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, 23, 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].

ICFP '13, September 25–27, 2013, Boston, MA, USA.

http://dx.doi.org/10.1145/2500365.2500575

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 $show : A \to String$ for a data type A.

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

$SimpleDT$ 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.[1] 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.

1.1 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

[1] Code available on `http://people.cs.kuleuven.be/dominique.devriese/permanent/tsmp.zip`.

```
data Constant : (arity : ℕ) → Set where
  (empty for now)
data Binder : Set where Π Λ : 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 Λ
```

Figure 1. The representation of 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 defined as a standard inductive data type [21], with an enumeration of its constructors and their types. The *set* constructor represents the type of types in the object language and free variables are embedded through *var*. There is a standard function application constructor *appl* and constants applied to a fixed number of arguments (as determined by the constant's *arity*) through term constructor *constant*. $Vec\ A\ n$ is another Agda standard library type representing a vector of precisely n values of type A. In what follows, we use $[\,]$ 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 $\lambda(x : T) \to b$[2] and dependent function types $(x : T) \to T'$, constructed as $bind\ \Lambda$ and $bind\ \Pi$ 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 $s \to t$ can be represented as dependent function $(\_ : s) \to t$. Finally, note in the type of *constant*, *pi* and *lambda* that we bind some arguments using curly brackets, indicating that they can be omitted in calls. Agda will then infer their value from the types of the remaining arguments.

[2] We use Agda notation for lambdas, not the more standard $\lambda x : T.b$.

```
Sub      : ℕ → ℕ → Set
_/_      : {m n : ℕ} → Expr m → Sub m n → Expr n
weaken   : {n : ℕ} → Expr n → Expr (suc n)
_[_]     : {n : ℕ} → Expr (suc n) → Expr n → Expr n
```

Figure 2. Substitutions (implementations omitted).

```
data _⇝₀_ {n} : Expr n → Expr n → Set where
  reduceApplication : ∀ {s} b val →
    appl (lambda s b) val ⇝₀ b [val]
data _⇝_ {n} : Expr n → Expr n → Set where
  ⋯ (congruence closure of _⇝₀_)
_⇝*_ : {n : ℕ} → Expr n → Expr n → Set
_⇝*_ = ⋯ (transitive-reflexive closure of _⇝_)
_≈_ : {n : ℕ} → Expr n → Expr n → Set
x ≈ y = ∃ (λ n → x ⇝* n × y ⇝* n)
```

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

Substitutions We use a library of substitutions that is part of the Agda standard library [17], based on a technique by McBride [37]. Figure 2 shows a type of substitutions $Sub\ m\ n$ that will substitute terms with n free variables for all m free variables of other terms. More concretely, the function $\_/\_$ applies a substitution ϕ of type $Sub\ m\ n$ to a term t typed $Expr\ m$ to obtain term $t\ /\ \phi$, typed $Expr\ n$. Note that for example $\_/\_$ is Agda notation for a *mixfix* operator that is applied to two arguments t and ϕ in the form t/ϕ [18]. The function *weaken* uses the substitutions infrastructure to increase free de Bruijn indices by one and $\_\,[\_]$ substitutes term v for de Bruijn variable 0 in term t, to obtain term $t\ [v]$, shifting other free de Bruijn indices downward in the process.

Convertibility The next thing we define is an untyped notion of strong β-reduction and β-equivalence of terms in Figure 3. It is technically convenient to define primitive reductions in judgement $\_\rightsquigarrow_0\_$, a congruence closure of it in $\_\rightsquigarrow\_$ and a transitive-reflexive closure of that in $\_\rightsquigarrow_*\_$. The *reduceApplication* rule uses the substitution function $\_\,[\_]$ we saw before. In the type of *reduceApplication* we use Agda's $\forall$ shorthand notation, which desugars to a normal dependent type. For example $\forall\ \{n\} \to \cdots$ or $\forall\ n \to \cdots$ is short for $\{n : \_\} \to \cdots$ and $(n : \_) \to \cdots$ respectively, i.e. an implicit or normal argument n whose type is inferred by Agda. One $\forall$ symbol can apply to more than one argument. In $\_\approx\_$, we use the $\exists$ and $\times$ types: for a type A and predicate P typed $A \to Set$, $\exists P$ represents a dependent sum type containing tuples (v, pv) with v of type A and pv of type $P\ v$. For types A and B, $A \times B$ represents the cartesian product type of A and B (containing (a, b) with a of type A and b of type B). Two terms t and t' are defined to be convertible ($t \approx t'$) iff there exists a third term n that both t and t' reduce to.

Typing Contexts Figure 4 contains the definition of typing contexts and the more general notion of telescopes. A telescope is a sequence of expressions, each representing the type of a bound variable. The entries may refer to a number of free variables, assumed to be bound outside the telescope. The first index i of the *Telescope* type indicates how many such *initial* variables are as-

```
data Telescope (i : ℕ) : ℕ → Set where
  ε     : Telescope i i
  _◁_ : {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) (_ ◁ Γ) = weaken (lookup n Γ)
```

