Practical Reflection and Metaprogramming for Dependent Types

David Raymond Christiansen

Advisor: Peter Sestoft
Submitted: November 2, 2015

IT UNIVERSITY OF COPENHAGEN

Abstract

Embedded domain-specific languages are special-purpose programming languages that are implemented within existing general-purpose programming languages. Dependent type systems allow strong invariants to be encoded in representations of domain-specific languages, but it can also make it difficult to program in these embedded languages. Interpreters and compilers must always take these invariants into account at each stage, and authors of embedded languages must work hard to relieve users of the burden of proving these properties.

Idris is a dependently typed functional programming language whose semantics are given by elaboration to a core dependent type theory through a tactic language. This dissertation introduces *elaborator reflection*, in which the core operators of the elaborator are realized as a type of computations that are executed during the elaboration process of Idris itself, along with a rich API for reflection. Elaborator reflection allows domain-specific languages to be implemented using the same elaboration technology as Idris itself, and it gives them additional means of interacting with native Idris code. It also allows Idris to be used as its own metalanguage, making it into a programmable programming language and allowing code re-use across all three stages: elaboration, type checking, and execution.

Beyond elaborator reflection, other forms of compile-time reflection have proven useful for embedded languages. This dissertation also describes *error reflection*, in which Idris code can rewrite DSL error messages before presenting domain-specific messages to users, as well as a means for integrating quasiquotation into a tactic-based elaborator so that high-level syntax can be used for low-level reflected terms.

Resumé

Indlejrede domænespecifikke sprog er skræddersyede programmeringssprog, som er implementeret inde i eksisterende alment anvendelige programmeringssprog. Afhængige typer tillader, at stærke invarianter indkodes i datarepræsentationer, herunder repræsentationer af programmeringssprog. Dog kan denne kontrol også besværliggøre programmeringsprocessen, da brugeren altid skal demonstrere, at invarianterne er overholdt. Fortolkere og oversættere skal tage disse invarianter i betragtning i hvert trin. Når indlejrede sprog bliver designet, skal der arbejdes hårdt for at brugerne ikke bebyrdes med tungt bevisarbejde.

Idris er et funktionsprogrammeringssprog med afhængige typer, hvis semantik er givet ved elaborering til en afhængigt-typet kerneteori ved brug af et taktiksprog. Denne afhandling introducerer *elaboratorrefleksion*, hvor elaboratorens grundlæggende operationer er realiseret som en type, hvis bereginger bliver afviklet under Idris's egen elaborering. Dertil findes et rigt API for refleksion om termer, datatyper, og andre definitioner. De reflekterede elaboreringsoperationer muliggør, at domænespecifikke sprog kan implementeres med Idris's elaboreringsteknologi, og således får de elaborerede sprog flere muligheder for interaktion med Idris. Desuden bliver Idris anvendelig som sit eget metasprog, hvilket gør Idris til et programmerbart programmeringssprog og muliggør genbrug af kode i alle tre stadier: elaborering, typetjek og afvikling.

Udover elaboreringsrefleksion har andre former for statisk refleksion vist sig at være nyttige for indlejring af sprog. Denne afhandling præsenterer desuden *refleksion af fejlmeddelelser*, hvor kode skrevet i Idris kan omskrive fejlmeddelelser med oprindelse i domænespecifikke sprog til domænespecifikke fejlmeddelelser. Derudover præsenteres en teknik for integrering af kvasicitering i en taktikbaseret elaborator, så højtniveausyntaks kan anvendes for termer i kernesproget.

Acknowledgments

This dissertation represents the culmination of a long process of learning and hard work that was made possible largely by the generosity and caring of other people.

I would like to thank my supervisor, Peter Sestoft, for being a wonderful example of how to communicate clearly and how to let theory and practice inform one another. Even when my work diverged somewhat from our original plans, he has been supportive and helpful, always ready to discuss what I've been working on.

Our partners at Edlund A/S, especially Henning Niss and Klaus Grue, got me off to a good start and kept me on the right track. Their feedback and good example was immensely valuable.

I would also like to thank Edwin Brady for being so helpful as I've extended and modified his language, and the `#idris` channel on Freenode for being such a great community.

I would especially like to thank my fellow Ph.D. students Hannes Mehnert and Ahmad Salim Al-Sibahi. Hannes's hacker spirit was always an inspiration, and we had many good discussions about both programming languages and wider themes. Ahmad's encyclopedic knowledge helped me find many papers that I didn't even know existed, and his feedback on an earlier draft of this dissertation made it much better.

If not for the Danish welfare society, I may have never had the chance to go to graduate school. The Danish public sector paid my tuition and stipend while I was working towards my master's degree, and my work was funded as part of the Actulus project, Danish National Advanced Technology Foundation (*Højteknologifonden*) grant 017-2010-3. *Højteknologifonden* has since become part of *Innovationsfonden*. Tusind tak!

And thanks, Lisbet, for support both moral and material, and for your patience during the last few months.

Contents

Chapter 1

Introduction

Static type systems in programming languages are becoming more and more expressive over time. As type systems get more and more expressive, they offer greater abilities to express complex invariants, ruling out more classes of bugs. Additionally, an expressive static type language can enable better programming tools. If the system knows more about what the programmer is trying to build, then it is in a better position to help her get there!

As type systems become more and more expressive, the job of the programmer when writing a type begins to resemble the job of the programmer when writing a program. In other words, types begin to need to do *computation*. Unfortunately for programmers, most type systems were not designed as programming languages, and systems in which the ability to write programs is accidental tend to not be very pleasant to write programs in.

Enter dependent types. Dependently typed languages use the *same* language for type-level and program-level computation. Because this language is designed explicitly for programming, type-level programming is no longer a difficult subject reserved for the most advanced users. Additionally, it becomes possible to re-use code in both stages: the same list datatype that is used to contain run-time values from a database can be used to represent a statically-checked typing context for an embedded domain-specific language.

Martin-Löf's type theory and its descendants have been recognized as programming languages since the 1980s [Mar84]. However, early implementations of type theory, including Coq [Coq04], Nuprl [Con+86], LEGO [LP92], Alf [MN94], and the early versions of Agda, were typi-

cally designed as proof assistants, rather than as programming languages. While it was possible to use *extraction* to recover computational content from proofs, it was the proofs that were the star of the show: rather than programming with dependent types, dependent types were used first and foremost as a logic. This changed with Augustsson's Cayenne [Aug98], and then later Epigram [MM04], Agda 2 [Nor07], and their descendant Idris [Bra13b]. These languages represent an alternative tradition, in which type theory is intended to be used directly as a programming language. This dissertation describes a series of techniques that are implemented as part of Idris, and basic reading knowledge of Idris is necessary to understand the code samples. Appendix B summarizes some of Idris's features that differ from Haskell or Agda.

In a dependently typed language, datatypes can be represented as codes in universes [AM03], allowing generic programming internally in the system. It is even possible to remove datatype definitions as a primitive, representing new datatypes as codes in a self-representing universe of definitions [Cha+10b], which allows generic programming over all datatypes — a technique dubbed *levitation*. One might therefore think that dependent types are powerful enough to subsume all other compile-time programming.

However, systems for representing type theory inside of type theory suffer from a number of shortcomings. Generic programming with universes relies on all datatypes being represented using the universe encoding, and current implementations do not automatically encode datatypes as codes in a universe. It is also unclear to what extent these very expressive universe encodings can be used for practical programming, which requires things like compilers that generate efficient code and error reports that are given in terms of the code that the user wrote rather than code that was generated behind the scenes. For example, Al-Sibahi implemented levitation for Idris in his M.Sc. thesis [AlS14], and encountered very poor performance. Furthermore, given the variety of universe encodings that exist, privileging one of them in the implementation will probably not satisfy all the desired use cases.

Domain-specific languages, which are discussed as "little languages" in a classic paper by Bentley [Ben86], are small programming languages designed to solve one particular problem very well. These domain-specific languages need not be suitable for general-purpose computing. In 1996, Paul Hudak [Hud96] coined the term "domain-specific embedded language" to refer to libraries or APIs in a general-purpose language that

somehow work more like a separate language. Since then, there has been an explosion in the number of these embedded languages.

Embedded languages can be easier to develop than they would be as stand-alone languages because they can re-use aspects of their host language, such as development tools, parsers, compilers, and documentation systems. Dependently typed languages are a particularly interesting host for these languages: not only do dependent types allow the embedded language to have a rich type system that is enforced by the type system of the source language, they also provide a means to specify programs in a form amenable to proof automation. Oury and Swierstra [OS08] describe an embedding of part of Cryptol into Agda, a data description language, and a database query language. The Bedrock project [Chl13] developed a DSL embedded in Coq that supports verified development of low-level code with an abstraction level that is similar to a macro assembler. In Idris, Brady and Hammond [BH12] implemented a resource-safe imperative language, and Brady later built an algebraic effects language [Bra13c] where resource protocols can depend on the results of effectful operations [Bra14].

As the implementation of an embedded language increases in complexity, the issues faced by the implementer begin to approach those faced by the implementer of a stand-alone language. It becomes more and more difficult to provide error messages that make sense in the context of the embedded language, non-trivial type checking and elaboration may be required, and it may not be easy to arrange for extrinsic proofs required by the DSL's internal representation to be constructed transparently.

Rather than using the richness of type theory as a very big hammer to drive in a screw, this dissertation explores the application of more traditional forms of compile-time metaprogramming to a dependently typed language. Some of the technology developed, such as error reflection, is not specific to dependent types. On the other hand, the reflected elaborator only makes sense in the context of a system that is implemented using the same technique as Idris itself.

Some portions of this dissertation are based on previously published papers. We indicate each of these in the text.

Summary of Contributions

This dissertation is divided into two parts. Part I describes the context within which the work should be understood, and Part II describes the specific new contributions of this dissertation. In particular,

- Chapter 6 presents a general means of extending commonly-used pretty printing libraries with semantic information, which enables features like interactive error messages;
- Chapter 7 presents a means of implementing quasiquotation in a tactic-based elaborator, so that the results of elaboration are reified as a datatype;
- Chapter 8 presents *error reflection*, a method for allowing users and library authors to improve error messages;
- Chapter 9 describes *elaborator reflection*, in which Idris's elaborator is reified into Idris and made into a means of language extension; and
- Chapter 9 also contains a number of examples of elaborator reflection, demonstrating its utility.

All of these features have been incorporated into Idris itself. At the time of writing, the source code to Idris is available from `https://www.github.com/idris-lang/Idris-dev`.

Typographical and Spelling Conventions

This dissertation uses the following typefaces:

- `Monospace text` is used for code in Idris or any other language, as a user might type it, where an anonymous function that adds one to its argument is written `\x => x + 1`.
- Mathematical notation is used for internal representations of type theory, where an anonymous function that adds one to its argument is written $\lambda\ x : \mathrm{Nat}\ .\ x{+}1$.
- Identifiers are typically colored according to the standard Idris color scheme, itself based on Conor McBride's color scheme for Epigram. Data constructors are red, type constructors are blue, user-defined

constants and operators are green, and bound variables are purple. Implicitly bound variables are *italic purple*.

- Program text that has not yet been type checked, along with programs written in non-dependently-typed languages such as Haskell, are written in `black monospace text`.
- Code samples that contain errors are written with a red wavy underline.
- Lines of code that simulate an interactive session begin with a boldface **>**, and the response from the system is immediately beneath.
- Tactics in the idealized tactic metalanguage are written in SMALL CAPS.
- Nonterminals in a grammar are written in *black italic text*, while concrete terminals are written either in `monospace text` when the grammar represents a user-facing programming language or in mathematical notation when the grammar represents an internal representation of type theory.

Due to its origin at the University of St. Andrews, Idris libraries conventionally use UK spellings of English words. In particular, the "-ise" spelling is preferred over the "-ize" spelling. Thus, literal operator names will sometimes occur with these spellings, even though the text is written according to US spelling conventions.

Part I

Background

Chapter 2

Embedding Languages

The original motivation for the work presented in this dissertation was to narrow the usability gap between embedded and stand-alone domain-specific languages, as well as to provide tools to implement type checkers more easily. Even though the tools that were developed have broader applicability, knowing their roots will help in understanding them.

Embedded languages are significantly easier to develop than stand-alone languages. They can inherit from the substantial work that has been put into the development tools and libraries that exist for their host language.

However, embedded languages are also restricted by their host language. Just as their syntax and tools can build upon those of the host, they are also limited by them. Features that are fairly straightforward to add to a stand-alone language implementation, such as reporting source code locations in runtime error messages, can be difficult or impossible when the language is embedded. Even worse, details about the host language can "leak", leading to inscrutable error messages.

This chapter contains an overview of commonly-used techniques for embedding languages, with a special focus on their applicability to dependently typed languages such as Idris.

2.1 Embeddings, Deep and Shallow

One of the first decisions that must be made in the course of embedding a DSL in some other language is whether to implement a *deep* or *shallow* embedding, a distinction that goes back to Boulton et al. [Bou+92].

While Boulton et al. were writing in the context of embedding hardware description languages in proof assistants, the terminology has spread to language embeddings that are performed in other contexts.

Shallow embeddings of DSLs directly use the features of the host language where they coincide with the features of the embedded language. For example, addition in an embedded language may be implemented directly as addition in the host language. One common advantage of shallow embeddings is a syntax for programs that is closer to that of the host language's syntax for programs. Additionally, because the embedded language can be extended by defining new host-language programs, it can be easier to extend than a deep embedding, which can require adding new constructors to a datatype.

A deep embedding uses an ordinary datatype to represent the abstract syntax of the embedded language. Because DSL programs are represented as explicit tree structures, it is straightforward to use them in multiple ways, such as interpretation and compilation. Additionally, generalized algebraic datatypes (GADTs) or indexed families provide a means of using the host language's type system to implement the DSL's type system, either in part or in whole, which can reduce the burden of implementing type inference or type checking for the embedded language. Compared to shallow embeddings, deeply embedded languages often require more code to be written, and it is more difficult to extend them because host language features cannot be re-used as readily.

It was previously believed that it was not possible to implement the kinds of static type guarantees that are provided by using GADTs or indexed families to represent terms in the embedded language. However, techniques like Carette et al.'s final tagless encoding [CKS09] and Hofer et al.'s related polymorphic embedding technique [Hof+08] provide a means of encoding expressive embedded type systems while still enabling multiple interpretations of the same code.

2.2 Representing Binders

Many useful languages include some notion of variables and the scopes within which they are bound. However, representing binding and substitution is fundamentally difficult, both because binding trees are usually considered according to their α-equivalence classes and because name capture can cause difficult-to-diagnose bugs. Embedded languages, whe-

ther they are deeply or shallowly embedded, have an additional challenge: they must be readable and writable by their intended users, which rules out approaches such as the direct use of de Bruijn indices [deB72].

Many embedded languages re-use the host language's variable binding mechanisms, a technique referred to as higher-order abstract syntax, or HOAS [PE88]. Not only does HOAS allow the embedded language to "piggyback" on the host language's implementation of substitution, it can also provide a much nicer syntax for variables and binders. Unfortunately, typical representations of HOAS lead to datatypes that are not strictly positive (that is, datatypes where a recursive reference to the type is to the left of a function arrow in a constructor argument), rendering them unfit for use in dependently typed languages like Idris. Even worse, using host-language lambda expressions for variable scopes can allow the representation of *exotic terms*, which are term representations that do not correspond to actual terms. For example, in a datatype that uses host language functions to represent DSL variable bindings, it is possible to use a function that does a case analysis on its argument, instantiating completely different terms. Some solutions to this problem exist, such as Chlipala's parametric HOAS [Chl08]. The field of tools and languages for reasoning about syntax trees with binders is vast, and giving a complete survey here would detract from the presentation of the work on reflection and metaprogramming.

Idris has a feature called *DSL notation* [BH12] that allows the direct reification of Idris's nominal syntax for bindings to datatypes that represent a form of de Bruijn indices. When using DSL notation, expressions are provided that correspond to the zero and successor constructors of the de Bruijn indices, and constructors are specified for each overloadable binding form. By suitably indexing the term and variable index datatype with contexts and type information, this approach can provide the syntactic convenience of a HOAS encoding along with the semantic convenience of de Bruijn indices.

2.3 Syntactic Re-Use

Because embedded languages derive much of their benefit from the ability to re-use aspects of their host language, techniques for embedding should ideally allow as much of the host language's syntax to be re-used as possible. Indeed, this is one of the reasons why shallow embeddings

are useful, absent a Lisp-style macro system.

Repurposing the host languages' features is also useful in languages without the full syntactic abstraction of the Lisp tradition. Rompf et al. [Rom+13] describe the selective re-use of components of the host language as *language virtualization*. Their paper contains a survey of language mechanisms that allow virtualization, including C++'s expression templates, Haskell's `do`-notation, and F#'s computation expressions [PS14].

An alternative version of the Scala compiler, which is called Scala-virtualized [Rom+13], enables almost all of Scala's syntax to be rewritten to method calls. This makes it possible to repurpose features such as Scala's pattern matching and variable definition syntax. Scala-virtualized is used to implement the Lightweight Modular Staging framework [RO12], which is a reusable deep embedding of a core language that has been used to implement a number of embedded languages. Svenningsson and Axelsson [SA13] describe another approach to achieving a convenient syntax for deeply embedded DSLs that does not require modifications to the host language. They combine deep and shallow embeddings such that the shallow embedding produces terms in a deeply embedded core language. This approach has the extensibility and convenient syntax of shallow embeddings and the potential for code generation of deep embeddings.

Najd et al. [Naj+15] describe an alternative to language virtualization that relies on quotation mechanisms to enable the re-use of host-language syntax. Their methodology, referred to as Quoted Domain-Specific Languages (QDSL), consists of using typed quotations of the host language together with a normalization procedure that allows Gentzen's subformula property to be used to reason about the presence or absence of certain features in the embedded programs. Because of the use of quotation, essentially arbitrary sections of the host language can be re-used by the embedded language.

2.4 Elaborating Embedded Languages

Much of the work in this dissertation grew out of a project to define a dependent type system for the modeling language for life insurance and pension products [Chr13b; Chr+14] that was developed as part of the Actulus project. This type system was never completed. However, in the course of this work, it became apparent that we needed a way to re-use parts of the implementation of a language like Idris or Agda, and that ex-

isting paradigms of language embedding would be insufficient. Work on reflection proceeded originally from this intention.

Chapter 3

Reflection and Metaprogramming

In this dissertation, we use the term *metaprogramming* to refer to the development of programs that generate or modify other programs, and the term *reflection* to refer to the representation of aspects of a system in itself. Metaprogramming and reflection are often intimately connected, so that metaprograms can use reflected information about their context in the production of new programs. However, the specific facilities that are available or desirable depend on the system in which they occur.

Smith [Smi84] founded the current tradition of research on reflection in programming languages. He focused on making a Lisp interpreter capable of reflecting on its own operational semantics. This style of reflection is typically referred to as *computational reflection* or *dynamic reflection*, and it has since seen wide application in a variety of contexts. Demers and Malenfant [DM95] document some of the further development of this idea in functional, object-oriented, and logic programming. However, as this dissertation focuses primarily on reflection that occurs at compile time, we will not document the breadth of work on computational reflection.

3.1 Quasiquotations in Programming Languages

The notion of quasiquotation was invented by Quine in his 1940 book *Mathematical Logic* [Qui81, pp. 33–37]. While ordinary quotations allow one to *mention* a phrase rather than *using* it, quasiquotations allow these quoted expressions to contain variables that stand for other expressions, just as mathematical expressions can contain variables that stand for val-

ues. In other words, a specific class of subexpression is treated as a *use* within a context that is *mentioned*. Quine used Greek letters to represent variables in quasiquotations. Søndergaard and Sestoft [SS90] explore referential transparency, an idea also due to Quine, as it relates to programming languages, including in languages with quotations.

The paradigmatic instance of quasiquotation in programming languages is that found in the Lisp family. Bawden's 1999 paper [Baw99] summarizes the history and semantics of the quasiquotation mechanism found in both the Scheme family of languages and in Common Lisp. In the Lisp family, program code is represented in a uniform manner, using lists that contain either atomic data, such as symbols, strings, and numbers, or further lists. In Lisp parlance, these structures are referred to as "S-expressions". Because S-expressions are simply ordinary data, it makes sense to quote them, yielding a structure that can easily be manipulated. Additionally, most Lisps have a quasiquotation system, in which specially marked subexpressions of a quotation are evaluated, with the result substituted into the quotation. Unlike Quine's quasiquotation, the Lisp family of languages allow arbitrary expressions to be inserted into quasiquotations.

Languages outside of the Lisp family have also used quasiquotation to implement language extension. The Camlp4 system [Rau03] provides quasiquotation for the OCaml language, among other extensions. Quasiquotations in Camlp4 consist of arbitrary strings that are transformed by a *quotation expander* to either a string representing valid concrete syntax or to an abstract syntax tree. These quotations support antiquotation, which invokes the parser to read an OCaml expression or pattern inside of the quotation. Template Haskell's quasiquotations [Mai07] work on similar principles. Both systems fully expand all quotations at compile time, and both check that the generated code is well-typed.

The MetaML family of metaprogramming facilities [TS00], including MetaOCaml [MOC] and F# [Sym06], implement a style of quotation in which the type of quoted expressions is parameterized over the type that would be inhabited by the quoted expression if it were spliced into a program. These features are intended for use in staged computation. In addition to representing the types of the quoted expressions, these staging annotations feature static scope, so a quotation that contains a name contains the version of that name from the scope in which the term was quoted.

Barzilay [Bar06] defined a form of quasiquotation called *operator shift-*

ing to implement reflection in the Nuprl proof assistant. Like Lisp, Nuprl has a uniform representation of terms as operators applied to a collection of operands. Unlike Lisp, the Nurpl term representation has a uniform, built-in means of representing binding structures. The shifted representation of an operator simply decorates the operator as denoting a quoted version of itself, and the binding structure is left intact, in a form of higher-order abstract syntax where exotic terms are ruled out through a syntactic restriction.

Scala quasiquotations [SBO13] are very much like Lisp quasiquotations. While their syntax resembles that of strings, this is a consequence of their implementation using Scala's string interpolators and they are in fact expanded to tree structures at compile time. The quasiquotations were initially intended to serve as a technique for implementing Scala macros [Bur13], but they are also useful for both runtime code generation as well as generating program text. Scala macros closely resemble Lisp macros, in that they do not intend to allow arbitrary strings to be used as syntax, but instead implement transformations from one valid parse tree to another. Unlike Lisp, Scala programs that contain macros are type checked after macro expansion, and they are represented by a conventional tree structure that macros manipulate. Quasiquotations are a means of constructing and destructuring these trees using the syntax of the high-level Scala language.

Like Scala, C# is an object-oriented language with a notion of quotation [C#E]. In C#, quotation can be applied to an anonymous function by annotating it with the `Expression` type, which causes a datatype representing the function's AST to be generated instead of the function itself. However, this feature cannot properly be considered quasiquotation, as there is no mechanism for escaping the quotation and inserting a sub-tree that has been generated elsewhere.

3.2 Static Reflection

Racket has perhaps the most advanced static reflection features of any language. Racket's "languages as libraries" [Tob+11] features allow entire new languages can be defined in Racket. Each module specifies the language in which it is written, and languages are defined by libraries that control both parsing and the expansion of modules into a core language. These libraries can make use of arbitrary Racket code during expansion,

and they have access to a rich environment of metadata about the system.

Both Template Haskell [SJ02] and Scala's macro system [Bur13] support querying the global state of the compiler during metaprogram execution, using this reflection to inform code generation and metaprogramming.

While popular object-oriented languages such as Java and C# generally have poor support for static reflection, the research community has produced some static reflective metaprogramming systems for them. A popular approach has been pattern-based reflection, where classes can be defined by recursion over the structure of other classes. Examples of this approach include Genoupe [DLW05], class morphing [HZS07], and Miao and Siek's [MS14] metaprogramming system for Java. Research has typically focused on increasing the safety of metaprograms. However, it is somewhat unclear to what extent these tools should be classified as reflection according to the definition used in this dissertation, because the language used to represent and process reified definitions is typically not the programming language itself, but rather a special-purpose DSL.

3.3 Reflection and Dependent Types

With inductive-recursive families, dependently typed languages are powerful enough to represent dependent type systems internally. In other words, in sufficiently rich dependently typed languages, it is possible to define a datatype of terms that represent dependently typed languages such that only the well-typed terms can be represented. This line of work was begun by Danielsson [Dan07], and it was continued by Chapman [Cha09] and McBride [McB10]. Additionally, closed type theories like the one planned for Epigram 2 [Cha+10b] remove the distinction between datatypes that are represented in some universe and those that are defined ahead of time as extensions to the type theory. Rather than attempting to model the semantics of type theory in type theory, Devriese and Piessens [DP13] construct a model of well-typed terms and datatypes that is explicitly intended for use in metaprogramming. They provide two examples: a code generator that derives a function to render elements of a type to strings as well as a type-safe implementation of Coq's `assumption` tactic. While these encodings have good type safety properties, they unfortunately require a titanic effort to implement and use.

The Nuprl team have experimented with multiple implementations of

reflection. Knoblock [Kno87] focused on reflecting the metalogic of Nuprl, implementing Nuprl-level versions of the judgment forms and refiner, resulting in an infinite tower of reflected proof systems. His focus was on creating verified proof tactics, such that a tactic to be used at level n could be verified by the system found at level $n + 1$. Allen et al. [All+90] describe an alternative notion of reflection of the logic in which a fixed-point construction is used to represent proofs within a single language. Both of these approaches rely on a datatype that represents terms as well as proofs - in some sense, they provide a deep embedding of aspects of Nuprl in itself.

All of the above implementations of reflection in dependent types are instances of what Barzilay [Bar06] refers to as *indirect reflection* in his Ph.D. thesis on reflection in Nuprl. Indirect reflection is a form of reflection in which a language or parts of a language are reimplemented in the reflective model. Barzilay identifies a number of downsides of indirect reflection:

- implementing a complex system, like a type checker, is a great deal of work
- ensuring a perfect correspondence between the reflected portions of the implementation and the original implementation is a major task
- the size of a term's representation will typically grow exponentially with each level of quotation that is applied

Instead, he suggests the use of *direct reflection*, in which the existing implementation is exposed to the object language.

3.3.1 Agda's Reflection System

Agda, a close relative and predecessor of Idris, has a reflection system that has been used to implement non-trivial proof automation and code generation features. Though the system is primarily documented in Agda's release notes [Agda], some examples of its use exist, including its use for deriving instances in Ulf Norell's alternative standard library[1] and in a

[1] Avaliable from `https://github.com/UlfNorell/agda-prelude/` at the time of writing

convenient interface to the Agda standard library's semiring solver by Jedynak.[2] Additionally, there have been applications and descriptions in the academic literature. van der Walt's M.Sc. thesis [vdW12] and the associated paper [WS12] contains a description and analysis of Agda's reflection capabilities *anno* 2012. In 2012, Agda's reflection mechanism lacked key features, such as the ability to generate new pattern-matching definitions, which meant that it was unsuitable for tasks such as automating the Bove-Capretta transformation [BC03]. However, it has since been extended with the ability to unquote definitions as well as terms. Additionally, van der Walt pointed out that transformations written using Agda's reflection are inherently untyped, which means that Agda's type system cannot be used to enforce their correctness — a feature shared by the reflection system described in this dissertation. The advantage of an untyped reflection system is that it allows an escape from the type system, and traditional proof by reflection can still be used to implement verified proof automation.

Later, Kokke and Swierstra [KS15] implemented a general-purpose proof search, modeled on Prolog, and used reflection to enable it to be employed for proof automation. Because this proof search is implemented entirely in Agda itself, it can be extended and customized using Agda's excellent development environment and tools. However, Kokke and Swierstra had to implement essentially the entire system themselves, using reflection at the borders to connect their model of terms to actual Agda terms. Because of this, the framework supports only first-order unification, failing even on simple cases such as Σ types being used to model non-dependent pair types. This is an example of the difficulty of implementing indirect reflection.

[2] Available from `https://github.com/wjzz/Agda-reflection-for-semiring-solver` at the time of writing

Chapter 4

The Idris Elaborator

Idris is a quite complex language, with many interacting features. Rather than attempting to write a correct type checker for all of Idris, the compiler is implemented as a multi-stage process in which the desugared high-level Idris program is first elaborated to a simple core theory, called TT. In TT, all global names are fully qualified, all local names are resolved to de Bruijn indices, all arguments are explicit, and features such as type classes have been completely removed. If elaboration succeeds, the fully-explicit program is then type checked again. Because TT is such a simple language, its type checker can be similarly simple — in particular, it need not perform unification. This reduces the size of the trusted code base.

Additionally, the strict separation between the high-level language and the core language frees implementers to extend the high-level language without fear of undermining the safety of the system as a whole. New extensions to Idris should therefore be explained by giving their translations to TT.

The overall architecture of the Idris compilation process is described in Figure 4.1. First, the high-level Idris language is desugared to a form that Brady [Bra13b] refers to as Idris^{-}, in which features like conditional expressions have been replaced by function calls and implicitly quantified variables have been added. Because this dissertation is concerned with the elaboration and type checking process, details about the subsequent code generation process will not be described.

The Idris elaborator, described in detail in Brady's 2013 paper [Bra13b], uses proof tactics to translate desugared Idris to the core type theory TT. When elaborating a language such as Idris, there are a number of language features that can interact in complex ways to obtain the informa-

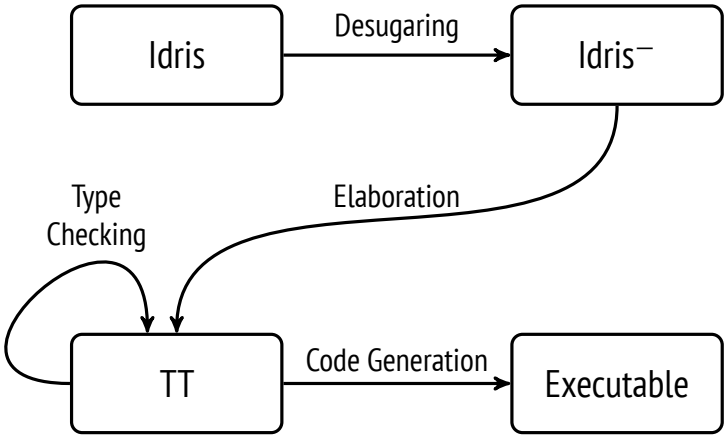

Figure 4.1: The Idris compilation process.

tion necessary to produce a term in TT. For instance, resolving a multi-parameter type class might lead to one of its arguments becoming known, which may enable the solving of an implicit argument, which may cause a `where`-bound definition to type check, which may itself provide information that makes it possible to resolve further type class instances. This complex interplay of language features could quickly lead to special-case code that is difficult to maintain. When writing the present version of Idris, Brady's solution to this problem was to incrementally build terms in a manner that allows the tracking of dependencies between language features without requiring special-case handling of each of them.

The Idris term elaborator is implemented in Haskell, using a monad that has both state and error effects. Rather than a general exception-handling mechanism, the error handling of the elaborator supports only recovery, with no means of distinguishing between thrown exceptions. The state consists of

- a *goal type*, which is the type of the term that is under construction;
- a possibly-incomplete *proof term*, which should inhabit the goal type at the end of elaboration;
- a *hole queue*, tracking the incomplete portions of the proof term;
- a collection of open *unification problems*, representing recoverable failures of unification that may yet unify once more variables are solved; and

- various deferred operations, such as the bodies of case blocks that need to be elaborated.

