Typed Syntactic Meta-programming

Dominique Devriese  Frank Piessens
iMinds – DistriNet, KU Leuven
firstname.lastname@cs.kuleuven.be

Abstract
We present a novel set of meta-programming primitives for use in a dependently-typed functional language. The types of our meta-programs provide strong and precise guarantees about their termination, correctness and completeness. Our system supports type-safe construction and analysis of terms, types and typing contexts. Unlike alternative approaches, they are written in the same style as normal programs and use the language’s standard functional computational model. We formalise the new meta-programming primitives, implement them as an extension of Agda, and provide evidence of usefulness by means of two compelling applications in the fields of datatype-generic programming and proof tactics.

Categories and Subject Descriptors D3.3 [Programming Languages]: Language Constructs and Features; Data types and structures; F3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs: Specification Techniques; F4.1 [Mathematical Logic]: Lambda Calculus and Related Systems

Keywords meta-programming; dependent types; datatype-generic programming; tactics.

1. Introduction
Meta-programming means writing programs that write or manipulate other programs. It is an important software engineering technique that is widely used in practice. The term covers a wide variety of techniques and applications, including parser generators [29], reflection and byte-code generation in Java-like languages [8, 40], macro’s in Lisp-like languages [53], eval primitives in languages like JavaScript [45], special-purpose meta-programming or generic programming primitives [6, 11, 13, 26, 33, 48, 52], tactics in proof assistants [20, 50, 51] and term representations in advanced type systems [9, 15, 38]. Meta-programming jargon distinguishes between the meta-language, that meta-programs are written in, and the object language, that the programs being manipulated are in.

Meta-programming can often be used to implement features in a library that would otherwise require ad hoc compiler support. This ranges from meta-programs that generate small amounts of boilerplate code to give libraries a more native feel (e.g. [31, 34, 46]) to languages built from the ground up using meta-programming [53].

In many applications, meta-programs must not only be able to produce new code but also analyse existing terms, types or type contexts. Applications in e.g. datatype-generic programming or tactics for proof assistants involve meta-programs that analyse the syntactic structure of object language data types [13, 33], types [26, 50], types and contexts [20]. Some systems allow analysing terms [11], terms and types [48] or all three [9, 15, 38, 51].

Type-safety in the context of meta-programming can mean different things. In some approaches, generated code is type-checked upon completion of the meta-program, either at compile-time or run-time [8, 20, 48]. This can be sufficient to guarantee type-correctness of the resulting program. In this text, we are interested in a stronger form of type-safety, in which a meta-program’s type can guarantee type-correctness of all programs it will ever generate [11, 26, 33, 50–52]. This stronger form of type-safety provides meta-program authors and users with greater correctness assurance.

Sometimes, it also enables additional applications. For example, MetaML runs meta-programs and compiles the generated code at run-time, but type errors during this run-time compilation are ruled out by its strong type-safety [52]. In the context of a dependently-typed proof assistant, where proofs and programs are equated, Chlipala argues that the stronger form of type-safety has a performance advantage because proofs generated by meta-programs do not need to be calculated as long as they can be trusted to exist [14]. Note that for this last application, the meta-program must be guaranteed to terminate, as well as produce well-typed code.

However, this stronger form of type-safety puts a high demand on representations of object code and the meta-language type system, especially for object languages with strong (e.g. dependent) type systems and if meta-programs can construct and analyse both terms, types and typing contexts. Most approaches use an explicit syntactic representation of object language terms and/or types. To achieve strong type safety, they employ advanced type-system features of the meta-language, including GADTs [11, 52], strong type systems with powerful type-level languages [13, 23, 50, 51] and an advanced feature of dependent type systems called induction-recursion [9, 15, 38]. However, even the most powerful approaches have to make certain compromises, simplifying the resulting system at the cost of expressivity. For example, many approaches provide syntactic models of only types [33] or only terms [11, 52] or types and terms but not typing contexts [50].

Particularly technically ambitious are those meta-programming systems that use a syntactic model of a dependently-typed language within another one [9, 15, 38]. Their term encoding represents type-correctness internally, i.e. only well-typed terms are represented. To support this, they require an advanced dependent type system with support for induction-recursion in the meta-language and even then have trouble fitting the interpretation function in it [9, 15]. McBride presents a model that is accepted by Agda but has to significantly limit the dependent nature of the object language’s type system in the process [38]. The objective in this work is generally to prove meta-theory for the object language and the authors work
hard to fit their encodings into the advanced, but general and previously studied schema of inductive-recursive definitions.

However, beside their meta-theoretical value, syntactic models of a typed object language with a well-typed interpretation function are also promising for meta-programming applications. Unfortunately, the full potential of this has not been explored or demonstrated so far because researchers have not yet managed to build syntactic models of dependently-typed programming languages that support a big enough subset of a dependently-typed language and still have provably sound interpretation functions. In this paper, we ignore the aim of building meta-theory for dependent type theories within themselves and instead focus on applying such techniques to meta-programming. We will show that this approach has some very compelling qualities.

We use Agda [39], a pure functional dependently-typed language, as both the meta- and object language and we start from a conventional representation of the object language based on de Bruijn-encoded lambda terms and an external typing judgement. We make an interpretation function available as a new meta-programming primitive. This puts us on shakier ground, because the soundness of the primitive is not guaranteed by existing meta-theory, but it allows us to side-step the unsolved problem of syntactically representing a dependent type theory within itself with a provably sound interpretation function. As such, we gain the ability to explore and demonstrate our approach’s potential for meta-programming and present novel techniques for it.

Our choice to keep the meta- and object language the same (known as homogeneous meta-programming [48, 52]) contrasts with systems where meta-programs use a different computational model than object programs. Often this is an imperative model [20, 48, 50, 51], but some systems even use a logic programming-like model derived from the meta-programs’ interaction with type inference [26, 33]. Our meta-programs use the same functional model as normal programs and dependent pattern matching [25] for syntactically analysing terms, types and typing contexts. This choice keeps the system smaller, makes techniques, tools and knowledge for normal programming directly reusable in meta-programs and it allows meta-programs to use other meta-programs to do their work. It does not exclude imperative, generally recursive, non-deterministic or unification-based reasoning in meta-programs. Research has demonstrated functional models of such algorithms [16, 30, 32] and such ideas could be combined with our work.

In the dependently typed meta-language, meta-programs have strong and precise types that guarantee termination and correctness. Termination is standard for Agda functions (Agda is total). For strong type safety, our primitives require meta-programs to provide type-correctness proofs together with generated code and they can exploit type-correctness proofs for the code they analyse.

Some homogeneous meta-programming systems couple meta-programming with multi-stage programming [6, 48, 52], which allows object code programs to explicitly invoke meta-programs and use the generated expressions as if they were hand-written (unquoting) and allows meta-programs to include references to existing terms in generated code (quoting). A linear hierarchy of staging levels exists when meta-programs may unquote expressions generated by other meta-programs. The bottom stage is the program executed at run-time, while other stages execute at compile-time or run-time, depending on the system. Our interpretation function for encoded terms is analogous to an unquoting primitive and we will demonstrate how object-level terms can be referenced in generated code. The question of when meta-programs are executed becomes a matter of choice and a special case of partial evaluation.

We demonstrate the properties of our system by applying it to two important application domains: datatype-generic programming and proof tactics. For the first, we define a syntactic representation $SimpleDT$ of inductive data types that can be used to write general datatype-generic meta-programs. As proof-of-concept, we present a meta-program $deriveShow$ that syntactically derives a serialisation function $show : A \rightarrow String$ for a data type $A$.

$deriveShow : (A : Set) \rightarrow SimpleDT A \rightarrow A \rightarrow String$

$SimpleDT$ and $deriveShow$ do not require compiler support beyond our (general) meta-programming primitives, although the value of type $SimpleDT A$ could be provided by the compiler for additional convenience. The type of $deriveShow$ guarantees its correct termination and well-typedness of generated programs (modulo the primitives’ soundness). To the best of our knowledge, this is the first demonstration of strongly typed, general datatype-generic meta-programs, with support for syntactic analysis of terms and types and using the language’s standard computational model.

The second application domain is proof tactics. A tactic is a meta-program that analyses the type of a proof obligation and produces a proof term (possibly including remaining proof obligations) using general or domain-specific reasoning. Several proof assistants provide special-purpose languages for writing custom tactics [20, 50, 51]. These are often imperative and only guarantee weak type-safety (generated code is checked after execution of meta-program) or partial strong type-safety (generated code is guaranteed type-correct but meta-programs may not terminate). Gonthier et al. argue that tactics without strong type-safety can be hard to maintain and compose [26]. Chlipala discusses a performance advantage of precisely-typed and terminating meta-programs since generated proofs do not need to be calculated if they are known to exist [14]. To demonstrate that we can do better, we present an account of Coq’s $assumption$ tactic with a very precise type, guaranteeing that it will always terminate and produce a guaranteed type-correct term under a precise condition. The tactic uses a functional computational model and dependent pattern matching for syntactic analysis of terms, types and typing contexts.