Figure 4. Telescopes and Contexts

```
data _⊢_:_ {n} (Γ : Context n) :
       Expr n → Expr n → Set where
  typeSet : Γ ⊢ set : set
  typeVar : ∀ {i} → Γ ⊢ var i : lookup i Γ
  typePi : ∀ {s t} → Γ ⊢ s : set →
    (s ◁ Γ) ⊢ t : set → Γ ⊢ pi s t : set
  typeLam : ∀ {s b t} → Γ ⊢ s : set →
    (s ◁ Γ) ⊢ b : t → Γ ⊢ lambda s b : pi s t
  typeAppl : ∀ {s f t val} → (s ◁ Γ) ⊢ t : set →
    Γ ⊢ f : pi s t → Γ ⊢ val : s →
    Γ ⊢ appl f val : appl (lambda s t) val
  typeConv : ∀ {e t t'} → t ≈ t' →
    Γ ⊢ e : t' → Γ ⊢ t : set → 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 $(n : \mathbb{N})\ (t : Expr\ n)$, 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 $\Gamma \vdash v : t$ 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_l, the typing rule that $set_l : set_{suc\ l}$ for all l, and a level-indexed typing judgement $\Gamma \vdash_l v : t$ with l such that $\Gamma \vdash_{suc\ l} t : set\ l$ 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 $(x : S) \to T$ is a type if S and T are types, with $x : S$ added to the context for T. For lambda expressions, *typeLam* says that $\lambda\ (x : S) \to b$ is typed $(x : S) \to T$ if b has type T in a context extended with $x : S$. According to *typeAppl*, a function application $f\ val$ has type $((\lambda\ (x : S) \to T)\ val)$ if f has type $(x : S) \to T$ 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-in dependent sum types (like the $\exists$ type we have already seen), identity types $x \equiv y : A$ (which contain proofs that x and y of type A are definitionally equal) and the empty type $\bot$ (which does not contain any value). These are modelled by adding suitable constructors for the types, their constructors and eliminators to the *Constant* data type, together with appropriate typing and reduction rules.

```
data ⊢_ : ∀ {n} → Context n → Set where
  ty_ε : ⊢ ε
  ty_◁ : ∀ {n e} {Γ : Context n} →
         ⊢ Γ → Γ ⊢ e : set → ⊢ (e ◁ Γ)
data _⊢_:_ {n} (Γ : Context n) :
  {m : ℕ} (ρ : Sub m n) (tel : Context m) → Set where
  ··· (omitted)
```

Figure 6. Well-typed Contexts