Additionally, the elaborator has read-only access to portions of the global Idris state, such as the global definition context.

In addition to the term elaborator, there is also a definition elaborator for each form of top-level definition. The definition elaborators invoke the term elaborator for Idris terms that occur in definitions, and then use the resulting TT terms to produce definitions in the global context. For example, to elaborate a function definition

```
plus : Nat → Nat → Nat
plus Z     k = k
plus (S j) k = S (plus j k)
```

the definition elaborator must first invoke the term elaborator with the goal `Type` to elaborate the type signature. When this is done, the left- and right-hand-sides of each definition clause can be elaborated as terms, taking into account the type that was elaborated for `plus`.

The concrete manipulations of the elaboration state are performed by a collection of operations that are called *tactics*, by analogy to the operations available in proof assistants. These tactics can place a term into a hole, create new holes, introduce binders, and more. While Appendix A contains a complete list of these tactics, we introduce them as they are first used in this example.

4.1 The Core Language

To prevent confusion with high-level Idris, TT will be written in mathematical syntax. The term language of TT is given by three syntactic categories: terms, constants, and binders. These are defined in Figure 4.2. In addition to this syntax, when x is not free in t_2, we will sometimes write $t_1 \to t_2$ as an abbreviation for $\forall\ x\ :\ t_1\ .\ t_2$.

The category c of constants includes additional primitive types and their canonical values, such as characters, machine integers, and floating-point numbers. The universe hierarchy is predicative, with cumulativity; however, universe annotations are not required because a solver is used to ensure that there are no cycles in the universe graph. The details of the predicative hierarchy are outside the scope of this dissertation, so $\star$

t	$::=$	c	Constants
	$\mid$	x	Variables
	$\mid$	$b\ .\ t$	Variable bindings
	$\mid$	$t\ t$	Application
	$\mid$	T	Type constructors
	$\mid$	C	Data constructors
b	$::=$	$\lambda\ x : t$	Functions
	$\mid$	$\mathsf{let}\ x \mapsto t : t$	Let-bindings
	$\mid$	$\forall\ x\ :\ t$	Dependent function types
c	$::=$	$\star_i$	Type universes
	$\mid$	i	Integer literal
	$\mid$	Integer	Integer type
	$\mid$	s	String literal
	$\mid$	String	String type
	$\mid$	...	

Figure 4.2: The grammar of terms t, binders b , and constants c in Idris's core language TT.

will be written without subscripts for the remainder. The typing rules and operational semantics of TT are completely standard; please refer to Brady's 2013 paper [Bra13b] for details.

4.2 Pattern-Matching Definitions and Data Types

Evaluation and type checking of TT terms occurs within a global context that defines inductive families [Dyb94] and pattern-matching definitions. Note that the TT term language has no built-in means of destructuring inductively-defined data types, such as a `case` expression — all pattern-matching occurs in top-level definitions.

A pattern-matching definition in TT consists of two parts: a type declaration and zero or more clauses. Each clause consists of zero or more pattern variable bindings, a left-hand side, and a right-hand side.

For example, the type declaration for addition on the natural numbers is written

$$\mathrm{plus} : \forall\ j\ :\ \mathrm{Nat}\ .\ \forall\ k\ :\ \mathrm{Nat}\ .\ \mathrm{Nat}$$

and its clauses are written

$$\begin{array}{l}\mathsf{pat}\ k : \mathrm{Nat}\ . \\ \quad \mathrm{plus}\ \mathrm{Z}\ k \mapsto k \\ \mathsf{pat}\ j : \mathrm{Nat}\ .\ \mathsf{pat}\ k : \mathrm{Nat}\ . \\ \quad \mathrm{plus}\ (\mathrm{S}\ j)\ k \mapsto \mathrm{S}\ (\mathrm{plus}\ j\ k)\end{array}$$

where the argument names need not coincide between the declaration and the various cases. All clauses must apply the function being defined to the same number of arguments, and the left and right sides of each clause must have convertible types.

The datatype Nat of Peano-style natural numbers is represented with the following definition in TT:

$$\begin{array}{l}\mathsf{data}\ \mathrm{Nat}\ :\ \star\ \mathsf{where} \\ \quad \mathrm{Z}\ :\ \mathrm{Nat} \\ \quad \mathrm{S}\ :\ \forall\ k\ :\ \mathrm{Nat}\ .\ \mathrm{Nat}\end{array}$$

Inductive families can have both parameters and indices. Parameters are used consistently across all constructors and depend only on fixed types and other parameters; indices can vary across constructors. For pragmatic reasons, neither TT nor Idris require that all parameters occur syntactically before all indices in type constructors.

McBride's heterogeneous equality type, also known as *John Major equality* due to an obscure joke about British politics in the 1990s, is represented as follows in TT:

$$\begin{array}{l}\mathsf{data}\ (=)\ :\ \forall\ A\ :\ \star\ .\ \forall\ B\ :\ \star\ .\ \forall\ x\ :\ A\ .\ \forall\ y\ :\ B\ .\ \star\ \mathsf{where} \\ \quad \mathrm{Refl}\ :\ \forall\ A\ :\ \star\ .\ \forall\ x\ :\ A\ .\ (=)\ A\ A\ x\ x\end{array}$$

For the sake of readability, we abbreviate the homogeneous propositional equality $(=)\ t_1\ t_1\ t_2\ t_3$ as $t_2 =_{t_1} t_3$.

4.3 The Development Calculus

Conor McBride's 1999 thesis [McB99] describes a *development calculus* in which a core dependently typed language is extended with new binding forms representing holes and guesses. The TT development calculus has the following additional binders:

$$\begin{array}{lcll} b & ::= & \ldots & \\ & | & ?x\ :\ t & \text{Hole} \\ & | & ?x \approx t\ :\ t & \text{Guess}\end{array}$$

Representing holes as a binding form allows both their types and any suggested values to refer to variables from the surrounding context. Additionally, because holes bind new variables, dependencies between terms and types are naturally represented. The identity function applied to a proof of the reflexivity of equality for some unknown natural number might then be:

$$?k \ : \ \mathrm{Nat} \ . \ \mathit{id} \ (k =_{\mathrm{Nat}} k) \ (\mathrm{Refl} \ \mathrm{Nat} \ k)$$

This term, even though it is incomplete, can be type checked. At the same time, it is not necessary to have any special machinery for representing metavariable contexts or types.

4.4 Elaboration Example

In this high-level description of elaboration, the elaborator language is represented using some typographical conventions to enhance readability. Following Brady [Bra13b], object-language names are written undecorated in the metalanguage. The operation $\mathcal{E}\,[\![\cdot]\!]$ is the the term elaborator, which elaborates an Idris$^-$ term into the currently-focused hole. The related operation $\mathcal{P}\,[\![\cdot]\!]$ is the pattern elaborator, which elaborates Idris$^-$ terms that occur in pattern contexts. The pattern elaborator is almost identical to the term elaborator, except that holes remaining after elaboration are converted into pattern variables rather than reported as errors. Finally, there are elaboration procedures for each variant of top-level definitions of the high-level Idris language, such as datatype definitions, record declarations, function type declarations, pattern-matching definitions, type classes, and instances.

Names that are written in `typewriter font` denote names that occurred in the user's program, while names that are written in mathematical *italic* denote fresh names. A Haskell-style `do`-notation is used, with the typical monadic semantics, but the code is not intended to be Haskell — rather, it is an idealized language whose interpreter is written in Haskell. As the example proceeds, successive additions to the elaboration script are highlighted in yellow.

The purpose of this section is not to define the elaboration procedure; for that, please consult Brady [Bra13b]. Rather, it is to explain the procedure through a concrete example of a realistic function. Later, in Chap-

ter 7, we will explain how to implement quasiquotation in this framework, and in Chapter 9, we will make this language available to Idris itself.

4.4.1 Desugaring

The example definition to be elaborated is the following:

```
replicate : (n : Nat) → a → Vect n a
replicate Z     x = []
replicate (S k) x = x :: replicate k x
```

Prior to elaboration, the implicit arguments are discovered and syntactic sugar is removed, yielding:

```
replicate : {a : _} → (n : Nat) → a → Vect n a
replicate {a} Z     x = []
replicate {a} (S k) x = (::) {a=_} {n=_} x (replicate {a=_} k x)
```

In Idris^{-}, all implicit bindings are made visible, `do`-notation is replaced with appeals to the bind operator `>>=`, and the `if-then-else` syntax is replaced with a call to the appropriate function. In Idris, function arguments declared with curly braces (such as `a` above) are implicit arguments that will be found by the elaborator. As in Haskell, integer literals are desugared to applications of an operator `fromInteger`, but unlike Haskell, this operator can be overloaded on an *ad hoc* basis and does not require an implementation of the entire `Num` type class.

4.4.2 Elaborating the Type

The first step following desugaring is to elaborate the type that was provided for `replicate`. Because a type declaration is declaring a type, the elaboration goal is $\star$. Elaboration proceeds by recursion over the syntactic structure of the declared type. At the beginning of elaboration, the state is:

Goal $\star$

Term $?h \ : \ \star \ . \ h$

Holes h

and the high-level Idris term being elaborated is `{a : _} → (n : Nat) → a → Vect n a`.

The topmost node in the Idris AST is the function type. When elaborating a function type, the first step is to produce a new hole for the argument type, after which this hole is used as the argument type in a fresh function space binder. Following this, the Idris representation of the argument type is elaborated into the hole. In this case, the type of the argument `a` is a type, so we can elaborate it with the goal $\star$. The CLAIM tactic is used to introduce a new hole t_a that is available in the current hole h's scope. Running CLAIM t_a : $\star$ thus yields the following elaborator state:

Goal $\star$

Term $?t_a : \star \,.\, ?h : \star \,.\, h$

Holes h, t_a

Now that there is a representation for the type of `a`, the dependent function type itself can be elaborated. To produce a dependent function type, the elaborator uses the FORALL tactic, which takes a name and a type and surrounds the focused hole with a dependent function binding. The tactic script is, thus far,

```
do CLAIM t_a : ⋆
   FORALL a : t_a
```

and the resulting elaborator state is

Goal $\star$

Term $?t_a : \star \,.\, ?h : \star \,.\, \forall\, a : t_a \,.\, h$

Holes h, t_a

The next step is to elaborate the value of `a`'s type, which is currently represented by the hole t_a. The elaborator therefore focuses on t_a using the FOCUS tactic, which brings its argument to the beginning of the hole queue. The tactic script is, thus far,

```
do CLAIM t_a : ⋆
   FORALL a : t_a
   FOCUS t_a
```

and the resulting elaborator state is

Goal $\star$

Term $?t_a : \star . ?h : \star . \forall\ a : t_a . h$

Holes t_a, h

Note that t_a is now at the head of the hole queue, and thus in focus.

The type of `a` is the placeholder term `_`, because the user did not provide an annotation. When the elaborator encounters a placeholder, it abandons the hole, expecting that it will be solved later via unification constraints. This abandonment is achieved through the use of the UNFOCUS tactic, which sends its argument to the end of the hole queue, bringing the next hole into focus.

The outermost binding of the type (the binder `{a : _} →`) has now been elaborated. What remains to be elaborated is the body `(n : Nat) → a → Vect n a`. Once again, CLAIM is used to establish a hole for the type of the argument `n` and FORALL establishes a dependent function binding around the hole h. This time, however, a concrete type is available, so it can be elaborated immediately instead of being deferred with UNFOCUS.

The FILL tactic places a term in the current hole. The global Idris name `Nat` resolves to the TT type constructor Nat, so the tactic script and elaborator state are now:

```
do CLAIM t_a : ⋆
   FORALL a : t_a
   FOCUS t_a
   UNFOCUS t_a
   CLAIM t_n : ⋆
   FORALL n : t_n
   FOCUS t_n
   FILL Nat
```

Goal $\star$

Term $?t_a : \star . ?h : \star . \forall\ a : t_a . ?t_n \approx \mathrm{Nat} : \star . \forall\ n : t_n . h$

Holes t_n, h, t_a

The hole t_n now contains a *guess*: namely, Nat. The SOLVE tactic substitutes this guess in the scope of the hole binder and removes it from the hole queue, yielding

```
⋮
CLAIM t_n : ⋆
FORALL n : t_n
FOCUS t_n
FILL Nat
SOLVE
```

Goal $\star$

Term $?t_a : \star . ?h : \star . \forall a : t_a . \forall n : \mathrm{Nat} . h$

Holes h, t_a

What remains to be elaborated into h is the type `a -> Vect n a`. Because the user has not provided a name for this binding, a fresh name x is used. Otherwise, this process is the same as for earlier dependent function bindings.

```
⋮
FILL Nat
SOLVE
CLAIM t_x : ⋆
FORALL x : t_x
FOCUS t_x
FILL a
SOLVE
```

Goal $\star$

Term $?t_a : \star . ?h : \star . \forall a : t_a . \forall n : \mathrm{Nat} . \forall x : a . h$

Holes h, t_a

Finally, it is time to elaborate the result type of the function, `Vect n a`, into h. `Vect` has no implicit arguments, so we need only establish holes for its visible arguments, again using CLAIM. Just as the elaborator determined that `Vect` has no implicit arguments by consulting the global context, it also discovered the elaborated forms of these arguments types. After establishing argument holes, FILL is used to place the operator applied to its operand holes into the present hole.

```
⋮
FILL a
SOLVE
CLAIM a₁ : Nat
CLAIM a₂ : ⋆
FILL Vect a₁ a₂
SOLVE
```

Goal $\star$

Term $?t_a : \star\ .\ \forall\ a : t_a\ .\ \forall\ n : \mathrm{Nat}\ .\ \forall\ x : a\ .\ ?a_1 : \mathrm{Nat}\ .\ ?a_2 : \star\ .\ \mathrm{Vect}\ a_1\ a_2$

Holes t_a, a_1, a_2

After solving h, the elaborator focuses on each argument in turn, elaborating the argument from the source code. The first of these is a_1, which the elaborator will fill and solve based on the source term `n`, after which it focuses on a_2.

```
⋮
FILL Vect a₁ a₂
SOLVE
FOCUS a₁
FILL n
SOLVE
FOCUS a₂
```

Goal $\star$

Term $?t_a : \star\ .\ \forall\ a : t_a\ .\ \forall\ n : \mathrm{Nat}\ .\ \forall\ x : a\ .\ ?a_2 : \star\ .\ \mathrm{Vect}\ n\ a_2$

Holes $a_2\ , t_a$

When a_2 is filled with `a`, the elaborator unifies the type of the hole with the type of the expression being placed in it. Unification succeeds immediately, yielding the constraint $t_a = \star$, which induces a substitution of the hole t_a. Thus, both holes are dispatched at once. The final tactic

script and proof term are:

```
do CLAIM t_a : ⋆
   FORALL a : t_a
   FOCUS t_a
   UNFOCUS t_a
   CLAIM t_n : ⋆
   FORALL n : t_n
   FOCUS t_n
   FILL Nat
   SOLVE
   CLAIM t_x : ⋆
   FORALL x : t_x
   FOCUS t_x
   FILL a
   SOLVE
   CLAIM a_1 : Nat
   CLAIM a_2 : ⋆
   FILL Vect a_1 a_2
   SOLVE
   FOCUS a_1
   FILL n
   SOLVE
   FOCUS a_2
   FILL a
   SOLVE
```

Goal $\star$

Term $\forall\ a : \star\ .\ \forall\ n : \mathrm{Nat}\ .\ \forall\ x : a\ .\ \mathrm{Vect}\ n\ a$

Holes None

Finally, the resulting term is re-checked against the goal type using the core type checker, which does not contain complex features such as unification, in the interest of reducing the trusted code base. When this succeeds, the declaration elaborator adds the type signature for `replicate` to the global environment along with a specification of which arguments are to be provided implicitly.

4.4.3 Elaborating the Cases

Because a pattern-matching function definition extends TT with new reduction rules, it is imperative that both sides of each equation have the same type so that subject reduction is preserved. However, Idris function type declarations do not distinguish between the arguments that occur on the left-hand side of the definition from those that will be processed by the value of the right-hand side, which makes it difficult to know the types of the two sides ahead of time. Additionally, the concrete constructors matched in patterns can refine the type of the application as a whole, which means that the type will vary from pattern to pattern.

This means that the two sides should be elaborated in a context in which their types can be inferred. However, the elaboration infrastructure that has thus far been presented requires an explicit goal type, and uses that goal type for things like solving implicit arguments by unification. The solution to this is to establish a context in which the type can be inferred. This is done using the following auxiliary datatype:

```
data Infer : ⋆ where
  MkInfer : ∀ t : ⋆ . ∀ x : t . Infer
```

By using Infer as the goal and leaving the first projection the constructor MkInfer as a hole, the term elaborator can build a term in the second projection while imposing constraints that can lead to the solving of the first projection automatically. Then, the elaborator can project the desired type and term from the constructor and use the type as the goal for the right-hand side.

The first case to elaborate is the Idris equation `replicate {a} Z x = []`. The first step in elaboration is to establish the type inference context, using the following script:

```
do CLAIM t : ⋆
   CLAIM h : t
   FILL (MkInfer t h)
   SOLVE
   FOCUS h
```

Then, the left-hand side is elaborated into h. Because it is an application, holes will be generated for each argument, and the provided argument term will be recursively elaborated into its hole. However, in a pattern context, the elaboration procedure for names is slightly different than in

an expression context: if filling the hole with the name doesn't work, the hole is instead filled with a pattern variable that has the given name.

Once h has been filled with the concrete term, the elaborator projects the left hand side and its type out of the MkInfer constructor. Then, the type can be used as a goal for the right hand side, under the same pattern variables.

When the pattern-matching cases have been elaborated, they are rechecked just as the type signature was. Then, they can be added to TT as new reduction rules.

4.5 Binders in the Elaborator

In the interest of simplicity, the previous section papered over a detail about the elaboration process. Tactics such as FORALL, INTRO or LETBIND that produce new binders around the focused hole have an additional precondition: the scope of the hole binder must consist precisely of a reference to the hole. If this precondition is not met, then they fail.

To see why this precondition is necessary in general, take the focus to be $?h \ : \ t_1 \to t_1 \ . f \ h$ and the high-level Idris term being elaborated to be `\x ⇒ x`. Following the ordinary rules of elaboration, the next step would be to use the INTRO tactic. However, this would result in the term $\lambda \ \mathtt{x} : t_1 \ . ?h \ : \ t_1 \ . f \ h$. In other words, the application of f has been moved underneath the lambda, rather than being applied to the function as a whole.

One possibility would be to change the semantics of the binding tactics to wrap the *references* to the hole in a binder. However, if f were to reduce to a form that had multiple references to h, then the elaborator would need to perform more work to track down all copies of h. The implementation of the binding tactics in terms of the low-level term language would also become quite a bit more complex. Additionally, performing the wrapping in the scope of the hole binding may lead to unwanted computation, as the two-stage FILL-SOLVE process is subverted.

The solution, first presented by McBride [McB99], is an additional tactic called ATTACK. If the focus is a hole h with type t, then ATTACK fills h with a guess consisting of a new hole binding h' with type t.

In the above example, running ATTACK yields the term

$$?h \approx (?h' \ : \ t_1 \to t_1 \ . h') \ : \ t_1 \to t_1 \ . f \ h$$

with a focus on h'. Now, applying the INTRO tactic results in the following term:

$$?h \approx (\lambda\ \mathsf{x} : t_1\ .\ ?h'\ :\ t_1\ .\ h')\ :\ t_1 \to t_1\ .\ f\ h$$

The hole h' is then filled and solved by the just-introduced x, and focus returns to h. The term is now:

$$?h \approx (\lambda\ \mathsf{x} : t_1\ .\ \mathsf{x})\ :\ t_1 \to t_1\ .\ f\ h$$

and h can be dispatched with SOLVE, yielding the correct term $f\ (\lambda\ \mathsf{x} : t_1\ .\ \mathsf{x})$.

The general recipe for introducing bindings is thus to bracket the initial binding tactic and the solution of the contained hole with ATTACK and SOLVE. The elaborator calls ATTACK when entering a new scope in the term being elaborated, and it calls SOLVE when leaving again.

Chapter 5

Reflection in Idris

By *reflection*, we mean the representation of aspects of a programming language, such as terms or error messages, in the language itself. Perhaps confusingly, there is also a technique for structuring proofs in a dependently typed system that is known as *proof by reflection*, in which objects in type theory are mapped to some simpler domain in which there exists a simpler or faster decision procedure along with appropriate proofs to ensure the soundness of the mapping. Proof by reflection is accessibly described by Bertot and Castéran [BC04] and Chlipala [Chl11]. To make the terminological confusion worse, reflection in the sense of self-representation is frequently used to automate the application of the proof technique reflection. In this dissertation, when confusion could result, these two concepts are referred to as *language reflection* and *proof by reflection*, respectively.

In the context of Idris, there is an experimental form of *well-typed reflection*, in which functions decorated with a certain modifier are allowed to match intensionally against arbitrary syntax rather than against canonical values when executed during type checking. This feature, described by Brady [Bra13a], has somewhat unclear semantics due to the fact that the procedures defined with it do not respect functionality, and further description is outside the scope of this dissertation. Indeed, as it currently stands, this feature makes Idris inconsistent. This should come as no surprise, because it requires experimental extensions to the core language. The new features described in Chapters 7 and 9 are at the level of the elaborator and thus require no such changes, increasing our confidence in their safety.

Prior to this dissertation work, Idris had a simple tactic language with

basic support for reflection [Idr14]. This tactic language had only rudimentary support for constructing new tactics from old ones, requiring a fairly convoluted appeal to reflected tactics in a manner that made recursive tactics exceedingly difficult to define. The original reflection mechanism translated the core type theory to two separate representations, `TT` and `Raw`, corresponding to representations inside the compiler. The first of these representations, `TT`, is used for terms that the compiler has type checked. In this locally-nameless representation, all binders and all references to global names have complete type annotations, global names indicate whether they refer to constructors, type constructors, or functions, and universes are annotated with their level in the predicative hierarchy. The second, `Raw`, is used for terms that have not yet been type checked. This representation uses explicit names for both bound and free variables and its universes do not mention levels, as the type checker is expected to assign the appropriate levels.

The datatype `TT` is defined in Figure 5.1. The constructor `P` represents free variables to be looked up in the global context. The `NameType` states whether the looked-up name denotes a data constructor, a type constructor, an unresolved bound variable, or a function, and data and type constructors present information about their arities. Additionally, names referred to with a `P` contain the type that the name has in the global context. The constructor `V` represents local variables, here with de Bruijn indices. The constructor `Bind` takes a local name and a binder, binding that name in its scope. Because both representations of terms have the same binders available, `Binder` takes a type parameter for the terms that it contains. It is defined in Figure 5.3, and described later in this chapter. The `App` constructor represents an application of one term to another. The constructor `TConst` injects constants are injected from their own datatype `Const`, which defines the primitive types and their elements. The constructor `Erased` represents a portion of a term that has been eliminated. This occurs when the information can be reconstructed, or as a run-time optimization. The constructor `TType` is the type of types, at some universe level. Finally, the constructor `UType` represents the variant universes involved in uniqueness types.

The datatype `Raw`, defined in Figure 5.2, is similar to `TT`. Instead of the `P` and `V` constructors, it contains a single variable reference constructor `Var`, and its constructor `RType` for the type of types has no universe annotation.

In both kinds of reflected Idris terms, all binders are represented uniformly, using either the `Bind` or `RBind` constructors. Because there are mul-

```
data TT : Type where
  P : NameType → TTName → TT → TT
  V : Int → TT
  Bind : TTName → Binder TT → TT → TT
  App : TT → TT → TT
  TConst : Const → TT
  Erased : TT
  TType : TTUExp → TT
  UType : Universe → TT
```

Figure 5.1: The TT datatype, representing terms received from the type checker.

```
data Raw : Type where
  Var : TTName → Raw
  RBind : TTName → Binder Raw → Raw → Raw
  RApp : Raw → Raw → Raw
  RType : Raw
  RUType : Universe → Raw
  RConstant : Const → Raw
```

Figure 5.2: The Raw datattype, representing terms to be submitted to the type checker.

tiple ways to bind variables, but only a single constructor for binding, an auxiliary datatype is used to track the various binders, defined in Figure 5.3. In this datatype, the constructor `Lam` represents lambda abstractions, and its type argument is the type of the variable being bound. The constructor `Pi` represents function types. Its first field is the type of the argument to the function and its second field is the kind of the arrow, representing its interactions with Idris's experimental uniqueness types. Because uniqueness typing is outside the scope of this dissertation, we can safely ignore this field. The constructor `Let` represents `let` bindings with type annotations and the values of bound variables. The constructors `Hole` and `Guess` are used to represent partial terms during elaboration, as described in Section 4.4. The constructor `GHole` is a representation of a hole that will not be filled out during elaboration, but instead should be converted to a top-level definition applied to all variables in scope. This is used to implement Idris's interactive features and user-visible holes, and having it present in the reflection API makes it possible to write metapro-

```
data Binder : Type → Type where
  Lam : (ty : a) → Binder a
  Pi : (ty, kind : a) → Binder a
  Let : (ty, val : a) → Binder a
  Hole : (ty : a) → Binder a
  GHole : (ty : a) → Binder a
  Guess : (ty, val : a) → Binder a
  PVar : (ty : a) → Binder a
  PVTy : (ty : a) → Binder a
```

Figure 5.3: Reified binders.

grams that do a great deal of work and then delegate to the user to carry out some final steps. The constructors `PVar` and `PVTy` are used to represent pattern variables and their types. They will occur around the terms that represent the left and right sides of a clause in a pattern-matching definition.

Unlike Agda, which takes great pains to ensure that all names represented in a program have been directly written by a user, and thus keeps the representation of reflected names completely abstract, Idris generates many names itself. They have a structure and can be manipulated. For example, metaprograms can generate auxiliary definitions and assign them names that no user could ever type. Idris reflection represents names using the datatype in Figure 5.4. The `UN` constructor, short for "user name," represents a concrete name that a user has typed. When wrapped in a `NS` constructor, a name is considered to be in a namespace, which is represented in reverse order, which is because a suffix of a namespace is used for disambiguation. Thus, a name that is written `Prelude.Nat.(-)` is represented in reflection as `NS (UN "-") ["Nat", "Prelude"]`. The constructor `MN` represents a machine-generated name, with a unique integer corresponding to the `GENSYM` counter in a Lisp along with a display hint. In addition to these names, Idris has a collection of special names, representing names for auxiliary definitions produced during the elaboration of `with` patterns, `case` expressions, constructors of the record types underlying type classes, and so forth. These special names are injected into the ordinary name type using `SN`. In addition to the names used for the products of Idris's elaborator, `SpecialName` has a constructor `MetaN` with two fields holding names. This constructor is intended to be used by metaprograms when they need

```
data TTName : Type where
  UN : String → TTName
  NS : TTName → List String → TTName
  MN : Int → String → TTName
  SN : SpecialName → TTName
```

Figure 5.4: Reified names.

```
record FunArg where
  constructor MkFunArg
  name    : TTName
  type    : Raw
  plicity : Plicity
  erasure : Erasure
```

Figure 5.5: Reified argument specifiers.

another kind of internal name.

As a part of introducing new metaprogramming features to Idris, we added additional datatypes representing reflections of definitions. These datatypes are described here for the sake of having a single place to look up definitions and details; please refer to Section 9.7 for a discussion of the design.

Because TT does not have features such as implicit arguments and type classes, the TT notion of a dependent function type is insufficient to capture an Idris type declaration. In addition to the elaborated form of the declaration, which is used for type checking, Idris also keeps track of argument metadata. This metadata is represented in the `FunArg` record type, defined in Figure 5.5. The `plicity` field tracks whether the argument is implicit, explicit, or a type class instance, and the `erasure` field tracks whether Idris should issue a warning if it is unable to erase the argument at run time. The term *plicity* is Idris jargon for whether an argument is explicit or implicit. Type declarations for functions are represented by the `TyDecl` record type in Figure 5.6, which provides a function name, an argument list, and a result type. Each argument's type should be understood as being in the scope of the previous arguments, and the result type is in the scope of all of the arguments.

Function definitions do not need any extra metadata about high-level

```
record TyDecl where
  constructor Declare
  name       : TTName
  arguments  : List FunArg
  returnType : Raw
```

Figure 5.6: Reified function declarations.

```
data FunClause : Type where
  MkFunClause : (lhs, rhs : Raw) → FunClause

record FunDefn where
  constructor DefineFun
  name    : TTName
  clauses : List FunClause
```

Figure 5.7: Reified pattern-matching definitions.

Idris, so they are represented as in Figure 5.7 by a pair of a name and a list of function clauses, each of which is simply a pair of a left and right-hand side. The bound pattern variables are represented by `PVar` binders, and the system ensures that they are consistent prior to admitting a new definition by ensuring that the two sides of the clause have convertible types (the type of a `PVar` binding is a `PVTy` binding).

Reflected datatype definitions contain both the kinds of high-level information about arguments that reflected type declarations contain, as well as information that is not readily apparent in Idris datatype declarations but must be discovered by the compiler. Unlike Coq and Agda, Idris does not make a syntactic distinction between parameters and indices to inductive families. Instead, the distinction is discovered by the compiler based on their mode of use in the definition. Thus, reflected definitions tag constructor and type constructor arguments with information about their status as parameters.

```
data TyConArg = TyConParameter FunArg
              | TyConIndex FunArg

data CtorArg = CtorParameter FunArg
             | CtorField FunArg

record Datatype where
  constructor MkDatatype
  familyName : TTName
  tyConArgs : List TyConArg
  tyConRes : Raw
  constructors : List (TTName, List CtorArg, Raw)
```

Figure 5.8: Reified datatype definitions.

Part II

Metaprogramming Idris

Support for metaprogramming, understood to mean programs that generate or modify other programs, is an essential feature of a modern, mature programming language. In particular, compile-time metaprogramming allows generating or modifying programs during the type checking and compilation process, but does not provide facilities for doing so while programs are running. For a programming language with a static type system, this allows the type system to make the same kinds of guarantees about the result of a metaprogram as it would about any other program. Additionally, very expressive type systems can necessitate boilerplate code for explaining to the type checker exactly why a program is well-typed. A metaprogramming system that runs at compile time can eliminate some of the need for this boilerplate.

Background

The design space for a metaprogramming system is exceedingly large. In some cases, the language used to write metaprograms (the *metalanguage*) is the same as the language in which ordinary programs are written (the *object language*). For example, the broad Lisp family, including Racket, Scheme, Clojure, Common Lisp, and Emacs Lisp, provide rich facilities for defining *macros*, which are programs that extend the syntax of the language by translating the new operators to other programs. In Scheme and its descendant Racket, a great deal of work has gone into the concept of *hygiene*, which is enabling a macro system to automatically respect the naming and scoping rules of its programming language. In other Lisps, metaprogrammers must manually manage scope and take care to avoid accidental variable capture, typically using side-effectful operations such as `GENSYM`, which generates a name that is guaranteed to be fresh. Template Haskell provides a computational context, the `Q` monad, in which compile-time effects such as generating fresh names are available, as well as a primitive operator for running a `Q` action during compilation, inserting its result.

Other languages, such as C++ and Coq, have separate languages for programming and metaprogramming. In some cases, such as C++, this has occurred because the support for metaprogramming was an accidental consequence of a language extension that was not intended to be a complete programming language. In others, such as Coq, it is because the metalanguage was intended to be more ergonomic than the object language.