We have implemented our primitives in Agda and our example meta-programs are accepted by Agda’s type-checker. Unfortunately, this does not mean our work is readily usable. The practicality of our implementation is currently hampered by long compilation times. However, we will argue that this problem is not intrinsic, but caused by the inefficient evaluation strategy of Agda’s compile-time evaluator. The soundness of our approach depends on the soundness of our primitives, which we can currently not provide guarantees about. We believe that our work gives a strong motivation to investigate both of these aspects further, since we provide strong evidence for the additional power that the system offers for meta-programming in general and the hard problems of well-typed tactics and datatype-generic programming in particular.

11 Contributions

Our first contribution in this work is the definition of novel meta-programming primitives in a dependently-typed language, starting from a partial formalisation of the language’s meta-theory. We also contribute the (to our knowledge first) demonstration of using such a formalisation for meta-programming, with compelling examples in two important application domains: datatype-generic programming and proof tactics. Our meta-programming model works with the language’s standard functional computational model, and meta-programs are written in the same way as normal programs. Modulo the soundness of our primitives, meta-programs can be given strong and precise guarantees of termination and correctness of the generated code. Finally, our proof-of-concept applications in these two application domains are interesting in their own right. For both

data Constant : (arity : N) → Set where
(early for now)
data Binder : Set where Π A : Binder
data Expr (n : N) : Set where
set : Expr n
var : Fin n → Expr n
appl : Expr n → Expr n → Expr n
constant : {arity : N} → Constant arity →
 Vec (Expr n) arity → Expr n
bind : Binder → Expr n → Expr (suc n) → Expr n
pi : {n : N} → Expr n → Expr (suc n) → Expr n
pi = bind Π
lambda : {n : N} → Expr n → Expr (suc n) → Expr n
lambda = bind A

Figure 1. The representation of terms.
data _→_₀₀ n {a} : Expr n → Expr n → Set where
 reduceApplication : ∀ {a} b val → appl (lambda a b) val →₀₀ a b [val]
data _→₋₋ n {a} : Expr n → Expr n → Set where
 ⋮ (congruence closure of _→₋₋₀₀)
→₋₋₀₀ {a} : {n : N} → Expr n → Expr n → Set
→₋₋₋₋ = ⋮ (transitive-reflexive closure of _→₋₋₀₀)
∞ ≃ : {n : N} → Expr n → Expr n → Set
x ≃ y = ∃ λ n → x ≈₋₋ n x y ≈₋₋ n

Figure 2. Substitutions (implementations omitted).
data _₋₋₋₋₋₋ n {a} : Expr n → Expr n → Set where
 reduceApplication : ∀ {a} b val → appl (lambda a b) val →₋₋₋₋₋₋ a b [val]
data _₋₋₋₋₋₋ n {a} : Expr n → Expr n → Set where
 ⋮ (congruence closure of _₋₋₋₋₋₋₀₀)
₋₋₋₋₋₋₀₀ n {a} : {n : N} → Expr n → Expr n → Set
₋₋₋₋₋₋₋₋₋₋ = ⋮ (transitive-reflexive closure of _₋₋₋₋₋₋₀₀)
∞ : {n : N} → Expr n → Expr n → Set
x ≃ y = ∃ λ n → x ≈₋₋₋₋₋₋ n x y ≈₋₋₋₋₋₋ n

Figure 3. Full β-reduction and β-equivalence for untyped terms.

Datatype-generic programming and proof tactics, the prospect of writing general meta-programs with strong and precise guarantees about termination, correctness and completeness and using the language’s standard computational model is compelling and novel.

1.2 Outlook

We present the representation of our object language in Section 2. In Section 3, we show how the represented terms and types are brought to life in the meta-language using our meta-programming primitives. In Section 4, we present applications to the fields of datatype-generic programming and proof tactics. We discuss issues like soundness and performance in Section 5, related work in Section 6 and we conclude in Section 7.

2. Self-representation

As discussed, we start from a representation of Agda terms in Agda using a notion of lambda expressions representing terms as well as their types and a typing judgement linking the two together.

Terms Figure 1 shows the definition of Expr, our representation of Agda terms and types as lambda terms, using de Bruijn indices. We represent de Bruijn indices as integers between 0 and n − 1 using the Agda standard library type Fin n [17]. The type Expr is parameterised by the number of free variables in scope. It is a set of its constructors and their types. The set constructor represents the type of a sequence of expressions, each representing the type of a bound variable. In what follows, we use \([\cdot]\) for the empty vector and for example \([x, y]\) for the vector with elements x and y. Similarly, we write literal Fins as numbers.

The final Expr constructor in Figure 1, bind, is a common representation of two separate binding constructs: lambda expressions \(\lambda (x : T) \rightarrow b\) and dependent function types \((x : T) \rightarrow T',\) constructed as bind A and bind Π respectively. They take two arguments: the type T of the bound variable and the body of the construct (b or T' respectively) with the bound variable additionally in scope in the body. Note by the way that a standard non-dependent function type \(s \rightarrow t\) can be represented as dependent function type \(\lambda x : T.b\).

\(\text{Sub} : N \rightarrow N \rightarrow Set\)
\(_\sim / : \{m : N\} \rightarrow \text{Expr } m \rightarrow \text{Sub } m n \rightarrow \text{Expr } n\)
\(_\sim [-] : \{n : N\} \rightarrow \text{Expr } (\text{suc } n) \rightarrow \text{Expr } n \rightarrow \text{Expr } n\)

\(\text{Sub} : N \rightarrow N \rightarrow Set\)
data Telescope (i : ℕ) : ℕ → Set where
  ε : Telescope i i
  ν : (n : ℕ) → Expr n →
  Telescope i n → Telescope i (suc n)
Context : (n : ℕ) → Set
Context = Telescope 0
lookup : ∀ {n} → Fin n → Context n → Expr n
lookup zero (t ≈ _ ) = weaken t
lookup (suc n) (ν ≈ _ ) = weaken (lookup n Γ)

Figure 4. Telescopes and Contexts

data _⊢_ : {n} (Γ : Context n) :
  Expr n → Expr n → Set where
  typeSet : Γ ⊢ t ⊢ set
  typeVar : ∀ {i} → Γ ⊢ var i : lookup i Γ
  typePi : ∀ {s t} → Γ ⊢ s : set →
  (s ⊦ t) ⊢ Γ ⊢ pi s t : set
  typeLam : ∀ {s b t} → Γ ⊢ s : set →
  (s ⊪ t) ⊢ Γ ⊢ lam s b : pi s t
  typeAppl : ∀ {s f t val} → (s ⊦ t) ⊢ Γ ⊢ f pi s t →
  Γ ⊢ val s →
  Γ ⊢ appl f val (lam (val s t) val)
  typeConv : ∀ {e t t’} → t ≈ t’ ⊢ Γ ⊢ e : t

Figure 5. Typing Judgements.

sumed. Telescopes are dependent: subsequent types can mention variables bound earlier in the telescope. This allows us to represent e.g. the telescope (n : ℕ) (t : Expr n), where the type of t depends on the value of n. As a consequence of this dependence, each additional entry in a telescope has an additional variable in scope.

The second index n of the Telescope type is the number of final variables: if i variables are initially bound, and we add the bindings of a Telescope i n, then in total n variables will be bound, so the telescope contains precisely n – i entries. A typing context Context n is a telescope with zero initial and n final bound variables. The lookup function looks up the type of a variable in a context. lookup’s dependent type ensures that only de Bruijn variables lower than the length of the context can be looked up.

**Typing Judgements**

In Figure 5, we show the typing judgement \( Γ ⊢ v : t \) stating that term \( v \) has type \( t \) in typing context \( Γ \). The typing judgement models a fairly standard dependent type system, except for the first rule typeSet. This rule expresses that set has type set in any context, a rule which is known as type-in-type and a known source of paradox in dependent type theories [28]. However, we use this rule only for ease of presentation. Our full code avoids type-in-type using a predicative hierarchy of universes similar to Agda’s [39]. It uses a level-indexed set\( i \), the typing rule that \( set_i : set_{i+1} \) for all \( i \), and a level-indexed typing judgement \( Γ ⊢ v : t \) with \( i \) such that \( Γ ⊢ set_i : set \) must hold.