```
weaken−inj− ≈ : ∀ {n} {x y : Expr n} →
      weaken x ≈ weaken y → x ≈ y
≈−trans : ∀ {n} {x y z : Expr n} →
      x ≈ y → y ≈ z → x ≈ z
≈−/ : ∀ {n} {x y} {m} (ρ : Sub Expr n m) →
      x ≈ y → x / ρ ≈ y / ρ
weakenJudgementTop : ∀ {n} {Γ : Context n} {v t t'} →
      Γ ⊢ v : t' → t ◁ Γ ⊢ weaken v : weaken t'
substJudgementTop : ∀ {n} {Γ : Context n} {t' e t v} →
      t' ◁ Γ ⊢ e : t → Γ ⊢ v : t' → Γ ⊢ e [v] : t [v]
substContext : ∀ {n} {Γ : Context n} {e t} {t' t''} →
      t' ≈ t'' → Γ ⊢ t' : set → t' ◁ Γ ⊢ e : t →
      t'' ◁ Γ ⊢ e : t
⊢−/ : ∀ {m n} {e t} Γ1 Γ2 → (ϕ : Sub m n) →
      Γ2 ⊢ ϕ : Γ1 → Γ1 ⊢ e : t → Γ2 ⊢ e / ϕ : t / ϕ
⊢−var : ∀ {n} → {Γ : Context n} → ⊢ Γ →
      (i : Fin n) → Γ ⊢ lookup i Γ : set
typesAreSets : ∀ {n} {Γ : Context n} {e t} {l} →
      ⊢ Γ → Γ ⊢ e : t → Γ ⊢ t : set
substJudgementType : ∀ {n} {Γ : Context n} {e t t'} →
      t ≡ t' → Γ ⊢ e : t → Γ ⊢ e : t'
```

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

More typing judgements In addition to the typing judgement for terms above, we also define typing judgements for contexts and for substitutions. Figure 6 shows the judgement $\vdash\ \Gamma$ expressing that context Γ is well-typed, i.e. that all context entries are sets. Its rule ty_ϵ states that the empty context is always well-typed and $ty_\triangleleft$ says that subsequent entries should be types in their preceding context. We omit the definition of judgement $\Gamma \vdash \phi : tel$[3], which expresses that the terms substituted by substitution ϕ satisfy the type requirements of telescope *tel* in context Γ.

[3] For ease of presentation, we overload the notation $\_\vdash\_:\_$ in this text.

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. $weaken - inj- \approx$ shows that weakening is injective with respect to convertibility. $\approx\!\!-trans$ shows that convertibility is transitive. $\approx -trans$ 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 $\approx\!\!-/$ states that convertibility is invariant under substitutions. Theorems $weakenJudgementTop$, $substJudgementTop$, $substContext$ and $\vdash\!\!-/$ 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. $\vdash\!\!-var$ 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 $t \equiv t'$).

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

$$id \;:\; \forall\ (A : Set) \to A \to A$$
$$id = \lambda\ (A : Set) \to \lambda\ (v : A) \to v$$

The type and definition of this function are given by closed expressions $idTyTm$ and $idTm$.

$$idTm \;:\; Expr\ 0$$
$$idTm = lambda\ set\ (lambda\ (var\ 0)\ (var\ 0))$$
$$idTyTm \;:\; Expr\ 0$$
$$idTyTm = pi\ set\ (fun\ (var\ 0)\ (var\ 0))$$

We can prove that the term $idTm$ satisfies type $idTyTm$ using the typing rules from Figure 5.

$$ty_{idTm} \;:\; \varepsilon \vdash idTm : idTyTm$$
$$ty_{idTm} = typeLam\ typeSet\ (typeLam\ typeVar\ typeVar)$$

By the $typesAreSets$ theorem, it follows that $idTyTm$ is a type.

$$ty_{idTyTm} \;:\; \varepsilon \vdash idTyTm : set$$
$$ty_{idTyTm} = typesAreSets\ ty_\epsilon\ ty_{idTm}$$

3. Bringing Terms to Life

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

$$interpCtx : \{n : \mathbb{N}\}\ \{\Gamma : Context\ n\} \to \vdash\Gamma \;\to\; Set$$
$$interpSet : \{n : \mathbb{N}\}\ \{\Gamma : Context\ n\}\ \{A : Expr\ n\} \to$$
$$\Gamma \vdash A : set \;\to\; (ty_\Gamma : \vdash\Gamma) \;\to\; interpCtx\ ty_\Gamma \;\to Set$$
$$interp : \{n : \mathbb{N}\}\ \{\Gamma : Context\ n\}\ \{v\ t : Expr\ n\} \to$$
$$(ty_v : \Gamma \vdash v : t) \;\to\; (ty_\Gamma : \vdash\Gamma) \;\to$$
$$(asmpts : interpCtx\ ty_\Gamma) \;\to$$
$$interpSet\ (typesAreSets\ ty_\Gamma\ ty_v)\ ty_\Gamma\ asmpts$$

$interpCtx$ 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. $interpSet$ interprets an encoded type, yielding a Set, and $interp$ interprets a term v typed t. In the result type of $interp$ for a proof ty_v of judgement $\Gamma \vdash v : t$, we use the previously mentioned theorem $typesAreSets$ to calculate $typesAreSets\ ty_\Gamma\ ty_v$, a proof that $\Gamma \vdash t : set$. The result of $interp$ is then of type t, interpreted using $interpSet$ 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 $idTm$ and $idTyTm$ encode the function and its type and the proofs ty_{idTm} and ty_{idTyTm} witness the typing judgements $\varepsilon \vdash idTm : idTyTm$ and $\varepsilon \vdash idTyTm : set$. Both proofs assume only an empty context, which is always well typed according to the rule ty_ϵ in Figure 6. We will discuss the reduction behaviour of our primitives further, but $interpCtx\ ty_\epsilon$ (the assumptions in the empty context) reduces to unit type $\top$ (with canonical inhabitant tt). With all of this, we can interpret the encoded type $idTyTm$ to obtain the type $intrp_{idTyTm}$:

$$intrp_{idTyTm} = interpSet\ ty_{idTyTm}\ ty_\epsilon\ tt$$

More details follow, but $intrp_{idTyTm}$ reduces to $(x : Set)\ (x_1 : x) \to x$, alpha-equal to the intended type $(A : Set) \to A \to A$. Similarly, we can interpret term $idTm$ and its typing proof ty_{idTm} to obtain $intrp_{idTm}$ of type $intrp_{idTyTm}$.

$$intrp_{idTm} = interp\ ty_{idTm}\ ty_\epsilon\ tt$$

As we intended, $intrp_{idTm}$ reduces to $\lambda\ (x : Set) \to (x_1 : x) \to x_1$, alpha-convertible to our intended $\lambda\ (A : Set) \to \lambda\ (x : A) \to x$.

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 $suc\ 2$ from the pre-existing value 2 and function suc. To do this, the meta-program clearly needs to refer to the type $\mathbb{N}$, the function 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 Γ_{ex}:

$$\Gamma_{ex} = (pi\ (var\ 1)\ (var\ 2)) \triangleleft\ (var\ 0) \triangleleft\ set \triangleleft\ \varepsilon$$

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

$$ty_{\Gamma ex} = ty_\triangleleft\ (ty_\triangleleft\ (ty_\triangleleft\ ty_\epsilon\ typeSet)\ typeVar)$$
$$(typePi\ typeVar\ typeVar)$$

We will fill in the appropriate values for this context's assumptions with the value $asmpts_{\Gamma ex}$ of type $interpCtx\ ty_{\Gamma ex}$:

