  : BA, [],  $\Gamma 1 \vdash \alpha \cdot \perp \approx \perp$ 
 $\perp 1$  = -- proof omitted
double-comp : BA, [],  $\Gamma 1 \vdash \alpha' \approx \alpha$ 
double-comp = -- proof omitted
demorgan1 : BA, [],  $\Gamma 2 \vdash (\alpha + \beta)' \approx \alpha' \cdot \beta'$ 
demorgan1 = -- proof omitted

```

In all cases, Waldmeister returned within seconds. But, due to splitting equational proofs into lemmas, the reflection layer proof terms tend to become very long, and their reconstruction can take several minutes. There are some other theorems of Boolean algebra that Waldmeister could easily verify, but where proof reconstruction failed, e.g., when Waldmeister chose to introduce new undeclared constants for non-obvious reasons.

As a final example we show the mother of all verification proofs for functional programs, namely that the reverse of the reverse of a list yields back that list. This example is especially interesting for two reasons. First, lists are two sorted structures and it is shown that Waldmeister can handle this situation. Second, the proof requires induction, which is beyond first-order logic and cannot directly be executed by Waldmeister. However, it can handle the purely equational reasoning required in the base case and the induction step.

```

infixl 15 _'::_
infixl 20 _'++_
_[] : Op-0  $\Sigma$ -List (# 1)
_[] = con (# 0) [] x
_''::_ : Op-2  $\Sigma$ -List (# 0) (# 1) (# 1)
x'':: xs = con (# 1) (binExprVec x xs)
_'++_ : Op-2  $\Sigma$ -List (# 1) (# 1) (# 1)
xs'+ ys = con (# 2) (binExprVec xs ys)
_rev : Op-1  $\Sigma$ -List (# 1) (# 1)
_rev xs = con (# 3) (unaryExprVec xs)
_List : AxiomVec

```

```

‘List = AlgSt Σ-List axioms
  where
    +-nil = Γ1, ‘[] ⊢ α ≈ α
    +-cons = Γ3, (α :: β) ⊢ γ ≈ α :: (β ⊢ γ)
    rev-nil = Γ0, ‘rev [] ≈ []
    rev-cons = Γ2, ‘rev (α :: β) ≈ ‘rev β ⊢ (α :: ‘rev [])
    axioms = (+-nil :: +-cons :: rev-nil :: rev-cons :: [])

```

Lists are essentially monoids with respect to append and nil and we first show that the empty list is indeed a right identity.

```

rident-nil : ‘List, [], Γ0 ⊢ [] ⇒ ‘[] ⊢ ‘[] ≈ []
rident-nil = fromJust (reconstruct ((inj1 (# 0), true, eq-step ([ ]l)
  (con (# 0) ([ ]x) ::s [ ]s)) ::l [ ]l))
rident-cons : ‘List, [], Γ2 ⊢ β ⊢ ‘[] ≈ β :: []
rident-cons ⇒ (α :: β) ⊢ ‘[] ≈ (α :: β)
rident-cons = fromJust (reconstruct ((inj1 (# 1), true, eq-step
  ([ ]l) (con (# 4) ([ ]x) ::s con (# 5) ([ ]x) ::s con (# 0)
  ([ ]x) ::s [ ]s)) ::l (inj1 (# 4), true, eq-step (1 ::l
  [ ]l)([ ]s)) ::l [ ]l))

```

The base case and the induction step can, of course, be tied together by a case split at the Agda layer. The induction step goes beyond pure equational reasoning, but can still be handled by Waldmeister. The reason is that the implication in the proof goal is skolemised, which yields constants, and the antecedent of the resulting ground formula is then added to the list of axioms. This is captured in our implementation by the derived type  $E, L, \Gamma \vdash H \Rightarrow s = t$ , where  $H$  contains the ground equations resulting from the inductive hypothesis.

The remaining lemmas are proved in a similar way. Previously proved lemmas can be added as hypotheses to prove goals. Again, this is managed automatically by Agda.