Operations that only make sense while metaprogramming (such as pattern matching on hypotheses in Coq's LTac) can be added to a separate metaprogramming language. Additionally, having a syntactic distinction between phases can make it easier to identify which programs will run in which phase.

However, there are major disadvantages to having separate languages for separate phases: users must learn multiple programming languages, tools such as syntax highlighters and IDEs must account for not only both syntaxes but also the borders between them, and code cannot be re-used between phases. In this part of the dissertation, we describe a collection of metaprogramming subsystems for Idris that allow orthogonal extensions to the language in support of code generation and domain-specific programming. This metaprogramming system has been used to implement the deriving of boilerplate code such as induction principles for datatypes, to automate proofs in the style of Idris's previous tactic language, and to implement domain-specific languages that re-use portions of the Idris elaborator. In some cases, it has replaced code that was previously a part of the compiler, allowing more of Idris to impelemented in itself.

Chapter 6

A Pretty Printer that Says What it Means

This chapter is not based on previously-published papers; however, the contents have been presented at two informal workshops. This is noted in the section corresponding to each of the workshops presentations.

A programming language is more than a mathematical abstraction that maps strings in some formal language to configurations of a Turing machine. A large part of what makes programming languages interesting is that we can implement them, yielding a usable system in which ideas can be explored.

Previously, it was popular to look at an implementation of a programming language as being essentially a batch-mode program that converts input strings to either error messages or executable machine code, through compilation or interpretation. While this model is still sometimes assumed, it is increasingly common to view a language as more than just its syntax and semantics. Today, the entire interactive environment that is available to programmers matters. This approach, pioneered in the Lisp and Smalltalk communities, makes the programming environment into a kind of assistant for the programmer, rather than an adversary whose arbitrary discipline is harshly imposed.

Interactive proof assistants, as one might conclude from the name, have a similarly strong tradition of attempting to assist their users, in this case because of the incredible tedium of trying to write fully-explicit proofs in small formal systems whose derivations are easy to check with a short, understandable program. These have typically followed two broad styles of user interface: a tactic interface in the broad tradition of LCF, or direct constructions of terms or derivations with machine assistance. Tactic languages such as ML and LTac, being programming languages in

their own right, often support interfaces that are not very far divorced from the read-eval-print loops (REPLs) and batch-mode processors of older languages. Languages that assist in the direct construction of proofs (and programs!) in a high-level notation, such as Agda 1, Alf [MN94], Epigram [MM04], and Agda 2 [Agda], provide interfaces based on partial programs that are built interactively by their users.

Environments such as Lisp machines, later stand-alone Lisp implementations, and Smalltalk images have provided excellent access to the context within which a program is developed, including the exploration of the accessible APIs and viewing metadata such as all callers or callees of a given function or method. Their support for the interactive construction of programs has been comparatively limited, often supporting no more than symbol completion — however, this is perhaps to be expected, due to their lack of a static typing discipline to guide the construction. On the other hand, the aforementioned implementations of type theory provide comparatively poor interactive explorations of the environment, while providing excellent assistance for writing programs.

While working on Idris, we have attempted to bridge the gap between Agda-style interactive editing of programs and Lisp-style awareness of a program's environment. However, we have attempted to avoid the trap of requiring a particular text editor as Epigram and (until recently) Agda have done. This chapter describes an implementation strategy for making semantic information from a compiler available to a variety of clients, without preferring one over another. The presented technique is quite straightforward; however, it has enabled a variety of useful features. The technique is by no means specific to implementing dependently typed languages, and should be just as applicable to other language implementations.

6.1 Presentations

The Dynamic Windows user interface toolkit on the Genera operating system for Symbolics Lisp machines supported a user interface convention known as the *presentation* [MYM89]. First defined by Ciccarelli [Cic84], presentations are regions of program output that retain their association with the underlying application object that they signify. For example, a user could enter an expression at a REPL that evaluates to the name of a function, then right-click the returned function name and look up meta-

data about it from the resulting menu. Later, when a command is issued that requires a function name as an argument, the previously-output function names "light up" under the user's mouse pointer, and they can be selected as the argument to the command. Because the interactive Lisp environment relied on pervasive mutability, it is important to note that what was presented was a pointer to an object — two presentations of the same object are aliases, so mutations will affect both presentations.

In Dynamic Windows, presentations were not limited to output at a REPL. McKay et al. [MYM89] give an example of a circuit diagram editing application where each component in the diagram is a presentation of an underlying data model object. The very same components could also be represented as text in a list of components, and the semantic link ensures that any command that applies to one representation also applies to the others. This association between signifier and signified means that the same commands are uniformly available at each representation of the same data, and there is no need for a separate parsing step to re-recognize previously-output information.

The Idris plugin for GNU Emacs[1] supports a form of presentations pervasively throughout the interface. When a term occurs in compiler output, no matter whether it occurs at the REPL, in a proof context, or in an error message, it can be right-clicked, which provides a list of commands that includes normalizing the term, displaying the contents of implicit arguments, and viewing its representation in TT. When names are output by the compiler, they are associated with the underlying name object, enabling contextual access to commands such as looking up documentation and viewing definitions, without a risk of confusion due to overloading.

Figures 6.1 through 6.4 demonstrate the kinds of interaction enabled by this semantic association. In Figure 6.1, a user has attempted to apply an evaluator that only works on closed terms to an open term. In the code, `Program` is an alias for the type `Expr 0`. The resulting error message points out the specific unification failure and displays the types that occurred in the users's code. However, the connection between `Program` and `Expr` is not particularly clear from the error message. While this could be avoided by normalizing all terms that occur in error messages, that would cause aliases written by users to be far less useful. Likewise, normalized terms can be vastly larger than the terms that actually occur in programs. In

[1]Available from `https://www.github.com/idris-hackers/idris-mode` at the time of writing

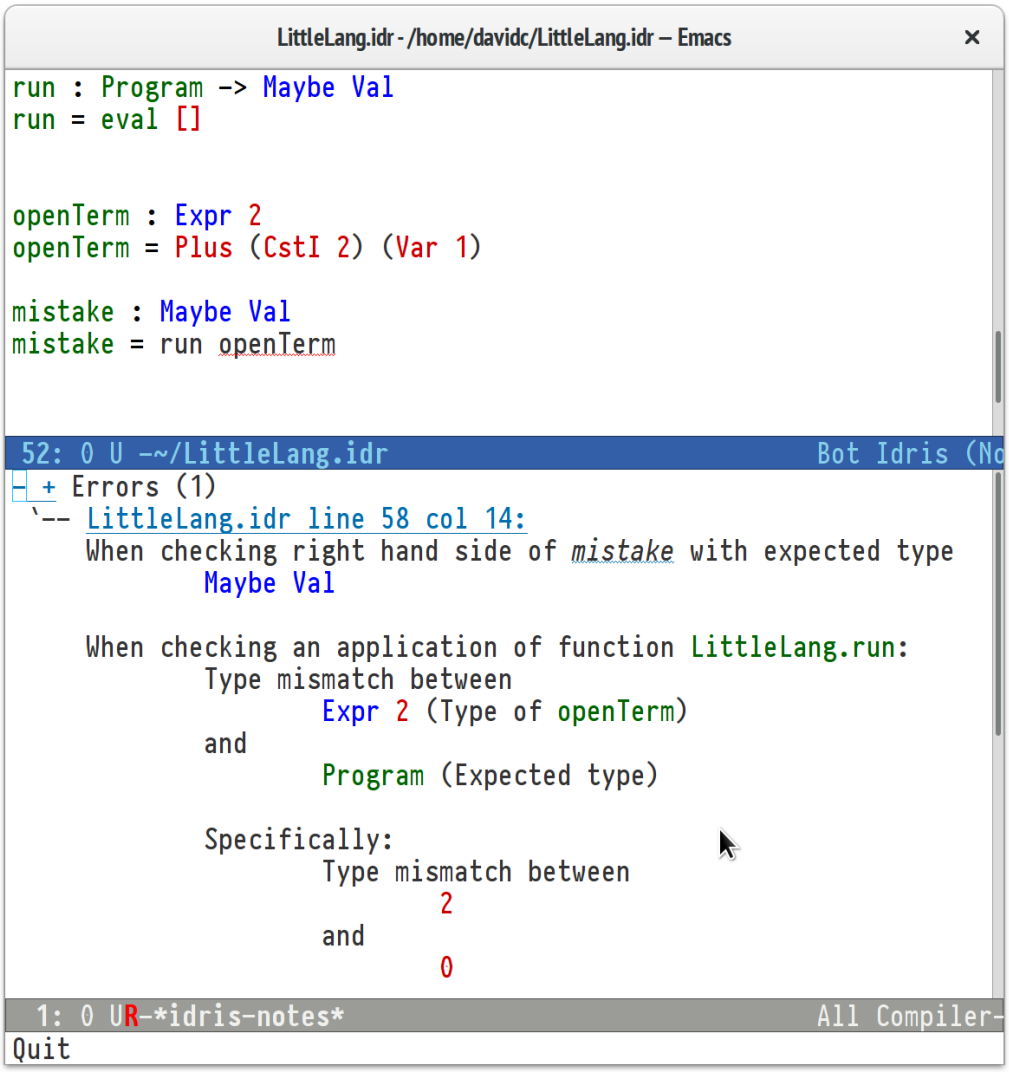

Figure 6.1: An Idris type error. The user may not know where the 0 and 2 come from, nor what Program refers to.

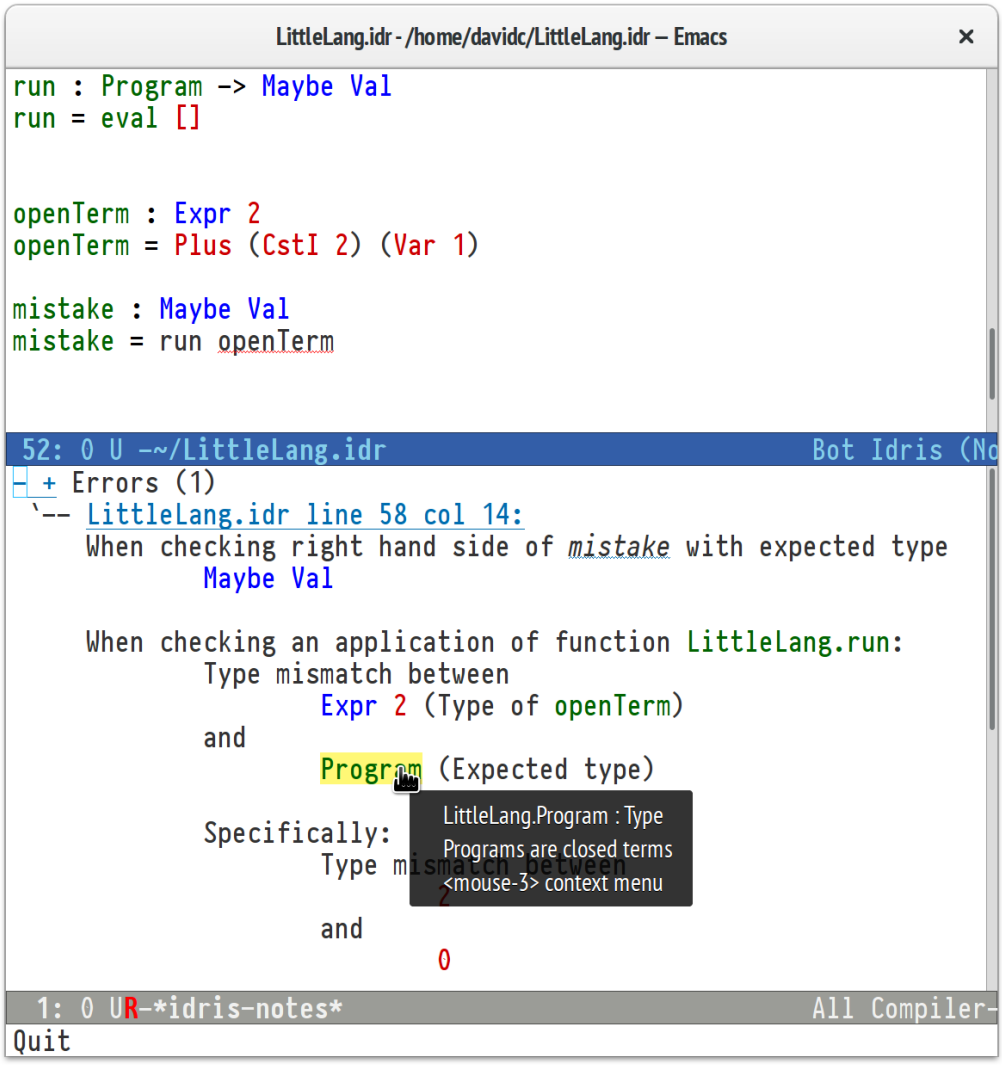

Figure 6.2: Hovering over `Program` indicates that a menu is available and provides information about the name under the pointer.

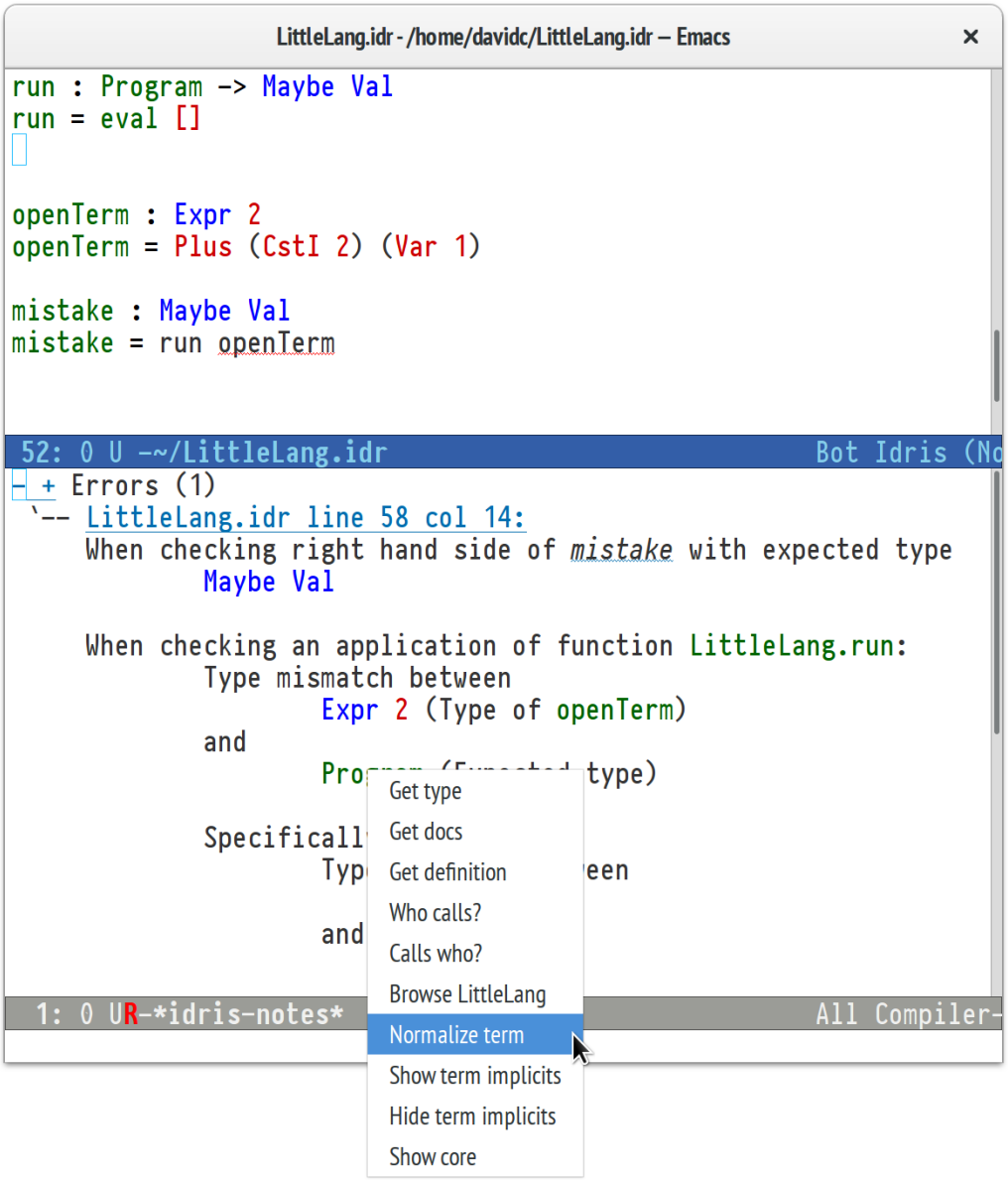

Figure 6.3: Right-clicking `Program` displays the available commands.

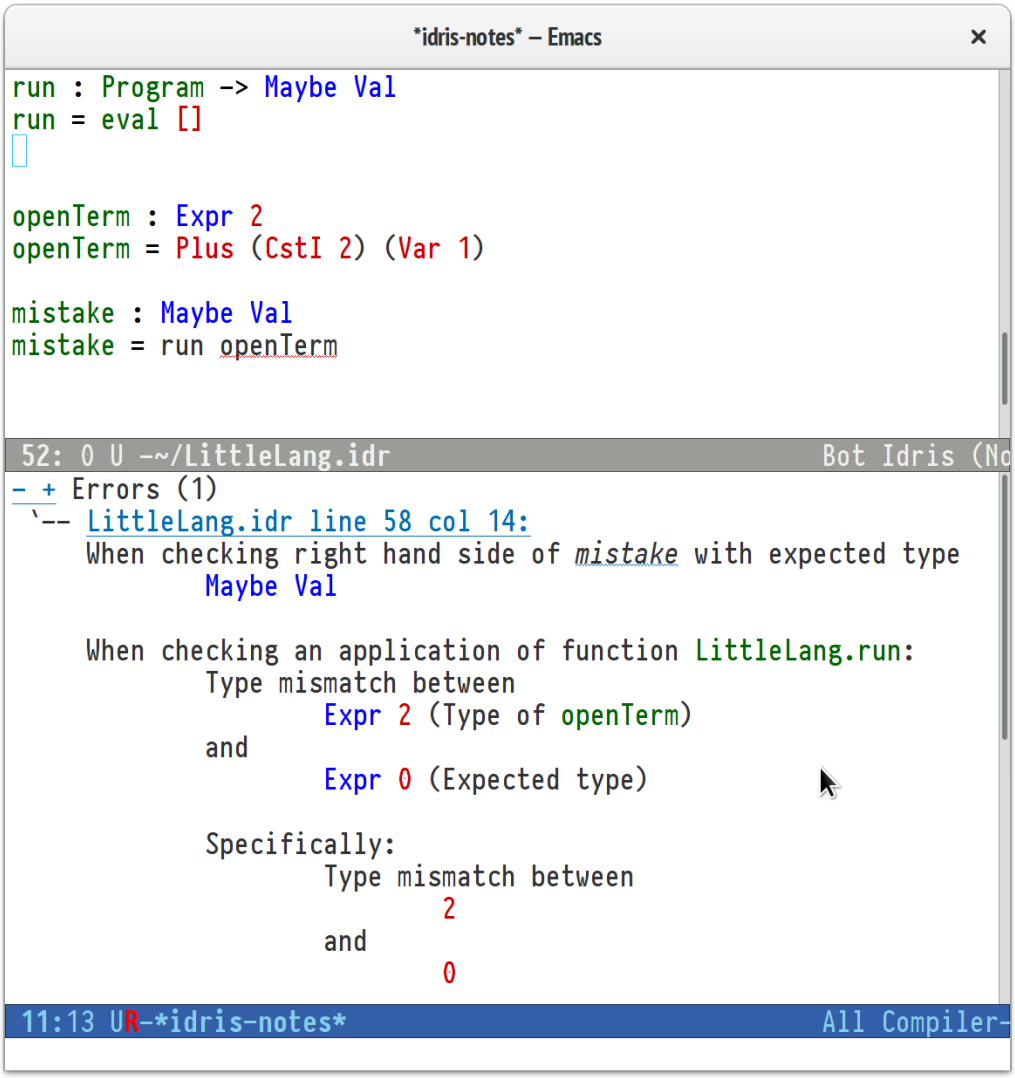

Figure 6.4: Selecting “Normalize term” makes the source of 0 clear.

our interface for Idris in Emacs, the user has more possibilities. Figure 6.2 shows what happens when the user points her mouse at `Program`: Emacs displays a tooltip with the documentation and type signature for `Program` and the term lights up, indicating the presence of a menu. This menu contains a number of commands, some of which pertain to the name `Program` and some of which pertain to the term consisting of a reference to it. In Figure 6.3, the user selects "Normalize term". The result of this can be seen in Figure 6.4, where `Program` has been replaced by its definition.

There is nothing Emacs-specific about this means of interacting with Idris: any user interface toolkit that can be programmed to remember an association between a region of the screen on which output has been performed and an opaque token that represents Idris's view of the semantics can be used. The remainder of this chapter describes an extension to the standard methods of building pretty printers that enables them to be used as part of a presentation-style interface.

6.2 The Idris IDE Protocol

This section is based on joint work with Hannes Mehnert and has been previously presented as a demonstration at the 2014 Workshop on Dependently Typed Programming in Vienna, Austria.

The overall design of the Idris IDE for Emacs is based on the SLIME[2] and DIME[3] environments for Common Lisp and Dylan, respectively. Instead of being an integrated, monolithic system like the Lisp machines or like Smalltalk environments, these environments use a client-server architecture to enable interfaces such as Emacs to have the same high degree of access to the compiler. In particular, editor commands are performed using remote procedure calls over a socket.

The Idris IDE protocol continues this tradition. Commands for looking up type information, submitting user programs to the type checker, and editing programs with assistance from the compiler are encoded in a machine-accessible format. In some sense, the REPL has been given a machine-readable syntax in addition to its human-readable syntax, which enables machines to assist humans in constructing appropriate requests and interpreting the response.

[2] Available from `https://common-lisp.net/project/slime` at the time of writing
[3] Available from `https://github.com/dylan-lang/dylan-mode` at the time of writing

To achieve the colors and easy accessibility of commands in Figures 6.1 through 6.4, output from the compiler is decorated with metadata describing its semantics prior to sending it to the text editor. Each string sent from the compiler to the text editor is accompanied by a collection of offset-length-properties triples. The properties applied to names include their unique, fully-qualified representations, a summary of their documentation, their type signatures, and whether the represent bound variables, functions, data constructors, or type constructors. Constants are highlighted with information including their type, alternative representations (e.g. hexadecimal notation for decimal machine integers), and computed metadata such as the length of strings in characters. Whole terms are annotated with a unique identifier for this term that can be used to query the Idris compiler for more information about it. Because Idris terms are immutable, there are no concerns about aliasing, so the unique identifier can be a serialized binary representation of the term and its environment, freeing the compiler from remembering all of the previously-displayed terms.

6.3 Annotated Pretty Printing

This section was previously presented as a talk at the Haskell Implementers' Workshop 2015 in Vancouver, British Columbia, Canada.

A key component of the user interface of most programming language implementations is the pretty printer, which is responsible for converting from internal tree-structured data to a textual representation that is suitable to show to users. If there should be any hope of adding presentations to a user interface, it is vital that the pretty printer can "say what it means": it must record the *semantics* of the sub-regions of the resulting strings, and communicate this link to the user interface.

Pretty printers that are written using combinator libraries, such as those of Hughes [Hug95] and Wadler [Wad03], are one of the successes of that approach to library design. In these libraries, there is a datatype `Doc` of *documents*, which are abstract representations of sets of concrete strings. A rendering process selects which of these strings is best for some particular context, taking into account how many columns are available for display. The library provides a collection of combinators for constructing a `Doc` that express features such as concatenation of documents, optional line breaks, and including a string in a document.

The interfaces to Hughes's and Wadler's libraries are not identical. In addition to differences in interface, Wadler's approach requires a rendering step prior to outputting the document as a string. Because the technique described in this section is applicable to either style of pretty printer, it will be presented in a slightly idealized manner. The additions to the pretty printer API are orthogonal to the differences between the two approaches.

The first step in adding semantic information to a pretty printer document is to represent it as a datatype. To avoid committing to a datatype that might be inappropriate for some uses, the Idris pretty printer accomplishes this by abstracting the type of pretty-printer documents over a type of semantic *annotations*. In other words, the `Doc` type should take a parameter, and elements of `Doc a` are documents that signify an `a`.

The second step is to add a new primitive operator to the pretty printing library: `annotate :: a → Doc a → Doc a`. The `annotate` operator adds an annotation to a document, which may later become a subdocument of another document.

Finally, the rendering process must be extended to enable the conveyance of the annotations onward to the user interface. However, this process will vary depending on exactly which user interface is to be used. For example, if the annotations are to be used to compute colors on a Unix console, then the strings that result from annotated subdocuments must be decorated with ANSI color escape codes. HTML and LaTeX should be decorated similarly, except the contents of strings must be escaped. If they are to be used to produce colored output on Microsoft Windows, then the display process must run inside the `IO` monad so that side effects can be used to change colors. If the annotations are to be sent over Idris's IDE protocol, then the string must be rendered as if there were no annotations, but the annotations must be collected along with the regions in the string that they correspond to.

There is a single rendering interface that is sufficient for all of the above as well as being convenient to use. Its signature is:

```
displayDecorated :: (Applicative f, Monoid o)
                 ⇒ (a → f o) → (a → f o)
                 → (String → f o)
                 → Doc a
                 → f o
```

It takes three type parameters: `a`, which is the type of annotations, `f`, which

represents the effects that can be used to display the data, and `o`, which is the type of intermediate results. The parameter `f` is an applicative functor, which is an abstraction that is weaker than a monad, described by McBride and Paterson [MP08]. The `Applicative` class supports sequencing of effects from left to right, rather than the arbitrary order of effects given by `Monad`, and this is sufficient to render a document. The result type must be a `Monoid`, because the rendering process needs a way to represent empty documents and a means of concatenating results. The first two arguments are instructions for beginning and ending annotated subdocuments. The third argument declares how to display an atomic string. The fourth argument is the document to render.

To recover a pretty printer that ignores annotations, we invoke `displayDecorated` as follows:

```
display :: Doc a → String
display = runIdentity . displayDecorated doNothing doNothing pure
    where doNothing = pure ""
```

In this invocation of `displayDecorated`, `f` is the trivial functor `Identity` and `o` is `String`, and the `Monoid` instance for `String` is used to concatenate them. It would be possible to use a writer monad for `f` and set `o` to `()`; however, such an interface would be far less convenient to use. To output colored strings on Windows, we invoke `displayDecorated` to use side effects:

```
display :: Doc Annot → IO ()
display = displayDecorated start end putStr
```

Here, `start` and `end` make the appropriate API calls to set colors based on names. If we wanted to additionally highlight overlapping regions, some state would be necessary as well. The `putStr` action is used to write strings to standard output. In this case, `f` is `IO` and `o` is the trivial monoid for `()`, because we are not accumulating a result.

Sometimes, however, neither the computational context `f` nor the result `o` should be trivial. The function that computes both a string and a collection of offset-length-annotation triples for Idris's IDE protocol needs to use a state monad to keep track of the position in the string (via tracking the lengths of output strings), the current stack of open annotated regions, and the thus-far-collected closed annotation regions. Thus, it can use `State (Int, [(a, Int)], [(Int, Int, a)])` as `f` and `String` as `o`.

The type constructor `Doc` is a functor. This means that it is possible to transform annotations using `fmap`, which means that we need not have

all the information that we'd like to include in a display when a pretty-printer is executed. Prior to rendering a document for output, `fmap` can be used to populate the annotations with information from the global context, such as documentation and type signatures. It can also be used to combine annotated documents produced in different parts of a compiler whose notions of annotations do not need to agree.

6.4 Conclusions

Annotated pretty printing allows pretty printers to say what they mean by retaining enough of a compiler's internal representation of data to provide an unambiguous means of indicating to the compiler which data were intended. Just as pretty printing libraries allow code that generates strings to be blissfully unaware of details like the width of the context in which the string will be displayed, annotated pretty printing allows the code that generates the strings to be unaware of how metadata will be displayed and take a declarative approach.

By annotating each document with the object used to produce it, an annotated pretty printer enables the implementation of presentations, even in editors that only have a text-driven interface to the underlying compiler. This enables an exploratory style of user interface in which terms in error messages, proof contexts, and other output from the compiler can be directly manipulated. These interactive interfaces are not tied to a specific frontend, and can be implemented in a variety of editors or other tools.

Chapter 7

Quasiquotation

> *"[P]rograms must be written for people to read, and only incidentally for machines to execute."*
>
> Abelson and Sussman, *Structure and Interpretation of Computer Programs*

This chapter is an expanded and revised version of a paper presented at the 26th Symposium on Implementation and Application of Functional Languages (IFL 2014) [Chr14b]. Since the publication of that paper, we have extended Idris quasiquotations to be polymorphic, able to generate both the `Raw` and `TT` representations of TT that are presented in Chapter 5. This new extension is described in Section 7.4.

The metaprogramming features described in this dissertation allow Idris to be used to implement extensions to itself. However, because all type checking and evaluation occur using the core language TT, metaprograms must be able to both construct and destruct TT terms. Without special support from the compiler, this process is error-prone and incredibly tedious — even very simple Idris terms can expand to quite complicated terms in TT. Furthermore, the Idris elaborator makes use of two different representations of TT for different purposes (see Chapter 5 for details). Keeping track of which representation is being used, and how that particular representation is used for some term of interest, diverts attention from the actual work that the user is trying to accomplish.

Even worse, the correspondence between high-level Idris terms and their TT equivalents is not always obvious, even to expert users of the language. If one believes that programs exist first and foremost as a means

of communication between humans, and thereafter for machines, then another means of working is desirable. Luckily, Idris already contains an elaborator that can transform high-level Idris into TT.

This chapter describes a technique for augmenting the high-level Idris language with *quasiquotations*, in which the Idris elaborator is invoked to transform high-level Idris into reflected TT terms using the very same translation that produces TT terms for the type checker. Within quasiquoted terms, *antiquotations* allow other reflected terms to be spliced into the quotation. In a pattern context, antiquotations become patterns to be matched by the reflected term at the corresponding position. These quasiquotations allow the best of both worlds: high-level syntax for the uninteresting parts, with details filled in by type-directed elaboration, but with full control over the details of term construction when and if it matters.

7.1 Example

To illustrate the difference in verbosity and complexity between a term in the high-level Idris language and TT, it is sufficient to compare the natural number 1, expressed in the typical Peano encoding, in each notation. In the high-level Idris language, it is represented as `S Z`, the application of the successor operation (named `S`) to zero (named `Z`). The representation of this term in the `TT` datatype can be seen in Figure 7.1. Please refer to Chapter 5 for the meanings of the constructors of `TT` and `TTName`.

Correctly constructing these reflected terms can be tedious. Additionally, one must be careful to encode precisely the right details when pattern-matching on reflected terms. In the above term, the type annotation on `S` includes a machine-generated name, because the constructor's type `Nat → Nat` is a special case of the dependent type `(x : Nat) → Nat`. The name `x` is constructed by the implementation during elaboration and is not predictable. In other cases, however, the particular name in a binding may be important. Other details that most pattern matches should ignore include tag values and universe level indicators. We expect that the type annotation on the `P` constructor will typically be irrelevant, though some metaprograms or proof search procedures may be able use them to avoid repeated lookups in the global context.