$$asmpts_{\Gamma ex} = ((tt, \mathbb{N}), 1), suc$$

In context Γ_{ex}, we can now construct the value $suc\ 2$ as a term ex. It is an $Expr\ 3$, since it may refer to Γ_{ex}'s three assumptions, and applies the postulated suc function to the postulated value 2.

$$ex = appl\ (var\ 0)\ (var\ 1)$$

We construct a proof ty_{ex} of judgement $\Gamma_{ex} \vdash ex : var\ 2$, i.e. that the constructed term ex has the first postulated value ($\mathbb{N}$) as its type, in three steps. First typing rules $typeAppl$ and $typeVar$ give us proof ty'_{ex}, showing that ex has a more complicated type. We then prove this type convertible to $var\ 2$ in (partly omitted) proof $conv_{ex}$. ty_{ex} then uses typing rule $typeConv$ to replace the convertible type.

$$
\begin{array}{l}
ty'_{ex}\ :\ \Gamma_{ex} \vdash ex : appl\ (lambda\ (var\ 2)\ (var\ 3))\ (var\ 1) \\
ty'_{ex} = typeAppl\ typeVar\ typeVar\ typeVar \\
conv_{ex}\ :\ appl\ (lambda\ (var\ 2)\ (var\ 3))\ (var\ 1) \approx var\ 2 \\
conv_{ex} = \cdots (reduceApplication\ (var\ 3)\ (var\ 1)) \\
ty_{ex}\ :\ \Gamma_{ex} \vdash ex : var\ 2 \\
ty_{ex} = typeConv\ conv_{ex}\ ty'_{ex}\ typeVar
\end{array}
$$

We can then interpret object program ex to obtain a value of type $interpSet\ (typesAreSets\ ty_{\Gamma ex}\ ty_{ex})\ ty_{\Gamma ex}\ asmpts_{\Gamma ex}$:

$$exInt = interp\ ty_{ex}\ ty_{\Gamma ex}\ asmpts_{\Gamma ex}$$

The reduction behaviour of our primitives that we will talk about next ensures that $exInt$'s type and $exInt$ itself reduce to $\mathbb{N}$ and $suc\ 2$ 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 ascertain 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 $interp$.

$$
\begin{array}{l}
interp : \{n : \mathbb{N}\}\ \{\Gamma : Context\ n\}\ \{v\ t : Expr\ n\} \rightarrow \\
\quad (ty_v : \Gamma \vdash v : t)\ \rightarrow\ (ty_\Gamma : \vdash \Gamma)\ \rightarrow \\
\quad (asmpts : interpCtx\ ty_\Gamma)\ \rightarrow \\
\quad interpSet\ (typesAreSets\ ty_\Gamma\ ty_v)\ ty_\Gamma\ asmpts
\end{array}
$$

The primitive takes a context Γ, a term v and a type t as hidden arguments, followed by proofs ty_v, ty_Γ of typing judgements $\Gamma \vdash v : t$ and $\vdash \Gamma$ and a value $asmpts$ of the context's interpretation type $interpCtx\ ty_\Gamma$. The reduction rules in Figure 8 specify that for certain forms of the judgement ty_v, the primitive application reduces to appropriate right-hand sides. For $ty_v = typeSet$, which implies[4] $v = set$ and $t = set$, the first rule returns interpretation Set. For $ty_v = typeVar$, an interpretation of the ith context assumption is given by primitive $interpVar$, discussed below.

The rules for $ty_v = typePi\ ty_s\ ty_t$ and $typeLam\ ty_s\ ty_b$ interpret terms $pi\ s\ t$ and $lambda\ s\ b$ as respectively the corresponding Agda Π-type and lambda term, recursively constructed from interpretations of s and t resp. b. The bound variable x is made available for the interpretation of t resp. b by placing it in the interpretation of the extended context $s \triangleleft \Gamma$. 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 a $typeConv$ 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 $interpCtx$:

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

The primitive takes a context Γ 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 $interp$'s reduction rules for $typeLam$ and $typePi$, how an extended context $s \triangleleft \Gamma$ is interpreted by a tuple of the s value and the interpretation of Γ. This corresponds to $interpCtx$'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 Γ extended with a type t is interpreted by an interpretation $asmpts$ of Γ, and an interpretation of the type t. We use a dependent sum $\exists$ to specify the interpretation of t with respect to the interpretation $asmpts$ of the rest of the context.

Now that we know how to interpret a context, we can define reduction rules for $interpVar$, to project out a context's ith entry. Its reduction rules are not surprising, projecting out the top assumption for variable $zero$ and recursing for $suc\ i$. The primitive $interpSet$ is a version of $interp$ 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 $interpSet'$ and $interpVarSet$. We do not discuss their reduction behaviour as it is similar to $interp$ and $interpVar$ except that we require proof that the judgement's type is convertible to 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 reductions: 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:

$$absurdTerm = \lambda\ (prf : Int \equiv Bool) \rightarrow cast\ prf\ 3 \vee false$$

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

For our primitives, similar issues arise. We can for example assume a proof $tyAbsurd$ of judgement $\varepsilon \vdash set : pi\ set\ set$ even

[4] Note: pattern matches that imply equalities about other arguments are standard for dependent pattern matching [25].

$$
\begin{array}{llllll}
interp & typeSet & ty_\Gamma & asmpts & = & Set \\
interp & (typeVar\ \{i = i\}) & ty_\Gamma & asmpts & = & interpVar\ i\ ty_\Gamma\ asmpts \\
interp & (typePi\ ty_s\ ty_t) & ty_\Gamma & asmpts & = & (x : interpSet\ ty_s\ ty_\Gamma\ asmpts) \to interpSet\ ty_t\ (ty_\triangleleft\ ty_\Gamma\ ty_s)\ (asmpts, x) \\
interp & (typeLam\ ty_s\ ty_b) & ty_\Gamma & asmpts & = & \lambda\,(x : interpSet\ ty_s\ ty_\Gamma\ asmpts) \to interp\ (ty_\triangleleft\ ty_\Gamma\ ty_s)\ ty_b\ (asmpts, x) \\
interp & (typeAppl\ \mathtt{ty_t}\ \mathtt{ty_f}\ \mathtt{ty_{val}}) & ty_\Gamma & asmpts & = & interp\ ty_\Gamma\ ty_f\ asmpts\ (interp\ ty_\Gamma\ ty_{val}\ asmpts) \\
interp & (typeConv\ \mathtt{t{\sim}t'}\ \mathtt{ty_e}\ \mathtt{ty_t}) & ty_\Gamma & asmpts & = & interp\ ty_\Gamma\ ty_e\ asmpts
\end{array}
$$

$$
\begin{array}{lll}
interpCtx & ty_\epsilon & = \top \\
interpCtx & (ty_\triangleleft\ ty_\Gamma\ ty_t) & = \exists\,\lambda\,(asmpts : interpCtx\ ty_\Gamma) \to interpSet\ ty_t\ ty_\Gamma\ asmpts
\end{array}
$$

$$
\begin{array}{lllll}