In the remaining typing rules in Figure 5 we have typeVar, stating that the type of a variable is given by the corresponding entry in the typing context and typePi, stating that \( (x : S) → T \) is a type if \( S \) and \( T \) are types, with \( x : S \) added to the context for \( T \). For lambda expressions, typeLam says that \( λ (x : S) → b \) is typed \( (x : S) → T \) if \( b \) has type \( T \) in a context extended with

```plaintext
data _⊢_ : {n} (Γ : Context n) :
  Expr n → Expr n → Set where
  tyv : Γ ⊢ ε
typePi : ∀ {n e} {Γ : Context n} →
  Γ ⊢ e : set →
  Γ ⊢ e ≈ (e ≈ Γ)
tyconv : ∀ {n} (Γ : Context n) :
  m : ℕ (ρ : Sub m n) (tel : Context m) → Set where
  ttop : Γ ⊢ tel
```

Figure 6. Well-typed Contexts

```
weaken : irrefl ≈ set x y : Expr n →
  x ≈ y → y ≈ x → x ≈ y
≈− trans : set x y z : Expr n →
x ≈ y → y ≈ z → x ≈ z
≈−/: ∀ n {x y} {m : Sub n m} →
x ≈ y → x / y ≈ y / p
weakenJudgementTop : ∀ n {Γ : Context n} {v : Expr n} →
  Γ ⊢ v : t → t ⊢ Γ ⊢ weaken u : weaken t
subtJudgementTop : ∀ n {Γ : Context n} {t e t’} →
  t’ ⊢ e : t → t ⊢ Γ ⊢ e ≈ [v] : t [v]
subtContext : ∀ n {Γ : Context n} {e t t’} →
  t ≈ t’ → Γ ⊢ e : t →
  Γ ⊢ e ≈ t’
setp : ∀ {m} {e t} →
  e t ⊢ Γ ⊢ e : t →
  e t → Γ ⊢ e : t
setl : ∀ {n} {Γ : Context n} {e t t’} →
  t ≈ t’ → Γ ⊢ e : t →
  Γ ⊢ e : t’
```