The example function `isZeroR` returns `Just True` when its argument is reflection of `Z`, `Just False` when its argument is a reflection of an applica-

```
App (P (DCon 1 1)
       (NS (UN "S") ["Nat", "Prelude"])
       (Bind (MN 0 "_t")
             (Pi (P (TCon 0 0)
                    (NS (UN "Nat")
                        ["Nat", "Prelude"])
                    Erased)
                 (TType (UVar -1)))
             (P (TCon 0 0)
                (NS (UN "Nat") ["Nat", "Prelude"])
                Erased)))
    (P (DCon 0 0)
       (NS (UN "Z") ["Nat", "Prelude"])
       (P (TCon 0 0)
          (NS (UN "Nat") ["Nat", "Prelude"])
          Erased))
```

Figure 7.1: `S Z`, represented in the `TT` datatype.

tion of `S` to any other term, and `Nothing` when it is any other term. Such a function might be useful in a proof tactic. Even this simple function is quite verbose:

```
isZeroR : TT → Maybe Bool
isZeroR (P _
           (NS (UN "Z") ["Nat", "Prelude"])
           _)   = Just True
isZeroR (App (P _
                (NS (UN "S") ["Nat", "Prelude"])
                _)
             n) = Just False
isZeroR _       = Nothing
```

Compare this to its equivalent for non-reflected natural numbers:

```
isZero : Nat → Bool
isZero Z     = True
isZero (S n) = False
```

This version can return `Bool` instead of `Maybe Bool` because the type system guarantees that it will never be called with a non-`Nat` argument. How-

ever, the largest decrease in complexity comes from using Idris's high-level notation to define the patterns, rather than a datatype representing core terms.

In contrast to the somewhat verbose definition above, the quasiquotation mechanism described by this paper allows a definition of `isZeroR` that is much more like the non-reflected version:

```
isZeroR : TT → Maybe Bool
isZeroR `(Z)    = Just True
isZeroR `(S ~n) = Just False
isZeroR _       = Nothing
```

The quotations, indicated by the backquote characters, cause the elaborator to produce patterns that are precisely equivalent to those in the original definition of `isZeroR`. The antiquotation of `n`, indicated by preceding `n` with a tilde, causes the elaborator to treat the expression `n` normally; that is, `n` becomes an ordinary pattern variable.

7.2 Idris Quasiquotations

Our quasiquotations extend both the high-level Idris language and its desugared form Idris^{-}. We extend the expression language with three new productions:

$$
\begin{array}{rll}
e, t ::= & \ldots & \\
 & \mid \ \texttt{`(}\ e\ \texttt{)} & \text{(quasiquotation of } e\text{)} \\
 & \mid \ \texttt{`(}\ e\ \texttt{:}\ t\ \texttt{)} & \text{(quasiquotation of } e \text{ with type } t\text{)} \\
 & \mid \ \texttt{\~{}}\ e & \text{(antiquotation of } e\text{)}
\end{array}
$$

The parts of a term between a quotation but not within an antiquotation are said to be *quoted*. Every antiquotation must have a corresponding enclosing quotation; that is, it is an error if the depth of nesting of antiquotations exceeds the depth of nesting of their enclosing quotations. The quoted regions of a term are elaborated in the same way as any other Idris expression. However, instead of being used directly, the elaborated TT terms are first reified, and this representation is then used. Antiquoted regions are elaborated directly into the reified terms, which are inserted as usual.

Names occurring in the quoted portion of a term do not obey the typical lexical scoping rules of names in Idris. This is because quoted terms

are intended to be used in places other than where they are constructed, and the location in the program where they are spliced may have completely different bindings for the same names. Therefore, all free names in the quoted portion are taken to refer to the global scope. Because antiquotations are ordinary terms, they obey the ordinary scoping rules of the language.

Idris supports type-driven disambiguation of overloaded names. This feature is used for everything from literal syntax for number- and list-like structures to providing consistent naming across related libraries. This is also used to allow "punning" between some types and their constructors. For instance, the syntax `()` represents both the unit type and its constructor in Idris, and `(Int, String)` can represent either a pair type or a pair of types. In ordinary Idris programs, all top-level definitions are required to have type annotations, so type information is available to aid in disambiguation. Because of this, Idris's expression language does not include type annotations on arbitrary subterms. In quasiquoted terms, however, no top-level type annotation is available. Thus, the second variant of quasiquotation above allows an explicit *goal type* to be provided. Like a quoted term, it is elaborated in the global environment. Because the goal type does not occur in the final reflected term and simply exists as a shorthand to avoid explicitly annotating names, goal types may not contain antiquotations.

7.3 Elaborating Quasiquotations

We now describe quasiquote elaboration under the assumption that all quasiquotations will be elaborated to the datatype `TT`. In Section 7.4, the technique is extended so that quotations can produce both `TT` and `Raw`.

In addition to the elaborator tactics already described in Chapter 4, this section makes use of four new meta-operations:

- ANYTHING, which introduces a hole whose type must be inferred;
- EXTRACTANTIQUOTES, which replaces antiquotations in a quasiquoted Idris⁻ term with references to fresh names, returning both the modified term and the mapping from these fresh names to their corresponding antiquotation terms;
- REIFY, which returns a term corresponding to the reification of its argument; and

- reifyP, which returns a pattern corresponding to the reification of its argument.

The operation anything n can be defined as follows:

$$\begin{aligned} \textsc{anything}\ n = \mathsf{do}\ \ & \textsc{claim}\ n' : \star \\ & \textsc{claim}\ n : n' \end{aligned}$$

This operator serves a different purpose than the Infer type described in Chapter 4: while Infer is intended to serve as the goal type of an elaboration process, in which both the resulting term and its type are desired as output, anything is intended for situations in the middle of elaboration where only the resulting term is of interest.

The operation extractAntiquotes is a straightforward traversal of an Idris$^-$ term, replacing antiquotations with variables and accumulating a mapping from these fresh variables to the corresponding replaced subterms.

The operators reify and reifyP each take a term and a collection of names of antiquotations (see Section 7.3.1) and return a quoted version of the term. Antiquotation names, however, are not quoted. Additionally, reifyP inserts universal patterns in certain cases — see Section 7.3.3 for details.

7.3.1 Elaboration Procedure

We implement quasiquotations by extending the elaboration procedures for expressions and patterns: $\mathcal{E}[\![\cdot]\!]$ and $\mathcal{P}[\![\cdot]\!]$ respectively. Elaborating the quoted term proceeds through four steps, each of which is described in detail below:

1. Replace all antiquotations by fresh variables, keeping track of the antiquoted terms and their assigned names
2. Elaborate the resulting term in a fresh proof state, to avoid variable capture
3. Quote the elaborated term:
   a) When not in a pattern, quote the term, leaving antiquotation variables free
   b) When in a pattern, quote the term with additional universal patterns

4. Restore the local environment and elaborate antiquotations

Replace antiquotations We replace antiquotations with fresh variables because they will need to be treated differently than the rest of the term. Additionally, the expected types of the antiquotations must be inferable from the context in which they are found, because the quotations that will fill them provide no type information. Here, variables serve their typical function: they *abstract* over the antiquoted subterms, because the term that will be constructed to fill an antiquotation at run time is unknown at elaboration time. We remember the association between the antiquoted terms and the names that they were replaced by so that the result of elaborating them can later be inserted.

Elaborate in a fresh proof state Quotations can occur in any Idris expression. However, names that occur in quotations are resolved in the global scope, for reasons discussed in Section 7.2. Because the scopes of local variables are propagated using hole contexts in the proof state, it is sufficient to elaborate the quoted term in a fresh state. The replacement of antiquotations with references to fresh names means that there is no risk of elaborating the contents of the antiquotations too early. However, when the elaborator reaches these names, it will fail, because they are unknown. To fix this problem, we first use the ANYTHING meta-operation that was defined above to introduce holes for both these names and their types. Because this stage of elaboration occurs in term mode, rather than pattern mode, the elaboration will fail if the holes containing types don't get solved through unification.

Quote the term Quotation is the first step that differs between terms and patterns. In both cases, the term resulting from elaboration is quoted, with the names that were assigned to antiquotations left unquoted. However, if the term being elaborated is a pattern, then some aspects of the term are not quoted faithfully. See Section 7.3.3 for more information.

Elaborate the antiquotations The quoted term from the previous step is ready to be spliced into the original hole. What remains is to solve the variables introduced for antiquotations in the previous step. This is done by first introducing each name as a hole expecting a quoted term, and then elaborating them into their holes using the standard elaborator $\mathcal{E}\,[\![\cdot]\!]$.

$$
\begin{array}{lll}
\mathcal{E}\,[\![\texttt{`(}\ e\ \texttt{)}]\!] = \textsf{do} & (e', \vec{a}) \leftarrow \textsc{extractAntiquotes}\ e & (1) \\
& st \leftarrow \textsc{get} & (2) \\
& \textsc{newProof}\ T & \\
& \textsc{claim}\ T\ :\ \star & \\
& \textsc{anything}\ (names\ \vec{a}) & \\
& \mathcal{E}\,[\![e']\!] & \\
& qt \leftarrow \textsc{term} & \\
& \textsc{check}\ qt & \\
& \textsc{put}\ st & \\
& \textsc{claim}\ \left(names\ \vec{a}\ :\ \vec{\mathrm{T}}\mathrm{T}\right) & (3a) \\
& r \leftarrow \textsc{reify}\ qt\ \vec{a} & \\
& \textsc{fill}\ r & \\
& \textsc{solve} & \\
& \textsc{elabAntiquote}\ \vec{a} & (4)
\end{array}
$$

Figure 7.2: Elaboration of quasiquotations.

Formal Description

This four-step elaboration procedure is described in Figure 7.2, in Brady's notation [Bra13b] as described in Chapter 4. The individual tactics that correspond to each of the steps 1–4 above are numbered. Antiquotations are replaced in the first line of the tactic script, using the previously-described operation ExtractAntiquotes (1). Then, the ordinary state monad operations get and put are used to save and restore the original proof state. The region (2) bracketed by these operations corresponds to step 2 above — namely, elaboration of the quoted term in the global context, which is achieved using a fresh proof state introduced by newProof. Initially, the goal of the new proof is an unbound variable, but this variable is then bound as a hole expecting a type using the claim meta-operation. The quoted term is provided with hole bindings for each of the fresh antiquotation names by the anything meta-operation. Then, the quoted term is elaborated into the main hole. If this process is successful, it will result in the hole T being filled out with a concrete type as well. The result of elaboration is saved in the variable qt, and then type checked one final time with Check to ensure that no errors occurred.

After the original proof state is restored with put, the actual quoting must be performed and the antiquotations must be spliced into the re-

sult (3). Each antiquotation name is now established as a hole of type `Term`, the datatype representing reflected terms, because the elaborated form must be a quotation. Now that the holes for the antiquotations are established, it is possible to insert the reflected term into the initial hole. The operation REIFY is invoked, which quotes the term, leaving references to the antiquotation variables intact as references to the just-introduced holes. This quoted term is then filled in as a guess, and SOLVE is used to dispatch the proof obligation.

Finally, the antiquotations can be elaborated (4). This is done by focusing on their holes and elaborating the corresponding term into that hole. In the above script, this is represented by the tactic ELABANTIQUOTE, which can be defined as follows:

$$\begin{array}{lcll} \textsc{elabAntiquote}\ (n, t) & = & \mathsf{do} & \textsc{focus}\ n \\ & & & \mathcal{E}\,[\![t]\!] \end{array}$$

A specific elaboration procedure for antiquotations is not necessary, because programs with antiquotations outside of quasiquotations are rejected prior to elaboration.

7.3.2 Elaborating With Goal Types

Elaborating a quasiquotation with an explicit goal type is a straightforward extension of the procedure in the previous section. After introducing a hole for the type of the term that will be elaborated prior to the actual quotation, the goal type is elaborated into this hole. Because this is occurring immediately after the establishment of a fresh proof state, names in the goal type will be resolved in the global scope, as intended.

The formal procedure is largely identical to that shown in Figure 7.2, with only the small addition shown in Figure 7.3. Thus, the lines immediately before and immediately after are included to show where the additions have occurred. This seemingly-simple change has far-reaching effects, because type information is now available to the subsequent elaboration of e'. This type information can, for instance, enable implicit arguments to be solved due to unification constraints induced by the elaboration of t.

7.3.3 Elaborating Quasiquotation Patterns

Quasiquotations can also be used as patterns. Recall that the operation $\mathcal{P}\,[\![\cdot]\!]$ is a variation of $\mathcal{E}\,[\![\cdot]\!]$ that is used on the left-hand side of definitions

$$
\begin{aligned}
\mathcal{E}\,[\![\texttt{`(}\, e \;:\; t \,\texttt{)}]\!] \;=\; \texttt{do}\;\; & \\
& \vdots \\
& \textsc{claim}\ T \;:\; \star \\
& \textsc{focus}\ T \\
& \mathcal{E}\,[\![t]\!] \\
& \textsc{anything}\ (\overrightarrow{\textit{names}\ \vec{a}}) \\
& \vdots
\end{aligned}
$$

Figure 7.3: Elaborating quasiquotations with goal types (new steps highlighted).

in order to elaborate patterns. The primary difference is that $\mathcal{P}\,[\![\cdot]\!]$ does not fail when the elaborated term contains unknown variables. Instead, it inserts pattern variable bindings for these.

It is tempting, then, to simply use the pattern elaborator in the recursive elaboration clauses of the quasiquote elaboration procedures. However, this would not work. REIFY would simply quote these new pattern variables, leading to terms that contain explicitly quoted fresh pattern variables. Pattern elaboration must instead invoke ordinary expression elaboration when generating the term to be quoted, but then use pattern elaboration for the antiquotations.

For practical reasons, pattern elaboration must use a specialized reflection procedure REIFYP that introduces some universal patterns in strategic places. These universal patterns serve two purposes: preventing unnecessary pattern-matching of subterms that are uniquely determined by other subterms, and preventing elaborator-chosen features such as hidden names from making patterns too specific. In Idris, it is especially important to reduce the size of the subterms being scrutinized when possible, because the coverage checker can take significant time when checking deeply nested patterns. In particular, the constructor for references to global names contains three subterms:

- whether the name is a function, constructor or type constructor;
- the fully-qualified name; and
- a full type annotation.

The first and last of these subterms are, however, uniquely determined by the second, and they exist to simplify the type checker. Thus, when

$$
\begin{array}{lll}
\mathcal{E}\,[\![\texttt{`(}\ e\ \texttt{)}]\!] \;=\; \texttt{do} & (e', \vec{a}) \leftarrow \textsc{extractAntiquotes}\ e & (1) \\
& st \leftarrow \textsc{get} & (2) \\
& \textsc{newProof}\ T & \\
& \textsc{claim}\ T\ :\ \star & \\
& \textsc{anything}\ (\mathit{names}\ \vec{a}) & \\
& \mathcal{E}\,[\![e']\!] & \\
& qt \leftarrow \textsc{term} & \\
& \textsc{check}\ qt & \\
& \textsc{put}\ st & \\
& \textsc{claim}\ \left(\mathit{names}\ \vec{a}\ :\ \vec{\mathrm{TT}}\right) & (3b) \\
& r \leftarrow \textsc{reifyP}\ qt\ \vec{a} & \\
& \textsc{fill}\ r & \\
& \textsc{solve} & \\
& \textsc{elabAntiquoteP}\ \vec{a} & (4)
\end{array}
$$

Figure 7.4: Elaborating quasiquote patterns (changes highlighted).

pattern matching, there is no need to check them. Additionally, the elaborator will from time to time invent a fresh name or universe variable. For example, ordinary non-dependent function types are represented in TT as dependent functions types in which the bound name is not free in the type on the right hand side. In these cases, it does not actually matter which name was chosen, because the name does not appear in the term, and matching against the specific name chosen by the elaborator could mean that the pattern `` `(Nat → Nat) `` did not match the quoted input term `` `(Nat → Nat) ``.

There is no solid theoretical reason for the selection of these particular heuristics. However, they do work well in practice, and users who want to control the details of pattern matching can always override these defaults with an explicit antiquotation. For instance, one could use the pattern `` `(Nat → ~(Bind (UN "x") (Pi `(Nat) _) `(Nat))) `` to match all non-dependent functions in `Nat → Nat → Nat` whose second argument happens to be named precisely `x`.

Figure 7.4 demonstrates the formal procedure for elaboration of quasiquotation patterns. This procedure uses two variations on previously-seen meta-operations: reifyP, like reify, is a traversal of the resulting tree structure that implements step 4 above, and elabAntiquoteP is defined

as follows:

$$\textsc{elabAntiquoteP}\ (n, t) \quad = \quad \texttt{do} \quad \textsc{focus}\ n \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathcal{P}\,[\![t]\!]$$

The modifications necessary to elaborate a quasiquotation pattern with a goal type are identical to the non-pattern case.

7.3.4 Nested Quasiquotations

While nested quasiquotations are a useful idiom in Lisp macro programming, it is unclear to what extent they are useful in the context of metaprogramming Idris. It is not particularly common to build a complex infrastructure around the reflected term datatype itself, as reflection is primarily used to escape the confines of the type theory. However, in the interest of not introducing arbitrary restrictions, the elaboration procedure described in this section can be straightforwardly extended to support nested quasiquotations.

Only one modification is needed: the extractAntiquotes operation needs to keep track of the current quotation level. Crossing a quotation increments the quotation level, and crossing an antiquotation decrements it. Only antiquotations corresponding to the outermost quotation, i.e., only antiquotations at quotation level zero, are extracted. The remainder of the elaboration procedure is unchanged.

In the real implementation, of course, quasiquote elaboration with or without goal types and in pattern mode or expression mode is handled by one code path, with conditionals expressing the four possibilities. They were presented as four separate procedures above for reasons of clarity.

7.4 Polymorphic Quotations

In the past, the `Raw` datatype was not particularly useful for reflection in Idris. Since the paper upon which this chapter is based was written, specifically with the advent of the reflected elaborator (see Chapter 9), `Raw` has become much more prominent. Thus, it became necessary to extend quasiquotation to work for both `TT` and `Raw`. The procedure outlined in this chapter was straightforwardly adapted by introducing additional reflection and pattern reflection operations.

Figure 7.5 exhibits the elaboration procedure for polymorphic quotations; the pattern elaborator and goal types are added in the same manner

$$
\begin{array}{lll}
\mathcal{E}[\![\texttt{`(}\ e\ \texttt{)}]\!] = \textsf{do} & g \leftarrow \textsc{goal} & \\
& (e', \vec{a}) \leftarrow \textsc{extractAntiquotes}\ e & (1) \\
& st \leftarrow \textsc{get} & (2) \\
& \textsc{newProof}\ T & \\
& \textsc{claim}\ T\ :\ \star & \\
& \vec{\textsc{anything}}\ (names\ \vec{a}) & \\
& \mathcal{E}[\![e']\!] & \\
& qt \leftarrow \textsc{term} & \\
& \textsc{check}\ qt & \\
& \textsc{put}\ st & \\
& \vec{\textsc{claim}}\ \left(names\ \vec{a}\ :\ g\right) & (3a) \\
& r \leftarrow \textsc{try}(\textsc{reify}_{\texttt{TT}}\ qt\ \vec{a}) & \\
& \qquad\qquad (\textsc{reify}_{\texttt{Raw}}\ qt\ \vec{a}) & \\
& \textsc{fill}\ r & \\
& \textsc{solve} & \\
& \vec{\textsc{elabAntiquote}}\ \vec{a} & (4)
\end{array}
$$

Figure 7.5: Polymorphic quasiquotation (changes highlighted).

as they were in monomorphic quasiquotations. At the beginning of the quotation process, the elaboration goal type is remembered. Then, when holes are established for the fresh names that were inserted, the elaboration goal type is used for the type of the hole. Finally, the two quotation procedures are applied wrapped in the TRY tactical. Here, $\textsc{reify}_t$ represents a function that reflects its argument into term representation t. This assures that the first success will be used, which means that quotations will default to `TT` when no type constraints are available. To avoid defaulting, it would also be possible to explicitly check that the goal type was either `TT` or `Raw`, and employ the appropriate reflection procedure.

7.5 Quoted Names

The contents of this section are new developments since the IFL 2014 paper.

The complex structure of the name datatype in Idris has many of the same disadvantages as working with the `TT` or `Raw` datatypes. Names are

difficult to recognize when reading code. Additionally, because reflected names are not required to point at anything in particular, it is easy to forget to modify a metaprogram when the namespace structure of a project changes. Just as it is useful for reflected terms to be well-typed and to share the high-level syntax of Idris, it is useful for reflected names to be valid references to existing names and to share the syntax of Idris names.

Thus, we extended Idris with two additional syntactic forms, inspired by Agda's **quote** syntax:

$$
\begin{array}{rll}
e, t ::= & \ldots & \\
 & \mid \texttt{`\{}n\texttt{\}} & \text{(resolved quotation of name } n\text{)} \\
 & \mid \texttt{`\{\{}n\texttt{\}\}} & \text{(unresolved quotation of name } n\text{)}
\end{array}
$$

The first, a quotation of the name n, resolves n in the current scope during elaboration and fills in a reflected version of the same name. If n is a bound variable, then the quotation is filled with a reflected version of n. Otherwise, it is looked up in the global context, within which it must be unambiguous. To resolve ambiguities, the user must provide sufficient namespace information to disambiguate the name, as is standard in Idris. The second new syntactic form, unresolved quotation, simply elaborates to a reflection of the parser's notion of the name. This is convenient when constructing new names during a reflective procedure, or when using elaborator reflection interactively.

7.6 Future Extensions

There are a number of interesting and useful extensions to the quasiquotation mechanism described in this chapter that have not yet been implemented.

Idris's reflection mechanism is presently entirely untyped. Although the typing discipline of quasiquotations imposes a certain amount of sanity on metaprograms that use reflection, it is still not possible to use this quotation mechanism to implement automation libraries in the style of MTac [Zil+13] or VeriML [SS10] in which the type of a proof tactic guarantees that if the tactic succeeds, then it has solved a goal of a particular type. Decorating quotations with phantom types that can only originate from the elaborator might be useful for providing safer proof automation tools. We have developed a prototype implementation of this mechanism on a separate Idris branch, but it has not yet been merged into the

mainline Idris compiler. A more robust form of typed quotation, in which the structure of the reflected term itself has the desired typing properties, may also be an interesting extension. For example, quotation could be used to automate the production of Devriese and Piessens's term representations [DP13]. However, representing type theory in itself is still a very demanding exercise, and it would be difficult to make it practical for programming.

Najd et al.'s quoted domain-specific languages [Naj+15], described in Section 2.3, provide a greater degree of flexibility when repurposing a host language's syntax. However, these quoted DSLs need access to high-level Idris or Idris⁻ syntax, as they may impose different meanings on syntactic forms. For example, the quoted language may need to have rules for `case` expressions that cannot be recovered from the top-level pattern-matching definitions produced by the Idris elaborator. Implementing a QDSL in the manner described in Najd et al.'s technical report would also require the typed quasiquotations described in the previous paragraph. Additionally, given some syntactic sugar, this form of quotation could be used to implement a full-fledged macro system in the tradition of Lisp, and it could allow experimentation with new elaborator features if combined with a metacircular reflected elaborator using the reflected elaboration operators described in Chapter 9.

Presently, quotations apply only to terms. However, quoted definitions could also be very useful for implementing domain-specific languages, code generation tools, generic programming systems, and other alternative interpretations of code. In particular, when pattern matching against the definition of a datatype in the reflected elaborator, it would be very convenient to be able to use the concrete syntax instead of the reflected syntax described in Chapter 5.

The restriction that quoted terms are elaborated in the global environment is too strict for some applications. It would be convenient to expand them to support descriptions of a local environment. This could be accomplished by elaborating both terms and goal types in a context in which abstract hole bindings have been established for the free variables. These hole bindings would be forgotten when the elaboration state is restored.

Chapter 8

Error Reflection

This chapter is based on an unpublished paper [Chr14a], an earlier version of which was presented at the Symposium on Trends in Functional Programming in 2014.

8.1 Introduction

Much of the discourse about domain-specific languages (DSLs) has traditionally focused on allowing domain experts, rather than professional software developers, to describe non-trivial software in a notation that is close to their mental models. However, implementing a programming language is difficult and expensive. In an EDSL, an already-existing general-purpose language is used to express the domain-specific language, which may not require as large of an investment of time as a stand-alone implementation due to re-use of existing development tools. In recent years, another category of embedded domain-specific languages has come to the fore: EDSLs intended for professional software developers that mask some of the complexity of a particular area of programming. Examples of this category include Chafi et al.'s work with heterogeneous parallel computation [Cha+10a], the Repa array language [Kel+10], and the Feldspar language for digital signal processing [Axe+10].

Embedding DSLs in a host language with an expressive type system allows domain-specific type systems to be encoded in the host type system. This has a number of benefits: the DSL type system inherits properties from the host language such as type soundness and decidability of type checking, the DSL developers do not need to implement complicated fea-

tures such as type inference, and existing development tools can support the embedded language's type system without further extension or customization. The Achilles' heel of this approach is the convoluted error messages that can result from non-trivial encodings: they may bear little resemblance to the embedded surface syntax and they may be long and stereotypical. The utility of a type error that no user can understand is questionable at best.

One of the design goals of Idris is that it should be a good host for embedded languages. Examples of languages that have been embedded in Idris include Brady's algebraic effects language [Bra13c; Bra14] and his earlier language for resource-safe programming [BH12]. Dependent types have the potential to be a host for very expressive embedded type systems. Additionally, embedded languages have the potential to hide the complexity of full dependent types, allowing programmers to write code as if they were working in a simpler language, while automation procedures take care of the proofs behind the scenes. If this goal is to be achieved in practice, then the problem of error messages must be solved.

For example, take an embedded language for querying relational databases. This query language might require that projections from tuples select columns that actually exist in the schema. The mechanism described in this chapter can transform this generic error message:

```
Can't solve goal
        HasCol [("name", STRING), ("age", INT)] "naime" STRING
```

into this domain-specific error message:

```
The schema [("name", STRING), ("age", INT)] does not contain the
column "naime" with type STRING
```

8.2 Error Reflection

The primary contribution of this chapter is an extension to the Idris language, called *error reflection*, that expands the scope of the reflection mechanism used for proof automation to encompass error reporting. Error reflection enables developers to rewrite the compile-time error messages that result from complex APIs, particularly embedded domain-specific languages, so that the error messages can be consistent with the ideas and metaphors of the API or DSL. We present several examples of error messages that can be improved using error reflection. Though the technique

is implemented in Idris, there is reason to believe that it would be applicable to languages without dependent types. Error reflection is available in versions of Idris numbered 0.9.13 and higher.

8.2.1 Motivating Example

As a very simple motivating example, consider a tiny fragment of a database interface library that contains schemas, tuples, and relations, along with Cartesian products of relations and projection of individual elements from tuples. Following Oury and Swierstra [OS08], we define a simple universe of datatypes that will be supported in the database. For the sake of simplicity, the embedded query language will support only integers and strings:

```
data Ty = INT | STRING

interpTy : Ty → Type
interpTy INT    = Int
interpTy STRING = String
```

In this representation, schemas are simply lists of pairs of attribute names and codes from the universe. To avoid getting sidetracked, no uniqueness condition is imposed on attribute names.

```
Schema : Type
Schema = List (String, Ty)
```

Tuples are represented by the family `Row`, which is indexed by schemas:

```
data Row : Schema → Type where
  Nil : Row []
  (::) : interpTy t → Row s → Row ((c,t) :: s)
```

In Idris, naming the constructors `Nil` and `(::)` allows list literal syntax to be used to construct a `Row`. The following listing puts these pieces together, showing a concrete schema and tuple:

```
r : Row [("name", STRING), ("age", INT)]
r = ["Jane", 43]
```

Projections from a tuple are slightly more complicated: the type of the result depends on the schema, and the system should disallow projections of columns that do not exist in the schema. This check should occur

statically, so that a query that purports to be defined against a particular schema is in fact a query against that schema. A convenient way to achieve this is to define a type of witnesses that a particular attribute is present in a schema, and then arrange for Idris to construct these witnesses on demand. The type `HasCol s c t` represents that schema *s* contains a column named *c* with type *t*, using the standard technique.

```
data HasCol : Schema → String → Ty → Type where
  Here : HasCol ((c, t) :: s) c t
  There : HasCol s c t → HasCol ((c', t') :: s) c t
```

Note that it is impossible to show that the empty schema contains any attribute at all:

```
instance Uninhabited (HasCol [] c t) where
  uninhabited Here      impossible
  uninhabited (There _) impossible
```

Projection can now be defined by recursion over the structure of these witnesses:

```
project : (c : String) → (t : Ty) → (r : Row s) →
          (ok : HasCol s c t) → interpTy t
project c t []        ok          = absurd ok
project c t (x :: xs) Here        = x
project c t (x :: xs) (There ok') = project c t xs ok'
```

In practice, however, we cannot expect users of our embedded query language to construct a `HasCol` every time they want to project an element from a tuple. Thus, we redefine `ok` to be an implicit argument that should be inferred by the compiler. The **auto** keyword causes Idris to construct the `HasCol` witness using its built-in proof search.

```
project : (c : String) → (r : Row s) →
          {auto ok : HasCol s c t} → interpTy t
project c []        {ok = ok}          = absurd ok
project c (x :: xs) {ok = Here}        = x
project c (x :: xs) {ok = (There ok')} = project c xs {ok=ok'}
```

A *relation* is a collection of tuples with the same schema. Here, a relation containing *n* tuples with schema *s* is represented by a `Vect n (Row s)`.

Like SQL and unlike the relational algebra, this encoding allows for duplicate tuples. The Cartesian product, written here with the `(*)` operator, is the concatenation of each row from one relation with each row from another. Just like projection, the Cartesian product has a side condition: it is only defined for tuples whose collection of attribute names are disjoint. This enables attribute names to be used for projection later. As before, we represent this side condition using an automatically-solved implicit proof of disjointness.

```
(*) : Vect n (Row s1) -> Vect m (Row s2) ->
      {auto prf : Disjoint s1 s2} ->
      Vect (n * m) (Row (s1 ++ s2))
(*) []      ys             = []
(*) (r::rs) ys {prf = prf} = map (r++) ys ++ ((*) rs ys {prf=prf})
```

Ideally, users of an embedded database language would be able to work entirely within the abstractions of the database language, rather than needing to worry about the details of proof automation and implicit arguments. However, when they make mistakes, the error messages are expressed in terms of the underlying implementation of the static semantics. The purpose of this work is to improve on this situation.

8.2.2 Error Messages

The relation `humans` describes people and their ages, while `housing` lists the size of two apartments:

```
humans : Vect 2 (Row [("name", STRING), ("age", INT)])
humans = [ ["Alice", 37]
         , ["Bob",   23]
         ]

housing : Vect 2 (Row [("floorspace", INT)])
housing = [ [48]
          , [72]
          ]
```

The first column of the first row in `humans` can be extracted using the `project` function, but if a user misspells a column name, then the resulting error message can be difficult to decode:

```
> project "name" (head humans)
"Alice" : String

> project "naime" (head humans)
Can't solve goal
        HasCol [("name", STRING), ("age", INT)] "naime" t
```

Likewise, the Cartesian product of `humans` and `housing` contains the expected four tuples. However, trying to take the product of `humans` and itself results in an error that reflects details of the DSL implementation rather than domain concepts:

```
> humans * housing
[["Alice", 37, 48],
 ["Alice", 37, 72],
 ["Bob", 23, 48],
 ["Bob", 23, 72]] : Vect 4
                         (Row [("name", STRING),
                               ("age", INT),
                               ("floorspace", INT)])

> humans * humans
Can't solve goal
        Disjoint [("name", STRING), ("age", INT)]
                 [("name", STRING), ("age", INT)]
```

Careful naming of the required proof objects can sometimes lead to these errors being somewhat understandable, as above. However, good error messages should do more than simply provide a vague hint about why a problem arose. They should explain the problem, and do so in an accessible and straightforward manner.