interpVar : \forall\ \{n\}\ \{\Gamma : Context\ n\}\ i \to (ty_\Gamma : \vdash \Gamma) \to (asmpts : interpCtx\ ty_\Gamma) \to interpSet\ (\vdash\!\!-var\ ty_\Gamma\ i)\ ty_\Gamma\ asmpts \\
interpVar & zero & (ty_\triangleleft\ ty_\Gamma\ \mathtt{ty_t}) & (\_, asmpt) & = asmpt \\
interpVar & (suc\ i) & (ty_\triangleleft\ ty_\Gamma\ ty_t) & (asmpts, \_) & = interpVar\ ty_\Gamma\ i\ asmpts
\end{array}
$$

$$
interpSet\ ty_t\ ty_\Gamma\ asmpts = interpSet'\ ty_t \approx\!\!- refl\ ty_\Gamma\ asmpts
$$

$$
\begin{array}{llllll}
interpSet' : \forall\ \{n\}\ \{\Gamma : Context\ n\}\ \{A\ t\} \to \Gamma \vdash A : t \to t \approx set \to (ty_\Gamma : \vdash \Gamma) \to interpCtx\ ty_\Gamma \to Set \\
interpSet' & typeSet & eq & ty_\Gamma & asmpts & = Set \\
interpSet' & (typeVar\ \{i = i\}) & eq & ty_\Gamma & asmpts & = interpVarSet\ i\ eq\ ty_\Gamma\ asmpts \\
interpSet' & (typePi\ ty_s\ ty_t) & eq & ty_\Gamma & asmpts & = \\
\multicolumn{6}{l}{\quad (x : interpSet\ ty_s\ ty_\Gamma\ asmpts) \to interpSet\ ty_t\ (ty_\triangleleft\ ty_\Gamma\ ty_s)\ (asmpts, x)} \\
interpSet' & (typeAppl\ \mathtt{ty_t}\ \mathtt{ty_f}\ ty_{val}) & \mathtt{eq} & ty_\Gamma & asmpts & = interp\ ty_f\ ty_\Gamma\ asmpts\ (interp\ ty_{val}\ ty_\Gamma\ asmpts) \\
interpSet' & (typeConv\ t{\sim}t'\ ty'_A\ ty_t) & eq & ty_\Gamma & asmpts & = interpSet'\ ty'_A\ (\approx\!\!-trans\ (\approx\!\!-sym\ t{\sim}t')\ eq)\ ty_\Gamma\ asmpt
\end{array}
$$

$$
\begin{array}{llllll}
interpVarSet : \forall\ \{n\}\ \{\Gamma : Context\ n\}\ \{l\}\ i \to lookup\ i\ \Gamma \approx set \to (ty_\Gamma : \vdash \Gamma) \to interpCtx\ ty_\Gamma \to Set \\
interpVarSet & zero & \mathtt{eq} & (ty_\triangleleft\ ty_\Gamma\ \mathtt{ty_t}) & (\_, asmpt) & = asmpt \\
interpVarSet & (suc\ i) & eq & (ty_\triangleleft\ ty_\Gamma\ ty_t) & (asmpts, \_) & = interpVarSet\ i\ (weaken{-}inj{-}\approx\ (lookup\ i\ \Gamma)\ set\ eq)\ ty_\Gamma\ asmpts
\end{array}
$$

Figure 8. Reduction behaviour of our primitives. Patterns in `typewriter` font are required to be values.

though this type of proofs is empty. Clearly, $interp\ tySet\ ty_\epsilon\ tt$ should then not reduce to Set at type $Set \to Set$, but instead block on the open variable $tySet$. Similarly, if we assume a proof prf of judgement $pi\ set\ set \approx set$, and use it with $typeConv$ to construct a proof $tyAbsurd'$ of judgement $\varepsilon \vdash set : pi\ set\ set$, then our primitives should block on open variable prf.

By the value patterns in Figure 8, some rules require that certain arguments are values. We have checked for each rule that the right-hand side's type was equal to the declared type, assuming just the information from the left-hand side patterns, similar to how dependent pattern matching can be type-checked [25]. For the five rules with value patterns, this was not the case. In for example the rule for $interp\ (typeConv\ \mathtt{t{\sim}t'}\ \mathtt{ty_e}\ \mathtt{ty_t})\ ty_\Gamma\ asmpts$, the right-hand-side is of type

$$interpSet\ (typesAreSets\ ty_\Gamma\ ty_e)\ ty_\Gamma\ asmpts$$

i.e. the interpretation of t', not t and the convertibility assumption $t \sim t'$ is essential for returning a value of type t' as one of type t. We believe that the value patterns in Figure 8 solve this problem, making our reduction rules valid for open terms, even though the general question of soundness remains open.

The primitives' properties In addition to the reduction behaviour of our primitives, some of our meta-programs require additional properties about them listed in Figure 9. Property $castInterp{-}\approx'$ states that for convertible types A and A', the interpretations under $interpSet$ must be the same. The next two properties are related to the interpretation of a type after a well-typed substitution $\Gamma_2 \vdash \phi : \Gamma_1$ between well-typed contexts Γ_1 and Γ_2. $interpCompSubCtx$ says that an interpretation of Γ_1 can be constructed from one of Γ_1 and $interpCompSubSet'$ says that the interpretation of a type t in Γ_2 is the same as that of $t\ /\ \phi$ in Γ_1 using the interpretation of Γ_1 constructed by $interpCompSubCtx$.

We are currently using stub proofs of these properties, based on an Agda primitive called $primTrustMe$. $primTrustMe$ 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, $primTrustMe$ only reduces to $refl$ 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 $\vdash -/$.

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** *Show* mechanism [35, §4.3.3, §11], which allows a data type A to be annotated with the directive **deriving** *Show* to make the compiler derive an instance of the *Show* type class. Such an instance consists essentially of a function $show :: A \to String$, 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 *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:

$$
\begin{array}{l}
\textbf{record}\ SimpleDT\ (A : Set) : Set\ \textbf{where} \\
\quad \textbf{constructor}\ simpleDT \\
\quad \textbf{field}\ constructors\ :\ List\ (Constructor\ A) \\
\qquad\qquad folder\ :\ folderType\ A\ constructors
\end{array}
$$

$$
\begin{array}{l}
castInterp{-}{\approx}' : \forall\ \{n\}\ \{\Gamma : Context\ n\}\ \{A\ A'\} \to (ty_A : \Gamma \vdash A : set)\ \to\ (ty'_A : \Gamma \vdash A' : set)\ \to\ A \approx A'\ \to \\
\quad (ty_\Gamma : \vdash \Gamma)\ \to\ (asmpts : interpCtx\ ty_\Gamma)\ \to\ interpSet\ ty_A\ ty_\Gamma\ asmpts \equiv interpSet\ ty'_A\ ty_\Gamma\ asmpts \\
interpCompSubCtx : \forall\ \{m\ n\}\ \{\Gamma_1\ \Gamma_2\}\ \{\phi : Sub\ Expr\ m\ n\} \to \\
\quad \Gamma_2 \vdash \phi : \Gamma_1\ \to\ (ty\Gamma_1 : \vdash \Gamma_1)\ \to\ (ty\Gamma_2 : \vdash \Gamma_2)\ \to interpCtx\ ty\Gamma_2\ \to\ interpCtx\ ty\Gamma_1 \\
interpCompSubSet' : \forall\ \{m\ n\ t\}\ \{\Gamma_1\ \Gamma_2\}\ \{\phi : Sub\ Expr\ m\ n\} \to \\
\quad (comp : \Gamma_2 \vdash \phi : \Gamma_1)\ \to\ (ty\Gamma_1 : \vdash \Gamma_1)\ \to\ (ty_t : \Gamma_1 \vdash t : set)\ \to\ (ty\Gamma_2 : \vdash \Gamma_2)\ \to\ (asmpts_2 : interpCtx\ ty\Gamma_2)\ \to \\
\quad interpSet\ (\vdash -/comp\ ty_t)\ ty\Gamma_2\ asmpts_2 \equiv interpSet\ ty_t\ ty\Gamma_1\ (interpCompSubCtx\ comp\ ty\Gamma_1\ ty\Gamma_2\ asmpts_2)
