Integrating an Automated Theorem Prover into Agda

Simon Foster and Georg Struth

Department of Computer Science, University of Sheffield, UK
{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 theorem provers, SMT/SAT 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 Agda [7] or Epigram [15], which are based on dependent constructive type theory. Here, programs are obtained directly from type-level specifications and proofs via the Curry-Howard isomorphism. These languages are therefore, in ingenious ways, programming languages *and* interactive theorem provers. Program development can be based on the standard methods for functional languages, but the need of formal proof adds an additional layer of complexity. It requires substantial mathematical skill and user interaction even for trivial tasks. Increasing proof automation is therefore of crucial importance.

Interactive theorem provers such as Isabelle [17] are showing a way forward. 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 trustworthiness. But all this proof technology is based on classical logic. This has two main consequences. First, on the programming side the proofs-as-programs approach is not available in Isabelle, hence programs cannot be extracted from Isabelle proofs. Second, because of the absence of the law of excluded middle in constructive logic, proofs from ATP systems and SMT solvers are not generally valid in dependently typed languages. An additional complication is that proof reconstruction in dependently typed languages must be part of type-checking. This makes an integration certainly not straightforward, but at least not impossible.

Inspired by Isabelle we provide the first ATP integration into Agda. To keep it simple we restrict ourselves to pure equational reasoning, where the rule of excluded middle plays no role and the distinction between classical and constructive proofs vanishes. We integrate Waldmeister [10], the fastest equational ATP system in the world[1]. Waldmeister also provides detailed proofs and supports simple sorts/types. Our main contributions are as follows.

- We implement the basic data-types for representing equational reasoning 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 at Agda's reflection layer together with functions that parse Waldmeister proofs 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. Invalid proofs are nevertheless caught during proof reconstruction.
- We provide a series of small 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 templates 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 discussed 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 cleaner, faster and less error-prone.

This paper aims to explain the main ideas and features of our approach to a formal methods audience. Its more idiosyncratic aspects, which are mainly of interest for Agda developers, are contained in a technical report [8]; the complete code for our implementation can be found at our website[2].

[1] `http://www.cs.miami.edu/~tptp/CASC/`, 15/02/2011

[2] `http://simon-foster.staff.shef.ac.uk/agdaatp`

2 Agda

Agda [7] is a dependently typed programming language and proof-assistant. It is strongly inspired by 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. Additional information about Agda, including libraries and tutorials, can be found at the Agda Wiki[3].

The data-types featured in this section come from Agda's standard library. The following inductive data-type declaration introduces vectors.

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

In contrast to most other functional programming languages, Agda supports dependent data-types. The data-type of vectors is defined 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 : 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 supply the parameter n, such *hidden* parameters are indicated by braces. One can now define functions as usual.

```
head : ∀ {n} {A : Set} → Vec A (1 + n) → A
head (x :: xs) = x
```

Agda only accepts *total functions*, but head should only be defined when $n \neq 0$. The dependent type declaration captures this constraint. It thus allows a fine control of data validity in specifications. Predicates can also be data-types:

```
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⩽n and s⩽s are *names*. Agda is white-space sensitive, so they are parsed as one token, whereas zero ⩽ n is parsed as three tokens. The elements of this data-type are *inductive proofs* of ⩽. For instance, s⩽s (s⩽s z⩽n) is a proof of $2 \leqslant 3$. Hence, Agda data-types capture proofs as well as objects such as numbers or vectors. Similarly, one can define $n < m$ as suc n ⩽ m.

Agda provides two definitions of equality. *Propositional equality*, ≡, holds when two values and their types have the same normal forms. *Heterogeneous equality*, ≅, only requires equality of values. Two vectors xs : Vec A (m + n) and ys : Vec A (n + m) have different types in Agda, hence xs ≡ ys is not well typed. But xs ≅ ys would hold if xs and ys have same normal form.

[3] `http://wiki.portal.chalmers.se/agda/pmwiki.php`

As a constructively typed language, Agda uses the Curry-Howard Isomorphism to extract programs from proofs. The above data-types 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) → n < m)
greater n = ?
```