Figure 7. Meta-theoretic properties of our typing judgements.

```
x : S. According to typeAppl, a function application f val has type
(λ (x : S) → T) val if f has type (x : S) → T and val has type S.
Note that we could equivalently have given such an application the
type T [val]. Finally, the rule typeConv states that a type t can be
substituted for a convertible type t’ in any typing judgement.

In the full version of our code, we extend the calculus with built-
```
```
Meta-theory and helper functions We have proved quite some meta-theory about the reduction, convertibility and typing judgements. For full detail we refer to the full version of our code, but to give you an idea of what is there, Figure 7 shows the types of the most important results. weakening = inj_{\approx} \approx shows that weakening is injective with respect to convertibility, \approx \rightarrow \text{trans} shows that convertibility is transitive, \approx \rightarrow \text{trans} is a consequence of the Church-Rosser-property for our reduction rules, which we have proved using a technique for untyped lambda calculi by Tait, described by Martin-Löf [36]. Theorem \approx \rightarrow \text{trans} states that convertibility is invariant under substitutions. Theorems weakeningJudgementTop, substJudgementTop, substContext and \vdash \rightarrow \vdash state roughly that typings are preserved under weakening, instantiating a variable in the context, replacing a type in the context by a convertible one and applying a substitution to term and type. \vdash \rightarrow \vdash is a simple proof that entries in a well-typed context must be sets. By theorem typesAreSets, the type of a judgement in a well-typed context must in fact be a type. Finally, substJudgementType is not a theorem but a simple helper function that replaces a judgement’s type by a provably equal type (it is a special case of subst, the standard eliminator of Agda’s singleton type \text{t} \equiv \text{t}').

Some example terms Let us consider the encoding of a simple example term: the following polymorphic identity function:

\begin{align*}
\text{id} : \forall (A : \text{Set}) \rightarrow A \rightarrow A \\
\text{id} = \lambda (A : \text{Set}) \rightarrow \lambda (v : A) \rightarrow v 
\end{align*}

The type and definition of this function are given by closed expressions \text{idTyTm} and \text{idTm}.

\begin{align*}
\text{idTm} & : \text{Expr} 0 \\
\text{idTm} & = \lambda\text{expr} \text{set} \text{ lambda} (\text{var} 0) (\text{var} 0) \\
\text{idTyTm} & : \text{Expr} 0 \\
\text{idTyTm} & = \text{pi set} (\text{fun} (\text{var} 0) (\text{var} 0)) 
\end{align*}

We can prove that the term \text{idTm} satisfies type \text{idTyTm} using the typing rules from Figure 5.

\begin{align*}
\text{ty}_{\text{idTm}} & : \vDash \text{idTm} : \text{idTyTm} \\
\text{ty}_{\text{idTm}} & = \text{typeLam} \text{ typeSet} (\text{typeLam} \text{ typeVar} \text{ typeVar}) 
\end{align*}

By the typesAreSets theorem, it follows that \text{idTyTm} is a type.

\begin{align*}
\text{ty}_{\text{idTyTm}} & : \vDash \text{idTyTm} : \text{set} \\
\text{ty}_{\text{idTyTm}} & = \text{typesAreSets} \text{ ty}_v \text{ ty}_{\text{idTm}} 
\end{align*}

3. Bringing Terms to Life

With this infrastructure in place, we can define our meta-programming primitive \text{interp} together with auxiliary primitives interpCtx and interpSet. Their types are:

\begin{align*}
\text{interpCtx} : \{n : \text{Nat}\} \rightarrow \Gamma \rightarrow \text{Context} n \\
\text{interpSet} : \{n : \text{Nat}\} \rightarrow \Gamma \rightarrow \text{Context} n \rightarrow \{A : \text{Expr} n\} \\
\text{interp} : \{n : \text{Nat}\} \rightarrow \Gamma \rightarrow \text{Context} n \rightarrow \{A : \text{Expr} n\} \\
\text{interpCtx} \text{ ty}_v \rightarrow \text{interpCtx} \text{ ty}_v \rightarrow \text{set} \\
\text{interpSet} \text{ ty}_v \rightarrow \text{interpSet} \text{ ty}_v \rightarrow \text{set} \\
\text{asmspts} : \text{interpCtx} \text{ ty}_v \rightarrow \text{interpSet} \text{ typesAreSets} \text{ ty}_v \text{ ty}_v \text{ asmspts} 
\end{align*}

\text{interpCtx} turns the types in a well-typed context into a dependent sum type of the context entries’ interpretations. It is used by the two other judgements to require values for all of a context’s assumptions. \text{interpSet} interprets an encoded type, yielding a \text{set}, and \text{interp} interprets a term \text{v} typed \text{t}. In the result type of \text{interp} for a proof \text{ty}_v of judgement \Gamma \vdash v : t, we use the previously mentioned theorem typesAreSets to calculate typesAreSets \text{ty}_v \text{ ty}_v, a proof that \Gamma \vdash t : \text{set}. The result of \text{interp} is then of type \text{t}, interpreted using \text{interpSet} and this derived judgement.

Interpreting examples Before we go into more details, consider again the previously encoded polymorphic identity function. Remember that the closed terms \text{idTm} and \text{idTyTm} encode the function and its type and the proofs \text{ty}_{\text{idTm}} and \text{ty}_{\text{idTyTm}} witness the typing judgements \vDash \text{idTm} : \text{idTyTm} and \vDash \text{idTyTm} : \text{set}. Both proofs assume only an empty context, which is always well typed according to the rule \text{ty}_v in Figure 6. We will discuss the reduction behaviour of our primitives further, but \text{interpCtx \text{ ty}_v}, (the assumptions in the empty context) reduces to unit type \top (with canonical inhabitant \text{tt}). With all of this, we can interpret the encoded type \text{idTyTm} to obtain the type \text{interp}_{\text{idTyTm}}:

\begin{align*}
\text{interp}_{\text{idTyTm}} = \text{interpSet} \text{ ty}_{\text{idTyTm}} \text{ ty}_v \text{ tt} 
\end{align*}

More details follow, but \text{interp}_{\text{idTyTm}} reduces to (x : \text{Set}) \rightarrow (x : x) \rightarrow x, alpha-equal to the intended type (A : \text{Set}) \rightarrow A \rightarrow A. Similarly, we can interpret term \text{idTm} and its typing proof \text{ty}_{\text{idTm}} to obtain \text{interp}_{\text{idTm}} of type \text{interp}_{\text{idTyTm}}:

\begin{align*}
\text{interp}_{\text{idTm}} = \text{interp} \text{ ty}_{\text{idTm}} \text{ ty}_v \text{ tt} 
\end{align*}

As we intended, \text{interp}_{\text{idTm}} reduces to \lambda (x : \text{Set}) \rightarrow (x : x) \rightarrow x, alpha-equal to our intended \lambda (x : \text{Set}) \rightarrow \lambda (x : A) \rightarrow x.

Interfacing with the real world In real examples, generated code needs to interface with existing types and values. In staging meta-programming systems, this is supported with a built-in quoting primitive, but we use an alternative approach. Suppose for example that we want a meta-program to construct the term \text{suc} 2 from the pre-existing value 2 and function \text{suc}. To do this, the meta-program clearly needs to refer to the type \text{N}, the function \text{suc} and the value 2 in the generated object code, but our term encoding does not provide a way to refer to such outside definitions. One solution would be to build natural numbers into our calculus as primitives, but this is not a scalable approach, since we cannot expect to do this for all types we will ever need, let alone a user’s custom types.

A better solution lets the meta-program construct the object term in a suitable context, postulating values of the correct types. Real values can then be provided in the interpretation of this context. For our example, we need the context \Gamma_{ex}:

\begin{align*}
\Gamma_{ex} = (\pi (\text{var} 1) (\text{var} 2)) \circ (\text{var} 0) \circ \text{set} \circ \epsilon 
\end{align*}

This definition should be read right-to-left: \circ is right-associative and the left-most context entries are added last and may refer to the values of entries to their right. It starts with the empty context \epsilon and lists the types for which we want to postulate values. In order, these are a type (of type \text{set}), a value of this type (of type \text{var} 0) and a function from this type to itself (of type \text{pi} (\text{var} 1) (\text{var} 2)). The context is intended to be instantiated to values \text{N}, 2 and \text{suc} respectively. Note that the de Bruijn variables \text{var} 0, \text{var} 1 and \text{var} 2 in the context all refer to the value of the rightmost context entry of type \text{set}; subsequent context entries have an additional variable in scope and the body of a \text{pi} as well. Proof \text{ty}_{\text{ex}} of judgement \Gamma_{ex} shows that context \Gamma_{ex} is well-typed, i.e. all entries are in fact sets:

\begin{align*}
\text{ty}_{\text{ex}} = \text{ty}_2 (\text{ty}_\epsilon (\text{ty}_\epsilon \text{ typeSet} \text{ typeVar}) (\text{typePi} \text{ typeVar} \text{ typeVar})) 
\end{align*}

We will fill in the appropriate values for this context’s assumptions with the value \text{asmspts}_{\text{ex}} of type \text{interpCtx} \text{ ty}_{\text{ex}}:

\begin{align*}
\text{asmspts}_{\text{ex}} = ((\text{tt}, \text{N}), 2, \text{suc}) 
\end{align*}

In context \Gamma_{ex}, we can now construct the value \text{suc} 2 as a term \text{ex}. It is an \text{Expr} \beta, since it may refer to \Gamma_{ex}’s three assumptions, and applies the postulated \text{suc} function to the postulated value 2.

\begin{align*}
ex = \text{appl} (\text{var} 0) (\text{var} 1) 
\end{align*}
We construct a proof $t_y \vdash \text{ex}: \nu \exists v. 2$ i.e. that the constructed term $\text{ex}$ has the first postulated value ($\forall$) as its type, in three steps. First typing rules typeApp and typeVar give us proof $t_y \vdash \text{ex}$, showing that $\text{ex}$ has a more complicated type. We then prove this type convertible to $\nu \exists v. 2$ (partly omitted) proof $\text{conv}_\text{ex} : t_y \vdash \text{ex}$ then uses typing rule typeConv to replace the convertible type.

$$
\begin{align*}
\text{ty}_\text{ex} : \Gamma \vdash \text{ex} : & \text{appl (lambda (var 2) (var 3)) (var 1)} \\
\text{ty}_\text{ex} = \text{typeApp typeVar typeVar typeVar} & \\
\text{conv}_\text{ex} : & \text{appl (lambda (var 2) (var 3)) (var 1)} \approx \nu \exists v. 2 \\
\text{conv}_\text{ex} = & \cdots \text{reduce (Application (var 3) (var 1))} \\
\text{ty}_\text{ex} : \Gamma \vdash \text{ex} : & \nu \exists v. 2 \\