\end{array}
$$

Figure 9. Primitive properties

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:

$$
\begin{array}{l}
\mathbf{data}\ Constructor\ (A : Set) : Set\ \mathbf{where} \\
\quad mkConstructor : String \to (n : \mathbb{N}) \to \\
\qquad (tel : Telescope\ 1\ (n + 1))\ \to\ (ty_{tel} : \Gamma set \vdash tel) \to \\
\qquad \mathbf{let}\ ctorT = funCtx\ n\ tel\ (var\ 0) \\
\qquad\quad ty_{ctorT}\ :\ \Gamma set \vdash ctorT : set \\
\qquad\quad ty_{ctorT} = typeFunCtx\ n\ ty_{tel}\ typeVar \\
\qquad \mathbf{in}\ interpSet\ ty_{\Gamma_{set}}\ ty_{ctorT}\ (tt, A)\ \to\ Constructor\ A
\end{array}
$$

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 $\Gamma set = set \lhd \varepsilon$, 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 $Constructor$s, $SimpleDT$ 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.

$$
\begin{array}{l}
folderType : (A : Set) \to List\ (Constructor\ A) \to Set \\
folderType\ A\ constructors = (P : A \to Set) \to \\
\quad underFolderAsmpts\ A\ P\ constructors\ ((x : A) \to P\ x)
\end{array}
$$

Given a set A and a list of A's constructors, $folderType$ returns the type for a corresponding induction principle: it takes a predicate $P : A \to Set$ (the *motive* [25], describing what the induction principle should produce) and returns a function of type $(x : A) \to P\ x$ under a number of assumptions. For every constructor, the function $underFolderAsmpts$ 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:

$$
\begin{array}{l}
\mathbf{data}\ \bot : Set\ \mathbf{where} \\
\\
\quad foldBot : (P : \bot \to Set) \to (t : \bot) \to P\ t \\
\quad foldBot\ P\ ()
\end{array}
$$

Note the use of an absurd pattern $()$ 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 $\bot$:

$$
\begin{array}{l}
botDT\ :\ SimpleDT\ \bot \\
botDT = simpleDT\ []\ foldBot
\end{array}
$$

$botDT$ 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:

$$
\begin{array}{l}
\mathbf{data}\ \mathbb{N} : Set\ \mathbf{where}\ zero\ \ : \mathbb{N} \\
\qquad\qquad\qquad\qquad\quad suc\ \ : \mathbb{N} \to \mathbb{N} \\
fold\mathbb{N} : (P : \mathbb{N} \to Set) \to\ P\ zero\ \to \\
\quad (\forall\ n \to P\ n \to P\ (suc\ n))\ \to\ (n : \mathbb{N}) \to P\ n \\
fold\mathbb{N}\ P\ Pz\ Ps\ zero\ \ \ \ = Pz \\
fold\mathbb{N}\ P\ Pz\ Ps\ (suc\ n) = Ps\ n\ (fold\mathbb{N}\ P\ Pz\ Ps\ n)
\end{array}
$$

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

$$
\begin{array}{l}
zeroConstr = mkConstructor\ \texttt{"zero"}\ 0\ \varepsilon\ ty_\epsilon\ zero \\
sucConstr\ \ = mkConstructor\ \texttt{"suc"}\ 1\ (var\ 0 \lhd \varepsilon) \\
\qquad\qquad (ty_\lhd\ ty_\epsilon\ typeVar)\ suc
\end{array}
$$

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 $natDT : SimpleDT\ \mathbb{N}$.

$$
natDT = simpleDT\ [zeroConstr, sucConstr]\ fold\mathbb{N}
$$

$natDT$ lists $\mathbb{N}$'s constructors and provides induction principle $fold\mathbb{N}$, 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.

$$
\begin{array}{l}
deriveShow : \forall\ \{A\} \to SimpleDT\ A \to A \to String \\
deriveShow\ (simpleDT\ constructors\ folder) = \text{omitted}
\end{array}
$$

We omit the algorithm's implementation, which takes the description of data type A and exploits the induction principle with motive $P = \lambda\ \_ \to String$. 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 construct for the concrete motive $\lambda\ \_ \to String$ correspond to their expected types for a general predicate P when P is instantiated to $\lambda\ \_ \to String$ through the context interpretation. This essentially uses the $interpCompSubCtx$ and $interpCompSubSet$ primitive properties shown in Figure 9.

For our example data types, $deriveShow$ derives an (admittedly not very useful) $show$ function for $\bot$:

$$showBot\ :\ \bot \to String$$
$$showBot = deriveShow\ botDT$$

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

$$showNat = deriveShow\ natDT$$

Like for $showBot$, $showNat$'s definition reduces to the generated function $showNat' = fold\mathbb{N}\ (\lambda\ \_ \to String)$ `"zero"` (...) (final argument omitted). We can apply it to numbers with for example $showNat\ 2$ producing the string `"(suc (suc zero))"`.

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 $SimpleDT$ 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.

$SimpleDT$ and $deriveShow$ 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.

4.2 Tactics

Tactics are a form of meta-programs that solve or refine proof obligations 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 performance 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 $interpSet$ to specify the expected result type of the tactic.

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

$$\begin{aligned}
&assumptionTactic : \forall\ \{n\ T\}\ \{\Gamma : Context\ n\} \to \\
&\quad (ty_t : \Gamma \vdash T : set)\ \to \\
&\quad (ty_\Gamma : \vdash \Gamma)\ \to\ (asmpts : interpCtx\ ty_\Gamma) \to \\
&\quad ifYes\ (inContext?\ \Gamma\ T)\ (interpSet\ ty_t\ ty_\Gamma\ asmpts)
\end{aligned}$$

The tactic takes a type T, a well-typed context Γ 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 $Dec\ P$ type. It models a decision of proposition P, i.e. either a proof of P or a proof of $\neg P$:

$$\begin{aligned}
\mathbf{data}\ Dec\ (P : Set) : Set\ \mathbf{where}\ &yes : P\ \ \ \ \ \ \to Dec\ P \\
&no\ :\ \neg\ P\ \ \to Dec\ P
\end{aligned}$$

Based on a decision of some property, the $ifYes$ function returns either an argument type or unit type $\top$:

$$\begin{aligned}
&ifYes : \{P : Set\} \to Dec\ P \to Set \to Set \\
&ifYes\ (yes\ _)\ P' = P' \\
&ifYes\ (no\ _)\ _\ \ = \top
\end{aligned}$$

The $inContext?$ algorithm decides whether or not a certain type t is present in context Γ, i.e. if the ith entry in the context is equal to t for some i. It uses a general purpose decision procedure $any?$, which simply tries all i of the bounded type $Fin\ n$. For a given variable i, we use a general equality decision procedure for terms $\_\stackrel{?}{=}\_$ to check whether the ith context entry is equal to t.

