Type-Directed Specification Refinement

By
© 2011
Mark Huntington Snyder

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graudate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

____
Dr. Perry Alexander, Chairperson

____
Dr. Andrew Gill

____
Dr. Nancy Kinnersley

____
Dr. Prasad Kulkarni

____
Dr. Jeremy Martin

Date Defended:___________________

The Dissertation Committee for Mark Huntington Snyder
certifies that this is the approved version of the following dissertation:

Type-Directed Specification Refinement

____
Dr. Perry Alexander, Chairperson

____
Dr. Andrew Gill

____
Dr. Nancy Kinnersley

____
Dr. Prasad Kulkarni

____
Dr. Jeremy Martin

Date Approved:___________________

Abstract

Specification languages serve a fundamentally different purpose than general-purpose programming languages, and their type systems reflect these needs. Specification type systems must record and track more information for us to reason about a system adequately, and this added expressiveness may lead to an undecidable typing analysis. System level design begins with a high-level specification that is continually refined and expanded with implementation details, constraints, and typing information, down to a concrete specification. During this refinement process, the system is underspecified, and many static analyses aren't applicable until the system is fully specified. However, partial specifications contain valuable information that can inform the refinement process—we can locally inspect parts of the specification from a typing perspective to look for inferrable information or inconsistencies early on to aid the refinement process. This work defines a typing analysis that gathers constraints and typing information to inform the specification refinement process. It explores localized techniques such as local type inference and tracking of values as a means of influencing the specification refinement process.

Acknowledgements

I'd like to thank my advisor Dr. Perry Alexander for the years of guidance and seeing me through to the very end. Whether I needed another round of edits on some paper or a life lesson, he found the time and the words I sought. I'd also like to thank my committee for giving the considerable time and attention required for the lengthy process. I also am indebted to my family for being a lifeline across the thousand-plus miles, keeping me grounded and positive. I've also been fortunate to have found friendship with many people throughout my stay in Lawrence, Kansas – this has truly become a place I will miss, and remember fondly. I'd especially like to thank my labmates for all their help and camaraderie — Garrin Kimmell, Nicolas Frisby, Philip Weaver, Jennifer Lohoefener, Evan Austin, Megan and Wesley Peck, Kevin Matlage, Andrew Farmer, Michael Jantz, and everyone else who has shared deskspace in 145 Nichols over the past seven years. I've received exceptional support from the faculty as well – I received wonderful teaching guidance from Dr. Nancy Kinnersley, and I spent most of my class and research time around Dr. Andrew Gill and Dr. Prasad Kulkarni when I ventured beyond my research group and classes under Perry. I wouldn't have succeeded how I did without being surrounded by all these wonderful, supportive people.

Contents

Acceptance Page **ii**

Abstract **iii**

1 Introduction **1**

1.1 Thesis Experimentation and Evaluation . . . 2

1.2 Summary . . . 5

2 Related Work **6**

2.1 Type Systems . . . 6

2.1.1 The Lambda Cube . . . 6

2.1.2 Simply Typed Lambda Calculus . . . 7

2.1.3 System F_2 . . . 10

2.1.4 System λ_ω . . . 12

2.1.5 System F_ω . . . 14

2.1.6 LF, the Logical Framework . . . 16

2.1.7 System λ_P . . . 20

2.1.8 Calculus of Constructions . . . 20

2.1.9 Pure Type Systems . . . 21

2.1.10 System $F_C(\chi)$ . . . 25

2.1.11 System $F_{<:}$ . . . 26

2.1.12 Relation to Current Work . . . 27

2.2 Type Inference . . . 27

2.2.1 Algorithm W . . . 28

2.2.2 Local Type Inference . . . 29

2.2.3 Definitions and Typing Rules for Local Type Inference . . . 30

2.2.4 Type Inference of System F_2 . . . 32

2.3 Dependent Types . . . 33

2.4 Refinement . . . 34

2.5 Summary . . . 37

3 Background Work **38**
3.1 The Rosetta Type System . . . 38
3.1.1 Standard Types and Operators . . . 38
3.1.2 Composite Types, Constructed Types . . . 41
3.1.3 First-Class Types . . . 42
3.1.4 Building Blocks - Facets . . . 43
3.1.5 Reflection and Interactions . . . 44
3.1.6 Rosetta Type System Summary . . . 46
3.2 Rosetta Alpha Type Checker . . . 46
3.3 Preparatory Typing Work . . . 49
3.3.1 Local Type Inference on ASG's . . . 49
3.3.2 Error Reporting . . . 50
3.4 Summary . . . 50