8.2.3 Reflecting Errors

To solve this problem, we extended Idris's reflection mechanism to encompass error messages. Now, the standard library contains a datatype corresponding to the compiler's own internal representation of errors, and Idris functions can insert themselves into the error reporting mechanism to rewrite errors before they are shown to users. In some sense, these error handlers resemble exception handlers, except they can only raise a

new exception, rather than recovering from the error and continuing the program.

An error handler is a partial function from a representation `Err` of error messages to a rewritten error report. Accordingly, we might assign them the type `Err → Maybe String`. However, error messages have more structure than a String can express. Often, they will include Idris terms, or have a hierarchical structure. Error reports that result from reflection should be able to use the facilities of the compiler that already exist for rendering this structure. Thus, Idris defines a type `ErrorReportPart` that represents the various sorts of content that can appear in an error report.

```
data ErrorReportPart = TextPart String
                     | NamePart TTName
                     | TermPart TT
                     | RawPart Raw
                     | SubReport (List ErrorReportPart)
```

The constructor `TextPart` represents a string containing error explanations, `NamePart` contains a reflected Idris name to be highlighted, `TermPart` contains a reflected, fully-elaborated Idris term to be pretty-printed, `RawPart` contains a putative TT term that has not yet been type checked, and a `SubReport` contains further information to be rendered as additional explanation in an indented block.

The representation of errors `Err` is simply a subset of the constructors of the compiler's internal datatype that represents errors. The most important of these errors are conversion failures, unification errors, and proof search failures. It is a subset because error reflection should not rewrite errors that have already been rewritten, and because some types of errors exist primarily to keep track of things like source location, which rewritten errors should not lie about. Additionally, the compiler contains a number of unstructured error messages that only arise in specialized circumstances. Due to this lack of structure, it does not make sense to pattern-match these errors.

Error handlers map reflected errors to lists of the previously-described error report parts. Because not every handler will handle every error, error handlers should have the type `Err → Maybe String`. To avoid unintentional rewriting of errors, the keyword **`%error_handler`** is used to mark functions with this type as error handlers.

The function `dbErr` in Figure 8.1 maps proof search errors in the above code to domain-specific error messages. It relies on an auxiliary func-

tion `getHasColFields`, which simply extracts the three parameters of the reflected representation of a `HasCol` proof. Note that the error handler distinguishes between cases in which the elaborator has already discovered a value for the type of the column by checking for bound variables first. This easy interleaving of quotation and low-level syntax is a key advantage of quasiquote patterns.

As we saw in Section 8.2.1, the following definition quite obviously fails to satisfy the condition that the arguments of the Cartesian product of relations should be disjoint:

```
test3 : Vect 4 (Row [("name", STRING), ("age", INT),
                     ("name", STRING), ("age", INT)])
test3 = humans * humans
```

Now, however, the resulting error message refers specifically to the notion of disjointness:

```
When checking right hand side of test3:
The schemas [("name", STRING), ("age", INT)] and
[("name", STRING), ("age", INT)] are not disjoint.
```

Additionally, misspelling a column name when performing a projection provides a clear error. The definition:

```
floorspace : Int
floorspace = project "flodorspace" INT (index 2 (humans * housing))
```

yields the error:

```
When elaborating right hand side of floorspace:
The schema [("name", STRING), ("age", INT)] ++ [("floorspace", INT)]
does not contain the column "flodorspace" with type INT
```

which quite straightforwardly explains the problem. Because the reported error contains semantically interesting information, users can additionally use the Idris IDE features described in Chapter 6 to do things like normalizing subterms of error messages in-place. The above error message can be converted to the following error message with just a few clicks of the mouse:

```
When elaborating right hand side of floorspace:
The schema [("name", STRING), ("age", INT), ("floorspace", INT)]
does not contain the column "flodorspace" with type INT
```

```
%error_handler
total
dbErr : Err -> Maybe (List ErrorReportPart)
dbErr (CantSolveGoal `(Disjoint ~s1 ~s2) _) =
  Just [ TextPart "The schemas", TermPart s1
       , TextPart "and", TermPart s2
       , TextPart "are not disjoint." ]
dbErr (CantSolveGoal `(HasCol ~s ~c ~(P Bound _ _)) _) =
  Just [ TextPart "The schema", TermPart s
       , TextPart "does not contain the column"
       , TermPart c
       ]
dbErr (CantSolveGoal `(HasCol ~s ~c ~t) _) =
  Just [ TextPart "The schema", TermPart s
       , TextPart "does not contain the column"
       , TermPart c, TextPart "with type"
       , TermPart t
       ]
dbErr _ = Nothing
```

Figure 8.1: An error handler for the embedded database language. Quasiquotations are used to destructure the terms inside of reflected errors.

8.3 Applications

While error reflection was a useful technique for improving the usability of our fragment of a domain-specific language, we should hope that the technique is more broadly applicable. This section exhibits specific applications for error reflection in already-existing Idris code.

8.3.1 Beginner-Friendly Arithmetic

The code described in this section was added to the Idris standard library by Edwin Brady. Here, we describe it as an illuminating example.

Like Haskell, Idris has a `Num` type class. An instance of `Num` a explains how integer literals should be converted to elements of a as well as how to perform operations such as addition. However, new users without a background in Haskell can be confused by the error that results when an integer literal is used improperly:

```
foo : Int → Int
foo x = if 17 then x else 42

Can't resolve type class Num Bool
```

Suddenly the number of concepts that they need to know has grown to include both type classes and the default desugaring of integer literals. Thus, the Idris prelude contains the following error handler:

```
%error_handler
num_error : Err → Maybe (List ErrorReportPart)
num_error (CantResolve `(Num ~x)) =
    Just [ TermPart x
         , TextPart "is not a numeric type"
         ]
num_error _ = Nothing
```

Now, the above error is rewritten to:

```
Bool is not a numeric type
```

which refers to the error actually committed by the user, without reference to the language mechanisms by which the error is noticed. Advanced users can toggle the display of the original error message in connection with the rewritten one.

8.3.2 Integer Literals for Finite Set Elements

A typical example of dependent types are the finite sets. Given a natural number n, the type `Fin` n has exactly n canonical elements. In other words, `Fin 0` is uninhabited, `Fin 1` has precisely one element, `Fin 2` has precisely two elements, and so forth. The `Fin` family is often used for bounds-checked indexing into data structures. Thus, it can be convenient to use integer literals for them.

Unlike Haskell, type-driven ad hoc overloading is used to disambiguate the application of `fromInteger` that integer literals desugar to. This means that types need not have instances of the `Num` type class in order to support integer literals — they need only define an overloading of the name `fromInteger`. The Idris standard library contains the following definition:

```
fromInteger : (x : Integer) →
              {default ItIsJust
               prf : (IsJust (integerToFin x n))} → Fin n
fromInteger {n} x {prf} with (integerToFin x n)
  fromInteger {n} x {prf = ItIsJust} | Just y = y
```

Here, `IsJust : Maybe a → Type` is inhabited when its argument is built with the `Just` constructor. The **`default`** `ItIsJust` modification to the implicit argument causes Idris to use the constructor of these proofs to solve it. The function `integerToFin` simply returns the corresponding `Fin` if possible, or `Nothing` otherwise. This combination, then, statically ensures that `Fin` literals are within their bounds.

Unfortunately, the error message that results from this arrangement is somewhat opaque. The simple definition:

```
f : Fin 2
f = 3
```

results in a quite involved error, in which otherwise-hidden details of the implementation take center stage:

```
When checking argument prf to function fromInteger:
        Type mismatch between
                IsJust (Just x) (Type of ItIsJust)
        and
                IsJust (integerToFin 3 2) (Expected type)

        Specifically:
                Type mismatch between
                        Just x
                and
                        Nothing
```

Fortunately, it is straightforward to define an error handler that will rewrite this error to something understandable. The error handler is demonstrated in Figure 8.2. In the presence of this error handler, the above definition of `f` results in a much more explanatory error message:

```
When checking argument prf to function fromInteger:
        When using 3 as a literal for a Fin 2
                Could not show that 3 is less than 2
```

By further dissecting the argument `n` to `integerToFin`, the message can be further improved to distinguish between the cases where the literal is most definitely out of bounds and the cases where it simply cannot be statically demonstrated.

```
%error_handler
finTooBig : Err -> Maybe (List ErrorReportPart)
finTooBig (CantUnify x tm `(IsJust (integerToFin ~n ~m)) err xs y)
  = Just [ TextPart "When using" , TermPart n
         , TextPart "as a literal for a"
         , TermPart `(Fin ~m)
         , SubReport [ TextPart "Could not show that"
                     , TermPart n
                     , TextPart "is less than"
                     , TermPart m
                     ]
         ]
finTooBig _ = Nothing
```

Figure 8.2: An error handler for finite set literals.

8.3.3 Algebraic Effects

Idris includes a library for handling side effects compositionally, without explicitly using monad transformers. Briefly, `Eff a [E, F r, G]` is the type of an effectful computation that uses effects E, F, and G, yielding a value in a. Additionally, the effect F has some resource r, which might be a file handle or a state. Effectful operations can, but need not, change the type of the resource.

When one operation in `Eff` calls another, the library searches for a proof that the called operation's effects are a subset of the calling operation's effects. This ensures that all effects that might be performed are visible in the caller's type, but it does not require that the called operation have the exact same effect collection.

As an example of the effects library, consider a program that reads a name from standard input, and then greets the user by name. Its type signature lists that it uses `STDIO`, which is the representation of console input and output, as well as that it will not return an interesting value.

```
hello : Eff () [STDIO]
hello = do n ← getStr
           putStrLn ("Hello, " ++ n)
```

If this program is rewritten to read the name from a file, the type of `hello` must be updated. If it is not, as in the following program, then Idris will report a proof search error.

```
getName : Eff (Maybe String) [FILE_IO ()]
getName = do ok ← open "test" Read
             case ok of
               False ⇒ return Nothing
               True  ⇒ do name ← readLine
                          close
                          return (Just name)

hello : Eff () [STDIO]
hello = do Just n ← getName
             | Nothing ⇒ putStrLn "Can't read file"
           putStrLn ("Hello, " ++ n)
```

The error that occurs is quite long, because some details of the implementation of `Eff` are visible. Buried deep within is the fact that it is unable to implicitly find a value with the type:

```
SubList [FILE_IO ())] [STDIO]
```

While a great deal of thought has gone into making this error as readable as possible, the effect system is just a library, and it must juggle concerns that include generating efficient Idris code, automating proof obligations such as this one, and having a convenient syntax. The compiler can't possibly provide more useful suggestions.

An error handler can rewrite this to something more informative. Because the term that we care about is surrounded by other information, the first step is to extract the invocation of `SubList` for which proof search failed. This can be done through straightforward recursion over the `TT` representation, using a quasiquotation pattern to match the instance that is of interest. The actual error handler checks whether it is the result of a proof search failure. If so, it examines the failed type for `SubList` and constructs a friendlier error message.

```
findSubList : TT -> Maybe (TT, TT)
findSubList `(SubList {a=~_} ~l ~r) = Just (l, r)
findSubList (Bind n b tm) = findSubList tm
findSubList (App tm tm') = findSubList tm <|> findSubList tm'
findSubList _ = Nothing


%error_handler
sublist_err : Err -> Maybe (List ErrorReportPart)
sublist_err (CantSolveGoal tm xs) =
    do (required, found) <- findSubList tm
       return [ TextPart "Attempted to use an operation with effects"
              , TermPart required
              , TextPart "in a context where only"
              , TermPart found
              , TextPart "are available."
              ]
sublist_err _ = Nothing
```

With this error handler, the message becomes:

```
Attempted to use an operation with effects [FILE_IO ()] in a context
where only [STDIO] are available.
```

This message helps the user focus on the part of the error that was relevant and provides a much better hint as to the significance of the relationship between these lists.

8.4 Argument Error Handlers

The error reflection mechanism described thus far suffers from a major shortcoming: the risk of spurious matches. For example, it is perfectly reasonable to expect that a library other than the effects library might use the SubList family and its associated proof search procedure. However, the error handler described in Section 8.3.3 will also be used to rewrite error messages resulting from this new library, giving blatantly misleading results. The risk of false positives arises whenever a single type is used for multiple purposes, and the global nature of error handlers means that importing a new library can break the error messages for some other imported library if they share a mutual dependency.

The problem could be solved by copying and pasting the definitions of the types in question to a separate namespace, and being careful about matching namespaces in error handlers. However, copying and pasting is not typically regarded as a good code re-use practice. Even worse, users may receive a very confusing message if a library developer is not aware that a particular type is used in more than one location. Developers of error handlers need not be the original authors of a library.

The chance of false positives can be reduced by narrowing the scope of error handlers. Thus, Idris supports attaching them to specific formal parameters of specific functions. A comma-separated list of error handler names *h1*, ..., *hn* is attached to parameter *x* of the function, constructor, or type constructor *f* using the pragma:

```
%error_handlers f x h1, ..., hn
```

When an error results from the elaboration of a term that occurs as an argument to *f* in the position indicated by *x*, the error handlers *h1*, ..., *hn* will be preferred over global error handlers.

8.5 Implementation Considerations

While error reflection is implemented in Idris, there are no fundamental considerations that prevent it from being implemented in languages without dependent types. Nevertheless, a practical implementation requires a certain amount of compiler infrastructure that may not be available in every programming language.

Compile Time Evaluation Executing error handlers requires that the compiler be able to evaluate expressions while type checking. Thus, an interpreter for the language being type-checked should be available, and the type checker should have some facility for using it. This is available by definition in a dependently typed language, but many other languages or external static analysis tools will also be able to do this.

Ensuring Termination Running arbitrary code at compile time has the potential to cause the compiler to not terminate. Because dependently typed languages do this as a matter of course, they have evolved sophisticated techniques for ensuring that only terminating terms are evaluated at compile time.

Idris checks all programs for termination, but potentially non-terminating terms are simply not reduced by the type checker. To preserve the termination of the type checking process, error handlers that do not pass the termination checker are rejected. If other languages adopt error reflection, they should also implement a termination checker, or handle non-termination through some other mechanism, such as a timeout or through QuickCheck-style [CH00] property-based testing.

In some sense, the usability argument for mandatory termination is a bit of a red herring. A termination checker does not guarantee speedy execution, which means that timeouts and performance testing may be useful even in the presence of termination checking.

Reflection Facilities In order to support error reflection, the reflection capabilities of a host language should, as a minimum, support reification of both types and terms. In a dependently typed language like Idris, this is trivial, as there is no syntactic distinction between types and other terms, so a single mechanism suffices. In other systems, support for reflecting both syntactic categories can vary.

Some systems, such as Template Haskell [SJ02], support both straightforwardly. Some other languages, such as F# [Sym06], have one system for quoting terms and another for representing types (namely, .NET reflection). Scala's quasiquotations [SBO13] support both expressions and types, and could be a promising facility for implementing error reflection.

Error Origin Tracking In a dependently-typed language in which error handlers can be attached to specific function arguments, it is not sufficient to install error handlers as exception handlers in a traversal of the abstract syntax tree. This is because Idris's unifier accumulates a collection of unsolved unification problems, which may become solvable by a mix of later unifications and reductions. At the end of elaboration, the compiler must check that no open unification problems remain.

Unification errors in particular may first be signaled far from their source. In Idris, this is addressed by annotating every error with a stack of surrounding applications, and then using this information to decide which error handlers are eligible to rewrite the error. Other languages that seek to implement error reflection should adopt a similar method for ensuring that the correct handlers are associated with errors, based on these errors' origin in the source code.

8.6 Related Work

While the difficulties in interpreting error messages in embedded DSLs is well-known, there are comparatively few systems that attempt to address it. Here, we do not describe work on improving type errors in general, as the focus is on embedded languages and other situations where error messages could not possibly be improved by the authors of the programming language.

Heeren et al. [HHS03] present a system for constraint-based type inference in the context of the Helium language that aims at solving the same problems as Idris error reflection. Their system is defined at a higher level of abstraction, supporting the definition of custom typing rules that are then mechanically checked for consistency with the host language's type system. Additionally, their system supports defining "sibling functions" that are suggested as alternatives in the case of type errors. Finally, their system allows the order of type inference constraints to be controlled by library authors, making it easier to locate error messages at the real source of errors, rather than elsewhere in a library. Some of these techniques would be applicable in a language like Idris. In particular, sibling functions seem to be quite promising as a potential feature. However, there is no obvious way to apply these techniques to proof search failures, and checking that custom typing rules are a consequence of Idris's typing rules could require arbitrarily complicated computation due to dependent types. Additionally, the lack of global type inference in dependently-typed languages drastically reduces the utility of controlling the order in which constraints are checked.

The Scala-virtualized compiler described by Rompf et al. [Rom+13] supports an `@implicitNotFound` annotation that allows a custom error message to be displayed when the compiler cannot resolve an implicit argument. These new error messages can contain references to type variables in the declaration that they apply to. This feature can be seen as a special case of error reflection in which only one kind of error can be rewritten, with somewhat less ability to destructure the types involved.

8.7 Conclusion and Future Work

As demonstrated, the error reflection facility enables conversion of uninformative error messages to informative, domain-specific error messages.

This is useful both for embedded languages and for ordinary libraries. Quasiquote patterns enable a convenient syntax for destructuring terms that occur in reflected errors and reconstructing informative messages that contain terms.

However, there are still practical considerations to be worked out. Perhaps the most serious is the strong coupling between error handlers and the specific terms that occur in error messages. Both compiler updates and relatively small changes in an embedded language can cause fairly large changes in error messages. To make rewriting reflected errors more robust and flexible, it may be convenient to be able to use tools other than pattern matching to define error handlers. For example, it might be possible to develop a sort of query language for reflected terms that allows convenient and expressive extraction of sub-terms.

It can also be difficult to mentally map the displayed error message to the constructor that represents it in the error type. This could be solved by integrating error reflection more closely into Idris's IDE support. For instance, an interactive command to view the reflected form of an error that is displayed on screen might make it easier to determine the structure to be rewritten.

Chapter 9

Elaborator Reflection

In this chapter, we present *elaborator reflection*, in which the underlying tactics of the Idris elaborator are reified into a type of computations that are accessible from Idris itself. With the reflected elaborator, it becomes possible to re-use Idris's elaboration infrastructure to explain the meaning of embedded languages in terms of TT, allowing the language implementer to piggyback on Idris and re-use implementations of features such as higher-order unification and the built-in proof search.

Elaborator reflection is useful for more than just embedded languages. The ability to programmatically generate TT terms and definitions based on a rich API for static reflection means that the reflected elaborator can be used similarly to Template Haskell [SJ02]. As befits its basis in the techniques of interactive proof assistants, the reflected elaborator can also be used to implement proof automation in a manner reminiscent of Coq's LTac [Del00].

Template Haskell [SJ02] has proven to be a widely applicable tool for solving a variety of problems, including generating boilerplate code, prototyping higher-level generic programming systems [NJ04], generating bindings to foreign interfaces in a manner reminiscent of F#'s type providers [Sym+12], and much more. Unlike C++'s template metaprogramming, Template Haskell programs are written in ordinary Haskell rather than a special-purpose sub-language. In Template Haskell, the compiler supports a type of compile-time side effects, such as generating unique names, as well as access to meta-information, such as reflected definitions of datatypes. Generated terms or declarations can be spliced into ordinary programs.

Imperative tactic scripts in languages such as LTac [Del00] support the

incremental construction of proof terms, with commands for manipulating proof goals, introducing intermediate lemmas, automating the solution of simple goals *en masse*, and composing these procedures into large-scale solvers. In LTac and related tactic languages, the proof assistant allows the execution of tactic scripts to affect a global proof state, which contains things like the goals that remain to be proven, unification constraints, and intermediate proof terms. Additionally, scripts have access to control effects that include failure and recovery. Because the tactic language is not part of the trusted kernel of the proof assistant, tactic scripts need not follow the usual termination restrictions. When the tactic script has completed its task, the resulting term is type checked and saved.

Like Template Haskell, Idris's elaborator reflection mechanism uses Idris itself to implement compile-time metaprogramming. However, Template Haskell's `Q` monad is quite parsimonious, offering few effects and no additional control structures. Elaborator reflection is more expressive: it exposes the tactic-based proof assistant interface of Brady's [Bra13b] elaboration mechanism, allowing metaprograms written in Idris to make use of the same kinds of side effects as LTac scripts while still remaining in the same language.

9.1 Introductory Examples

To give a sense of the "flavor" of reflected elaboration in Idris, this section presents two quite different examples of its use: implementation of a small typed embedded language with a foreign-function interface (FFI) to Idris code and a simple proof search tactic.

9.1.1 A Simple Embedded Language

The reflected elaborator can be used to compile a DSL to TT. Indeed, it is powerful enough to elaborate at least a large subset of Idris's term language itself. This section presents a very simple language that is compiled to TT. The language in question has no static type information; thus, the elaboration process must also perform type inference.

The language is given by the datatype in Figure 9.1. The only constructor of this datatype that is not completely standard is `FFI`. Its argument, `TTName`, is a reflected Idris name. The intention is that this allows an Idris definition to be called directly, which enables additional primitive opera-

```
data Lang : Nat → Type where
  V : Fin n → Lang n
  Ap : Lang n → Lang n → Lang n
  Lam : Lang (S n) → Lang n
  CstI : Integer → Lang n
  FFI : TTName → Lang n
```

Figure 9.1: A simple language datatype, where the index represents the number of free variables.

tions that are not expressible in the language to be added without changing either the AST or the definition of the elaborator.

An example function in this language that adds two numbers follows:

```
exampleFun : Lang 0
exampleFun = Lam $ Lam $
               Ap (Ap (FFI `{prim__addBigInt})
                      (V 0))
                  (V 1)
```

Recall that the syntax `` `{n} `` means the quotation of the name n, once resolved to an actual name in scope. See Section 7.5 for details about these quoted names. In this case, it is a reference to Idris's built-in primitive addition on the arbitrarily large integer type.

When producing terms in TT, we will need concrete names to represent the de Bruijn indices in the `Lang` names. Because a term in `Lang` k can access k free variables, we can represent the corresponding TT names with a `Vect` k `TTName`. Thus, the elaboration procedure has type:

```
elabLang : Vect k TTName → Lang k → Elab ()
```

This means that, given a name for each de Bruijn index, it will produce a term through elaboration effects.

The reflected elaboration context, `Elab`, corresponds to the elaboration mechanism described in Chapter 4. At the beginning of elaboration, there will be a hole in focus whose type is determined by the elaboration of the surrounding high-level Idris. Throughout the course of elaboration, recursive calls will always be made with their hole in focus, and the elaborator for each syntactic production of `Lang` is expected to solve its corresponding hole.

The first case, variables, is solved by looking up the explicit name in the context and placing it in the hole:

```
elabLang ctxt (V i) = do fill (Var (index i ctxt))
                         solve
```

Here, `fill` corresponds to FILL (see page 29), `Var` is the constructor for reflected named variable references (in the datatype `Raw` described in Chapter 5), and `index` looks up a particular value from a `Vect`. The Idris operator `solve` corresponds to the SOLVE tactic, which substitutes the guess introduced by `fill` throughout its scope.

The second case, integer constants, is solved by filling the hole with the Idris integer inside the constructor:

```
elabLang ctxt (CstI x) = do fill (quote x)
                            solve
```

Here, `quote` is an overloaded operation that converts an Idris value into a corresponding reflected value. It is a method of the `Quotable` type class.

To elaborate a function, it is necessary to do slightly more work. First, a unique name is generated for the bound argument variable, to prevent variable capture. Then, the binding is introduced using `intro`, which will fail if the current hole does not have a function type. Finally, the body is elaborated into the new hole underneath the binder.

```
elabLang ctxt (Lam x) =
    do n ← gensym "argument"
       attack
       intro n
       elabLang (n :: ctxt) x
       solve
```

When elaborating an application, it becomes necessary to use the unifier in the Idris elaborator to discover all of the involved types. Four holes are created: two for the types, one for the function itself, and one for its argument. Here, `mkHole` is a derived tactic that generates a unique name and claims it with a particular type. Then, the current hole is filled with an application, and the function and argument holes are used for the elaboration of the function and argument, respectively.

This process is represented as follows:

```
elabLang ctxt (Ap x y) =
    do t1 <- mkHole `(Type)
       t2 <- mkHole `(Type)
       fun <- mkHole `(~(Var t1) -> ~(Var t2))
       arg <- mkHole (Var t1)
       fill (RApp (Var fun) (Var arg))
       solve
       focus fun; elabLang ctxt x
       focus arg; elabLang ctxt y
```

In the final case, the `FFI` constructor, the underlying Idris name is placed directly in the hole, which is then solved. If the name does not refer to a definition with the correct type, elaboration will halt with an error.

```
elabLang ctxt (FFI n) = do fill (Var n)
                           solve
```

Now, this elaborator can be used to compile our example DSL function to a real Idris function:

```
compiled : Integer -> Integer -> Integer
compiled = %runElab (elabLang [] exampleFun)
```

Evaluating `compiled` yields:

```
\argument1 => \argument2 => prim__addBigInt argument2 argument1
```

which is precisely the function that we would expect.

9.1.2 Simple Proof Search

As another example of elaborator reflection, we develop an automated procedure for solving certain simple proof goals. Although the Idris elaborator was not initially intended as a framework for proof automation, its similarities to traditional tactic languages allow it to be used in this manner as well. Following the initial development, we then extend it to perform a more general proof search.

Figure 9.2 contains the complete text of the initial solver. First, it queries the system to discover the type of the hole into which elaboration is occurring. Then, quasiquotation patterns are used to determine how to solve the hole. The type annotations in these quasiquotes are necessary due to

```
auto : Elab ()
auto =
    do g <- goalType
       case g of
         `(() : Type) =>
           do fill `(() : ())
              solve
         `((~A, ~B) : Type) =>
           do aH <- mkHole A
              bH <- mkHole B
              fill `(MkPair {A=~A} {B=~B} ~(Var aH) ~(Var bH))
              solve
              focus aH; auto
              focus bH; auto
         `(Either ~a ~b) =>
           left a b <|> right a b
         _ =>
           fail [ NamePart `{auto}
                , TextPart "can't solve the goal"
                , RawPart g
                ]

  where
    left : Raw -> Raw -> Elab ()
    left a b = do aH <- mkHole a
                  fill `(Left {a=~a} {b=~b} ~(Var aH))
                  solve
                  focus aH; auto
    right : Raw -> Raw -> Elab ()
    right a b = do bH <- mkHole b
                   fill `(Right {a=~a} {b=~b} ~(Var bH))
                   solve
                   focus bH; auto
```

Figure 9.2: A simple proof search, implemented using elaborator reflection.

the syntactic punning. If the goal is the unit type, the hole is filled with its trivial constructor. If the goal is a product type, new holes are generated for the left and right projections of the resulting pair. Then, `auto` is applied in each of these holes. If the goal is a coproduct type, the failure-recovery combinator `<|>` is applied, first attempting a procedure that would use the left constructor and then one that would use the right. Finally, in the case that none of these cases match, the `fail` tactic is used to deliver a friendly error message.

The ability to fail directly avoids the need for kludges like the error reporting mechanism in Kokke and Swierstra's proof search [KS15], in which blatantly type-incorrect values that happen to contain messages for users are substituted in contexts that are incorrect. Kokke and Swierstra's approach is to use a datatype like:

```
data Failure : String → Type where
  Fail : (msg : String) → Failure msg
```

If `Fail` *msg* is used to solve a goal, a compiler notification can be produced that contains the error message, such as:

```
Failure "proof search failure" !=< ⊥ of type Set
```

Even though it can report reasonable error messages, this approach to proof automation is unsatisfactory. Every time that we would like `auto` to support a new datatype, it must be extended with an additional pattern-match case. Then, each constructor of the datatype must be attempted (in a fashion similar to the `Either` case above), and each argument to these constructors must be supplied (in a fashion similar to the product type above). By using the datatype reflection capabilities of `Elab`, it is possible to retrieve sufficient information about its constructors to generalize this technique.

The first step in this generalization is to define two helper functions: `headName` and `inHole`, which respectively extract the name at the head of a term and execute a tactic in the context of a particular hole, if it is in fact a hole. The operator `inHole` is useful because some holes will be solved automatically according to unification constraints, and focusing on a non-existent hole would be an error.

To apply a constructor, we use `applyCtor`, which applies a constructor to the appropriate number of fresh holes, then solves each of them using some tactic that it receives as an argument. Iteration over the argument holes is performed using `for_`, which is a standard Idris idiom that

```
headName : Raw -> Maybe TTName
headName (Var n)     = Just n
headName (RApp tm _) = headName tm
headName _           = Nothing

inHole : TTName -> Elab () -> Elab ()
inHole h tac = do hs <- getHoles
                  if h `elem` hs
                    then do focus h; tac
                    else return ()

applyCtor : TTName -> Nat -> Elab () -> Elab ()
applyCtor cn argCount tac =
  do holes <- apply (Var cn) (replicate argCount True)
     solve
     for_ holes $ \h =>
       inHole h tac
```

Figure 9.3: Helpers used in the proof search example.

is described in Appendix B. `headName`, `inHole`, and `applyCtor` are defined in Figure 9.3.

Finally, we have the tools to construct a general-purpose depth-first constructor-based proof search. To ensure termination, the search takes a bound on the depth of recursion. Additionally, the tactic takes a list of inductive families to consider and a fallback tactic for when the goal is not one of these families. When the goal type is the application of a name that is in the list of family names to consider, the reflected form of the definition is looked up and the `constructors` field is projected from it. Then, each reified constructor is converted into a computation that will apply it and continue searching recursively, and the first succeeding computation is selected using `choice`.

Not only is this new code much more general, it is also much shorter than the procedure in Figure 9.2. The behavior of that example can be recovered (modulo a limit on search depth) with the definition `auto` in Figure 9.4.

```
byConstructors : Nat → List TTName → Elab () → Elab ()
byConstructors Z _ _ =
  fail [TextPart "Search failed because the max depth was reached."]
byConstructors (S k) tns tac =
  do case headName !goalType of
       Nothing ⇒ tac
       Just n ⇒
         if not (n `elem` tns)
           then tac
           else do ctors ← constructors <$> lookupDatatypeExact n
                   choice (map (\(cn, args, _) ⇒
                                  applyCtor cn
                                            (length args)
                                            (byConstructors k tns tac))
                               ctors)

auto : Elab ()
auto = byConstructors 1000 [`{Unit}, `{Pair}, `{Either}] nope
  where
    nope : Elab ()
    nope = do g ← snd <$> getGoal
              fail [ NamePart `{auto}
                   , TextPart "can't solve the goal"
                   , TermPart g
                   ]
```

Figure 9.4: The proof automation example.