```

assoc-nil : ‘List, [], Γ2 ⊢ [] ⇒ ‘[] ⊢ (α ⊢ β) ≈ (‘[] ⊢ α) ⊢ β
assoc-nil = -- proof omitted
assoc-cons : ‘List, [], Γ4 ⊢ β ⊢ (γ ⊢ δ) ≈ (β ⊢ γ) ⊢ δ :: []
assoc-cons ⇒ (α :: β) ⊢ (γ ⊢ δ) ≈ ((α :: β) ⊢ γ) ⊢ δ
assoc-cons = -- proof omitted
rev-+-nil : ‘List, ((Γ1, (α ⊢ ‘[] ≈ α)) :: []), Γ1 ⊢ []
rev-+-nil ⇒ ‘rev (‘[] ⊢ α) ≈ ‘rev α ⊢ ‘rev []
rev-+-nil = -- proof omitted
rev-+-cons : ‘List, ((Γ3, α ⊢ (β ⊢ γ) ≈ (α ⊢ β) ⊢ γ) :: []),
  Γ3 ⊢ ((‘rev (β ⊢ γ) ≈ ‘rev γ ⊢ ‘rev β) :: [])
rev-+-cons ⇒ ‘rev ((α :: β) ⊢ γ) ≈ ‘rev γ ⊢ ‘rev (α :: β)
rev-+-cons = -- proof omitted
rev-rev-nil : ‘List, [], Γ0 ⊢ [] ⇒ ‘rev (‘rev []) ≈ []
rev-rev-nil = -- proof omitted
rev-rev-cons : ‘List, ((Γ2, ‘rev (α ⊢ β) ≈ ‘rev β ⊢ ‘rev α) :: []),

```

$$\Gamma \vdash ((\text{'rev } (\text{'rev } \beta) \approx \beta) :: []) \Rightarrow (\text{'rev } (\text{'rev } (\alpha' :: \beta))) \approx (\alpha' :: \beta)$$

rev-rev-cons = -- proof omitted

As previously, Waldmeister was very efficient with these proofs. Proof reconstruction succeeded within seconds on these examples, too.

## 10 Future Work

The main purpose of this work is to provide a framework for ATP integration together with a prototypical integration of a particular ATP system. Our experiments show that there is still much potential for extension and improvement.

The current approach does not take full advantage of reflection. It is therefore impossible to automatically construct a hypothesis set and goal based on the contents of a metavariable and the proof state. Full-scale reflection is a major future goal for Agda, with possible language extensions required to allow a fully transparent integration. The current Agda reflection system works via a primitive `quoteGoal`, which constructs an expression representation of the current goal. This system of reflection can be used to write internal provers for Agda, for instance domain specific solvers and decision procedures [7].

However, `quoteGoal` produces an untyped representation of a goal. Therefore a signature and its interpretation for the expression must be manually built to enable realisation of the proof. With a typed representation, these could be automatically constructed, thus enabling automatic proof of equational goals with the underlying reflective proofs being hidden. Furthermore, it is not currently possible to query the proof-state from within Agda. Therefore we cannot retrieve relevant hypotheses.

The current approach to proving a goal performs a complete proof reconstruction and drops the proof text into a meta-variable. Whilst this is fully transparent, it is inconvenient when the structure of the proof is irrelevant. Furthermore, complex equational proofs can contain large numbers of proof steps which make Agda files unwieldy. In cases like this a better solution would be a single primitive which would try and prove a goal, possibly

$$\text{waldmeister} : \forall A L \{ \Gamma \} e f \{ c : \text{WMCert } A L \Gamma e f \} \rightarrow A, L, \Gamma \vdash e \approx f$$

We cannot simply use `Maybe` for the output type, as this may break referential transparency, rendering Agda inconsistent. Instead we provide a special type `WMCert`, a pseudo record type which is only inhabited when Waldmeister succeeds. If Waldmeister does not find a proof in the allotted time, type-checking will fail since a timeout is an essentially vacuous result.

Waldmeister is limited to pure equational logic, whereas full classical first-order logic clashes with Agda's constructive type-system. But constructive logic can be seen as an extension of classical logic in the sense that some of its theorems are classically valid. For the syntactic class of Harrop formulas classical ATP systems can still be used and should be integrated. A theoretical framework, which is more abstract than ours, has already been provided [1]. Ultimately,

Agda’s internal proof search tool Agsy [14] could be combined with a variety of external tools for proof search. This combination would yield a powerful and versatile proof environment for semi-automated program construction in Agda.