4 Methodology **51**
4.1 Introduction . . . 51
4.2 Defining Rosetta Typing Rules . . . 52
4.2.1 A Note on Lists . . . 54
4.2.2 Qualified Names . . . 55
4.2.3 Design Units . . . 56
4.2.4 Use Clauses . . . 57
4.2.5 Quantified Parameters . . . 57
4.2.6 Applications . . . 58
4.2.7 Variables . . . 59
4.2.8 Functions . . . 60
4.2.9 Direct Functions . . . 60
4.2.10 Anonymous Functions . . . 61
4.2.11 Further Binding Sites: Functions and Lets . . . 61
4.2.12 Sequence Predicates and Operators . . . 62
4.2.13 Control-Flow Expressions . . . 63
4.2.14 Ascriptions . . . 64
4.2.15 Constructed Types . . . 64
4.2.16 Mathematical Operators . . . 65
4.2.17 Top and Bottom Literals . . . 66
4.3 Subtyping Relationship . . . 66
4.3.1 Subtyping for Primitive Types . . . 67
4.3.2 Subtyping for Composite (Structural) Types . . . 68

4.3.3 Subtyping for Functions . . . 69
4.3.4 Subtyping for User-Defined Datatypes . . . 70
4.3.5 Explicit Subtype Definitions . . . 70
4.4 Developing the Typing Analysis . . . 71
4.4.1 Reference Algebras . . . 71
4.4.2 Tasks for Basic Type Checking . . . 79
4.5 Partial Information Approaches . . . 82
4.5.1 Witnesses . . . 83
4.5.2 Nodes as Types . . . 84
4.5.3 Substitutions . . . 85
4.5.4 Size control . . . 87
4.5.5 Presburger Arithmetic . . . 87
4.5.6 Local Type Inference . . . 90
4.6 Error Reporting . . . 93
4.7 Summary of Methodology . . . 97

5 Evaluation **100**
5.1 The Type System . . . 101
5.2 Basic Typing . . . 102
5.3 Partial Typing . . . 107
5.3.1 Local Type Inference . . . 107
5.3.2 Witnesses . . . 108
5.3.3 Tracking Sizes . . . 108
5.3.4 Error Reporting and Graph Annotations . . . 109
5.4 Driving Specification Refinement . . . 110

A Rosetta Typing Rules **112**
A.1 Design Units . . . 112
A.2 Use Clauses . . . 115
A.3 Quantified Parameters . . . 116
A.4 Qualified Names . . . 117
A.5 Variables . . . 117
A.6 Applications . . . 117
A.7 Functions . . . 118
A.7.1 Direct Functions . . . 119
A.7.2 Anonymous Functions . . . 120
A.8 Let-expressions . . . 120
A.9 Sequence Predicates . . . 121

A.10 Control-Flow Expressions . . . 121
A.11 Ascriptions . . . 122
A.12 Constructed Types . . . 123
A.13 Rosetta-Standard Operators . . . 123
A.13.1 Composite Types . . . 123
A.14 Set Operations . . . 124
A.15 Multiset Operations . . . 125
A.16 Sequence Operations . . . 127
A.17 Bitvector Operations . . . 130
A.18 Mathematical Operators . . . 131
A.19 Addition . . . 131
A.20 Subtraction . . . 132
A.21 Multiplication . . . 132
A.22 Division . . . 133
A.23 Exponentiation . . . 133
A.24 Exponentials . . . 134
A.25 Negation and Identity for Numbers . . . 134
A.26 Bit and Boolean Operators . . . 134
A.27 Functions over Complex Numbers . . . 135
A.28 Trigonometry and More with Complex Numbers . . . 136
A.29 Mathematical Constants . . . 136
A.30 Real, Imaginary Math Ops . . . 137
A.31 Rational Number Operations . . . 138
A.32 Integer and Natural Number Operations . . . 138
A.33 Character Operations . . . 139
A.34 Top and Bottom Literals . . . 140

Chapter 1

Introduction

Complex languages often provide features that make typing analysis or type inference over them undecidable. Whether the language combines subtyping with inferred polymorphism, dependent types, or existential types, these languages still have a place in the programming and specification communities—we can express more in our programs and specifications, at the expense of interpretability. Specification languages in particular benefit from these advanced features, as the work itself is more invested in formality and provable correctness of properties than executability. If we cannot simply perform a complete typing analysis for type safety, what rôle should the type system have? It is still beneficial to provide partial typing information, and this is especially true in partially-defined specifications. Writing a top-level specification and refining towards a concrete representation is at the the heart of system-level design, and providing partial typing information early in the design process is worth the effort. Features that are decidably typable should still be type-checked, and restrictions of features that introduce typing undecidability ought to be explored to provide as much information as is practical. In particular, we look to techniques that are localized, techniques that meaningfully interpret the information available from a limited vantage point, especially techniques that do not assume completeness of information and that do not operate globally. The purpose is to interpret the information as it becomes available in order to find more concrete information and constraints on the under-constrained specification.