9.2 Elaborator Reflection, Defined

After the previous section's examples of how elaborator reflection can be used, we now turn to the definition and description of the feature itself. Elaborator reflection extends the high-level Idris language with one new syntactic form, one new type constructor, and a collection of primitive operations. The new syntax extends desugared Idris (that is, both Idris and Idris^{-}) with one additional production, the splicing operator **%runElab**:

$$\begin{array}{rcll} e, t & ::= & \ldots & \\ & | & \textbf{\%runElab}\ e & \text{(splice of } e\text{'s result)} \end{array}$$

The new type constructor, called `Elab`, is analogous to Template Haskell's `Q` monad [SJ02]. Computations in `Elab` provide side effects such as a fresh

$$
\begin{aligned}
\mathcal{E}\,[\![\texttt{\%runElab}\ e]\!] \quad = \text{do} \quad & \textsc{claim}\ s\ :\ \texttt{Elab Unit} \\
& \textsc{focus}\ s \\
& \textsc{attack} \\
& \mathcal{E}\,[\![e]\!] \\
& x \leftarrow \textsc{guess} \\
& \textsc{solve} \\
& \mathcal{X}\,[\![x]\!] \\
& \textsc{solve}
\end{aligned}
$$

Figure 9.5: Elaborating **%runElab**.

name supply and access to metadata about the global context.

The elaboration rule for the new Idris construct **%runElab** merely delegates to a new meta-operation $\mathcal{X}\,[\![\cdot]\!]$ which is responsible for executing reflected elaboration scripts. We must first *capture* the result of elaborating the script without otherwise disturbing the elaboration process, then *execute* the resulting TT term. This process is illustrated in Figure 9.5.

The first step is to elaborate the tactic script. Because the term elaborator $\mathcal{E}\,[\![\cdot]\!]$ assumes that there is already an in-focus hole with the correct type, CLAIM is used to create the hole and FOCUS is used to bring it into focus. However, it is too early to elaborate the script. At the end of the elaboration of each syntactic form, $\mathcal{E}\,[\![\cdot]\!]$ SOLVEs the hole into which it is elaborating its argument. Because there are no references to the hole s, this will result in the hole being eliminated along with the results of elaboration. Inserting a reference to the hole, e.g. in a `let` binding, is not an acceptable solution, because that would cause the reference to persist into the result of elaboration. The solution is to use ATTACK from Section 4.5 to create a hole that the current hole refers to as a guess. Once the result of elaboration is present as a guess in s, it can be retrieved using the meta-operation GUESS. After capturing the result, s can be eliminated with SOLVE and the results executed with $\mathcal{X}\,[\![\cdot]\!]$.

In the remainder of this section, we define $\mathcal{X}\,[\![\cdot]\!]$ for the operations of the reflected elaborator. The high-level descriptions of elaborator operations are provided in Idris syntax, but when detailed semantics are necessary, they are given in TT, as the elaboration of `Elab` operations is straightforward. Just like type theory, elaborator reflection is open-ended — it can be freely extended. These future extensions can be implemented by

extending TT with new inhabitants of Elab and providing them with a semantics in $\mathcal{X}\,[\![\cdot]\!]$.

9.2.1 Control Structures

To enable computations to be sequenced and to depend on one another, `Elab` provides primitive `pure` and `>>=` operators, which are sufficient to implement `Functor`, `Applicative`, and `Monad`. This enables `do`-notation and idiom brackets to be used to compose computations. Additionally, the primitives `<|>` and `fail` enable the implementation of `Alternative Elab` as a left-biased error handler.

$$
\begin{aligned}
\mathcal{X}\,[\![\texttt{pure}_{\texttt{Elab}}\ a\ v]\!] &= \textsc{return}\ v \\
\mathcal{X}\,[\![\texttt{>>=}_{\texttt{Elab}}\ a\ b\ v\ k]\!] &= \begin{array}{l}\texttt{do}\ x \leftarrow \mathcal{X}\,[\![v]\!] \\ \qquad \mathcal{X}\,[\![k\ x]\!]\end{array} \\
\mathcal{X}\,[\![\texttt{fail}_{\texttt{Elab}}\ a\ m]\!] &= \textsc{fail}\ m \\
\mathcal{X}\,[\![\texttt{<|>}_{\texttt{Elab}}\ a\ l\ r]\!] &= \textsc{try}\ (\mathcal{X}\,[\![l]\!])\ (\mathcal{X}\,[\![r]\!])
\end{aligned}
$$

9.2.2 Queries

Just like Template Haskell, reflected elaboration supports queries about the contents of the global context. Additionally, there are queries about the current local elaboration context.

- `getEnv : Elab (List (TTName, Binder TT))` reflects the lexical environment surrounding the focused goal, providing a list of name-binder pairs.
- `goal : Elab (TTName, TT)` provides the name and type of the focused hole. This type may refer to variables in the local environment.
- `holes : Elab (List TTName)` provides the hole queue as an Idris list.
- `guess : Elab TT` extracts the guess from the current hole, if one is available. If the focus is not on a guess, it fails.
- `lookupTy : TTName → Elab (List (TTName, NameType, TT))` resolves a name in the global context, returning a list of triples. Each element consists of a fully-resolved name, an indication of whether it refers

Name	Type	Page
pure	a → Elab a	105
>>=	Elab a → (a → Elab b) → Elab b	105
<|>	Elab a → Elab a → Elab a	105
fail	List ErrorReportPart → Elab a	105
getEnv	Elab (List (TTName, Binder TT))	105
goal	Elab (TTName, TT)	105
holes	Elab (List TTName)	105
guess	Elab (Maybe TT)	105
lookupTy	TTName → Elab (List (TTName, NameType, TT))	105
lookupDatatype	TTName → Elab (List Datatype)	107
fixity	String → Elab Fixity	107
solve	Elab ()	107
fill	Raw → Elab ()	107
apply	Raw → List Bool → Elab (List TTName)	107
matchApply	Raw → List Bool → Elab (List TTName)	108
focus	TTName → Elab ()	108
unfocus	TTName → Elab ()	108
attack	Elab ()	108
claim	TTName → Raw → Elab ()	108
intro	Maybe TTName → Elab ()	108
forall	TTName → Raw → Elab ()	108
letbind	TTName → Raw → Raw → Elab ()	108
patBind	TTName → Elab ()	108
patVar	TTName → Elab ()	109
converts	List (TTName, Binder TT) → TT → TT → Elab ()	109
normalise	(List (TTName, Binder TT)) → TT → Elab TT	109
whnf	TT → Elab TT	109
check	List (TTName, Binder TT) → Raw → Elab (TT, TT)	109
compute	Elab ()	109
rewriteWith	Raw → Elab ()	110
sourceLocation	Elab SourceLocation	110
currentNamespace	Elab (List String)	110
gensym	String → Elab TTName	110
resolveTC	TTName → Elab ()	110
search'	Int → List TTName → Elab ()	110
debugMessage	List ErrorReportPart → Elab a	111
metavar	TTName → Elab ()	111
declareType	TyDecl → Elab ()	111
defineFunction	FunDefn → Elab ()	111
addInstance	TTName → TTName → Elab ()	111
runElab	Raw → Elab () → Elab (TT, TT)	111

Figure 9.6: Operations in Elaborator Reflection.

to a data constructor, a type constructor, or a function, and its elaborated type.

- `lookupDatatype : TTName → Elab (List Datatype)` resolves a name and returns reflected definitions for all inductive families whose names are overloadings of the input.

- `fixity : String → Elab Fixity` discovers the declared fixity for an operator, allowing this to be used by code generators when displaying infix constructors.

These queries will not be defined formally with the $\mathcal{X}[\![\cdot]\!]$ notation because each would simply consist of a new meta-operation that retrieves the corresponding information from the global environment and reifies it to the datatypes described in Chapter 5.

9.2.3 Modifying the Focused Hole

Some operations modify the focused hole. These are:

- `solve : Elab ()` is the SOLVE operation that substitutes a guess in the scope of a hole binding. It is an error when the focus is not on a guess.

- `fill : Raw → Elab ()` places a reflected term into the focused hole, creating a guess. It is an error if there is not a hole at the focus.

- `apply : Raw → List Bool→ Elab (List TTName)` takes as arguments a term and an argument specifier. For each intended argument, it produces a new hole, and marks the hole as solvable by unification if the Boolean value is true, and it fills the focused hole with the application of the term to these new holes. It returns a list of the names of the holes into which the arguments should be placed. Some of these holes may have been solved automatically by unification constraints.

  This operation is somewhat complicated, and it does not appear in Brady's description of the Idris metalanguage [Bra13b]. However, this operation is exceedingly useful: under the hood, it is used to implement the solving of implicit arguments. Implementing it correctly using simpler primitives would be possible, yet tedious and

error-prone, involving the replication of much of the internal compiler infrastructure. In particular, it would need to re-implement the typing rules for dependent functions, substituting each hole for its argument name in the remainder of the type of the operator being applied.

- `matchApply : Raw → List Bool → Elab (List TTName)` is equivalent to `apply`, except it uses one-directional matching instead of unification to automatically solve holes.

- `focus : TTName → Elab ()` moves the focus to some specified hole. The operation fails if the provided name is not a hole.

- `unfocus : TTName → Elab ()` causes the hole named by its argument to be moved to the end of the hole queue. The operation fails if the provided name is not a hole.

- `attack : Elab ()` implements the ATTACK meta-operation described in Section 4.5.

9.2.4 Adding Binders

Some meta-operations introduce binders around the focused hole. In some cases, it is necessary to use `attack` first to prevent improper scoping. These cases are noted as "requiring an immediate hole".

- `claim : TTName → Raw → Elab ()` introduces a new hole binding, given a name and type.

- `intro : Maybe TTName → Elab ()` introduces a lambda around the current hole, which should have a function type. This operation requires an immediate hole.

- `forall : TTName → Raw → Elab ()` wraps the current hole in a function type binding the given name with the given type. This operation requires an immediate hole.

- `letbind : TTName → Raw → Raw → Elab ()` wraps the current hole in a let binder with a given type annotation and value.

- `patBind : TTName → Elab ()` is the analogue of `intro` for pattern variable binders. If the present hole's type indicates that a pattern variable binder is expected, this tactic will introduce it with the given name.

- `patVar : TTName → Elab ()` converts the current hole to a pattern variable with the provided name. This changes both the proof term and the goal type, wrapping them in the appropriate pattern variable binder and pattern variable type, and replaces the hole with a reference to the pattern variable.

9.2.5 Computation

Some of the operators in `Elab` support invoking the evaluator. While terms provided by the users are typically expected to be in the `Raw` format, these operations expect their arguments to be in `TT`. This is because they require that their arguments are well-typed, and therefore expect that the user has already type checked them.

- `converts : List (TTName, Binder TT) → TT → TT → Elab ()` checks that two type-checked terms are convertible in the environment. That is, that they have α-equivalent normal forms.

- `normalise : (List (TTName, Binder TT)) → TT → Elab TT` computes the normal form of a term relative to an environment.

- `whnf : TT → Elab TT` computes the weak head normal form of a closed term.

9.2.6 Invoking the Type Checker

- `check : List (TTName, Binder TT) → Raw → Elab (TT, TT)` invokes the type checker to convert a term to its fully annotated representation. The first projection of the result is the type checked term and the second projection is its type.

9.2.7 Goals

- `compute : Elab ()` normalizes the present goal.

- `rewriteWith : Raw → Elab ()` rewrites the goal using an equality proof. While this could be implemented as a derived tactic that invokes the substitution operator `replace`, using the built-in implementation ensures consistency with the **`rewrite`** … **`in`** … syntax of Idris. This syntax is described in more detail in Appendix B.

9.2.8 Source Contexts

- `sourceLocation : Elab SourceLocation` returns the source location of the invocation site of the tactic script, which can be useful for reporting error messages.
- `currentNamespace : Elab (List String)` returns the default lexically-declared namespace at the invocation site of the tactic script, which allows definitions produced from tactic scripts to easily conform to the namespacing rules of ordinary Idris code.

9.2.9 Names

- `gensym : String → Elab TTName` corresponds to the same operator in Template Haskell [SJ02], which is itself based on the venerable operator from Lisp. It produces a unique unqualified name based on the hint provided. This can be used à la Lisp to avoid variable capture.

9.2.10 Proof Search

The Idris compiler contains proof automation features that are used to fill out function arguments using various strategies. Elaborator reflection makes these features available to Idris code.

- `resolveTC : TTName → Elab ()` solves the current goal using type class resolution, or fails if this is not possible. The argument is a name to exclude from the search, which is used to prevent direct self-recursion when constructing an instance dictionary.
- `search' : Int → List TTName → Elab ()` attempts to solve the current goal using Idris's built-in proof search. The first argument is a bound on the search depth and the second argument is a list of additional hints to use, beyond datatype constructors. A simpler operator, `search : Elab ()`, can be defined as `search = search' 1000 []`.

9.2.11 Development Tools

- `debugMessage : List ErrorReportPart → Elab a` halts the elaborator, dumping the current elaborator state in a readable format. Additionally, a pretty-printed message is rendered using the error message printing facilities described in Chapter 8.
- `metavar : TTName → Elab ()` fills the current hole with a top-level Idris metavariable hole, obligating the user to solve it in some other manner. This allows elaborator scripts to be composed with other means of proof automation.

9.2.12 Global Definitions

- `declareType : TyDecl → Elab ()` adds a new type declaration to the global context. The `TyDecl` record type has fields for the name of the declaration, its arguments' names, types, plicity (see page 41), and erasure status, and return type.
- `defineFunction : FunDefn → Elab ()` defines the pattern matching and reduction behavior of a declared function. The `FunDefn` type, which was defined in Figure 5.7, contains a list of pairs of pattern-matching terms.
- `addInstance : TTName → TTName → Elab ()` adds a previously-declared function to the type class instance search database.

9.2.13 Recursive Invocations

- `runElab : Raw → Elab () → Elab (TT, TT)` constructs a term using another elaboration script. The input is the goal type and the script; the output is the resulting term and its fully-explicit type. This is useful for producing terms to be used in helper definitions.

These represent the lowest level of tactics. In addition to these, a number of derived tactics have proven useful. These are discussed in Section 9.4.

9.3 Implementation Considerations

One concern when defining a reflection system on top of a dependent type theory is to what extent the type system should be used to reason

about metaprograms. Quite intentionally, our reflection library uses only simple datatypes, ruling out complicated dependent types at the border between Idris and the reflected elaborator. This is because dependently typed representations of programming languages are typically best suited to one particular mode of use. For example, if the reflected term datatypes ensured that all name references were well-scoped, then users of the library would need to maintain that invariant at all times. If the type of terms ensured that all reflected terms were well-typed, then users of the library would need to employ complex machinery in the style of Danielsson [Dan07], Chapman [Cha09], and Devriese and Piessens [DP13] at all times. While this might improve our confidence that metaprograms only generate meaningful programs, the resulting complexity would also reduce the number of people who can use the library. Additionally, one of the main reasons to have a tactic language or another metaprogramming system on top of type theory is precisely to allow an escape from the internal reasoning of the system in situations where it is inconvenient, while still producing checkable results.

Our elaborator reflection API is far closer to the idealized tactic language described by Brady [Bra13b] than it is to the real, underlying Idris elaborator. For instance, the real elaborator provides explicit, precise control over unification constraints, while they are implicit in the interface to the `apply` tactic in the reflected elaborator. This is intentional: we have striven to attain a good balance between ease of use and expressive power, and directly annotating constraints is not necessary to achieve any task that has yet been attempted with the reflected elaborator.

Reflecting the elaborator has, in many ways, pushed the current implementation of Idris's expression elaborator to its limits. The expression elaborator runs in a restricted context in which the only effects that are available are elaboration effects. This has been extended with support for modifications to the global Idris compiler state, such as when defining new functions through reflection, but the present design prohibits many useful features and imposes limitations on the effects that can be encoded.

Additionally, operations such as termination checking rely on operators that are not available in the expression elaborator context. Presently, the expression elaborator accumulates a list of instructions for the definition elaborator. These instructions include definitions of helper functions to be elaborated as well as information about global state changes performed by reflected elaborator scripts.

While many of these could be moved from the definition elaborator to

the term elaborator, or made sufficiently polymorphic to be able to run in either context, it might be better to enrich the control structures of the term elaborator to make it interruptable and resumable. This would allow breakpoint-style debugging of reflected elaboration scripts as well as the ability to elaborate helper functions immediately, rather than waiting until after the term elaborator had completed. Additionally, it would provide a clean means of escaping to a context in which more effects, such as logging or running IO actions, are available, without complicating the set of effects typically available in the term elaborator.

9.4 The Pruviloj Library

This section describes *Pruviloj*, a library of derived tactics built from the primitive tactics of Section 9.2. These tactics are used in later developments. Additionally, seeing how they are defined in terms of the primitive tactics may help build intuition for the process of reflected elaboration. This library of tactics, inspired by the standard Coq tactics, is far from the only mode of use for `Elab`. It can also be used to implement other reflection or tactic systems, such as the type-safe tactic language MTac or Agda's non-effectful reflection system.

The first two operations in Pruviloj are not specific to the reflected elaborator. They could be used in other situations, and may eventually be moved to elsewhere in the Idris libraries. The first, `ignore`, causes the return value of an effectful operation to be ignored. The second, `skip`, is included for compatibility with Idris's previous tactic language. It does nothing and has no effects.

```
ignore : Functor f ⇒ f a → f ()
ignore x = map (const ()) x

skip : Applicative f ⇒ f ()
skip = pure ()
```

The next utility in Pruviloj is a means of converting `TT` to equivalent `Raw` terms. Because malformed terms may cause scope errors, this operation may fail. Therefore, it runs in `Elab`. This operation is called `forget` and has type `TT → Elab Raw`.

A common task when working with `Elab` is to destructure the current goal, using parts of it in a solution. However, the goal has been type

checked, so it is provided in `TT`, while operators like `fill` expect `Raw`. Additionally, the primitive `getGoal` returns the name of the goal along with its type, but the name is often irrelevant. The `goalType` tactic encapsulates this common pattern:

```
goalType : Elab Raw
goalType = do g ← getGoal
              forget (snd g)
```

The `hypothesis` tactic solves the current goal using one of the binders in scope, if applicable. First, it retrieves the names of all local binders in scope at the current hole. Then, it attempts to solve the hole using each bound variable in turn. The function `choiceMap` is a fusion of `map` and `choice`, which is described in Appendix B.

```
hypothesis : Elab ()
hypothesis =
    do hyps ← map fst <$> getEnv
       flip choiceMap hyps $ \n ⇒
         do fill (Var n)
            solve
```

The `newHole` tactic captures the common pattern of generating a fresh name and introducing it as a hole:

```
newHole : (hint : String) → (ty : Raw) → Elab TTName
newHole hint ty =
    do hn ← gensym hint
       claim hn ty
       return hn
```

The `exact` tactic immediately solves the current hole with a complete term:

```
exact : (tm : Raw) → Elab ()
exact tm = do fill tm
              solve
```

The `intros` tactic introduces names so long as the goal type is a function. It calculates the bound name for the lambda from the one in the function type. It uses the helper `nameFrom`, which returns a fresh name that is similar to its argument.

```
intros : Elab (List TTName)
intros = do g ← snd <$> getGoal
            go g
  where go : TT → Elab (List TTName)
        go (Bind n (Pi _ _) body) =
            do n' ← nameFrom n
               intro n'
               (n' ::) <$> go body
        go _ = return []
```

Because the reflected elaborator maintains a queue of open holes rather than a tree-structured collection of goals and subgoals, some operations return a list containing the names of the holes that they introduced for later processing. However, solving one of these holes may cause others to be solved by unification, so many tactic scripts need to check whether a hole still exists before focusing on it. Thus, Pruviloj includes the `inHole` tactical that was discussed in Section 9.1.2, which runs a tactic in a hole if that hole still exists.

The `equiv` tactic replaces the current goal with a new goal that is convertible with the old goal. It does this by creating a new hole with the desired type and immediately filling the old hole with the new one, triggering a conversion check.

```
equiv : (newGoal : Raw) → Elab TTName
equiv newGoal =
    do h ← gensym "goal"
       claim h newGoal
       fill (Var h); solve
       focus h
       return h
```

One difficulty of using a hole for elaboration is that holes disappear when solved if there is no reference to them. On the other hand, explicitly setting up the infrastructure to keep them around is tedious. Pruviloj provides a tactic `remember` that produces a hole whose result is `let`-bound in the current scope. When this hole is solved, its value can still be referenced by the name given to `remember`. After `remember` has run, the new hole is in focus, ready to receive the results of elaboration.

```
remember : (n : TTName) → (ty : Raw) → Elab TTName
remember n ty =
    do todo ← gensym "rememberThis"
       claim todo ty
       letbind n ty (Var todo)
       focus todo
       return todo
```

The Pruviloj tactical `repeatUntilFail` corresponds to the `repeat` tactic in Coq. It repeats some tactic until a failure, succeeding if the argument tactic succeeds at least once.

```
repeatUntilFail : Elab () → Elab ()
repeatUntilFail tac =
    do tac
       repeatUntilFail tac <|> return ()
```

Type inference, in the style described in Section 4.4.3, is supported by the `Infer` datatype and the `inferType` tactical. This tactical runs the tactic that it receives as an argument in a context in which the focused hole has another hole as its type, the expectation being that the hole for the type will be solved by unification constraints. It returns the resulting term and type.

```
data Infer : Type where
  MkInfer : (a : Type) → a → Infer

inferType : (tac : Elab ()) → Elab (TT, TT)
inferType tac =
    case fst !(runElab `(Infer) (do startInfer; tac)) of
        `(MkInfer ~ty ~tm) ⇒ return (tm, ty)
        _ ⇒ fail [TextPart "Type inference failure"]
  where
    startInfer : Elab ()
    startInfer =
        do [_, tmH] ← apply (Var `{MkInfer}) [True, False]
              | _ ⇒ fail [TextPart "Type inference failure"]
           solve
           focus tmH
```

The `andThen` tactical supports using one tactic to attempt to make progress in all subgoals introduced by another tactic. The tactic that introduces subgoals should follow the convention of returning a list of holes. The overall result is a list containing the results of the tactics that were run in each hole.

```
andThen : (first : Elab (List TTName)) →
          (after : Elab a) → Elab (List a)
andThen first after =
    do hs ← first
       catMaybes <$> for hs (flip inHole after)
```

Finally, the core of Pruviloj provides the `unproduct` tactic. This tactic takes a reflected term that represents a nested structure of tuples as its argument and let-binds all of its projections, recursively. This can be used together with `hypothesis` to automate the extraction of results from helper tactics that compute more than is strictly necessary.

In addition to these base tactics, Pruviloj defines some tactics that generate helper functions if necessary. These include `induction`, which uses the eliminator generation described in Section 9.5.1 to produce induction principles, and then returns a list with a hole for each constructor. This tactic automatically abstracts the current goal over the scrutinee, using that as the motive for the eliminator. Pruviloj also defines tactics that generate and appeal to proofs of constructor injectivity and disjointness, respectively called `injective` and `disjoint`.

9.5 Other Applications

9.5.1 Deriving Eliminators

Elaborator reflection can be used to replace part of the language implementation, written in Haskell, with Idris library code. Not only does this enable more rapid experimentation with new features, as the entire compiler does not need to be re-built and patches to the compiler do not need to be maintained separately — the resulting drop in complexity makes it easier to maintain the compiler. Additionally, Idris's dependent types and DSL support can be used to assist the development of these extensions. In this section, we outline the replacement of one of the Idris compiler's features with equivalent features written in Idris itself.

Idris supports a rich variety of datatype definitions, including inductive and inductive-recursive families [Dyb94; Dyb00]. To support an induction tactic in Idris's previous tactic language, the compiler contains support for deriving induction principles for some of these types. In particular, it can derive induction principles for inductive families defined according to Dybjer's 1994 scheme [Dyb94]. The code that implements this feature is around 250 lines of dense, complex Haskell.

With the reflected elaborator, it is possible to define eliminator derivation as part of the Pruviloj library, and use it to implement an induction tactic. While this is useful for automating the construction of proofs as well as some programs, it is also a good test case for a programming tool because an induction principle is the most general operation that can be performed on a datatype. If it is possible to derive an eliminator, then we should expect to be able to derive any other program of interest. Furthermore, it is sometimes easier to simply apply the eliminator than it is to generate a fresh helper function during some other code generation task.

The approach that we have taken to derive the eliminator is:

- Convert the representation of type constructors to one in which the internal names have been made unique. The arguments are already tagged by their role as either parameters or indices.
- Convert the representation of data constructors into one in which the arguments representing parameters to the family have the same names as in the type constructor, and all other fields have been made unique.
- Compute the type of the eliminator, and declare it.
- For each processed constructor, derive a pattern-match case. Add the definition to the global context.

Each of these steps is described below.

In an Idris datatype, there is no syntactic distinction between parameters and indices. Instead, their role is inferred by the compiler when it inspects their mode of use in the constructors. This arrangement allows greater freedom in the ordering of type constructor arguments, which can be convenient for type class resolution.

In the following highly-explicit definition of `Vect`, the binding sites of parameters in the type constructor and in each data constructor have been

underlined. Note that the parameters need not occur in any particular position.

```
data Vect : Nat → Type → Type where
  Nil : {a : Type} → Vect Z a
  (::) : {n : Nat} → {a : Type} →
         (x : a) → (xs : Vect n a) →
         Vect (S n) a
```

Preprocessing the Type Constructor and Constructors

For purposes of deriving, a type constructor is represented by the following record:

```
record TyConInfo where
  constructor MkTyConInfo
  args : List TyConArg
  result : Raw
```

`TyConArg` is the datatype described in Chapter 5. The expectation is that all instances of this record will have uniquely named type constructor arguments, but this invariant is not guaranteed in the type. This is because the underlying `Raw` representation of TT does not provide sufficient flexibility to define custom representations of bound variables. While it would be possible to work around this by cleverly indexing the type of lists, it would complicate the code. Exploring the tradeoffs between highly dependent representations and simple, ML-style datatypes for reflection is left to future work.

Preprocessing of constructors is a similar process. Because Idris's reflection API distinguishes between the arguments to constructors that are parameters to the family and those that are not, the only thing that needs to be done is to ensure consistency of naming between them and the `TyConInfo`.

Because the code that performs these operations is a completely ordinary functional program, it is elided here.

Computing the Eliminator's Type

The eliminators constructed with this library take the following arguments:

- the parameters, which are quantified over the whole eliminator;

```
vectElim : (a : Type) →
           (n : Nat) → (xs : Vect n a) →
           (motive : (n' : Nat) → Vect n' a → Type) →
           (nil : motive Z Nil) →
           (cons : (n' : Nat) →
                   (y : a) →
                   (ys : Vect n' a) → motive n' ys →
                   motive (S n') (y :: ys)) →
           motive n xs
```

Figure 9.7: The type of the eliminator for `Vect`.

- the instance being eliminated, with its indices, here referred to as the *scrutinee*;
- the motive, which explains the context targeted by the eliminator; and
- the methods, which explain how to achieve the motive for each potential constructor of the scrutinee, given the inductive hypotheses.

For example, the type of the eliminator of the previous definition of `Vect` is given in Figure 9.7.

Deriving Pattern-Match Cases

Elaboration of the type occurs in a hole with goal type `⋆`, using a recursively-invoked elaboration script to ensure that it is valid in the top-level environment. The elaborator for the eliminator type is listed in Figure 9.8. The first step, binding the parameters, is performed with the simple operation `bindParams`. Because the pre-processing of the type constructor and the constructor ensured that the parameters have the same name across all instances, it is sufficient to bind each of them with `forall`. No `attack` is necessary, because we are aware of the shape of the proof term, having just created it.

```
bindParams : TyConInfo → Elab ()
bindParams info = traverse_ (uncurry forall) (getParams info)
```

Next, the scrutinee is quantified. This is done using the `bindTarget` operator, which computes fresh names for all indices of the datatype along

```
getElimTy : TyConInfo → List (TTName, List CtorArg, Raw) → Elab Raw
getElimTy info ctors =
  do ty ← runElab `(Type) $
             do bindParams info
                (scrut, iren) ← bindTarget info
                motiveN ← gensym "P"
                motiveH ← newHole "motive" `(Type)
                forall motiveN (Var motiveH)
                focus motiveH
                elabMotive info

                for_ {b=()} ctors $ \(cn, cargs, cresty) ⇒
                  bindMethod info motiveN cn cargs cresty
                let ret = mkApp (Var motiveN)
                                (map (Var . fst)
                                     (getIndices info) ++
                                 [Var scrut])
                apply (alphaRaw iren ret) []
                solve
     forget (fst ty)
```

Figure 9.8: Elaborating the Eliminator Type.

with a renaming from the previous index names. It then binds each index using `forall` and then binds the scrutinee as the type constructor applied to the global parameters and the renamed indices. This renaming is saved for later when it will be used in the conclusion of the eliminator's type. Here, `alphaRaw` is a function that applies a renaming to a `Raw` term.

```
bindIndices : TyConInfo → Elab Renamer
bindIndices info = bind' (getIndices info) noRenames
  where bind' : List (TTName, Raw) → Renamer → Elab Renamer
        bind' []              ren = return ren
        bind' ((n, t) :: ixs) ren = do n' ← nameFrom n
                                       forall n' (alphaRaw ren t)
                                       bind' ixs (extend ren n n')
```

```
bindTarget : TyConInfo → Elab (TTName, Renamer)
bindTarget info = do ren ← bindIndices info
                     tn ← gensym "target"
                     forall tn (alphaRaw ren $ result info)
                     return (tn, ren)
```

The next step is to bind the motive. Because its type must be computed, a new hole (`motiveH`) is established, and the helper script `elabMotive` is used to populate it. This script once again quantifies over the indices as well as an instance of the constructor applied to the parameters and indices. Because the result application is explicitly represented in `TyConInfo`, all that needs to be done is to rename the indices.

```
elabMotive : TyConInfo → Elab ()
elabMotive info = do attack
                     ren ← bindIndices info
                     x ← gensym "scrutinee"
                     forall x (alphaRaw ren $ result info)
                     fill `(Type)
                     solve
                     solve
```

To construct the method types, the `bindMethod` helper is applied to each pre-processed constructor in turn. It constructs a hole to receive the type of the method, binds it, and then fills the hole using `elabMethodTy`.

```
bindMethod : TyConInfo →
             (motiveName, cn : TTName) →
             List CtorArg → Raw → Elab ()
bindMethod info motiveName cn cargs cty =
  do n ← nameFrom cn
     h ← newHole "methTy" `(Type)
     forall n (Var h)
     focus h; elabMethodTy info motiveName cargs cty (Var cn)
```

The helper `elabMethodTy` constructs the type of the method corresponding to a particular constructor. It iterates over the list of constructor arguments, emitting `forall` bindings as it encounters constructor fields and accumulating the final application of the constructor to these bound variables that the method must demonstrate the motive for. Additionally, at each field, the procedure invokes `mkIH`, which checks whether the current

argument is an instance of the family being eliminated and, if so, additionally emits a binding of the relevant inductive hypothesis. At the end of the argument list, the hole is filled with the application of the motive. It is only explicitly applied to one argument — the accumulated application of the constructor to the bound variables — and its other arguments are all solved using Idris's unifier, as the indices forced by the constructor's definition are all apparent in its type.

```
elabMethodTy : TyConInfo →
               TTName → List CtorArg →
               (res, ctorApp : Raw) → Elab ()
elabMethodTy info motiveName [] res ctorApp =
  do argHoles ← apply (Var motiveName)
                      (replicate (length (getIndices info)) True ++
                       [False])
     argH ← last argHoles
     focus argH; fill ctorApp; solve
     solve
elabMethodTy info motiveName (CtorParameter arg  :: args) res ctorApp =
  elabMethodTy info motiveName args  res (RApp ctorApp (Var (name arg)))
elabMethodTy info motiveName (CtorField arg :: args) res ctorApp =
  do let n = name arg
     let t = type arg
     attack; forall n t
     mkIh info motiveName n t (result info)
     elabMethodTy info motiveName args res (RApp ctorApp (Var n))
     solve
```

Finally, the eliminator concludes with its conclusion: that the motive applied to the scrutinee must indeed hold. The hole is finally filled and solved with precisely that application.