Our experiments show that proof reconstruction remains a considerable bottleneck towards ATP integration. In particular, Agda proof checking is by orders of magnitude slower than Waldmeister proof search. There are several ways of addressing this. Based on profiling, one could optimise the existing functions and data-types, for instance by using term indexing or efficient binary data structures. In addition, a variant of the unfailing completion procedure [2] underlying Waldmeister could be implemented in Agda to make at least micro-step reconstruction more efficient. Finally, Agda itself is currently an experimental system and future optimisation will have immediate impact on proof reconstruction. A similar integration of SAT solvers into Agda is currently undertaken [12]. Its main difference is that proof reconstruction is sacrificed for the sake of efficiency, but at the expense of trust-worthiness, in the tradition of interactive theorem provers such as PVS [19].

## 11 Conclusion

We have presented a framework for integrating external ATP systems into Agda. Some parts of it are generic while others are specific to the Waldmeister implementation provided. First experiments show that our integration works, but should further be optimised to make proof reconstruction faster and more powerful.

These results show the potential of ATP integration in increasing Agda’s proof automation and making program development and analysis in Agda easier and less error-prone. One of the main program development methods in functional programming is documented in Bird and de Moor’s book on *The Algebra of Programming* [3]. It has already been integrated into Agda [16], but is still based on interactive theorem proving. Related work with the Isabelle interactive theorem prover, however, shows that the underlying algebra of relations and functions can effectively be automated [8]. This suggests that our Agda ATP integration has considerable potential in this direction, and could lift program development in dependently typed languages to a new level of automation.

*Acknowledgements.* We would like to thank Thorsten Altenkirch, Nils Anders Danielsson, Peter Dybjer, Ulf Norrell, Anton Setzer and Makoto Takeyama for helpful discussions and advice. This work has been funded by EPSRC grant EP/G031711/1.

## References

1. Abel, A., Coquand, T., Norell, U.: Connecting a logical framework to a first-order logic prover. In: Gramlich, B. (ed.) FroCoS 2005. LNAI, vol. 3717, pp. 285–301. Springer (2005)

2. Bachmair, L., Dershowitz, N., Plaisted, D.A.: Resolution of Equations in Algebraic Structures, chap. Completion Without Failure, pp. 1–30. Academic Press (1989)
3. Bird, R., de Moor, O.: The Algebra of Programming. Prentice-Hall (1997)
4. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer (2010)
5. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning. LNCS, vol. 6173, pp. 107–121. Springer (2010)
6. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer (2010)
7. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda - a functional language with dependent types. In: TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer (2009)
8. Foster, S., Struth, G., Weber, T.: Automated engineering of relational and algebraic methods in Isabelle/HOL (2010), Submitted.
9. Harrop, R.: On disjunctions and existential statements in intuitionistic systems of logic. *Mathematische Annalen* 132, 347–361 (1956)
10. Hillenbrand, T., Buch, A., Vogt, R., Löchner, B.: Waldmeister: High performance equational deduction. *Journal of Automated Reasoning* 18(2), 265–270 (1997)
11. Hurd, J.: System description: The Metis proof tactic. In: Benzmueller, C., Harrison, J., Schuermann, C. (eds.) ESHOL2005. pp. 103–104. arXiv.org (2005)
12. Kanso, K., Setzer, A.: Integrating automated and interactive theorem proving in type theory. In: Bendisposto, J., Leuschel, M., Roggenbach, M. (eds.) AVOCS 2010 (2010), 2 pp. Available from <http://www.cs.swan.ac.uk/~csetzer/articles/kansoSetzerAvocs2010.pdf>
13. Klop, J.W., de Vrijer, R.: First-order term rewriting systems, Cambridge Tracts in Theoretical Computer Science, vol. 55, chap. 2. Cambridge University Press (2003)
14. Lindblad, F., Benke, M.: A tool for automated theorem proving in Agda. In: Filiâtre, J.C., Paulin-Mohring, C., Werner, B. (eds.) Types for Proofs and Programs. LNCS, vol. 3839, pp. 154–169. Springer (2006)
15. McBride, C., McKinna, J.: The view of the left. *Journal of Functional Programming* 14(1), 69–111 (2004)
16. Mu, S.C., Ko, H.S., Jansson, P.: Algebra of programming using dependent types. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 268–283. Springer (2008)
17. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
18. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
19. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNAI, vol. 607, pp. 748–752. Springer (1992)