Thesis Statement:

> Using localized type analyses, we can gather and report information from partial, incomplete specifications that will guide the specification refinement process.

In order to test this thesis, we explore the type system for the Rosetta specification language [5, 6]. Rosetta has a non-trivial type system that has not been fully formally defined. This work formally defines elements of the Rosetta type system, and implements a typing analysis over Rosetta. A key goal of this research is to complement the specification process with partial typing information; specifications written in top-down fashion often lack enough information to perform traditional type checks, and so the nature of the specification refinement process will guide the typing implementation efforts.

The Rosetta type system is quite expressive; from simple expressions, functions, components and facets to meta-programming facilities, universally quantified parameters and dependent typing, there are many cross-cutting concerns to address. We discuss the foundations of formal type system definitions and lay out an approach to defining the type system of a language such as Rosetta. This work also implements a partial typing analysis for Rosetta, to test the effects of local type inference on the specification refinement process.

1.1 Thesis Experimentation and Evaluation

In order to test the thesis statement, this work provides a formal definition of the Rosetta type system a typing analysis over Rosetta specifications. The formal definition of Rosetta's type system gives a solid foundation for reasoning, and the type information analysis provides the actual hints and cues expected to motivate the thesis statement itself.

In this work, we create a set of type rules that allow deriving a Rosetta expression's type judgement. As with any type system, this is not required to be an algorithmic presentation of *how* to find the types, but presents a set of rules that can be applied in some appropriate order to arrive at a typing conclusion. It is a separate effort to provide algorithmic rules to perform typing in Rosetta. Rosetta allows dependently typed entities, and dependent typing leads to non-decidable type analysis unless usage is sufficiently restricted. This means that there may not be such a set of rules for Rosetta in its entirety. For this reason, this proposal does not claim it can provide an algorithmic set of typing rules for Rosetta. We define appropriate restrictions

for Rosetta and for the typing analysis performed to retain decidability in typing. The Rosetta language in general does not yet have a fully formally defined type system, however, and so this type system definition will provide opportunities to clarify the language's type system.

There are pre-existing partial definitions of the Rosetta type system. This proposal will rely on all existing documents on the Rosetta language in defining the Rosetta type system, as long as they are collectively coherent. This proposal will focus these efforts by collecting and extending definitions of the language's type system. It can function as a single source of typing semantics for the language, rather than typing being an implication of definitions strewn throughout the language documents.

This work also provides a typing analysis capability for Rosetta. As mentioned above, simply adding dependent typing to a language can make typing analysis undecidable; in implementing a typing analysis for Rosetta, there of course must be limitations on what the typing analysis can guarantee to test, or limitations to what the Rosetta language should allow. The focus is towards analyses that can utilize partial results, such as local type inference or collection of constraints and tracking values.

Typing is one of the few analyses that makes sense to apply over the entirety of Rosetta. Since Rosetta provides for a confluence of domains, any analysis that is defined in terms of one domain in particular cannot, by definition, be defined for all of Rosetta. For instance, timing concerns are non-sensical in the static domain; integrity over non-security domains is a meaningless concept. What really separates these different analyses from typing is a trade-off between generality and specificity. While typing Rosetta must interact with all features of the language in some fashion, the very fact that it must be defined over the entire language sometimes means that it will be a simple analysis. Recursively defined `let`-bindings in Rosetta are simpler to type using the required ascription than to evaluate in other analyses. Typing analysis must interact with all features of the language, yes, but at the same time typing Rosetta almost exclusively needs to consider the types.

In implementing typing as a Rosetta analysis, this work must address the usage of dependent types common to Rosetta specifications. Rosetta types are first-class values, without a common separation between the world of term values and the world of type values. Type checking cannot devolve into a Rosetta "evaluator" in order to calculate these uses of dependent types, so finding an appropriate cut-off to its usage will be key to achieving an actual implementation.