Generating Pattern-Match Clauses

As described in Chapter 4, when Idris functions are elaborated, the left-hand side is first elaborated into the constructor of a datatype that aids in type inference. After this process, remaining holes are converted to pattern variables. Then, this type is used as the goal for the right-hand side. The helper `elabPatternClause` implements this pattern. It first establishes the application of the constructor of `Infer`, then executes the tactic script that will produce the left-hand side. Finally, all remaining holes are con-

```
elabPatternClause : (lhs, rhs : Elab ()) -> Elab FunClause
elabPatternClause lhs rhs =
  do (pat, _) <- runElab `(Infer) $
                 do th <- newHole "finalTy" `(Type)
                    patH <- newHole "pattern" (Var th)
                    fill `(MkInfer ~(Var th) ~(Var patH))
                    solve
                    focus patH
                    lhs
                    for_ {b=()} !getHoles $ \h =>
                      do focus h; patvar h
     (pvars, `(MkInfer ~rhsTy ~lhsTm)) <- extractBinders <$>
                                          forget pat
        | fail [TextPart "Couldn't infer type of LHS"]
     rhsTm <- runElab (bindPatTys pvars rhsTy) $
              do repeatUntilFail bindPat <|> return ()
                 rhs
     realRhs <- forget (fst rhsTm)
     return $ MkFunClause (bindPats pvars lhsTm) realRhs
```

Figure 9.9: Helper function for elaborating pattern clauses.

verted to pattern variables. If this process was successful, the resulting term will be a collection of pattern variables bound around the application of `MkInfer` to the type and the left-hand side. This type, under the same pattern binders, is used as a goal for the script that will produce the right-hand side. Finally, the clause consisting of the left- and right-hand sides is returned.

The Induction Tactic

The induction tactic is now straightforward to implement. The scrutinee is type checked, and an eliminator is generated for the family if necessary. Then, the goal type is generalized over the term to be eliminated by replacing all references to it and its indices by variables, and these are λ-bound to produce the motive. A hole is generated for each method, and the eliminator is applied to the scrutinee along with the motive. The method holes are subsequently returned from the tactic so that the caller can solve them.

9.5.2 Deciding Equality

To show that a property is decidable is to provide a function that either provides a witness for the property or a witness for its negation, across the domain of interest. In Idris, this is captured in the family `Dec`:

```
data Dec : Type → Type where
  Yes : {A : Type} → A → Dec A
  No : {A : Type} → (A → Void) → Dec A
```

The notion of decidable equality is represented in Idris by the type class `DecEq`:

```
class DecEq a where
  total decEq : (x, y : a) → Dec (x = y)
```

An instance of `DecEq` t provides a verified means of checking whether any two elements of type t are propositionally equal. Just as a Haskell library developer should provide `Show`, `Read`, `Ord`, `Eq`, `Functor`, and other ordinary instances, Idris library developers should provide instances of `DecEq` for their datatypes.

However, these instances are highly tedious to write. The definition must pattern-match both of its arguments, resulting in n^2 cases for a datatype with n constructors. In each case where the constructors do not coincide, their disjointness must be witnessed by appealing to the fact that reflexivity is trivially impossible. When the constructors do coincide, then each field of the constructors must be checked pairwise for equality. When the two constructor fields are determined to be equal, then the equality is used to make progress towards the eventual equality of the entire terms being matched. When two corresponding constructor fields are determined to be not equal, that fact can be combined with the injectivity of constructors to show that the entire terms being matched must not be equal. This process is formulaic, and there is no room for interesting deviations. Figure 9.10 contains an example of a hand-written decidable equality procedure for a simple binary tree type.

The formulaic, tedious nature of these functions makes them into a perfect candidate for automation. Because we are willing to accept *any* total function that has the correct type, it makes sense to automate large portions of the code generation using tactics and searching. The strategy employed is to:

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)

injBranch : Branch l x r = Branch l' x' r' ->
            (l = l', x = x', r = r')
injBranch Refl = (Refl, Refl, Refl)

instance DecEq a => DecEq (Tree a) where
  decEq Leaf Leaf = Yes Refl
  decEq Leaf (Branch l x r) = No (\(Refl) impossible)
  decEq (Branch l x r) Leaf = No (\(Refl) impossible)
  decEq (Branch l x r) (Branch l' x' r') with (decEq l l')
    decEq (Branch l x r) (Branch l' x' r')
      | No contra =
          No (\h => case injBranch h of
                      (prf, _, _) => contra prf)
    decEq (Branch l x r) (Branch l x' r')
      | Yes Refl with (decEq x x')
        decEq (Branch l x r) (Branch l x' r')
          | Yes Refl | No contra =
              No (\h => case injBranch h of
                          (_, prf, _) => contra prf)
        decEq (Branch l x r) (Branch l x r')
          | Yes Refl | Yes Refl with (decEq r r')
            decEq (Branch l x r) (Branch l x r')
              | Yes Refl | Yes Refl | No contra =
                  No (\h => case injBranch h of
                              (_, _, prf) => contra prf)
            decEq (Branch l x r) (Branch l x r)
              | Yes Refl | Yes Refl | Yes Refl =
                  Yes Refl
```

Figure 9.10: A hand-written procedure for deciding equality of binary trees.

- Generate all n^2 pairs of constructors
- For each constructor pair, universally quantify over all constructor arguments with pattern variables, relying on the elaborator's unifier to ensure that constructor fields forced by the types are filled out appropriately
- On the right hand side of each case, inspect the type that occurs. If it expects a decidability result for the equality of disjoint constructors, construct a disjointness lemma and use that together with the `No` constructor. If it expects a decidability result for the equality of the same constructor, then examine the fields pairwise, performing induction on the result of comparing the fields for equality.

Our implementation of decidable equality derivation requires the user to specify a type signature that provides the type class constraints that will be in scope. This means that elaborator reflection is only used to derive the cases. Each case is produced using the `elabPatternClause` helper defined in the previous section. In cases where the two constructors are not the same, the `disjoint` tactic is used to produce a helper function that serves as the argument to the `No` constructor.

Cases in which the same constructor is applied are more interesting. The first step is to construct the sequence of equality tests that will eventually either prove or disprove the equality of the terms. For each matching pair of arguments, either the generated function is called recursively or `decEq` is applied. Then, the resulting terms are destructed using the induction tactic from the previous section. The method for the `Yes` constructor rewrites with the contained equality proof, advancing the state of the goal, while the method for the `No` constructor exploits an injectivity tactic to get the assumption in scope that is necessary to transfer the negative result for the equality of the arguments to be a negative result for the entire equality.

Figure 9.11, which contains part of the derivation code, demonstrates how this tactic-driven approach to code generation is written using Pruviloj. The first definition in the figure is `noCase`, a tactic for dispatching the case when one of the fields does not match. As an argument, it takes the name of the in-scope witness that they do not match, which will be discovered by performing induction on the result of the call to `decEq`. First, it applies the `No` constructor, and then focuses on the argument. Because the argument must witness a negation, and thus be a function, it introduces

```
noCase : TTName -> Elab ()
noCase contra =
  do [_, nope] <- apply (Var `{No}) [True, False]
       | _ => fail [ TextPart "Bad holes from"
                   , NamePart `{Tactics.apply}
                   ]
     solve
     focus nope

     h <- gensym "h"
     inj <- gensym "inj"
     attack
     intro h
     injective (Var h) inj
     unproduct (Var inj)
     ignore $ refine (Var contra) `andThen` hypothesis
     solve

matchCase : List Raw -> Elab ()
matchCase []          = search
matchCase (tm :: tms) =
    do (y :: n :: _) <- induction tm

       focus n; compute
       contra <- gensym "contra"
       attack; intro contra
       noCase contra; solve

       focus y; compute
       prf <- gensym "prf"
       attack; intro prf
       rewriteWith (Var prf)
       matchCase tms
       solve
```

Figure 9.11: Tactic script to dispatch matching constructor cases when deriving equality decision procedures.

as a hypothesis `h` that the two applications of the constructor are equal. It then uses the `injective` tactic from Pruviloj to let-bind a product of the pairwise equalities of the constructor arguments in the assumption, and `unproduct` to bring all the elements of this product individually into scope. Finally, it refines the goal by the witness that one of the arguments doesn't match, and uses the `hypothesis` tactic to select the correct equality.

When the pattern match is being generated, `matchCase` is used when the implementation of `decEq` is being applied to two terms with the same constructor at their head. The tactic is called with a list of terms that is the pairwise combination of the constructor arguments with `decEq` — that is, it consists of the subgoals that need to be checked. If there are no further arguments to compare, then the result must be true, so the `search` tactic will solve it with `Yes Refl`. If there is an argument, the `induction` tactic is applied, resulting in obligations for the case where the equality holds and the case where it does not. When the equality does not hold, the `noCase` tactic is sufficient. When it does, rewriting with the equality brings the derivation procedure one step closer to that final `search`.

This implementation is very different in character from the derivation of eliminators. It illustrates a different mode of use of the reflected elaborator, relying to a much greater extent on search and automation. This style of code generation is much more useful as our types increase in specificity.

9.5.3 Proof Automation

Some very simple tactics can take care of a large variety of proofs. In this section, we demonstrate a simple tactic that can nevertheless prove many of the theorems that are included in Idris's standard library about the functions that it exposes. Additionally, this demonstrates that partial tactics can produce useful total definitions.

Figure 9.12 contains the complete source code to this tactic. The `auto` tactic normalizes the goal, introduces as many times as possible, and then attempts to rewrite with every equality in scope. It then checks if any variable in scope solves the goal, and if none do, it invokes Idris's built-in proof search. Then, a tactic `autoInduction` performs induction on the first thing in scope using `auto` to solve each of the returned holes.

This short tactic is able to prove many easy theorems, including the associativity of addition on `Nat`, the left and right identity rules for multi-

```
partial
auto : Elab ()
auto = do compute
          attack
          try $ repeatUntilFail intro'
          hs ← map fst <$> getEnv
          for_ hs $
            \ih ⇒ try (rewriteWith (Var ih))
          hypothesis <|> search
          solve


partial
autoInduction : Elab ()
autoInduction =
    do n ← gensym "n"
       intro n
       try intros
       ignore $ induction (Var n) `andThen` auto
```

Figure 9.12: A simple automation procedure for proofs by induction.

plication, that `map` on `List` preserves the length of the list, that the length of two appended lists is the sum of their lengths, and more.

Additionally, elaborator reflection provides the tools necessary to make ordinary proof by reflection convenient. This example, a monoid equality solver, roughly follows the structure of the one given by Chlipala [Chl11], though it has been adapted quite heavily to Idris.

Figure 9.13 defines a representation of monoids, along with three instances. The first step in solving equalities of monoid expressions is to construct a reflected expression type. this type has three constructors, corresponding to the `neut` and `op` methods of the `IsMonoid` type class as well as arbitrary other expressions.

```
data MonoidExpr a =
  NEUT | VAR a | OP (MonoidExpr a) (MonoidExpr a)
```

These reified monoid expressions can be straightforwardly interpreted as actual monoid expressions by interpreting each constructor using the corresponding method of the `IsMonoid` class.

```
class IsMonoid a where
    neut : a
    op : a -> a -> a
    neutLeftId : (x : a) -> neut `op` x = x
    neutRightId : (x : a) -> x `op` neut = x
    assoc : (x, y, z : a) -> op x (op y z) = op (op x y) z

instance IsMonoid () where
    neut = ()
    op () () = ()
    neutLeftId () = Refl
    neutRightId () = Refl
    assoc () () () = Refl

instance IsMonoid Nat where
    neut = Z
    op = plus
    neutLeftId _ = Refl
    neutRightId = plusZeroRightNeutral
    assoc = plusAssociative

instance [multMonoid] IsMonoid Nat where
    neut = 1
    op = mult
    neutLeftId = multOneLeftNeutral
    neutRightId = multOneRightNeutral
    assoc = multAssociative
```

Figure 9.13: A representation of monoids with laws that enable instances to be used for proof by reflection.

```
interpExpr : (IsMonoid a) => MonoidExpr a -> a
interpExpr NEUT = neut
interpExpr (VAR x) = x
interpExpr (OP x y) = op (interpExpr x) (interpExpr y)
```

The reflection proof technique is to show that these expressions can be flattened to a normal form, namely lists, and then show that if two expressions have equal normal forms then they also have equal interpretations. This reduces the problem of proving equalities between monoid expressions to the problem of proving equalities between lists.

We interpret this normal form of lists by folding it using the monoid operations:

```
interpList : (IsMonoid a) => List a -> a
interpList xs = foldr op neut xs
```

Likewise, flattening a monoid expression to a list is a straightforward walk over the tree:

```
flattenExpr : MonoidExpr a -> List a
flattenExpr NEUT = []
flattenExpr (VAR x) = [x]
flattenExpr (OP x y) = flattenExpr x ++ flattenExpr y
```

It is not difficult to show that flattening and interpretation commute. This is demonstrated in Figure 9.14.

Now, given a representation of a proof goal as two `MonoidExpr`s, we can simplify it automatically, with full confidence. Unfortunately, it is exceedingly tedious to write these `MonoidExpr`s by hand. Luckily, this reification can be automated using reflection. The first step is to write the reification procedure, which pattern matches on a monoid term and constructs the corresponding `MonoidExpr`. This procedure is given in Figure 9.15.

Then, the `asMonoid` tactic reflects on the current goal type, reifying each side. This tactic is defined in Figure 9.16. As seen in Section 9.4, the tactic `remember` `let`-binds the result of an elaboration script, while `equiv` replaces the goal with a syntactically different but convertible goal. Having rewritten the proof goal to an equality of the interpretation of two monoid expressions, the monoid reflection theorem is then used to rewrite the proof in terms of the flattened form. To ensure that spurious differences are not introduced into the equality by type class resolution, `asMonoid` takes a reflected dictionary as an argument and ensures that it is used consistently.

```
opAppend : (IsMonoid a) => (xs, ys : List a) ->
           op (interpList xs) (interpList ys) =
           interpList (xs ++ ys)
opAppend [] ys = neutLeftId _
opAppend (x :: xs) ys =
  rewrite sym $ opAppend xs ys in
  sym $ assoc x (interpList xs) (interpList ys)


flattenOk : (IsMonoid a) =>
            (e : MonoidExpr a) ->
            interpExpr e = interpList (flattenExpr e)
flattenOk NEUT = Refl
flattenOk (VAR x) = sym $ neutRightId x
flattenOk (OP x y) =
    rewrite flattenOk x in
    rewrite flattenOk y in
    opAppend (flattenExpr x) (flattenExpr y)


monoidReflection : (IsMonoid a) =>
                   (x, y : MonoidExpr a) ->
                   interpList (flattenExpr x) =
                   interpList (flattenExpr y) ->
                   interpExpr x = interpExpr y
monoidReflection x y prf =
    rewrite flattenOk x in
    rewrite flattenOk y in
    prf
```

Figure 9.14: Flattening and interpretation of monoid expressions commute.

```
reifyExpr : Raw → Elab ()
reifyExpr `(op {a=~a} @{~dict} ~x ~y)
    do [l, r] ← apply `(OP {a=~a})  [False, False]
       solve
       focus l; reifyExpr x
       focus r; reifyExpr y
reifyExpr `(neut {a=~a} @{~dict}) =
    do fill `(NEUT {a=~a})
       solve
reifyExpr tm =
    do [_, h] ← apply (Var `{VAR}) [True, False]
       solve
       focus h; fill tm
       solve
```

Figure 9.15: Reification of monoid expressions to the custom expression type.

Now, somewhat complicated monoid expressions can be solved directly:

```
test1 : (IsMonoid a) ⇒
        (w, x, y, z : a) →
        (( w `op` x) `op` (y `op` z)) =
        (w `op` (x `op` (y `op` (z `op` IsMonoid.neut))))
test1 @{dict} w x y z = %runElab (do asMonoid (Var `{dict})
                                     reflexivity)
```

This is all well and good if one is trying to prove properties of monoids in general, but it's not very useful for other kinds of equalities that naturally arise during proof work. Expressions that are already expressed in terms of the underlying type need to have custom reification procedures to enable this technique. Writing a reification procedure for `Nat` and `plus` enables the following proof to be automated:

```
test2 : (x, y, z : Nat) →
        plus (plus (plus z 13) z) (plus x (plus y Z)) =
        z `plus` (13 `plus` ((z `plus` x) `plus` y))
test2 x y z = %runElab (do dict ← natPlusAsMonoid
                           asMonoid dict
                           reflexivity)
```

```
asMonoid : (dict : Raw) → Elab ()
asMonoid dict =
    case !goalType of
      `((=) {A=~A} {B=~B} ~e1 ~e2) ⇒
        do l ← gensym "L"
           r ← gensym "R"

           remember l `(MonoidExpr ~A); reifyExpr e1
           remember r `(MonoidExpr ~B); reifyExpr e2

           equiv `((=) {A=~A} {B=~B}
                       (interpExpr {a=~A} @{~dict} ~(Var l))
                       (interpExpr {a=~B} @{~dict} ~(Var r)))

           [h] ←
             apply `(monoidReflection {a=~A} @{~dict}
                                      ~(Var l) ~(Var r))
                   [True]

           solve
           focus h
```

Figure 9.16: A tactic for simplifying equalities of monoid expressions using reflection.

The reification procedure returns the dictionary that it used to reify the expression.

9.6 Agda-Style Reflection

Like Idris's implementation of reflection, Agda's reflection API (described in Section 3.3.1) uses an untyped representation of terms, relying on a later type checking pass to ensure that metaprograms have generated meaningful programs. However, this reflection system nonetheless occupies a quite different region of the design space than Idris's elaborator reflection. First off, Agda's reflection system is not modeled as a collection of compile-time side effects, but rather as a fixed collection of metaoperators for quoting and splicing. Secondly, Agda reflection uses high-level

Agda terms for metaprogramming, rather than a low-level core language. This is part of what seemingly allows it to avoid side effects, because it becomes possible to use high-level Agda features, such as hidden and instance arguments. These features are then resolved in the ordinary fashion when reflected terms or definitions are spliced. Thirdly, Agda's reflection system only represents the normal form of terms. This allows its users to ignore that the same underlying open term can have many different syntactic representations, and only worry about one of them. On the other hand, it can necessitate manually running computation backwards in some cases, where a non-normalized term is more convenient to work with. Finally, Agda's quotation mechanism is not as flexible as the quasiquotations we describe in Chapter 7. To compensate for the lack of quasiquotation pattern matching, pattern synonyms are frequently used to make matching against reified terms look more like matching against their high-level syntax. Additionally, van der Walt [vdW12] implemented an automatic quotation library that, given a declarative description of how to map Agda terms to constructors of some expression type, will generate the mapping itself, after which a convenient, domain-specific type can be used.

Agda's reflection mechanism provides the following operators:

- `quote` *n* quotes a global name *n* to its abstract reflected representation. It is an error to quote a locally-bound name.
- `quoteTerm` *e* quotes the normal form of *e* to the `Term` datatype.
- `unquote` *e* splices the result of evaluating *e*.
- `quoteGoal` *x* `in` *e* binds the name *x* to the reflected representation of the type that is expected for the term in the scope *e*.
- `quoteContext` *x* `in` *e* binds the name *x* to a list of the reflected names that are available in scope.
- `unquoteDecl` *d* splices a quoted declaration *d*
- `unquoteDef` *cs* splices a list of quoted function clauses *cs* as the definition of an already-declared function
- `tactic` *e* is an abbreviation for `quoteGoal` *x* `in unquote (`*e x*`)`.

- `tactic` e `|` e_1 `|` … `|` e_n is an abbreviation for `quoteGoal` x `in unquote` (e x) e_1 … e_n. The intention is that `unquote` (e x) will produce a function, whose arguments correspond to subgoals to be filled by e_1 … e_n.

Additionally, primitive operators are provided to allow reflection on definitions and to look up the type of a name.

Because Agda's reified terms represent the high-level language and can contain features such as unresolved implicit arguments, splicing them can have effects beyond the terms themselves by modifying the global metavariable context. While these operators may appear to be pure, they have effects nonetheless.

A simple Agda tactic to solve trivial goals, corresponding to the one built in Section 9.1.2, is defined in Figure 9.17. Instead of using van der Walt and Swierstra's AutoQuote library [WS12], the example achieves readable pattern matching on terms by defining pattern synonyms that syntactically resemble the term whose quotation is being matched. The ellipsis `...` is Agda notation for repeating the pattern that is being refined using the `with` rule. Like the example Idris tactic, this tactic computes an inhabitant for a small collection of datatypes. It can be employed using the `tactic` syntax:

```
test : ⊤ × ℕ × Either ⊤ ⊥ × Either ⊥ ⊤
test = tactic trivial
```

Normalizing `test` yields the value `tt , zero , left tt , right tt`.

It is possible to implement a large subset of Agda's reflection API using the reflected elaborator. In particular, the following operators can be implemented: `quote`, `quoteTerm`, `unquote`, `tactic`, `quoteGoal` and `quoteContext`. The `quote` operator already exists as a language feature — it is the quoted name syntax described in Section 7.5. The `quoteTerm` operator can be defined as an elaborator script that quotes from Idris's own core language to an implementation of an Agda-style term type. This must be combined with a syntax rule that `let`-binds the term to be quoted and then runs an elaborator script to access the bound term and fill the current hole with it's quotation.

Like `quoteTerm`, the operators `quoteGoal` and `quoteContext` can be implemented by using a syntax rule to let-bind the body of the quotation operator, lambda-abstracted over the variable to which the quotation is

```
pattern `⊤ = def (quote ⊤) []
pattern `ℕ = def (quote ℕ) []
pattern _`×_ a b =
    def (quote _×_) (arg _ a ∷ arg _ b ∷ [])
pattern `Either a b =
    def (quote Either) (arg _ a ∷ arg _ b ∷ [])

trivial' : Term → Maybe Term
trivial' `⊤ = just (quoteTerm tt)
trivial' `ℕ = just (quoteTerm zero)
trivial' (a `× b) with trivial' a | trivial' b
... | just x  | just y  =
    just (con (quote _,_)
              (arg (arg-info visible relevant) x ∷
               arg (arg-info visible relevant) y ∷ []))
... | just x  | nothing = nothing
... | nothing | _       = nothing
trivial' (`Either a b) with trivial' a
... | just x =
    just (con (quote left)
              (arg (arg-info visible relevant) x ∷ []))
... | nothing with trivial' b
...   | just x =
    just (con (quote right)
              (arg (arg-info visible relevant) x ∷ []))
...   | nothing = nothing
trivial' _ = nothing

record Failure : Set where
  constructor ItFailed

attempt : Maybe Term → Term
attempt (just x) = x
attempt nothing = quoteTerm ItFailed