\text{ty}_\text{ex} = \text{typeConv conv}_\text{ex} t_y \text{ty}_\text{ex} \text{typeVar}
\end{align*}
$$

We can then interpret object program $\text{ex}$ to obtain a value of type $\text{interpSet}$ (typesAreSets $t_y \text{ty}_\text{ex} \text{ty}_\text{ex} \text{asmps}_\text{tyex}$).

$$
\text{exInt} = \text{interp \text{ty}_\text{ex} \text{ty}_\text{ex} \text{asmps}_\text{tyex}}\]
$$

The reduction behaviour of our primitives that we will talk about next ensures that $\text{exInt}$'s type and $\text{exInt}$ itself reduce to $\mathbb{N}$ and $\text{suc} 2$ respectively, precisely as we intended.

Sometimes, a meta-program does not just need to refer to an external function $f$ in generated code, but also depends on information about such a function's reduction behaviour to prove well-typedness of the generated code. Without going into much detail, the ideas of this section can support this if we add singleton types to the object calculus. Concretely, a context could postulate the external function $f$ together with proofs of its reduction behaviour. Such proofs could then be used in the typing of generated programs and the invocation of the interpretation primitive would require actual proofs of the reduction behaviour in the context interpretation.

**Reduction behaviour** The reduction behaviour of our primitives is an important part of their definition and crucial for the functioning of the previous examples. We present the reduction rules in Figure 8. In general, these rules interpret encoded types, terms and contexts, but only when the well-typedness of the result can be guaranteed. To achieve the latter, we need to assert that the provided well-typedness proofs are valid and do not rely on assumptions that might not hold. This is non-trivial because a language like Agda applies strong reductions during type-checking, i.e. reductions can be applied to open terms as well as closed. Non-closed proofs are not necessarily valid, since they may rely on invalid assumptions. We will provide more insight further on and discuss our solution based on the value patterns in Figure 8. These are the patterns written in typewriter font in the left-hand sides of some reduction rules. Such a value pattern indicates that the rule must only be applied if the corresponding argument is a value. The types of these arguments are conversion or typing judgements and their values are finite trees of constructor applications (see Figures 3 and 5). As such, the property of value-ness can easily be checked in the primitives' implementation. But before we discuss the role of the value patterns further, let us take a better look at the reduction rules.

Recall the type of our most important primitive $\text{interp}$. $\text{interp} : \{ n : \mathbb{N} \} \{ \Gamma : \text{Context n} \} \{ v \ t : \text{Expr n} \} \rightarrow (t_y, \Gamma \vdash v \ t) \rightarrow (t_y, \vdash \Gamma) \rightarrow (\text{asmps} : \text{interpCtx t_y}) \rightarrow \text{interpSet} (\text{typesAreSets t_y} t_y \text{ty} \text{asmps})$

The primitive takes a context $\Gamma$, a term $v$ and a type $t$ as hidden arguments, followed by proofs $t_y, \Gamma \vdash v \ t$ and $\vdash \Gamma$ and a value $\text{asmps}$ of the context's interpretation type $\text{interpCtx t_y}$. The reduction rules in Figure 8 specify that for certain forms of the judgement $t_y \ t$, the primitive application reduces to appropriate right-hand sides. For $t_y = \text{typeSet}$, which implies $v = \text{set}$ and $t = \text{set}$, the first rule returns interpretation $\text{Set}$. For $t_y = \text{typeVar}$, an interpretation of the 4th context assumption is given by primitive $\text{interpVar}$, discussed below.

The rules for $t_y = \text{typePi} t_y, t_y$ and $\text{typeLam} t_y, t_y$ interpret terms $\pi s t$ and $\lambda t s h$ as respectively the corresponding Agda $\Pi$-type and lambda term, recursively constructed from interpretations of $s$ and $t$ resp. $h$. The bound variable $x$ is made available for the interpretation of $t$ resp. $b$ by placing it in the interpretation of the extended context $s \approx_i \Gamma$. For an application of a function to a value, we apply the interpretation of the function to the interpretation of the value. Note the value patterns on the left-hand side that we will come back to further on. Finally, the interpretation of $\text{interpSet}$ is simply the interpretation of the judgement whose type it substitutes, on the condition that the arguments are values.

Recall also the type of primitive $\text{interpCtx}$:

$$
\text{interpCtx} : \{ n : \mathbb{N} \} \{ \Gamma : \text{Context n} \} \rightarrow \Gamma \rightarrow \text{Set}
$$

The primitive takes a context $\Gamma$ as a hidden argument and a well-typedness proof for it and returns its interpretation, i.e. a type that contains all the context’s assumptions. We saw in $\text{interp}$’s reduction rules for $\text{typeLam}$ and $\text{typePi}$, how an extended context $s \approx_i \Gamma$ is interpreted by a tuple of the $s$ value and the interpretation of $\Gamma$. This corresponds to $\text{interpCtx}$’s reduction behaviour, that we look at now. The first reduction rule interprets an empty context by the unit type $\top$. More interestingly, a context $\Gamma$ extended with a type $t$ is interpreted by an interpretation $\text{asmps}$ of $\Gamma$, and an interpretation of the type $t$. We use a dependent sum $\exists t$ to specify the interpretation of $t$ with respect to the interpretation $\text{asmps}$ of the rest of the context.

Now that we know how to interpret a context, we can define reduction rules for $\text{interpVar}$, to project out a context’s $t$th entry. Its reduction rules are not surprising, projecting out the top assumption for variable $\text{zero}$ and recursing for $\text{suc} t$. The primitive $\text{interpSet}$ is a version of $\text{interp}$ that works on types only. Its role is to break the circularity in the types of the primitives. It is implemented in terms of helper primitives $\text{interpSet’}$ and $\text{interpVarSet’}$. We do not discuss their reduction behaviour as it is similar to $\text{interp}$ and $\text{interpVar}$ except that we require proof that the judgement’s type is convertible to $\text{set}$ and that this proof is a value in some cases.

**Soundness in the presence of open terms** To understand the value patterns in five of the reduction rules in Figure 8, we have to explain the powerful form of type-level computation that a dependently typed language like Agda uses. It uses a strong form of reduction, strong reductions can be applied even inside the body of lambda or pi terms. The term $\lambda x \rightarrow 0 + x$, for example, is considered equal to $\lambda x \rightarrow x$, because $0 + x$ is reduced to $x$ despite the open variable $x$. However, such strong reductions can be dangerous because, in the presence of open variables, we may be reasoning under absurd assumptions. Consider the following function:

$$
\text{absurdTerm} = \lambda (\text{prf} : \text{Int} \equiv \text{Bool}) \rightarrow \text{cast prf} 3 \lor \text{false}
$$

The function $\text{absurdTerm}$ takes a proof $\text{prf}$ that $\text{Int} \equiv \text{Bool}$, modelling an equality proof of types $\text{Int}$ and $\text{Bool}$. This proof type is of course empty, but the type-checker is not aware of that. With $\text{prf}$ and an appropriate $\text{cast}$ function, we can use a value $3$ as a $\text{Bool}$. However, this is not problematic, because a correct definition of the $\text{cast}$ function will never reduce $\text{cast prf} 3$ to $3$. Instead, it will block on the open variable $\text{prf}$ until a value (i.e. $\text{refl}$) is somehow substituted for it. This mechanism effectively protects values like $3$ from being used at wrong types like $\text{Bool}$.

For our primitives, similar issues arise. We can for example assume a proof $t_y \text{Absurd}$ of judgement $\epsilon \vdash \text{set} : \pi \text{set} \text{set}$ even

---

4: Note: pattern matches that imply equalities about other arguments are standard for dependent pattern matching [25].
the interpretation of a type after a well-typed substitution $\Gamma_{t} \vdash \phi; \Gamma_{f}$, between well-typed contexts $\Gamma_{f}$ and $\Gamma_{t}$, $\text{interpCompSubCtx}$ says that an interpretation of $\Gamma_{t}$ can be constructed from one of $\Gamma_{f}$ and $\text{interpCompSubCtxSet}$ says that the interpretation of a type $t$ in $\Gamma_{t}$ is the same as that of $t / \phi$ in $\Gamma_{f}$ using the interpretation of $\Gamma_{f}$ constructed by $\text{interpCompSubCtx}$.

We are currently using stub proofs of these properties, based on an Agda primitive called $\text{primTrustMe}$. $\text{primTrustMe}$ is an unsafe primitive that proves equalities $a \equiv b$ for any set $A$ and values $a, b$ of type $A$. However, during type-checking, $\text{primTrustMe}$ only reduces to $\text{refl}$ when $a$ and $b$ are definitionally equal. It is future work to ascertain that these properties follow from the reduction rules of Figure 8 and the proofs of theorems like $\vdash \bot$. 

4. Applications

Our approach allows definitions of powerful meta-programs, manipulating both code and types, in a functional style and with very precise types. In this section, we demonstrate this for two important applications: datatype-generic programming and tactics.

4.1 Datatype-Generic Programming

The field of datatype-generic programming studies the definition of algorithms that work for a wide variety of data types. An example is Haskell’s deriving $\text{Show}$ mechanism [35, §4.3.2, §11], which allows a data type $A$ to be annotated with the directive $\text{deriving Show}$ to make the compiler derive an instance of the $\text{Show}$ type class. Such an instance consists essentially of a function $\text{show} :: A \rightarrow \text{String}$, derived syntactically by the compiler from the data type’s constructors and their types. The goal of datatype-generic programming is to allow functions like $\text{show}$ to be defined in a generic way, i.e. such that they can be defined once but used with a wide variety of data types.

Representing data types To apply our techniques to the field of datatype-generic programming, we start from a syntactic representation of an inductive data type:

```haskell
record SimpleDT (A : Set) : Set where
  constructor simpleDT
  field constructors : List (Constructor A)
  folder : folderType A constructors
```

Figure 8. Reduction behaviour of our primitives. Patterns in typewriter font are required to be values.
According to this definition, a data type $A$ is syntactically described by a list of its constructors and a folder or induction principle (List is a standard type of finite lists). To keep things simple, we omit well-formedness requirements (like positivity of the definition) and proofs about the reduction behaviour of the folder function, which are required to completely describe a data type, but not needed for our example application. Constructor is the syntactic representation of a single constructor:

```
data Constructor (A : Set) : Set where
mkConstructor : String → (n : ℕ) →
(tel : Telescope 1 (n + 1)) → (ty_set : Γset ⊢ tel) →
let ctorT = funCtx n tel (var 0)
ty_ctorT = Γset ⊢ ctorT : set
ty_ctorT = typeFunCtx n ty_set ty_var typeVar
in interpret ty_set ty_ctorT (tt, A) → Constructor A
```

We describe a constructor by its name as a String, its arity $n$ and a telescope $tel$ containing the types of its arguments. The telescope has one initial variable in scope: the data type $A$ itself, so that it can be referenced in the types of constructor arguments. The telescope $tel$ must be well-typed in the context $\Gamma set = set \triangleq e$, i.e. with the premise that $A$ is a set. From $tel$, we can calculate the full type $ctorT$ of the constructor as the function that takes the arguments given by $tel$ and produces a value of type $A$ (using omitted helper function $funCtx$). We prove that $ctorT$ is a set (using omitted lemma $typeFunCtx$), interpret it and require a value of it, i.e. the actual constructor. Note how our meta-programming primitives provide the crucial link between the syntactically represented types and the normal type of the actual constructor.

In addition to the list of Constructors, SimpleDT contains an eliminator or folder for the data type. Every inductive data type comes with such an induction principle, which models a general way of perform structural induction over the data type. The function $folderType$ syntactically derives the type of this induction principle from the types of the constructors and their interpretations.

```
folderType : (A : Set) → List (Constructor A) → Set
folderType A constructors = (P : A → Set) →
underFolderAsmts P constructors (x : A) → P x
```

Given a set $A$ and a list of $A$’s constructors, $folderType$ returns the type for a corresponding induction principle: it takes a predicate $P : A → Set$ (the motive [25], describing what the induction principle should produce) and returns a function of type $(x : A) → P x$ under a number of assumptions. For every constructor, the function $underFolderAsmts$ syntactically derives the type of an assumption from the constructor’s type. This is fairly involved, but presents no fundamental difficulties and we omit it for space reasons.

Let us immediately show some data types and their representations. The simplest example is the empty type, which has zero constructors. Its definition and induction principle look as follows:

```
data ⊥ : Set where
foldBot : (P : ⊥ → Set) → (t : ⊥) → P t
foldBot P ()
```

Note the use of an absurd pattern ($\lambda$) in the definition of $foldBot$. This pattern communicates to Agda that no value can ever be given for the argument of type $\bot$, so that a right-hand-side is not needed. It is easy to provide a value of SimpleDT for $\bot$:

```
botDT : SimpleDT ⊥
botDT = simpleDT [] foldBot
```

$botDT$ specifies that $\bot$ has no constructors and $foldBot$ is its induction principle. Agda successfully type-checks $foldBot$ against the folder type calculated for the empty list of constructors.

For a more complex example, consider the standard definition of natural numbers and its induction principle:

```
data ℕ : Set where zero : ℕ
suc : ℕ → ℕ
foldℕ : (P : ℕ → Set) → P zero →
(∀ n → P n → P (suc n)) → (n : ℕ) → P n
foldℕ P Ps zero = Ps
foldℕ P Ps (suc n) = Ps n (foldℕ P Ps P n)
```

The constructors $zero$ and $suc$ of data type $\mathbb{N}$ are described by $zeroConstr$ and $sucConstr$ of type Constructor $\mathbb{N}$:

```
zeroConstr = mkConstructor "zero" 0 ε ty_zero zero
sucConstr = mkConstructor "suc" 1 (var 0 ε) ty_var typeVar suc
```

The constructor $zero$ is of arity 0, with the empty telescope describing its arguments. The actual constructor $zero$ is then provided and Agda checks its type against the one calculated from the syntactic description. Constructor $suc$ is of arity 1, taking one value of type $\mathbb{N}$ as its argument (recall that $\var 0$ in the constructor telescope refers to the data type itself). The constructor telescope is well-typed under $Γset$’s assumption that $\var 0$ is a set. Again, the actual constructor is given and checked against the type calculated from the description. We can now describe $\mathbb{N}$ with $natDT : SimpleDT \mathbb{N}$.