The Rosetta typing analysis targets the specification refinement process. Therefore, the focus is in collecting any type-related information and presenting it to the specifier. Some typing information will be available in traditional ways, but other means of inference expects and accepts partial specifications. Even on under-specified systems, this typing analysis will collect information and check for any inconsistencies it can recognize. The expectation is that this information can somehow be organized and presented to the specifier to indicate areas of underspecification or inconsistent specifications, in the form of warning and error messages. For example, local type inference can help infer omitted type parameters, even when other parts of a specification are incomplete. If a specification defines a type with no information about the type, then uses of the type will constrain the actual semantics of the type; this information can be collected and presented to the specifier. Overall, the purpose of the analysis is not just a correctness check, but a source of inferred and collected knowledge about the specification's types that further informs the refinement procedure.

This work yields two artifacts — a type system definition in the form of a set of typing rules, and a typing analysis implementation for Rosetta in the form of an executable analysis that plugs into the current architecture and framework of Rosetta tools. The purpose of the Rosetta typing analysis is to facilitate the specification refinement process. Therefore, the focus of the typing analysis is in what ways typing information aids the refinement process. As for the type system definition, if it is consistent with itself and covers all the typing features expressed and checked by the analysis, then it is by construction a definition of the Rosetta type system. To what degree we need qualifications and assumptions to provide a type system will be as good an indication as any that the work is faithfully defining the Rosetta type system. Similarly, the degree to which actual specifications are covered by the typing analysis tool will dictate how successful the implementation is, and how successfully this work achieves its goals. We look for common situations in refinement that the typing analysis can address as a source of its effectiveness. Some possible sources of those situations are described later, but some examples are local type inference,tightly controlled evaluation for dependent types, and tracking of values. These situations are the topics that allow testing the thesis statement.

1.2 Summary

This work formally defines the type system of the Rosetta specification language, a much needed reference for a language as complex as Rosetta is. Accompanying this formal definition will be a typing analysis over Rosetta specifications. Typing is an analysis that naturally ranges over all of the language definition, which by Rosetta's inclusion of dependent typing will not actually be able to decidably cover all of Rosetta's features. Rosetta is a specification language, so typing will focus on aiding the specification refinement process through partial analysis, constraints collection, and traditional typing analysis where applicable. This implementation must be faithful to the formal definition of the language while covering the language as fully and decidably as it can. The typing analysis focuses on applying typing concepts to partially specified specifications, in order to test the thesis statement that such information can aid the specification refinement process.

Chapter 2

Related Work

2.1 Type Systems

Languages are mixes of representation and meaning. They are normally represented as syntax in the form of grammars and given meaning through sets of rules that define typing, evaluation, and any other semantics inherent in the language. We often study languages by adding features to simple existing languages that are theoretically well-understood, incrementally expanding our understanding of the impact that individual features have on the expressiveness of languages. In this section, we survey a number of languages, exploring the significance of their prominent features. We must have an understanding of these implications in order to adequately address a language such as Rosetta that admits such features, and consequently to discuss a typing analysis over such a language.

These languages are only presented to introduce specific features such as type level computation, so we do not include definitions such as Booleans, numerical types, lists or records. Even more complex additions such as abstract data types (ADTs) and references do cross-cut the concerns of these languages, but we omit them in order to discuss these languages' features in their simplest form. We adopt syntax for these languages as is found in Pierce's invaluable book [38].

2.1.1 The Lambda Cube

Henk Barendregt formalized a group of well-studied languages [8], placing them all into a single framework called the *lambda cube* (fig. 2.1) or alternatively the *Barendregt cube*. In this survey of common theoretical

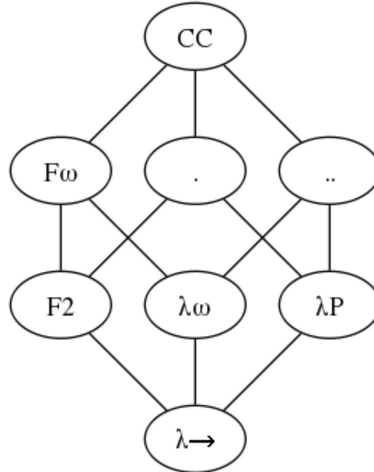

Figure 2.1: The Lambda Cube.

languages, we analyze this cube by investigating the languages comprising its corners. The lambda cube codifies four basic abstractions[1] – terms abstracted over terms, terms abstracted over types, types abstracted over types, and types abstracted over terms. At the 'base' of the cube is the simply typed lambda calculus, where terms may abstract over terms. The other three abstractions correlate to the three axes of the lambda cube. Combinations of features account for the remaining corners. Using the lambda cube as a road map for this discussion, we discuss the simply typed lambda calculus, the three other axes, as well as a few other languages. This will incrementally expand the discussion up to the level of languages involved in the thesis and the implementation.