The type of greater specifies that for every natural number n there exists a natural number m greater than n. In the function body, ? indicates a meta-variable for which a program must be constructed through proof. More precisely, Agda requires a natural number m constructed in terms of n and a proof that $n < m$. Agda provides a variety of tools for proof support. If the user invokes the *case-split* command, two proof obligations are generated from the inductive definition of natural numbers:

```
greater zero = { } 0
greater (suc n) = { } 1
```

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

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

This is now a pair consisting of a natural number m and a proof that this witness satisfies $zero < m$. The following code displays a value and proof:

```
greater zero = 1, s≤s z≤n
```

In this case, $m = 1$ and s≤s z≤n are the names of the inference rules needed for the proof. By the first rule, $zero \leqslant zero$, by the second rule, therefore, $suc\ zero \leqslant suc\ zero$, whence $zero < suc\ zero$ by the definition of $<$.

This proof style lends itself naturally to incremental program construction, where writing a program and proving its correctness are one activity. To further automate this, Agda provides 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)))
```

The functions $proj_1$ and $proj_2$ project on the value and the proof of the proof goal. However, Agsy struggles with non-trivial proof goals. Increasing the degree of automation is therefore highly desirable to free programmers from trivial proof and construction tasks.

3 Integration of Automated Theorem Proving

ATP systems have already significantly increased proof automation in interactive theorem provers. Isabelle [17], for instance, can use a tactic called Sledgehammer to call external ATP systems. In contrast to interactive provers, which consist of a relatively small inference engine, ATP systems are complex tools that depend on a large number of heuristics. They are less trustworthy than interactive provers. Consequently, Isabelle internally *reconstructs* all ATP proofs with the internally verified ATP system Metis [11]. Since Metis is less efficient than the external ATP systems, a relevance filter minimises the number of hypotheses given to it. Metis then performs proof search to derive the goal from the hypotheses. In practice, however, this *macro-step* proof reconstruction sometimes fails.

Evidently, Agda could benefit from a similar approach, but all state of the art ATP systems are designed for classical predicate logic. The resolution principle, which underlies most of these systems, is directly based on the law of excluded middle. Since constructive proofs are needed for Agda, we have based a first integration on Waldmeister [10], an ATP system for pure equational logic, where the difference between classical and constructive proof disappears. Equational logic needs only rules for reflexivity, symmetry, transitivity, congruence, substitution and a number of structural rules that are all present in constructive logic.

Our integration can serve as a basis for integrating full first-order ATP systems, which could still be used on subclasses of constructive formulae, such as Harrop formulae [9] where classical and constructive proofs coincide.

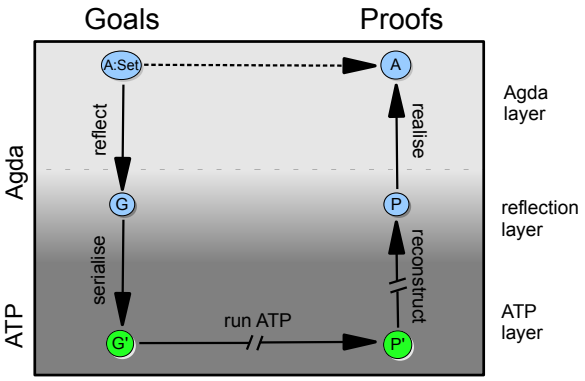

Fig. 1. Overview of automated theorem prover integration in Agda.

A Waldmeister integration into Agda is still not straightforward, for two main reasons. Firstly, the built-in Agda normaliser changes hypotheses and proof goals, but these are hidden within the proof state and cannot be accessed from within Agda. Secondly, for both goal extraction and proof reconstruction, Agda syntax must explicitly be manipulated as part of the type-checking process. We see two main approaches to integrating an ATP system into Agda. The *internal approach* performs most of the goal extraction and proof reconstruction within Agda itself. Reflection, which we explain below, provides a way of mediating Agda proofs with ATP proofs. In the *external approach*, proof proceeds by ac-

cessing the Agda proof state with an external tool, for instance Haskell, passing this state to the ATP system, and writing an ATP proof back into Agda.

Each approach offers advantages and disadvantages. In this paper, we use the former because it is conceptually cleaner and all the steps of proof reconstruction are internally verified in Agda itself. An additional design decision is whether to use Metis style macro-step proof reconstruction, or micro-step reconstruction of individual proof steps. Again, we take the latter approach. The former would require an internally verified equational prover, and we expect Agda's internal proof tools to be efficient enough to handle single proof steps. Fortunately, Waldmeister output is sufficiently detailed for this purpose.

Our internal approach uses the mechanism of *reflection* which is similar to a quoting mechanism in programming languages, lifting syntax to an internal meta-level, protecting it from evaluation and allowing manipulation within Agda. Therefore all Agda data-types needed in proofs are represented at Agda's reflection level before and after passing them to Waldmeister. Our approach therefore uses the three layers illustrated in Figure 1: The *Agda layer* contains the initial proof goal and realises the final proof. The *reflection layer* represents the Agda goal and reconstructed ATP proof output. The *ATP layer* runs the serialised proof goal and outputs an ATP proof.

Agda's quoting mechanism, however is still experimental. Currently we can reflect 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.

4 Proof Cycle Example

This section provides an overview of our Waldmeister integration. It shows how a simple inductive proof is passed from the Agda layer through the reflection layer to Waldmeister, and how the proof obtained by Waldmeister is passed back and reconstructed within Agda.

Consider the following Agda proof goal:

```
assoc : ∀ (x y z : ℕ) → (x + y) + z ≡ x + (y + z)
```

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

```
assoc zero y z = { } 0
assoc (suc n) y z = { } 1
```

Waldmeister can solve each individual goal. Within Agda, they must first be lifted to a reflected signature Σ-Nat for natural numbers and their operations.

```
Nat : HypVec                    -- Proof environment for natural numbers
Nat = HyVec Σ-Nat axioms        -- Construct it from signature and axioms
  where
  +-zero = Γ1, ‘0 ‘+ α ≈ α      -- Quotes indicate reflection layer
  +-suc  = Γ2, ‘suc α ‘+ β ≈ ‘suc (α ‘+ β)
  axioms = (+-zero :: +-suc :: [])
assoc-zero : Nat, [], Γ2 ⊢ [] ⇒ (‘0 ‘+ α) ‘+ β ≈ ‘0 ‘+ (α ‘+ β)
assoc-zero = ?
assoc-suc : Nat, [], Γ3 ⊢ (α ‘+ β) ‘+ γ ≈ α ‘+ (β ‘+ γ) :: []
          ⇒ (‘suc α ‘+ β) ‘+ γ ≈ ‘suc α ‘+ (β ‘+ γ)
assoc-suc = ?
```

The reflection layer types of assoc-zero and assoc-suc represent the proof goals including environments for axioms, lemmas and variables used. The question marks indicate meta-variables which need to be instantiated with a proof term. This reflection layer data-type provides sufficient information for generating a Waldmeister input file for the first proof obligation:

```
NAME            agdaProof
MODE            PROOF
SORTS           Nat
SIGNATURE       suc: Nat -> Nat
                plus: Nat Nat -> Nat
                zero,a,b:  -> Nat
ORDERING        LPO a > b > zero > suc > plus
VARIABLES       x,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))
```

Waldmeister instantly returns with the following proof:

```
Axiom 1: plus(zero,x1) = x1
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. Waldmeister's output contains sufficient information to reconstruct this proof step-by-step at the reflection layer and instantiate the first meta-variable:

```
assoc-zero =
  fromJust (  reconstruct ((inj₁ (# 0), true, eq-step (0 ::ᴵ [] ᴵ) (con (# 3) ([] ˣ)
           ::ˢ [] ˢ))::ᴵ (inj₁ (# 0), false, eq-step ([] ᴵ) (con (# 2) (con (# 3)
              ([] ˣ) ::ˣ con (# 4) ([] ˣ) ::ˣ [] ˣ)::ˢ [] ˢ)) ::ᴵ [] ᴵ))
```

The syntax in this proof need not concern us in this paper; it essentially expresses the equational steps above. The function reconstruct uses the reflection layer inference rules for equational logic, which we have internally proved to be sound. To realise this proof in Agda it needs to be translated back to $\mathbb{N}$.

```
ℕ-⟦Σ⟧ : ⟦Signature⟧ Σ-Nat   -- code omitted
ℕ-Nat : ⟦HypVec⟧ Nat ℕ-⟦Σ⟧   -- code omitted
```

These functions link the reflection layer signature to concrete Agda functions and ground the axioms. The first function instantiates the reflection layer signature, the second one instantiates the hypotheses. The reflection layer proof term is instantiated to a valid Agda proof corresponding to the first meta-variable {}0:

```
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))
```

The proof cycle for the second meta-variable, {}1, is similar.