$$\begin{aligned}
&InContext : \{n : \mathbb{N}\}\ (\Gamma : Context\ n)\ (t : Expr\ n) \to Set \\
&InContext\ \Gamma\ t = \exists\lambda\ i \to lookup\ i\ \Gamma \equiv t \\
&inContext? : \{n : \mathbb{N}\}\ (\Gamma : Context\ n)\ (t : Expr\ n) \to \\
&\qquad Dec\ (InContext\ \Gamma\ t) \\
&inContext?\ \Gamma\ t = any?\ (\lambda\ i \to lookup\ i\ \Gamma \stackrel{?}{=} t)
\end{aligned}$$

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

$$\begin{aligned}
&assumptionTactic\ ty_\Gamma\ ty_t\ asmpts\ \mathbf{with}\ inContext?\ \Gamma\ t \\
&assumptionTactic\ \{n\}\ \{t\}\ \{\Gamma\}\ ty_\Gamma\ ty_t\ asmpts \\
&\quad \mid yes\ (i, eq_{\Gamma id}) = \\
&\quad \mathbf{let}\ ty_{vari} : \Gamma \vdash var\ i : t \\
&\qquad\ \ ty_{vari} = substJudgementType\ eq_{\Gamma id}\ typeVar \\
&\quad \mathbf{in}\ \ castInterp\ (typesAreSets\ ty_\Gamma\ ty_{vari})\ ty_t \\
&\qquad\qquad ty_\Gamma\ asmpts\ (interp\ ty_{vari}\ ty_\Gamma\ asmpts) \\
&assumptionTactic\ ty_\Gamma\ ty_t\ asmpts \mid no\ _ = tt
\end{aligned}$$

The first step is to use the proof $eq_{\Gamma id}$ that $lookup\ i\ \Gamma \equiv t$ from $inContext?$ and the $typeVar$ typing rule to produce a proof ty_{vari} of judgement $\Gamma \vdash var\ i : t$. We can then obtain the interpretation of the ith variable through the value $interp\ ty_{vari}\ ty_\Gamma\ asmpts$. Unfortunately, that value's type is

$$interpSet\ (typesAreSets\ ty_\Gamma\ ty_{vari})\ ty_\Gamma\ asmpts$$

What we need is a value of type $interpSet\ ty_t\ ty_\Gamma\ asmpts$, i.e. an interpretation of the same type t, but for a different proof that t is a set. $castInterp$, an omitted special case of property $castInterp\text{-}\approx$ from Figure 9, is precisely what we need to cast one to the other.

Tactic usage Currently, our tactics can be manually invoked with a context and goal type and well-formedness proofs. The tactic invocation appears as an expression in the code where the goal is needed. In future systems, compiler support can increase convenience 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 combinators 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 Danielsson'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 Danielsson, 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 $showNat' = fold\mathbb{N}\ (\lambda\ \_ \to String)$ `"zero"` $(...)$. 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 $fold\mathbb{N}$. 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 $A \to String$, with the value $\lambda\ \_ \to String$ provided through the interpretation of this context and the same type with an encoding of $\lambda\ \_ \to String$ 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 could 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 $\Gamma \vdash set : set$ 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

Theorem 1 (Gödel's Second Incompleteness Theorem)**.** *For any formal effectively generated theory T including basic arithmetical truths and also certain truths about formal provability, if T includes a statement of its own consistency then T 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 $\forall\ \{t\} \to \varepsilon \vdash t : constant\ bot\ [\,] \to \bot$, 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 T's consistency, to obtain a new theory 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.

Staging As discussed in the introduction, our meta-programming primitives do not use the concept of staging like some other solutions [6, 11, 48, 52]. Nevertheless, our $interp$ primitive performs the same function as an unquote primitive in such systems, allowing object programs to invoke a meta-program and use generated code as if it were normal code. The quote primitive in a staging meta-programming system allows to include references to object-level terms in generated code, something which we support in a different way, as discussed in Section 3. Finally, while in these systems, code at all staging levels runs at either compile-time or run-time, but not both. In our system, the question of when to execute meta-programs is an orthogonal matter, not different from the *partial evaluation* of normal functions. Conveniently, partial evaluation is relatively cheap in total dependently-typed languages and for example well supported in the language Idris [7].

We see the orthogonality of our meta-programming primitives w.r.t. staging considerations as an advantage. If desired, it is technically possible to require at compile-time that all invocations of the primitives be unfoldable (producing errors if arguments are not statically known). However, like for partial evaluation, executing a meta-program upfront is not *always* a good idea, especially if we are already sure that the generated code will be well-typed (see e.g. Chlipala's arguments about the performance advantages of reflective meta-programs [14]). It seems that annotations for partial evaluation like in Idris would combine well with our primitives to conveniently let the programmer control when meta-programs are executed. For example, a version of $deriveShow$ with the $SimpleDT\ A$ argument annotated as $[static]$ would generate $show$ functions at compile time instead of run-time.

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 meta-programs, 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. even exploit 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 terms, types 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 Thiemann 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 `letstatic` 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

[1] A. Abel. Irrelevance in type theory with a heterogeneous equality judgement. In *Foundations of Software Science and Computational Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2011.

[2] D. Abrahams and A. Gurtovoy. *C++ template metaprogramming: Concepts, tools, and techniques from Boost and beyond*. Addison-Wesley, 2004.

[3] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl*, pages 1–20. Kluwer, 2003.