trivial : Term → Term
trivial = attempt ∘ trivial'
```

Figure 9.17: A simple tactic in Agda's reflection system.

applied, and then applying this function to the reified information that results from an elaborator script. However, it is not possible to use `quoteGoal` together with `unquote` to implement `tactic` directly as syntactic sugar in the manner that it is done in Agda. This is because of the elaborator side effects that can result. The meta-operations must be composed monadically inside of the same **`%runElab`**.

We have not attempted to implement `quoteDecl` or `unquoteDecl`, but it should be a matter of finding the correct representation and then constructing it from Idris's internal reflection datatypes. These datatypes already contain enough information.

Because Agda-style reflected terms include features like implicit arguments, the `unquoteTerm` operator requires a part of the Idris term elaborator to be re-implemented to arrange for these features to be solved. Luckily, the infrastructure for this already exists: it was the original purpose for which the elaborator was designed.

9.7 Reflections on Elaborator Reflection

How can we situate Idris's elaborator reflection in the design space for reflection and metaprogramming mechanisms in general? How does it compare to others used for dependent types? This section discusses how we might answer these questions, contextualizing the reflected elaborator in the design landscape.

9.7.1 Safety

Reflected elaborator scripts need not pass Idris's totality checker. They are allowed to contain infinite loops and they are allowed to have non-covering pattern matches. Thus, they can crash while running and they introduce non-termination into the elaboration process. However, they cannot undermine the safety of Idris in general. This is because the result of elaboration must still pass the type checker and the totality checker in order to be accepted by Idris. If elaboration terminates, then the result will be checked as usual.

Even though the resulting programs are safe, non-termination at compile time can still be irritating. However, just as with run-time programs, users still have access to Idris's totality checker and are able to use it to ensure that metaprograms terminate.

Additionally, a termination analysis is a bit of a red herring. While it is important for the consistency of a logic, reflected elaboration scripts are not proofs: they are programs that are used to construct proofs. Even in strongly normalizing systems, it is possible to write programs with execution times that exceed the users' patience. In practice, the aim of a terminating compiler is perhaps better achieved through a timeout or a limit on the number of reduction steps that are allowed.

9.7.2 Datatypes

Elaborator reflection is an open-ended system. Additional effects can be added in the future by extending the interpreter. Thus, it need not be able to account for all desired use cases immediately. Nevertheless, it is instructive to evaluate its suitability for a variety of tasks, and there is one important feature that it presently lacks: the ability to define new datatypes.

This would allow the automated construction of Bove-Capretta predicates [BC03]. For this to work for nested recursive functions, it would need

to be possible to define inductive-recursive families as well as ordinary inductive families. This can be achieved by having a two-step process similar to the one used for adding function definitions. First, `declareDatatype` would take a family name and a representation of the type constructor arguments, just like `declareType`. Then, `defineConstructors` would define the type using its constructors. The current reflected datatype definition described in Chapter 5 is not suitable for this task, as it would require the user to identify details such as parameters and indices. This is a task better left to a machine.

9.7.3 Direct vs. Indirect

Barzilay [Bar06] distinguished between direct and indirect reflection mechanisms, where a direct reflection of some aspect of a language exposes the surrounding system's own implementation of a feature to itself, and an indirect reflection reimplements it internally. Barzilay points out that direct reflections are typically far shorter, not requiring heroic efforts to implement simple features, although their internals are opaque and thus not customizable by clients of the reflection system. There is also no need to worry that a direct reflection does not agree with the system, while indirect reflection runs the risk of not accurately modeling its host. Finally, indirect reflection can lead to an increase in the space needed to represent a term that is exponential with regards to the quotation level.

The elaborator reflection mechanism described in this chapter is primarily an instance of direct reflection, though its term representation datatypes are implemented indirectly. A disadvantage of direct reflection is that the language's ordinary means of constructing and destructing datatypes cannot be used, and additional primitives must be provided. The indirect representation of datatypes allows them to be used with ordinary Idris pattern matching, while the elaboration of quasiquotations described in Chapter 7 removes the exponential syntactic overhead of an indirect representation. The exponential increase in size is not a major problem, because real metaprograms use only a single level of quotation in practice. Additionally, TT is a very simple language that changes only infrequently, lessening the maintenance burden of indirect reflection. Finally, the use of an indirect representation of reified terms means that the core language did not need to be extended at all in order to implement elaborator reflection, which means that it cannot undermine the safety of the system, unlike the direct reflection used in Brady's experimental

typed reflection [Bra13a].

When metaprograms written using Idris quasiquotations and elaborator reflection are contrasted with the type-safe metaprogramming described by Devriese and Piessens [DP13], the advantages of direct reflection become apparent: deriving `Show` for `Nat` took approximately 1200 lines of Agda code in their framework, much of which was occupied by a proof obligation, and their description of datatypes supports only a small fraction of the datatypes supported by Agda, ruling out things like parameters, indices, and non-recursive constructor arguments. A similar program written using elaborator reflection and the Pruviloj library requires only around 300 lines of code, and it supports many more datatypes. Furthermore, Devriese and Piessens's description of datatypes does not take into account features that are important for actual implementations of `Show`, such as ensuring correct placement of parentheses and hiding implicit arguments.

Using the reflected elaborator mechanisms for proof automation further underscores the benefits of a direct reflection. While Kokke and Swierstra [KS15] needed to implement features such as unification themselves, automation procedures in Idris can simply re-use the existing unifier, automatically benefiting from improvements.

9.7.4 Typing and Reflection

The only reason we have to think that an element of `TT` represents a well-typed term is that it was generated by Idris's type checker, but that information about its provenance is not represented anywhere. The only way to ensure that a metaprogram has done what it should is to run it and check the result. Some metaprogramming systems use type annotations on reflected terms to ensure that metaprograms generate well-typed terms, including MetaML [TS00] and F#'s code quotations [Sym06]. The structure of these typed term representations is not made available to users, and they rely on the surrounding system's implementation of parsing and type checking. Thus, they are also instances of direct reflection. The term representation given by Devriese and Piessens [DP13] can be seen as another example of the well-typed term approach to metaprogramming, this time implemented as indirect reflection. In the world of proof assistants, tactic metaprogramming systems such as VeriML [SS10] and MTac [Zil+13] make guarantees about the types of goals that metapro-

grams are able to solve, while most other tactic languages (including Coq's LTac [Del00]) do not make these kinds of guarantees.

While Idris's elaborator reflection makes use of Idris's type system to prevent mistakes such as confusing a name with a term and nothing prevents users from using more advanced types in their own tactics, the reflected elaborator does not use types to rule out metaprograms that do not guarantee their results to be well typed. In some sense, this is a matter of taste: should metaprogramming provide a means of escaping a type system when it becomes inconvenient, or should it provide new ways of employing a type system? Neither option is obviously superior.

Elaborator reflection has special support from the compiler. Thus, it occupies a privileged position in Idris. Any other metaprogramming system that we would like to implement must be expressible either directly in Idris or in the reflected elaborator. If the reflected elaborator made the kind of strong typing guarantees that are made by systems like MTac, then it would not be able to be used to implement tactic systems that do not. At the same time, because the reflected elaborator provides access to the type checker and evaluator, it can be used to implement typed metaprogramming or proof automation systems.

9.7.5 Nested Elaboration

It is presently impossible to "escape" back to the standard elaborator and make use of the elaborated form of an ordinary Idris subterm. The present situation is analogous to a quotation mechanism that does not support quasiquotations. The ability to escape to the main elaborator and capture its results would allow elaborator scripts to not only produce values, but also to transform them. For instance, elaborator reflection could become usable for type-aware, domain- and library-specific program transformations and optimizations in the style of HERMIT [Far+12], which has been used to implement stream fusion [FHG14], optimizing generic traversals [AFM14], and automating standard transformations from the literature [SFG13]. Additionally, it would allow the addition of arbitrary new binding forms to Idris, which the current system only allows to a limited extent. Finally, embedded languages would expand their FFI abilities to Idris beyond mere references to Idris names.

This could be implemented by enriching the reflection datatypes defined in Chapter 5 with a representation of terms in the fully desugared high-level Idris language, perhaps called `IdrisTerm`. Next, the quotation

mechanism described in Chapter 7 would need to be extended with support for non-elaborated quotations of this high-level term datatype. Finally, the `Elab` language would need a new primitive operator `elabIdris` with type `IdrisTerm → Elab ()` that would invoke the Idris elaborator on the embedded term. The semantics would be given by:

$$\mathcal{X}\,[\![\texttt{elabIdris}\ e]\!] = \mathcal{E}\,[\![\textsc{unquote}\ e]\!]$$

where UNQUOTE is a meta-operation that transforms a TT datatype that represents Idris terms into an Idris term suitable for the elaborator. This fairly heavyweight approach would allow scripts written in the reflected elaborator to defer to Idris's own elaborator.

Unfortunately, there are a few problems with this approach. Unlike TT, high-level Idris, even in its desugared form, is a quite complicated language that changes on a regular basis. Maintaining an Idris mirror of the datatype would add significantly to the maintenance burden. Additionally, quoted Idris terms would need to be able to refer to variables in their lexical environment. This means that the some of the transformations applied prior to elaboration, which can involve renaming bound variables, must also be applied within quotations; however, the elaborator script around the quotation might have shadowed the external variable. Thus, relying on quotations and an elaboration primitive to access Idris's own elaborator would drastically limit the source-to-source transformations that could be applied prior to elaboration. On the other hand, metaprograms that rely on explicit name capture are notoriously difficult to write correctly, so it might be even better to elaborate these quoted terms without access to the lexical environment, relying on explicit λ-abstractions inside the quotations to apply the names.

An alternative approach would be to use direct reflection to embed the Idris term elaborator in a reflected elaborator script. Because elaboration scripts are written in the high-level Idris language, they must be elaborated prior to execution in order to ensure that all the implicit details and overloading have been resolved. Just as implementing the elaborator required TT to be extended to a development calculus with holes and guesses, this approach would require extending TT with a new kind of term that represents a suspended elaboration and is computationally neutral according to TT's reduction semantics. Then, just as the elaborator's output is presently checked to ensure that there are no development calculus features remaining, it would also need to be checked for suspended elaborations, and if they are ruled out, then the core language as checked

would remain unchanged. This direct reflection avoids the complications of maintaining a reflected `IdrisTerm` datatype, and the lack of an analyzable concrete datatype for the high-level terms makes it feasible to impose a hygiene discipline on names.

9.7.6 Interrupting and Resuming the Elaborator

The reflected elaborator provides an imperative language for metaprogramming, proof automation, and developing elaborators for domain-specific languages. Because it inherits the general design of Idris's own elaborator, it is able to re-use features like the type checker and the unifier. At the same time, because its operations are segregated in the `Elab` type, it is easy to reason about when metaprogramming is occurring.

Due to its imperative nature, however, elaborator reflection gives up on the straightforward, compositional execution semantics of pure functional programming. Side effects are useful, but they are also difficult to reason about without additional tools. For reasons discussed in Section 9.3, the implementation of an otherwise pure language is not necessarily a friendly environment to implement the tools that make imperative programming tractable, such as breakpoints and logging. Additionally, the embedding of `Elab` within the expressions of a functional language makes it difficult to provide a step-by-step interface in the style of Proof General, so we have implemented such an interface external to source buffers.

9.7.7 Hint Databases

One useful feature found in many tactic-based proof assistants that has no counterpart in our reflected elaborator is the notion of an extensible hints database. In these systems, tactics have access to a bit of global state that instructs them how to interact with features that did not exist when the tactics were written. For instance, the list of names provided to the `byConstructors` tactic might be stored in such a database, and users could add new types to the list as they are defined. The MetaPRL proof assistant [Hic01] has an advanced notion of hint databases known as *resources*, which are conceptually lists of some particular type that are assembled by the system from the corresponding resource values of all instances of that resource in scope. This can be emulated by using Idris's *ad-hoc* overloading, but that lacks appropriate static type checks when hints are defined.

Chapter 10

Conclusions

This dissertation describes a collection of related static reflection mechanisms for Idris. While they were originally developed as an implementation technique for embedded domain-specific languages, they ended up being generally useful. Our error reflection is already being employed to improve the error messages in the Idris standard library, and type class deriving implemented with the reflected elaborator will soon drastically reduce the amount of boilerplate required from users. Using our elaborator reflection, compiler features implemented in Haskell have been replaced with Idris code, and Idris's limited, special-purpose tactic language has been replaced with `Elab` scripts that have access to the full power of Idris. In that respect, these features have been a success.

However, some difficulties remain. Limitations in the implementation of elaboration reflection lead to staging restrictions that expose the internal behavior of Idris's elaborator. Additionally, the elaboration effects are at a very low level of abstraction, and users have experienced difficulties managing the hole queue and the need for `attack`. Likewise, type class deriving is far too verbose, requiring a few hundred lines to implement deriving of a simple instance.

As the implementation of a version of Agda's **`tactic`** system shows, it is possible to build higher-level proof automation languages using the reflected elaborator. It would be good to explore the implementation of alternative systems like MTac [Zil+13] as well as to provide a full implementation of an Agda-style reflection system. Likewise, it would be nice to use a more advanced form of generic programming to implement features like type class deriving, using the reflected elaborator in a manner similar to how Norell and Jansson [NJ04] used Template Haskell to pro-

totype new generic programming systems.

One weakness of the work presented here is that it appears as multiple separate language features. In the future, their utility could increase if they were combined. For instance, if reflected errors could be handled in `Elab`, then they could do things like spell-checking unknown identifiers against the names in scope, and error handlers could choose whether to display normalized terms. Additionally, if the elaborator were extended to support actual side effects in `IO`, potentially by making it interruptable as suggested in Section 9.7.6 and allowing it to delegate to the main Idris compiler, it would be able to subsume Idris's type providers [Chr13a]. If type providers had access to `Elab`, then they could perform both ordinary execution and code generation.

The story of dependently typed programming is just beginning. Dependent types both enable and necessitate new ways of interacting with our programs and the environments within which they compute. The mechanisms presented in this dissertation show that we need not yield the reflective high ground to dynamically typed languages, and that simply typed reflection combined with a dependently typed language can provide the slack needed to place the burden of proof on the broadest shoulders.

Bibliography

[AFM14] Michael D. Adams, Andrew Farmer, and José Pedro Magalhães. "Optimizing SYB Is Easy!" In: *Proceedings of the 2014 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '14. San Diego, California, USA: ACM, 2014.

[Agda] The Agda Team. *The Agda Wiki*. Accessed 2015.

[All+90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. "The Semantics of Reflected Proof". In: *Proceedings of Fifth IEEE Symposium on Logic in Computer Science*. June 1990, pp. 95–105.

[AlS14] Ahmad Salim Al-Sibahi. *The Practical Guide to Levitation*. M.Sc. thesis, IT University of Copenhagen. 2014.

[AM03] Thorsten Altenkirch and Conor McBride. "Generic Programming Within Dependently Typed Programming". In: *Proceedings of IFIP TC2/WG2.1 Working Conference on Generic Programming*. 2003.

[Aug98] Lennart Augustsson. "Cayenne — a Language with Dependent Types". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, USA: ACM, 1998, pp. 239–250.

[Axe+10] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. "Feldspar: A domain specific language for digital signal processing algorithms". In: *Formal Methods and Models for Codesign*. MEMOCODE. IEEE, July 2010, pp. 169–178.

[Bar06] Eli Barzilay. "Implementing Reflection in Nuprl". PhD thesis. Cornell University, 2006.

[Baw99] Alan Bawden. "Quasiquotation in Lisp". In: *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. Ed. by Olivier Danvy. 1999, pp. 4–12.

[BC03] Ana Bove and Venanzio Capretta. "Modelling General Recursion in Type Theory". In: *Mathematical Structures in Computer Science* 15.4 (Mar. 2003).

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[Ben86] Jon Bentley. "Programming Pearls: Little Languages". In: *Communications of the ACM* 29.8 (Aug. 1986), pp. 711–721.

[BH12] Edwin Brady and Kevin Hammond. "Resource-Safe Systems Programming with Embedded Domain Specific Languages". In: *Practical Aspects of Declarative Languages*. Ed. by Claudio Russo and Neng-Fa Zhou. Vol. 7149. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 242–257.

[Bou+92] Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. "Experience with Embedding Hardware Description Languages in HOL". In: *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*. Amsterdam, The Netherlands: North-Holland Publishing Co., 1992, pp. 129–156.

[Bra13a] Edwin Brady. *First-class Type-safe Reflection in Idris*. Talk at the 2013 ACM SIGPLAN Workshop on Dependently Typed Programming. Sept. 2013.

[Bra13b] Edwin Brady. "Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation". In: *Journal of Functional Programming* 23.05 (2013), pp. 552–593.

[Bra13c] Edwin Brady. "Programming and Reasoning with Algebraic Effects and Dependent Types". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 133–144.

[Bra14] Edwin Brady. "Resource-Dependent Algebraic Effects". English. In: *Trends in Functional Programming*. Ed. by Jurriaan Hage and Jay McCarthy. Vol. 8843. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 18–33.

[Bur13] Eugene Burmako. "Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming". In: *Proceedings of the 4th Workshop on Scala*. Montpellier, France: ACM, 2013.

[C#E] Microsoft. *Expression Trees (C# and Visual Basic)*. `http://msdn.microsoft.com/en-us/library/bb397951.aspx`. Accessed August, 2014.

[CH00] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. ACM, 2000, pp. 268–279.

[Cha+10a] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. "Language Virtualization for Heterogeneous Parallel Computing". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. ACM, 2010, pp. 835–847.

[Cha+10b] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. "The Gentle Art of Levitation". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pp. 3–14.

[Cha09] James Chapman. "Type Theory Should Eat Itself". In: *Electronic Notes in Theoretical Computer Science* 228 (2009), pp. 21–36.

[Chl08] Adam Chlipala. "Parametric Higher-order Abstract Syntax for Mechanized Semantics". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, B.C., Canada: ACM, 2008, pp. 143–156.

[Chl11] Adam Chlipala. *Certified Programming with Dependent Types*. Available online: `http://adam.chlipala.net/cpdt/`. MIT Press, 2011.

[Chl13] Adam Chlipala. "The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 391–402.

[Chr+14] David Raymond Christiansen, Henning Niss, Klaus Grue, Kristján S. Sigtryggsson, and Peter Sestoft. "An Actuarial Programming Language for Life Insurance and Pensions". Presented at the International Congress of Actuaries, Washington, D.C., USA. 2014.

[Chr13a] David Raymond Christiansen. "Dependent Type Providers". In: *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*. WGP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 25–34.

[Chr13b] David Raymond Christiansen. "Software Development for the Working Actuary". In: *Proceedings of 4th International Symposium on End-User Development*. Ed. by Yvonne Dittrich, Margaret Burnett, Anders Mørch, and David Redmiles. Vol. 7897. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 266–271.

[Chr14a] David Raymond Christiansen. "Reflect on your mistakes! Lightweight Domain-Specific Errors". Unpublished manuscript. 2014.

[Chr14b] David Raymond Christiansen. "Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection". In: *Proceedings of the 26th International Symposium on Implementation and Application of Functional Languages*. IFL '14. Boston, Massachusetts, USA, Oct. 2014.

[Cic84] Eugene Charles Ciccarelli. *Presentation Based User Interfaces*. Tech. rep. 794. Available from `http://hdl.handle.net/1721.1/6946`, Accessed October 24, 2015. Massachusetts Institute of Technology Artificial Intelligence Laboratory, Aug. 1984.

[CKS09] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages". In: *Journal of Functional Programming* 19.05 (2009), pp. 509–543.

[Con+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[Coq04] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Version 8.0. LogiCal Project. 2004.

[Dan07] Nils Anders Danielsson. "A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family". In: *Types for Proofs and Programs*. Ed. by Thorsten Altenkirch and Conor McBride. Vol. 4502. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 93–109.

[deB72] Nicolaas Govert de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem". In: *Indagationes Mathematicae* 34 (1972), pp. 381–392.

[Del00] David Delahaye. "A Tactic Language for the System Coq". In: *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*. LPAR'00. Reunion Island, France: Springer-Verlag, 2000, pp. 85–95.

[DLW05] Dirk Draheim, Christof Lutteroth, and Gerald Weber. "A Type System for Reflective Program Generators". English. In: *Generative Programming and Component Engineering*. Ed. by Robert Glück and Michael Lowry. Vol. 3676. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 327–341.

[DM95] François-Nicola Demers and Jacques Malenfant. "Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study". In: *Proceedings of the IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*. Vol. 95. 1995, pp. 29–38.

[DP13] Dominique Devriese and Frank Piessens. "Typed Syntactic Meta-programming". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 73–86.

[Dyb00] Peter Dybjer. "A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory". In: *The Journal of Symbolic Logic* 65.02 (2000), pp. 525–549.

[Dyb94] Peter Dybjer. "Inductive Families". In: *Formal Aspects of Computing* 6.4 (1994), pp. 440–465.

[Far+12] Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. "The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs". In: *Proceedings of the ACM SIGPLAN Haskell Symposium*. Haskell '12. Copenhagen, Denmark: ACM, 2012, pp. 1–12.

[FHG14] Andrew Farmer, Christian Höner zu Siederdissen, and Andy Gill. "The HERMIT in the Stream". In: *Proceedings of the 2014 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '14. San Diego, California, USA: ACM, 2014.

[HHS03] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. "Scripting the Type Inference Process". In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. Uppsala, Sweden: ACM, 2003, pp. 3–13.

[Hic01] Jason J. Hickey. "The MetaPRL Logical Programming Environment". PhD thesis. Ithaca, NY: Cornell University, Jan. 2001.

[Hof+08] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. "Polymorphic Embedding of DSLs". In: *Proceedings of the 7th international conference on Generative programming and component engineering*. GPCE '08. Nashville, TN, USA: ACM, 2008, pp. 137–148.

[Hud96] Paul Hudak. "Building domain-specific embedded languages". In: *ACM Computing Survey* 28.4es (Dec. 1996).

[Hug95] John Hughes. "The Design of a Pretty-printing Library". In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*. 1995, pp. 53–96.

[HZS07] Shan Shan Huang, David Zook, and Yannis Smaragdakis. "Morphing: Safely Shaping a Class in the Image of Others". English. In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 399–424.

[Idr14] The Idris Community. *Programming in Idris: A Tutorial*. Accessed 19 October, 2015. Covers version 0.9.15 of Idris. Oct. 2014.

[Kel+10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. "Regular, Shape-Polymorphic, Parallel Arrays in Haskell". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pp. 261–272.

[Kno87] Todd Knoblock. "Metamathematical Extensibility in Type Theory". PhD thesis. Cornell University, 1987.

[KS15] Pepijn Kokke and Wouter Swierstra. "Auto in Agda". English. In: *Mathematics of Program Construction*. Ed. by Ralf Hinze and Janis Voigtländer. Vol. 9129. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 276–301.

[LP92] Zhaohui Luo and Robert Pollack. *LEGO Proof Development System: User's Manual*. Tech. rep. ECS-LFCS-92-211. University of Edinburgh, May 1992.

[Mai07] Geoffrey Mainland. "Why It's Nice to Be Quoted: Quasiquoting for Haskell". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell*. Haskell '07. Freiburg, Germany: ACM, 2007, pp. 73–82.

[Mar84] Per Martin-Löf. "Constructive mathematics and computer programming [and discussion]". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 312.1522 (Oct. 1984), pp. 501–518.

[McB10] Conor McBride. "Outrageous but Meaningful Coincidences: Dependent Type-safe Syntax and Evaluation". In: *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*. WGP '10. Baltimore, Maryland, USA: ACM, 2010, pp. 1–12.

[McB99] Conor McBride. "Dependently Typed Functional Programs and their Proofs". PhD thesis. University of Edinburgh, 1999.

[MM04] Conor McBride and James McKinna. "The View from the Left". In: *Journal of Functional Programming* 14.1 (Jan. 2004), pp. 69–111.

[MN94] Lena Magnusson and Bengt Nordström. "The ALF Proof Editor and Its Proof Engine". In: *Proceedings of the International Workshop on Types for Proofs and Programs*. TYPES '93. Nijmegen, The Netherlands: Springer-Verlag New York, Inc., 1994, pp. 213–237.

[MOC] The MetaOCaml Team. *MetaOCaml*. `http://www.cs.rice.edu/~taha/MetaOCaml/`. Accessed 2014.

[MP08] Conor McBride and Ross Paterson. "Applicative Programming with Effects". In: *Journal of Functional Programming* 18.1 (Jan. 2008), pp. 1–13.

[MS14] Weiyu Miao and Jeremy Siek. "Compile-time Reflection and Metaprogramming for Java". In: *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*. PEPM '14. San Diego, California, USA: ACM, 2014, pp. 27–37.

[MYM89] Scott McKay, William York, and Michael McMahon. "A Presentation Manager Based on Application Semantics". In: *Proceedings of the Second Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*. UIST '89. Williamsburg, Virginia, USA: ACM, 1989, pp. 141–148.

[Naj+15] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. *Everything Old is New Again: Quoted Domain Specific Languages*. Tech. rep. University of Edinburgh, 2015.

[NJ04] Ulf Norell and Patrik Jansson. "Prototyping Generic Programming in Template Haskell". In: *Mathematics of Program Construction*. Ed. by Dexter Kozen. Vol. 3125. LNCS. Springer-Verlag, 2004, pp. 314–333.

[Nor07] Ulf Norell. "Towards a Practical Programming Language Based on Dependent Type Theory". PhD thesis. Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[OS08] Nicolas Oury and Wouter Swierstra. "The Power of Pi". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, B.C., Canada: ACM, 2008, pp. 39–50.

[PE88] Frank Pfenning and Conal Elliot. "Higher-order Abstract Syntax". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: ACM, 1988, pp. 199–208.

[PS14] Tomas Petricek and Don Syme. "The F# Computation Expression Zoo". In: *Proceedings of Practical Aspects of Declarative Languages*. PADL 2014. San Diego, CA, USA, 2014.

[Qui81] Willard van Orman Quine. *Mathematical Logic*. Revised. Harvard University Press, 1981.

[Rau03] Daniel de Rauglaudre. *Camlp4 Reference Manual*. 2003.

[RO12] Tiark Rompf and Martin Odersky. "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". In: *Communications of the ACM* 55.6 (2012), pp. 121–130.

[Rom+13] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. "Scala-virtualized: Linguistic Reuse for Deep Embeddings". In: *Higher-Order and Symbolic Computation* 25.1 (Sept. 2013), pp. 165–207.

[SA13] Josef Svenningsson and Emil Axelsson. "Combining Deep and Shallow Embedding for EDSL". English. In: *Trends in Functional Programming*. Ed. by Hans-Wolfgang Loidl and Ricardo Peña. Vol. 7829. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 21–36.

[SBO13] Denys Shabalin, Eugene Burmako, and Martin Odersky. *Quasiquotes for Scala*. Tech. rep. 185242. École polytechnique fédérale de Lausanne, 2013.

[SFG13] Neil Sculthorpe, Andrew Farmer, and Andy Gill. "The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language". In: *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*. Vol. 8241. Lecture Notes in Computer Science. Oxford, England, 2013, pp. 86–103.

[SJ02] Tim Sheard and Simon Peyton Jones. "Template Meta-programming for Haskell". In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania: ACM, 2002, pp. 1–16.

[Smi84] Brian Cantwell Smith. "Reflection and Semantics in LISP". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: ACM, 1984, pp. 23–35.

[SS10] Antonis Stampoulis and Zhong Shao. "VeriML: Typed Computation of Logical Terms inside a Language with Effects". In: *Proceedings of the 2010 ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, Sept. 2010, pp. 333–344.

[SS90] Harald Søndergaard and Peter Sestoft. "Referential Transparency, Definiteness and Unfoldability". In: *Acta Informatica* 27.6 (1990), pp. 505–517.

[Sym+12] Don Syme et al. *Strongly-Typed Language Support for Internet-Scale Information Sources*. Tech. rep. MSR-TR-2012-101. Microsoft Research, Sept. 2012.

[Sym06] Don Syme. "Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution". In: *Proceedings of the 2006 workshop on ML*. Portland, Oregon, USA: ACM, 2006, pp. 43–54.

[Tob+11] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. "Languages As Libraries". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 132–141.

[TS00] Walid Taha and Tim Sheard. "MetaML and Multi-Stage Programming with Explicit Annotations". In: *Theoretical Computer Science* 248.1 (2000), pp. 211–242.

[vdW12] Paul van der Walt. "Reflection in Agda". MA thesis. Utrecht University, 2012.

[Wad03] Philip Wadler. "A prettier printer". In: *The Fun of Programming: A Symposium in Honor of Professor Richard Bird's 60th Birthday*. Oxford, Mar. 2003.

[WS12] Paul van der Walt and Wouter Swierstra. "Engineering Proof by Reflection in Agda". In: *24th International Symposium on Implementation and Application of Functional Languages*. Ed. by Ralf Hinze. Vol. 8241. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012.

[Zil+13] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. "Mtac: A Monad for Typed Tactic Programming in Coq". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 87–100.

Appendix A

Elaborator Tactics

This appendix describes all of the meta-operations in our extended version of Brady's [Bra13b] idealized description of the Idris elaborator language. These operations are presented in alphabetical order for ease of reference. In this list, x ranges over names, while t, t_1, and t_2 range over terms and s ranges over entire elaborator states.

- ANYTHING x creates a hole x whose type is also a hole, to assist in type inference. This is a derived operator that is defined on page 66.
- ATTACK arranges for the current hole to immediately occur in its scope. This is a precondition for binding forms. This tactic is presented in detail in Section 4.5.
- CHECK t invokes the type checker on a term t, returning its type.
- CLAIM $x : t$ establishes a new hole named x with type t at the end of the hole queue.
- ELABANTIQUOTE (x, t) focuses on x and elaborates t into it. This is a derived operator that is part of the quasiquotation mechanism, defined on page 69.
- EXTRACTANTIQUOTES is a traversal over a quoted term that replaces all antiquotations with fresh variables, remembering the mapping. It is discussed on page 66.
- FAIL m fails immediately with message m. It is used in the interpretation of the `fail` primitive in the reflected elaborator on page 105.

- FILL t places t in the currently-focused hole, converting it to a guess. If the current focus is not a hole or if t does not have the right type, the tactic fails. It is first mentioned on page 29.

- FOCUS x moves the focus to the hole or guess x. The tactic fails if x is not in the hole queue. It is first used on page 28.

- FORALL $x : t$ surrounds the focused hole with a dependent function binding. Like the other binding tactics, it requires that the hole's scope be an immediate reference to itself, so it should be bracketed by an ATTACK and a SOLVE. It is first used on page 28.

- GET binds the entire proof state to a metalanguage variable. It is used to save the state during quasiquote elaboration on page 68

- GOAL binds the current goal type to a metalanguage variable. In this dissertation, it is used in the implementation of polymorphic quasiquotations in Figure 7.5.

- GUESS retrieves the contents of the current guess, or fails if the focus is not on a guess. It is used during the elaboration of reflected elaborator scripts in Figure 9.5.

- INTRO, when run with the focus on a hole that is expecting a dependent function type, wraps the hole with a lambda, putting the focus on the body of the lambda. If the focused hole does not have a function type or if the hole binding does not immediately contain a reference to itself, then INTRO fails. Thus, it should be bracketed with an ATTACK and a SOLVE.

- LETBIND $x : t_1 = t_2$ surrounds the current hole with a `let`-binding of the name x to term t_2 with type t_1. Like the other binding tactics, it requires that the hole's scope be an immediate reference to itself, so it should be bracketed by an ATTACK and a SOLVE. It is first mentioned on page 34.

- NEWPROOF t initializes the term elaborator with goal t, which must be a type. After running NEWPROOF t, the proof term is $?h : t \,.\, h$ and the hole queue is h. It is first used in the description of the elaboration of quasiquotations on page 68.

- PUT *s* replaces the current state with *s*. This is used during the elaboration of quasiquotations, to restore the current local context after elaborating the quoted term. It is first used in Figure 7.2.
- REIFY and its variant REIFYP quote a TT term to its representation as a datatype. In addition to the term to be reified, they also accept a collection of names to *not* be reified, which is part of the implementation of antiquotations. They are described on page 66.
- SOLVE substitutes the currently focused guess in its scope, potentially causing computation to occur. It is described on page 29.
- TRY *a b* first runs *a*. If *a* succeeds, then *b* is not run; if *a* fails, the elaboration state is restored to what it was at the start of executing *a* and *b* is executed. The TRY tactical is described in the discussion of error handling in the elaborator on page 22, though it is not mentioned by name. In this dissertation, it is first used on page 73 in the description of polymorphic quotations.
- UNFOCUS *x* causes *x* to be moved to the end of the hole queue.
- UNQUOTE transforms an element of the TT datatype representing TT terms into the corresponding actual TT term.

Appendix B

The Idris Language

This appendix documents aspects of the syntax, semantics, and standard library of Idris that differ from other, similar systems.

Bang Binds

In addition to Haskell's `do`-notation as syntactic sugar for applications of the bind operator `>>=`, Idris supports an alternative notation that can be freely combined with it called *bang binds*. In Idris, `!` (pronounced "bang") is a prefix operator that causes the expression it is applied to to be lifted to under the scope of the closest enclosing binding, then bound. For example,

```
main : IO ()
main = putStrLn (!getLine ++ !getLine)
```

is syntactic sugar for

```
main : IO ()
main = getLine >>= \x =>
         getLine >>= \y =>
           putStrLn (x ++ y)
```

If multiple sub-expressions are preceded by bangs, they are bound from left to right, deepest first.

Pattern-Matching Binds

Like Haskell, Idris supports pattern-matching binds inside of a `do` block. However, in many cases, the type being matched against has more than one possible constructor, with all but one expected result being considered errors that need handling. To account for this situation, Idris allows alternative failure patterns to be specified. For instance, a program that must read a file, but that might fail to do so, can be written:

```
main : IO ()
main = do Just info ← readInfoFromFile "magic-file.txt"
            | Nothing ⇒ putStrLn "Couldn't read file"
          processInfo info
          presentOutput info
```

This is equivalent to:

```
main : IO ()
main = do x ← readInfoFromFile "magic-file.txt"
          case x of
            Nothing ⇒ putStrLn "Couldn't read file"
            Just info ⇒
              do processInfo info
                 presentOutput info
```

Rewriting Expressions

The Idris standard library defines the following function:

```
replace : {x, y : a} → {P : a → Type} →
          x = y → P x → P y
replace Refl prf = prf
```

This can be used to rewrite a proof goal according to an equality proof. However, it is difficult to use, because Idris's built-in unification mechanism is typically not able to discover an appropriate value for `P`, which represents the context within which the rewrite is performed. Manual invocations of `replace` are thus typically verbose and tedious.

The Idris syntax **`rewrite`** e_1 **`in`** e_2 causes the elaborator to examine the type of e_1 to discover what is being rewritten, and then automatically ab-

stract the current goal over that to generate a `P`. It then fills its hole with an application of `replace` to the appropriate arguments.

Uninhabited

A common Idris idiom is to create instances of the type class `Uninhabited` for empty types. This is because these proofs are tedious to keep track of.

```
class Uninhabited a where
  uninhabited : a → Void
```

It is quite uncommon to appeal directly to `uninhabited`. Instead, idiomatic Idris will typically call `absurd`, which composes `uninhabited` with the eliminator for `Void`.

Functors and Idioms

Like early versions of Haskell, Idris generalizes `map` to arbitrary functors rather than having a separate `fmap`. Like recent versions of Haskell, Idris provides `(<$>)` as an infix synonym for `map`.

Idris implements the idiom bracket syntax suggested by McBride and Paterson [MP08]. In Idris, `[| f x1 ... xn |]` is syntactic sugar for

```
pure f <*> x1 <*> ... <*> xn
```

Alternatives

The `Alternative` class, found in both Haskell and Idris, is a means of expressing that an applicative has a monoidal structure. It defines an operation and a neutral element, and users expect that `empty` is a left and right identity of `<|>` and that `<|>` is associative.

```
class Applicative f ⇒ Alternative f where
  (<|>) : f a → f a → f a
  empty : f a
```

The standard library defines a number of convenience functions that use `Alternative`. In particular, this dissertation makes use of the following functions:

```
choice : (Foldable t, Alternative f) ⇒
         t (f a) → f a
choiceMap : (Foldable t, Alternative f) ⇒
            (a → f b) → t (f a) → f a
```

The `choice` function folds using the `Alternative` operators. In the case of `Elab`, this corresponds to keeping the result of the first succeeding computation. The convenience function `choiceMap` is a fusion of `choice` and `map`.

Foldable and Traversable

As in modern Haskell, Idris provides standard `Foldable` and `Traversable` type classes. An instance of `Foldable` provides pure left and right folds over a datatype, and an instance of `Traversable` provides an effectful traversal by adding an `Applicative` constraint.

This dissertation uses a number of control structures based on `Foldable` and `Traversable`. The first two are the standard right and left folds, `foldr` and `foldl`:

```
foldr : Foldable t ⇒ (elt → acc → acc) → acc → t elt → acc
foldl : Foldable t ⇒ (acc → elt → acc) → acc → t elt → acc
```

Effectful code, such as elaborator scripts, will typically make use of one of the following operations:

```
traverse : (Traversable t, Applicative f) ⇒
           (a → f b) → t a → f (t b)
traverse_ : (Traversable t, Applicative f) ⇒
            (a → f b) → t a → f ()
```

Additionally, `for` and `for_` reverse the order of arguments to `traverse` and `traverse_` respectively, which can be more readable in some contexts.

Named Instances and Explicit Instance Application

Unlike Haskell, Idris supports named instances of classes. Users can explicitly provide named instances as an alternative to the instances chosen by instance resolution, and they will never be automatically used. Named

instances are defined by providing a name in square brackets after the **class** keyword. At an application site, explicit instances are provided by preceding them with `@` and surrounding them with curly braces.

For example, let `HasOp a` be a class that provides some operator `op`, and let `squish` be a function that applies `op` to four arguments:

```
class HasOp a where
  op : a → a → a

squish : HasOp a ⇒ a → a → a → a → a
squish w x y z = (w `op` x) `op` (y `op` z)
```

There might be two separate instances for `Nat`, one using addition and one using multiplication. Addition is the most commonly-desired operation, so it will be the default, but users can still opt in to multiplication if they would like:

```
instance HasOp Nat where
  op = plus

instance [multiply] HasOp Nat where
  op = mult
```

Users can then choose the version they want to call:

```
> squish 1 2 3 4
10 : Nat

> squish @{multiply} 1 2 3 4
24 : Nat
```

Additionally, instance application syntax can be used on the left-hand side of a definition to bind the resolved dictionary to a name.

Glossary

antiquotations 62
a region of a quasiquotation that is not quoted

deep embedding 10
an embedding of a DSL in which the terms are represented indirectly as syntax trees

development calculus 25
a term calculus extended with hole and guess bindings

direct reflection 19
a reflection system in which the surrounding system is used for reflection (c.f. `eval` as a primitive operator in a Lisp implementation)

hygiene 47
the property of a macro system in which its macros respect the lexical scoping rules of the language

Idris^{-} 21
the desugared form of Idris that is the input to the elaborator

indirect reflection 19
a reflection system in which the language is reimplemented in itself (cf. metacircular evaluators)

levitation 2
a technique for representing datatypes using a self-representing universe of codes

metalanguage 47
the language used to write metaprograms, or reason about programs

metaprogramming 15
writing programs that generate or modify other programs

object language 47
the programming language in which the programs to be generated or manipulated by the metalanguage are written

plicity the property of being implicit or explicit 41
presentation 50
a region of program output that retains its association with the underlying object represented by the output
proof by reflection 37
a proof technique in which objects in type theory are mapped to some simpler domain in which there exists a verified decision procedure

quasiquotation 62
a form of quotation in which subterms of the quoted term can have ordinary rather than quoted semantics
quasiquotation 16
A form of quotation within which some subterms are not quoted and have the ordinary evaluation rules of the language

reflection 15
the representation of aspects of a programming language, such as terms or error messages, as a datatype in the language itself

shallow embedding 10
embeddings of DSLs that directly use the features of the host language where they coincide with the features of the embedded language