5 The Reflection Layer

Agda data-types and proofs must be lifted to the reflection layer to enable their manipulation within Agda. This section shows how data-types, theories and proofs that enable ATP proofs can be implemented at the reflection layer. First we describe the data-types and theories.

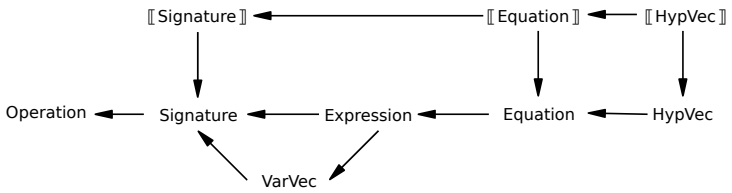

Fig. 2. Dependency graph of reflective components

Figure 2 shows the data-types required. They provide the reflection layer syntax for the terms and equations used in the proofs and the interpretations of these objects within Agda. We will now describe each of them. The basis of our reflection layer syntax are *operations* and *signatures*. Operations can either be Agda functions or data-type constructors; we need not distinguish them.

First we define operations, signatures and variable sets at the reflection layer.

```
record Operation (sorts : FinSet) : Set where
  field
    arity   : ℕ
    args    : Vec (El sorts) arity
    output  : El sorts
record Signature : Set where
  field
    sorts, ops : FinSet
    operations : Vec (Operation sorts) ops
```

```
record VarVec (Σ : Signature) : Set where
  open Signature Σ
  field
    vars : FinSet
    vvec : Vec (El sorts) vars
```

FinSet is a finite set and El indicates an element of a finite set. An n-ary operation is represented as a record parametrised over the set of sorts in its signature. It consists of its arity, its input sorts args and its output sort. A signature is a finite set of sorts together with a vector of operations. 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. Next we define terms.

```
mutual
  data Expr Σ (Γ : VarVec Σ) : VarSet Γ → Sort Σ → Set where
    con : (i : El (ops 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  -- code omitted
```

The expression data-type has four parameters: (i) the signature Σ, (ii) the variable context Γ which lists all variables available for building a term, (iii) the subset of variables ν drawn from the context which the expressions contains, represented by shorthand VarSet Γ and (iv) the sort s of the expression. Constructor con takes the operation index i, and an expression vector parametrised over its argument sorts; it yields an expression with the output sort of i. Constructor var takes a variable index and yields an expression with a singleton free variable {x} of the correct sort. Expressions and expression vectors are mutually inductive. It is now possible to define equations and hypotheses sets.

```
record Equation (Σ : Signature) : Set where
  constructor _≈_
  field
    {Γ}    : VarVec Σ
    {sort} : Sort Σ
    {ν₁ ν₂} : VarSet Γ
    lhs    : Expr Σ Γ ν₁ sort
    rhs    : Expr Σ Γ ν₂ sort
record HypVec : Set where
  constructor HyVec
  field
    Σ          : Signature
    {hyps}     : FinSet
    hypotheses : Vec (Equation Σ) hyps
```

We now move to the upper level of Figure 2, where corresponding data-types are implemented. The complete code can, again, be found at our website.

```
record ⟦Signature⟧ (Σ : Signature) : Set₁ where    -- code omitted
sem : ∀ {Σ} (⟦Σ⟧ : ⟦Signature⟧ Σ) {Γ : VarVec Σ} {s} {ν} (⟦ρ⟧ : ⟦Subst⟧ Γ ⟦Σ⟧)
      (e : Expr Σ Γ ν s) → ⟦Signature⟧.types ⟦Σ⟧ s    -- code omitted
record ⟦Equation⟧    -- code omitted
record ⟦HypVec⟧    -- code omitted
```

⟦Signature⟧, ⟦Equation⟧ and ⟦HypVec⟧ map the reflection layer objects indicated into the Agda layer, and provide soundness proofs. Function sem realises reflection layer terms, using an Agda layer signature ⟦Σ⟧ and substitution ⟦ρ⟧.