[4] B. Barras and G. Huet. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université de Paris 07, 1999.

[5] S. Boutin. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997.

[6] E. Brady and K. Hammond. A verified staged interpreter is a verified compiler. In *Generative Programming and Component Engineering*, pages 111–120. ACM, 2006.

[7] E. C. Brady and K. Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *International Conference on Functional Programming*, pages 297–308. ACM, 2010.

[8] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 2002.

[9] J. Chapman. Type theory should eat itself. In *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice*, volume 228 of *Electronic Notes in Theoretical Computer Science*, pages 21–36. Elsevier, 2009.

[10] J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *International Conference on Functional Programming*, pages 3–14. ACM, 2010.

[11] C. Chen and H. Xi. Meta-programming through typeful code representation. In *International Conference on Functional Programming*, pages 275–286. ACM, 2003.

[12] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*, pages 143–156. ACM, 2008.

[13] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *Programming Languages Design and Implementation*, pages 122–133. ACM, 2010.

[14] A. Chlipala. Certified programming with dependent types. online, 2012. URL `http://adam.chlipala.net/cpdt/`.

[15] N. A. Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2007.

[16] N. A. Danielsson. Operational semantics using the partiality monad. In *International Conference on Functional Programming*, pages 127–138. ACM, 2012.

[17] N. A. Danielsson and many others. The Agda standard library, 2009.

[18] N. A. Danielsson and U. Norell. Parsing mixfix operators. In *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 80–99. Springer, 2008.

[19] J. Darcy. JSR 269: Pluggable annotation processing API, 2011. URL `http://jcp.org/en/jsr/detail?id=269`.

[20] D. Delahaye. A tactic language for the system Coq. In *Logic Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 85–95. Springer, 2000.

[21] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4): 440–465, 1994.

[22] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.

[23] S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: indexed types now! In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 112–121. ACM, 2007.

[24] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik*, 38(1): 173–198, 1931.

[25] H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

[26] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *International Conference on Functional Programming*, pages 163–175. ACM, 2011.

[27] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. technical appendix, 2011. URL `http://www.mpi-sws.org/~beta/lessadhoc/appendix.pdf`.

[28] A. J. C. Hurkens. A simplification of Girard's paradox. In *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 266–278. Springer, 1995.

[29] S. C. Johnson. YACC. *UNIX Programmer's Manual*, 2b, 1979.

[30] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *International Conference on Functional Programming*, pages 192–203. ACM, 2005.

[31] R. Lämmel and S. Peyton-Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types in Languages Design and Implementation*, pages 26–37. ACM, 2003.

[32] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, page 35. ACM, 1994.

[33] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for Haskell. In *Haskell Symposium*, pages 37–48. ACM, 2010.

[34] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Haskell Symposium*, pages 67–78. ACM, 2010.

[35] S. Marlow. Haskell 2010 language report. online, 2010. URL `http://www.haskell.org/onlinereport/haskell2010/`.

[36] P. Martin-Löf. An intuitionistic theory of types. draft, 1972. URL `http://cs.ioc.ee/~james/ITT9200/martinlof72.ps`.

[37] C. McBride. Type-preserving renaming and substitution. draft, 2005. URL `http://strictlypositive.org/ren-sub.pdf`.

[38] C. McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *Workshop on Generic Programming*, pages 1–12. ACM, 2010.

[39] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, 2007.

[40] Oracle. Java core reflection. online, 1996. URL `http://docs.oracle.com/javase/1.5.0/docs/guide/reflection/spec/java-reflectionTOC.doc.html`.

[41] T. Parr and R. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[42] E. Pašalić and N. Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 136–167. Springer, 2004.

[43] B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *Principles and Practice of Declarative Programming*, pages 163–173. ACM, 2008.

[44] A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2008.

[45] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *European Conference on Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2011.

[46] A. Rodriguez, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *International Conference on Functional Programming*, 2009.

[47] J. Rudolph and P. Thiemann. Mnemonics: type-safe bytecode generation at run time. *Higher-Order and Symbolic Computation*, 23(3): 371–407, 2010.

[48] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Workshop on Haskell*, pages 1–16. ACM, 2002.

[49] T. Sheard and E. Pasalic. Meta-programming with built-in type equality. In *International Workshop on Logical Frameworks and Meta-Languages*, 2004.

[50] A. Stampoulis and Z. Shao. VeriML: typed computation of logical terms inside a language with effects. In *International Conference on Functional Programming*, pages 333–344. ACM, 2010.

[51] A. Stampoulis and Z. Shao. Static and user-extensible proof checking. In *Principles of Programming Languages*, pages 273–284. ACM, 2012.

[52] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[53] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Programming Languages Design and Implementation*, pages 132–141. ACM, 2011.

[54] Wikipedia. Gödel's incompleteness theorems, November 2012. URL `http://en.wikipedia.org/wiki/Goedels_incompleteness_theorems`.