```
natDT = simpleDT [zeroConstr, sucConstr] foldℕ
```

$natDT$ lists $\mathbb{N}$’s constructors and provides induction principle $foldℕ$, checked against the type calculated from the constructors.

**Derive Show** The type SimpleDT is a general syntactic description of inductive data types that permits a general form of datatype-generic meta-program. As a proof-of-concept, we show the function $deriveShow$ that derives a show function for a data type $A$.

```
deriveShow : ∀ (A) → SimpleDT A → A → String
deriveShow (simpleDT constructors folder) = omitted
```

We omit the algorithm’s implementation, which takes the description of data type $A$ and exploits the induction principle with motive $P = λ _ → String$. It syntactically derives arguments for the folder, specifying how values constructed using the different
constructors are to be serialised. The hardest part of the code is to
convince the type-checker that the folder arguments we con-
struct for the concrete motive \( \lambda \_ \rightarrow \text{String} \) correspond to their
expected types for a general predicate \( P \) when \( P \) is instantiated to
\( \lambda \_ \rightarrow \text{String} \) through the context interpretation. This essentially
uses the \texttt{interpCompSubCtx} and \texttt{interpCompSubSet} primitive
properties shown in Figure 9.

For our example data types, \texttt{deriveShow} derives an (admittedly
every useful) \textit{show} function for:

\[
\begin{align*}
\text{showBot} &: \bot \rightarrow \text{String} \\
\text{showBot} &= \text{deriveShow botDT}
\end{align*}
\]

\texttt{showBot}’s definition reduces to \texttt{foldBot} (\( \lambda \_ \rightarrow \text{String} \)),
the code that \texttt{deriveShow} syntactically generates. From \texttt{natDT}, we
can derive the function \texttt{showNat} of type \( \text{N} \rightarrow \text{String} \).

\[
\text{showNat} = \text{deriveShow natDT}
\]

Like for \texttt{showBot}, \texttt{showNat}’s definition reduces to the generated
function \texttt{showNat}’ = \texttt{foldN} (\( \lambda \_ \rightarrow \text{String} \)) "zero" (\( \ldots \)) (final
argument omitted). We can apply it to numbers with for example
\texttt{showNat} 2 producing the string "(\texttt{succ} (\texttt{succ} zero))".

\textbf{Discussion} This account of datatype-generic programming is
rudimentary, lacking support for indices and parameters and non-
recursive and more general recursive constructor arguments [21].
It does not exclude non-strictly-positive data types and does not
contain proofs about the induction principle’s reduction behaviour
(required to construct proofs about inductive functions). However,
we do not see fundamental obstacles for adding any of this.

From a methodological point of view, our account of datatype-
generic programming is compelling: meta-programs are written
in the language itself, using the language’s standard functional
computational model. The syntactic description of a data type in
\textit{SimpleDT} is general and could be automatically generated by the
compiler. Modulo correctness of our primitives, the meta-programs
come with strong guarantees about termination, well-typedness of
the generated programs and completeness.

\textit{SimpleDT} and \texttt{deriveShow} are implemented in \( \pm 1200 \) lines
of code and can be studied in the full version of our code (see the
footnote on page 2). This is still much more than what we would
like, and in Section 5 we discuss how this could be improved.

\section{4.2 Tactics}

Tactics are a form of meta-programs that solve or refine proof obli-
gations in proof assistants. In proof assistants based on dependent
type theory, solving a proof obligation is equivalent to producing
a program of a specified type in a specified context. Several proof
assistants provide support for writing tactics, often in the form of
a special-purpose sub-language. Such tactics are generally untyped
and provide little upfront guarantees about their correct operation.
Even though the correctness of the generated proofs can be checked
after generation, Gonthier et al. argue that untyped tactics can be
hard to maintain and compose and giving them more precise types
is a good approach to solve this issue [26]. There are also perform-
ance advantages to tactics that can be guaranteed to terminate
correctly without running them, as argued by Chlipala [14].

Our meta-programming primitives show promise for this field,
and they lend themselves to a typed form of tactics written in a
standard functional style. The input for a tactic is just a syntactic
representation of the proof obligation, i.e. a certain type in a certain
context. By additionally requiring a typing judgement for the type
and interpretations for the context’s values, we can use \texttt{interpSet}
to specify the expected result type of the tactic.

Consider the following analogue of Coq’s \texttt{assumption} tactic, a
simple tactic that solves proof obligations which appear literally in
the context. Our account of it enjoys a very precise type:

\[
\begin{align*}
\text{assumptionTactic} &: \forall \{ n \ T \} \{ \Gamma : \text{Context} \ n \} \rightarrow \\
&\{ \texttt{ty}_T : \Gamma \vdash \text{t} : \text{set} \} \rightarrow \\
&\{ \texttt{ty}_T : \Gamma \vdash \text{\texttt{P}} \} \rightarrow \{ \texttt{asms}\texttt{t} : \texttt{interpCtx} \ \texttt{ty}_T \} \rightarrow \\
&\text{ifYes (inContext} \ \Gamma \ \text{t} \} \{ \texttt{interpSet} \ \texttt{ty}_T \ \texttt{ty}_S \texttt{asms}\}.
\end{align*}
\]

The tactic takes a type \( T \), a well-typed context \( \Gamma \) and values for
its assumptions. The return type will be explained further, but it
specifies exactly what the tactic will return in all cases: either a
value of type \( T \) if \( T \) is present in the context or a value of the unit
type otherwise. Let us explain this in more detail.

We use the Agda standard library’s \texttt{Dec} \( P \) type. It models a
decision of proposition \( P \), i.e. either a proof of \( P \) or a proof of \( \neg P \):

\[
\text{data Dec} (P : \text{Set}) : \text{Set} \text{ where yes} : P \rightarrow \text{Dec} P \\
\text{no} : \neg P \rightarrow \text{Dec} P
\]

Based on a decision of some property, the \texttt{ifYes} function returns
either an argument type or unit type \( T \) :

\[
\begin{align*}
\text{ifYes} : \{ P : \text{Set} \} \rightarrow \text{Dec} P \rightarrow \text{Set} & \\
\text{yes} : P \rightarrow \text{Dec} P & \\
\text{no} : \neg P \rightarrow \text{Dec} P &
\end{align*}
\]

The \texttt{inContext}? algorithm decides whether or not a certain type
\( t \) is present in context \( \Gamma \), i.e. if the \( i \)th entry in the context is equal
to \( t \) for some \( i \). It uses a general purpose decision procedure \texttt{angl},
which simply tries all \( i \) of the bounded type \( \text{Fin} \ n \). For a given
variable \( i \), we use a general equality decision procedure for terms
\( \pm i \), to check whether the \( i \)th context entry is equal to \( t \).

\[
\begin{align*}
\text{InContext} : \{ n : \text{N} \} \{ \Gamma : \text{Context} \ n \} \{ t : \text{Expr} \ n \} \rightarrow \text{Set} & \\
\text{InContext} \ t = \exists i . \text{lookup} \ i \ \Gamma \equiv t & \\
\text{inContext}? : \{ n : \text{N} \} \{ \Gamma : \text{Context} \ n \} \{ t : \text{Expr} \ n \} \rightarrow \\
\text{Dec (inContext} \ \Gamma \ t) &
\end{align*}
\]

In our \texttt{assumptionTactic}, we use a \texttt{with} pattern match to
make a case distinction based on the decision from \texttt{inContext}!
If the type \( t \) is not found, we can simply return \( \top \) value \( t \). If it is
found at position \( i \), we essentially want to return the \( i \)th entry in the
context but we need to convince Agda that it has the desired type.

\[
\begin{align*}
\text{assumptionTactic} \ \texttt{ty}_T \ \texttt{ty}_S \texttt{asms} \texttt{with} \ \texttt{inContext}? \ \Gamma \ t & \\
\text{assumptionTactic} \ \{ n \} \{ t \} \{ \Gamma \} \ \texttt{ty}_T \ \texttt{ty}_S \ \texttt{asms} & \\
\text{yes} (i, \texttt{eq}_i) &= \\
\text{let} \ \texttt{ty}_\text{var} = \Gamma \vdash \text{var} \ i : t & \\
\texttt{ty}_\text{var} = \text{substJudgementType} \ \texttt{eq}_i \ \texttt{ty}_\text{var} \ \texttt{typeVar} & \\
\texttt{inContext} & \\
\texttt{assumptionTactic} \ \texttt{ty}_T \ \texttt{ty}_S \ \texttt{asms} \ | \ \text{no} \ _= \ _= t &
\end{align*}
\]

The first step is to use the proof \( \texttt{eq}_i \) that \( \text{lookup} \ i \ \Gamma \equiv t \) from
\texttt{inContext}! and the \texttt{typeVar} typing rule to produce a proof \( \texttt{ty}_\text{var} \)
of judgement \( \Gamma \vdash \text{var} \ i : t \). We can then obtain the interpretation
of the \( i \)th variable through the value \( \text{interp} \ \texttt{ty}_\text{var} \ \texttt{ty}_T \ \texttt{asms} \).
Unfortunately, that value’s type is

\[
\text{interpSet} \ \texttt{ty}_T \ \texttt{ty}_S \ \texttt{asms} \texttt{with} \ \texttt{inContext}! \texttt{ty}_T \ \texttt{ty}_S \texttt{asms} \texttt{with} \ \texttt{typeVar} \ \texttt{ty}_T \ \texttt{ty}_S \texttt{asms} \texttt{with} \ \texttt{inContext}!
\]

What we need is a value of type \( \text{interpSet} \ \texttt{ty}_T \ \texttt{ty}_S \ \texttt{asms} \), i.e. an
interpretation of the same type \( t \), but for a different proof that \( t \) is a
set. \texttt{castInterp}, an omitted special case of property \texttt{castInterp}– \( \approx \)
from Figure 9, is precisely what we need to cast one to the other.

\textbf{Tactic usage} Currently, our tactics can be manually invoked with
a context and goal type and well-formedness proofs. The tactic in-
vocation appears as an expression in the code where the goal is
needed. In future systems, compiler support can increase conven-
ience by automatically providing the goal type, context and their
typing proofs. This could e.g. extend Agda’s experimental and underdocumented quoteGoal construct. This construct allows the invocation of a reflective solver with the compiler providing a syntactic representation of the goal type. It does not however provide a syntactic representation of the context or a guarantee about well-formedness of the provided type. Also, a more developed tactic API could support returning unsolved sub-goals and tactic combiners like Coq’s ’;’.

5. Discussion

There are some more aspects of our approach that we believe deserve further discussion: the representation of the object language, the performance of our meta-programs, the overhead for writing meta-programs in our system and the soundness of our primitives.

Types and Guarantees

Considering our example meta-programs deriveShow and assumptionTactic, an important feature of our meta-programming approach is the strong guarantees that the meta-programs’ types provide, modulo the soundness of our primitives. First, meta-programs are strongly type-safe: any object code they generate must be well-typed, since they are required to provide a proof of well-typedness to the interpretation primitive. Second, our meta-language Agda checks termination and completeness of pattern matches for all function definitions to guarantee that all functions are total. This guarantee also applies to our meta-programs, so that additionally we automatically get a totality guarantee for our meta-programs. However, this does not completely exclude the use of general recursion in tactics, techniques like Daniëlssohn’s partiality monad [16] can be used to model such algorithms.

The representation

Meta-programming implies the syntactic analysis and construction of source code and/or types, and we have chosen a fairly well-understood representation to support this: a lambda calculus with de Bruijn indices and a standard separate encoding of typing judgements. However, many different encodings are equally possible, like those based on more advanced representations of binders [12]. It is future work to investigate the advantages that these alternatives might offer for our purposes. We also want to investigate merging interpSet and interp, but we cannot currently try this for technical reasons. Finally, we currently represent typing judgements externally, i.e. as a property that can be true or not for an untyped lambda term. This corresponds to standard presentations of type theory, but it may be interesting to explore the benefits of an internal encoding like Daniëlssohn, Chapman or McBride’s [9, 15, 38] in our setting.

Performance

We do not currently consider our implementation practical, because of performance reasons. For example, type-checking just the deriveShow example for the type of natural numbers currently takes about 2 minutes and 3GB of memory on our system. Such performance likely prohibits all practical applications. However, we do not think this bad performance is inherent to our approach, but rather a consequence of the inefficient call-by-name execution strategy that Agda uses during type-checking. Remember how we previously defined showNat using our deriveShow function. As we mentioned, showNat is definitionally equal to the generated program showNat′ \( \equiv \) foldN(\( \lambda \_ \rightarrow \text{String} \) "zero" \( \ldots \)). Nevertheless, applying showNat to the numbers 0 and 1 under Agda’s evaluator (which is also used during type-checking) takes 2.5 resp. 11 minutes while for showNat′, it is instantaneous for numbers up to at least 100. For larger numbers, showNat quickly runs out of memory.

This behaviour is a consequence of Agda’s call-by-name evaluation strategy, which repeats the normalisation of showNat for every reduction of foldN. If Agda were to use a more efficient strategy like call-by-need, then the normalisation of showNat to showNat′ would occur only once. Very likely, there is a lot more work being duplicated inside the normalisation of showNat and we believe the call-by-name evaluation strategy is responsible for the long execution and type-checking times there as well.

Overhead

Writing meta-programs in our approach entails a certain amount of programming overhead. The full code of our datatype-generic meta-programming application deriveShow is \( \pm 1200 \) lines of code (including the SimpleDT encoding and some reusable parts). This is a lot more than what it would take to write a corresponding untyped meta-program. A significant part is the correctness proof of the meta-program (i.e. the proof that it generates correct code for all inputs).