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. It provides a function which serialises a Waldmeister input file, as shown in Section 4, executes the prover and parses the resulting Waldmeister proof output back into Agda.

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 heterogeneous equality.

At the core of proof reconstruction is an implementation of equational reasoning, as performed by Waldmeister. We need some notation and concepts from term rewriting (cf. [19]). A *substitution* is a map ρ from variables to terms, which extends to a function on terms. A term t *matches* a term s (or s *subsumes* t) if $s\rho = t$ for some substitution ρ. We write $t \sqsubseteq s$ if s subsumes t. More specifically, to denote the ternary relation $s\rho = t$ between s, ρ and t, we write $t \sqsubseteq_\rho s$.

With subsumption we can model one-step rewrites of equational logic. Let E be a set of equations $l_i \approx r_i$. We write $E \vdash s = t$ if there is a substitution ρ, a context C and an equation $l \approx r \in E$ such that $s = C[l\rho]$ and $t = C[r\rho]$. Hence

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

for some substitution ρ, context C and $l \approx r \in E$. We extend this one-step rewrite relation by inductively defining $=$ as the transitive closure of $=^1$. To implement these concepts, we first provide a data-type for substitutions. The expression Subst Γ_1 Γ_2 ν represents a substitution map from the variables in Γ_2 to expressions with variables in Γ_1. The finite subset ν of Γ_2 indicates all those variables that are changed by the substitution. 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
    -- Code omitted
  data _ [ _⊑*_ ]    -- Code omitted
```

The inhabitants of this data-type are proofs that two terms match under a given substitution. If a user provides two terms and a substitution, this data-type

yields the proof obligations that the user must fulfill to establish the matching relation. These obligations correspond to the inductive definition of terms. The case of a term $f(t_1 \cdots t_n)$ requires mutual induction, as defined by $\sqsubseteq^*$, over the set of subterms. The complete code can be found at our website.

Using the subsumption data structure we can prove the following fact.

Lemma 1. $s \sqsubseteq_\rho t \implies \forall \sigma \,.\, [\![s]\!]\sigma \cong [\![t]\!](\sigma \circ [\![\rho]\!])$.

This lemma states that subsumption implies heterogeneous equality, where the additional substitution σ can be used for further instantiating the resulting expression in equational proofs. The proof of this lemma has been formalised in Agda. As an example, we show the Agda function type corresponding to the lemma.

```
⊑-to-≅ : ∀ {Σ} {Γ₁ Γ₂} {[[Σ]]} {ν₁ ν₂} {s₁ s₂} (f : Expr Σ Γ₂ ν₂ s₂)
         (e : Expr Σ Γ₁ ν₁ s₁) (ρ : Subst Γ₁ Γ₂ full) → ρ [ e ⊑ f ]
       → (∀ [[σ]] → sem [[Σ]] [[σ]] e ≅ sem [[Σ]] (sem-subst {[[Σ]] = [[Σ]]} ρ [[σ]]) f)
```

In the next steps we have implemented one-step equational reasoning with and without contexts and n-step equational reasoning, using the subsumption data-type. We have proved soundness of the resulting procedure within Agda.

Theorem 1. *Rewriting implies hetereogeneous equality.*

1. $u \approx v \vdash s = t \Longrightarrow \forall \rho. [\![s]\!]\sigma \cong [\![t]\!]\sigma$.
2. $E, L \vdash s = t \Longrightarrow [\![E]\!], [\![L]\!] \models [\![s]\!] \cong [\![t]\!]$.

The Agda proofs can be found at our website. The antecedent of the first statement expresses that s can be rewritten to t using the equation $u \approx v$ at the reflection layer. Its left-hand side essentially states that the interpretation of the reflection layer terms within Agda yield a valid equation. The interpretation of the equation $u \approx v$ is part of the proof state and therefore not visible on the right-hand side. The second statement lifts the first one to sets E of equational axioms, sets L of additional equational hypotheses and n-step proofs. This theorem provides the formal underpinning for proof reconstruction.

Although the data-types presented have been designed predominantly for equational reasoning, they can nevertheless easily be extended to full first-order logic by adding quantifiers, the usual boolean connectives and predicate symbols on top of our existing Equation data-type. Our implementation shows how Agda's reflection layer can be used to achieve such extensions.

6 Proof Reconstruction

We now show how Waldmeister proofs can be reconstructed within Agda as part of the type-checking process. Proof reconstruction can fail since Waldmeister can fail, types can be overabstracted, or Waldmeister introduces constants which have not been accounted for in Agda.

As shown in Section 4, 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, its orientation (left-right or right-left), and the substitution used. All these features have been implemented at the reflection layer in Section 5. 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 $e =^1 f$ is obtained by applying the equation $u \approx v$ under the context g of e, and using substitution ρ.

1. The function build-split uses the term position to split $e = g[e']$ and verify this equality.
2. The function build-$\vdash\approx^1$ takes an equation, substitution and split term, and rewrites e to f. A matching algorithm for computing subsumptions is used by this function.

These one-step reconstruction functions are then used together in reconstruct.

$$\begin{aligned}\mathsf{reconstruct} \;:\; &\forall\,\{\mathsf{E} : \mathsf{HypVec}\}\,\{\Gamma\}\,\{\nu_1\;\nu_2\}\,\{\mathsf{s}\}\,\{\mathsf{n}\}\,\{\mathsf{L} \;:\; \mathsf{Vec}\;(\mathsf{Equation}\;(\Sigma\;\mathsf{E}))\;\mathsf{n}\}\\ &\to \{\mathsf{e} \;:\; \mathsf{Expr}\;(\Sigma\;\mathsf{E})\;\Gamma\;\nu_1\;\mathsf{s}\}\,\{\mathsf{f} \;:\; \mathsf{Expr}\;(\Sigma\;\mathsf{E})\;\Gamma\;\nu_2\;\mathsf{s}\}\\ &\to \mathsf{EqProof}\;\mathsf{E}\;\Gamma\;\mathsf{L} \to \mathsf{Maybe}\;(\mathsf{E}, \mathsf{L}, \Gamma \vdash \mathsf{e} \approx \mathsf{f})\end{aligned}$$

The type EqProof represents the raw input from Waldmeister which has been reformatted by the Haskell module. 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. reconstruct applies the one-step reconstruction functions iteratively to yield the complete reflection layer proof.

Most first-order theorem provers support the TPTP format[4] as a standard input syntax. Some of them also support TSTP[5] as a proof output standard. For future implementations, Haskell modules supporting these standards should be used instead of the current proprietary Waldmeister ones.

7 Examples

This section shows the Waldmeister integration at work. We have tested it on simple examples about natural numbers (cf. Section 4), groups, Boolean algebras and lists. The following code implements group theory.

$$\begin{aligned}&\mathsf{Group} \;:\; \mathsf{HypVec}\\ &\mathsf{Group} \;=\; \mathsf{HyVec}\;\Sigma\text{-}\mathsf{Group}\;\mathsf{axioms}\\ &\quad \textbf{where}\\ &\quad \mathsf{assoc} \;=\; \Gamma 3, (\alpha \cdot \beta) \cdot \gamma \approx \alpha \cdot (\beta \cdot \gamma)\\ &\quad \mathsf{ident} \;=\; \Gamma 1, \mathsf{e} \cdot \alpha \approx \alpha\\ &\quad \mathsf{inv} \;=\; \Gamma 1, \alpha^{-1} \cdot \alpha \approx \mathsf{e}\\ &\quad \mathsf{axioms} \;=\; (\mathsf{assoc} :: \mathsf{ident} :: \mathsf{inv} :: [\,])\end{aligned}$$

As an example, we show one of the most basic facts.

[4] http://www.cs.miami.edu/~tptp/
[5] http://www.cs.miami.edu/~tptp/TSTP/