However, a big part of our deriveShow implementation consists of a rather tedious proof specific to our meta-programming primitives. It concerns the correspondence of a type in a context with a general predicate \( P \) of type \( A \rightarrow \text{String} \), with the value \( \lambda \_ \rightarrow \text{String} \) provided through the interpretation of this context and the same type with an encoding of \( \lambda \_ \rightarrow \text{String} \) already filled in. We expect quite some work can be saved in this proof, but long compilation times have prevented further investigation. On the bright side, our assumption tactic is only about 50 lines in total, for a big part because it reuses general functions like the decision procedure for syntactic term equality. It is likely that additional reusable functions can reduce the meta-programming effort further. For example, a verified type-inference algorithm can be combined with our primitives to obviate the need for manual typing proofs in many cases.

Finally, we also expect that more experience with the definition of interpretation primitives would provide further opportunities to reduce meta-programming effort. For example, it would likely simplify some things to merge interp and interpSet, but we currently cannot do so for technical reasons. Additionally, the irrelevant arguments [1] that Agda support offer the potential to make Agda understand that the type correctness proofs that our primitives require are only required to exist but do not influence their result value. We expect this could make a big difference for shortening tedious proofs like the one in our definition of deriveShow.

Soundness

The soundness of our primitives remains an open question, at least if we consider the full version that does not have the unsound \( \vdash \) set : set rule that we discussed in Section 2. However, we do think there is a relation to the field of foundational logic that we will try to informally explain here. What we are essentially doing is reasoning about Agda terms within Agda itself. In foundational mathematical logic, Gödel’s second incompleteness theorem has something to say about a similar situation for first-order logic [24]. An informal statement of the theorem (found on Wikipedia [54]) reads

\[ \text{Theorem 1 (Gödel’s Second Incompleteness Theorem). For any formal effectively generated theory } T \text{ including basic arithmetical truths and also certain truths about formal provability, if } T \text{ includes a statement of its own consistency then } T \text{ is inconsistent.} \]

A standard proof of this theorem constructs a proposition \( T \) in the object theory such that \( T \) asserts the unprovability of its own Gödel-encoding. In vague terms, it can be proven that such a term exists as soon as the object language is powerful enough to reason about natural numbers. Such a term leads to a contradiction in combination with the self-consistency proof of the theory.

It is fair to assume the theorem can be generalised to type theory, and applied to our object theory, perhaps after adding singleton types, an empty type and a type of natural numbers. Consistency of a dependent type theory is equivalent with the existence of a closed term of type \( \bot \). Using our primitives, it is not hard to construct a function of type \( \forall \{ t \} \rightarrow \bot : \text{constant} \ bot \ [\] \rightarrow \bot \), which means
that our meta-level primitives imply the consistency of our object theory. This begs the question whether Agda extended with our primitive must therefore necessarily be inconsistent, by the second incompleteness theorem, since it implies its own consistency. We conjecture that this implication is not there, for the reason that our object calculus does not contain the primitive itself, making it a fundamentally weaker theory. What we do is reminiscent of extending a first-order logical theory $T$ with an axiom asserting $\neg \text{Cons}(T)$. Such an extended theory $T'$ does not in fact prove its own consistency, just that of $T$, so that the second incompleteness theorem does not apply. Another question that Gödel’s result suggests is whether primitives like ours could in principle be implemented as normal functions within the bounds of a meta-language. Even with sufficient additional features like induction-recursion [22], this might not be possible as it would prove the language’s own consistency within itself.

For these reasons, we expect that our primitives are not implementable in pure Agda but do not compromise consistency. Because of Gödel incompleteness, we think there are only two options to gain more confidence in them: either prove consistency of the extended calculus in a strictly stronger logical system such as Zermelo-Fraenkel set theory or implement our primitives in pure Agda, relying on axioms that are easier to trust than our primitives. A non-computational axiom asserting strong normalisation of the calculus (as used by Barras [4]) is a good candidate, but it isn’t practical in our current implementation because Agda lacks a Prop universe like Coq’s.

We think these logical aspects of our work deserve further attention. Nevertheless, even if our primitives were to be proven unsound, we do not think our work would be useless. Our application of interpretation primitives to meta-programming remains relevant as long as the primitives can be restricted to regain soundness. Also, in some applications of a dependently-typed language for programming (rather than proof checking), full certainty about soundness can be less important than powerful meta-programming support.

### 6. Related Work

In the literature, we find different forms of programming language support for meta-programming. We discuss them according to the guarantees that are provided about object programs.

Many approaches represent code in an untyped way, i.e., without guarantees that the represented source code is well-typed. These techniques have no way of providing strong type-safety of metaprograms, i.e., a guarantee that all the code a meta-program will ever produce is well-typed. In this category, we include approaches that represent code textually, like parser generators [29, 41], C macro’s, eval primitives like JavaScript’s [45], Java’s pluggable annotation processors [19] (at least on the output side). Some approaches generate untyped bytecode [8]. Also in this category are macro approaches which receive and produce an untyped data structure representation of programs and types, like Template Haskell [48], Ltac proof tactics in Coq [20] and macro systems in Lisp-related languages (e.g. Racket [53]). Some provide specific language features for working with such representations. These systems provide the power of meta-programming at a comparatively low cost, but they make it hard to provide upfront guarantees of (strong) type-safety.

Not all meta-programming approaches are based on an explicit syntactic representation of terms or types. Some exploit type system features like Haskell type classes [33], Coq canonical structures [26] or C++ templates [2] to analyse types and produce code as part of the type inference process. These features provide (intentionally or not) a form of type-level computation with at least a notion of type analysis and structural recursion. Gonthier et al. explore canonical structures (non-trivially) to obtain a form of syntactic pattern matching and non-determinism with backtracking [26]. Meta-programming systems based on such primitives only support analysing types (but dependent types in Coq may contain terms). The computational model of these primitives is quite different from the underlying language’s (unification-based vs. functional), so that meta-programming requires special expertise and techniques. For canonical structures, the computational model is not so well understood [27] and the resulting meta-programs are tightly coupled to the precise behaviour of the inferencer. An advantage of using primitives exposed by the type inferencer is that strong type-safety can be guaranteed comparatively easily [26, 33]. The type class instance search always terminates (with common extensions), but not so for C++ templates and Coq canonical structures. Completeness of pattern matching is not statically checked in any system. More or less in this category, we also have Chlipala’s language Ur, which provides value-level folder functions for record types to support a practical form of meta-programming [13] with a form of syntactic analysis of record types, no explicit representation of object code and a functional computational model. Syntactic analysis of terms or general types is not supported.

Other approaches to meta-programming with strong type safety are based on explicit typed representations of code. This requires a powerful meta-language type system, as determined by the complexity of the object language and whether types, terms and typing contexts can all be syntactically constructed and analysed or only some of those. We discuss the related work according to the type system feature used in this representation.

Rudolph and Thieman represent typed JVM bytecode generators in the Scala Mnemonics library [47], exploiting various features of Scala’s type system. Taha and Sheard [52], Chen and Xi [11], Pašalić and Linger [42] and Sheard and Pašalić [49]’s systems are based on GADTs or explicit type equality proofs. Terms of a non-dependently-typed object language are syntactically represented as values of a data type indexed with the meta-level type of the term they represent. Without analysis of types, these techniques appear unsuitable for applications like proof tactics.
In VeriML, Stampoulis and Shao [50, 51] use a contextual type system, inspired by Beluga [43] and Delphin [44], in the meta-language to model a dependently-typed object language. They provide a syntactic model of terms and types, with a certain level of support for parameterising over and pattern matching on typing contexts. Nevertheless, contexts do not seem first class in VeriML’s type system. For example, tactics cannot have contexts as their return type, so meta-programs cannot construct them, only start from the ones they receive and extend them locally. Stampoulis and Shao use an imperative meta-language with general recursion because certain tactics use algorithms that are inherently imperative. We agree that such tactics exist, but we do not see why they cannot be modelled in a pure and/or total functional setting like ours, using models like those found in the literature [16, 30, 32]. VeriML tactics are partial: they can fail or loop forever. This has modularity disadvantages: if a tactic $t_1$ invokes another tactic $t_2$, then $t_1$’s author cannot be sure that $t_2$ will actually succeed when it is invoked at $t_1$’s run-time. Stampoulis and Shao partially solve this with a letstactic staging construct that forces tactic $t_2$ to be evaluated at $t_1$’s compile time instead. This works under certain restrictions on $t_2$’s arguments. Because our tactics’ types imply termination guarantees by default, we do not need such a system, while potential non-termination can still be modelled, e.g. using the non-termination monad [16]. Stampoulis and Shao link a proof assistant’s type checker with custom tactics to obtain the effect of a sound user-extensible conversion rule in the logic [51], allowing a term $t$ of type $A$ to be used at type $A'$ if the equality decision procedure (potentially a custom tactic) can find a proof that $A = A'$. This form of automatic triggering of tactics for solving constraints is interesting and could perhaps be combined with our work as well.

In a dependently-typed meta-language, it is possible to model non-dependent object languages with standard inductively-defined universes using the technique of reflection [5, 14]. Altenkirch and McBride [3] and Chapman et al. [10] provide syntactic models of data types, together with interpretation functions. Chapman et al.’s universe even describes itself as a data type. These authors do not consider syntactic models of terms or types that are not data types. Brady and Hammond [6] provide a universe that models a non-dependent object language. Terms, types as well as contexts are modelled and can be syntactically constructed and analysed.

This universe-based approach can be extended to dependently-typed object languages using the advanced type-theoretic concept of inductive-recursive definitions [22]. This has been studied by Danielsson [15], Chapman [9] and McBride [38]. These authors provide typed syntactic models of dependently-typed calculi in dependently-typed calculi, with different objectives than ours. Where we focus on the applicability of such a model in meta-programming primitives, they aim to prove properties of the modelled language in the meta-language. They use models based on advanced type-theory features like induction-recursion and mutual induction. All three authors use a model of the object calculus with terms indexed by encodings of their types, instead of an external typing judgement like ours. The models that they use are specifically tailored to enable proofs of deep properties like normalisation, and it is unclear if their models also fit our more practical objectives. Finally, these approaches generally try to stay within the limits of the features of an existing dependently typed language (albeit one with powerful features like inductive-recursive definitions). They try hard to fit their models and interpretation functions (more or less equivalent to the normalisation proof of the object language) in a known inductive-recursive schema, not fully successfully [9, 15]. McBride’s encoding is accepted by Agda but he has to significantly limit the dependent nature of his object language [38]. As discussed in the introduction, our use of interpretation primitives allows us to side-step the interesting but hard problems that these authors tackle, leaving us free to study the application of related techniques to concrete meta-programming applications. It also allows us to use a more conventional encoding of the object language based on external typing judgements.

7. Conclusion

Our primitives present a novel meta-programming model with several desirable characteristics. Our meta-programs use the same functional style and well-understood computational model as normal programs. They can be given precise types that guarantee termination and strong type-safety. Finally, they can construct and analyse terms, types and typing contexts in a type-safe way. Our proof-of-concept applications in the two important application domains of datatype-generic programming and tactics, demonstrate the generality of our approach. Still, we feel this work is only a first exploration of a new approach to meta-programming. Quite some interesting questions remain to be answered in future work.

Acknowledgments

This research is partially funded by the Research Foundation – Flanders (FWO), and by the Research Fund KU Leuven. Dominique Devriese holds a Ph.D. fellowship of the Research Foundation – Flanders (FWO).

References