```
ident-var : Group, [], Γ2 ⊢ α ⁻¹ · (α · β) ≈ β
ident-var = fromJust (reconstruct ((inj₁ (# 0), false, eq-step ([] ᴵ)
            (con (# 2) (var (# 0) ::ˣ [] ˣ) ::ˢ var (# 0) ::ˢ var (# 1)
            ::ˢ [] ˢ))::ᴵ (inj₁ (# 2), true, eq-step (0 ::ᴵ [] ᴵ) (var
            (# 0) ::ˢ [] ˢ)) ::ᴵ (inj₁ (# 1), true, eq-step ([] ᴵ) (var
            (# 1) ::ˢ [] ˢ)) ::ᴵ [] ᴵ))
```

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. Additional lemmas can be found at our website. On such very simple lemmas, Waldmeister returned almost instantaneously. Proof reconstruction required several seconds.

Proofs in Boolean algebra are more complex, and proof-search is more involved. In our experiments, Waldmeister returned within seconds. But reflection layer proofs tended to become very long, and their reconstruction sometimes took several minutes. There are some theorems that Waldmeister could easily verify, but where proof reconstruction failed, e.g., when Waldmeister chose to introduce new undeclared constants for non-obvious reasons. Further discussion can be found in our extended version [8].

Finally we show some simple proofs about lists. This is especially interesting for two reasons. First, lists are two-sorted structures and it is shown that Waldmeister can handle this situation. Second, proofs require induction, which is beyond first-order logic.

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

The base case and the induction step can 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 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.

Additional 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. Finally, we can automatically prove a classic.

```
rev-rev-nil : 'List, [], Γ0 ⊢ [] ⇒ 'rev ('rev '[]) ≈ '[]
rev-rev-nil =  -- proof omitted
rev-rev-cons : 'List, ((Γ2, 'rev (α '++ β) ≈ 'rev β '++ 'rev α) :: []),
                 Γ2 ⊢ (('rev ('rev β) ≈ β) :: []) ⇒ ('rev ('rev (α ':: β))) ≈ (α ':: β)
rev-rev-cons =  -- proof omitted
```

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

8 Conclusions and Future Work

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. The main purpose of this work is to explore how such integrations could be achieved by providing a prototype for one particular ATP system. Initial experiments show that our integration works, but should further be optimised to make proof reconstruction faster and more powerful.

First, reflection is experimental in Agda. It has already been used for integrating domain specific solvers and decision procedures [7], but does not suffice for automatically constructing ATP input from a metavariable and a proof state.

Second, a simple command, integrated with Agsy [14], and like Isabelle's Sledgehammer could greatly simplify ATP invocation and proof representation.

Third, full first-order theorem provers should be integrated and syntax checks (for Harrop formulae) could be used for applying them on safe fragments of constructive logic. A theoretical framework for this has already been provided [1].

Fourth, proof reconstruction requires further optimisation. As an alternative to the micro-step approach the unfailing completion procedure [2] underlying Waldmeister could be implemented. The external approach mentioned in Section 3 should also be explored. A similar integration of SAT solvers into Agda is currently undertaken [13]. Its main difference is that proof reconstruction is sacrificed for the sake of efficiency in the tradition of provers such as PVS [18].

Fifth, functional program development methodology [3] has already been integrated into Agda [16, 12]. Automating it could lift program development in dependently typed languages to a new level.

Acknowledgements. We would like to thank Thorsten Altenkirch, Nils Anders Danielsson, John Derrick, Peter Dybjer, Ulf Norrell and Makoto Takeyama for helpful discussions. 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.: Integrating an automated theorem prover into Agda. Tech. Rep. CS-10-06, Department of Computer Science, University of Sheffield (2010)
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: Benzmüller, C., Harrison, J., Schürmann, C. (eds.) ESHOL2005. pp. 103–104. arXiv.org (2005)
12. Kahl, W.: Dependently-typed formalisation of relation-algebraic abstractions. In: de Swart, H.C.M. (ed.) RaMiCS 2011. LNCS, Springer (2011), to appear.
13. Kanso, K., Setzer, A.: Integrating automated and interactive theorem proving in type theory. In: Bendisposto, J., Leuschel, M., Roggenbach, M. (eds.) AVOCS 2010 (2010)
14. Lindblad, F., Benke, M.: A tool for automated theorem proving in Agda. In: Filliâtre, 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 from 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. 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)
19. Terese: Term Rewriting Systems. Cambridge University Press (2003)