2.1.2 Simply Typed Lambda Calculus

The simply typed lambda calculus is the simplest typed language definition. Define terms `t` and types `T` as simple grammars:

```
t = x | λx:T.t | (t t)

T = T→T
```

[1] An abstraction represents an expression with a free variable we may later substitute a supplied value for throughout the expression. What kind of expression we represent in an abstraction and what kind of value for the free variable we may supply determine properties of the language. A function with one argument is a simple example of an abstraction.

x represents a variable. λ-abstractions allow us to create functions – the input parameter is to be named some variable x, it must have the annotated type T, and the output is the resulting body t of the λ-expression. To supply expressions as arguments to λ-abstractions, we construct applications with parentheses – first the function, then the argument. This simple mechanism of abstractions and applications with term variables to utilize the abstracted values gives us a simple language definition that strongly normalizes, meaning that evaluating a well-typed term in the language always terminates. It is therefore not Turing-complete, though the untyped lambda calculus is. Our types reflect the ability to constrain values to particular sets of values and to create functions that map elements of one type to elements of another type. If we were to add Booleans to our language, we could simply add to the term and type grammars:

```
t = ...  | true | false | if t then t else t
T = ...  | Bool
```

Now we have a type other than $T \to T$; by itself, the type grammar couldn't actually represent any finite or useful type, only infinitely substituting T with $T \to T$. Now we can create a function λx:Bool.x, whose type is Bool→Bool, indicating that when applied to a Bool value it will return a Bool value. This is the identity function for booleans. The simply typed lambda calculus does not allow for any recursion; we cannot represent a function that calls itself recursively, we cannot create any non-terminating behavior. Furthermore, we cannot reuse the identity function for any non-boolean values; if we want an identity function for numbers, or for functions or anything else, we must create a separate definition. This behavior is true for any function definition; for instance, if we wanted to provide a library of list functions, we would have to duplicate the entire library for lists of booleans, lists of naturals, lists of a particular function type, and so on. We will see ways to relax these concerns in the languages discussed below.

We now turn our attention to the typing rules for the simply typed lambda calculus. We define an environment, Γ, which we use to keep track of the names that are in scope.

$$\Gamma = \emptyset \mid x{:}T,\Gamma$$

Variables are added to this environment by lambda abstractions to link occurrence of the newly-bound variable to its type. Below are the typing rules for the three syntactic forms in the simply typed lambda calculus.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \texttt{(Var)}$$

$$\frac{\Gamma, x : T_D \vdash t_r : T_R}{\Gamma \vdash \lambda x : T_D . t_R : T_D \rightarrow T_R} \qquad \texttt{(Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_D \rightarrow T_R \quad \Gamma \vdash t_2 : T_D}{\Gamma \vdash t_1 t_2 \; : T_R} \qquad \texttt{(App)}$$

We see that variables are simply looked up in the environment; indeed, there is no other way to understand the meaning of the variable, and this is the only place in this language where we inspect the environment. For abstractions, we consider the domain type ($\mathtt{T}_D$) and range type ($\mathtt{T}_R$). We check that binding the variable (`x`) to the domain type can inform the context sufficiently to guarantee that $\mathtt{t}_r\mathtt{:}\mathtt{T}_R$, and conclude that the lambda abstraction is a mapping from $\mathtt{T}_D$ to $\mathtt{T}_R$. The bound variable must be added to the environment in order to derive the body's type — this is the only way in which the environment is extended in this language. Last, consider applications, in which we must see a mapping type ($\rightarrow$) and an argument of the correct type. The subscripts are again fashioned in terms of the domain and range of the abstraction function, and the argument must be of the domain type. If these constraints are met, the overall resulting type is the range type.

These rules do not tell us directly how to apply them to a particular term. Instead, they tell us that if we can find an appropriate application of these rules to our term, we can find a type for the term. These typing rules also dictate nothing of the evaluation of the language; there is no notion of call-by-value versus call-by-name implied by these typing rules, either of which could be supported by a similar set of rules for the semantics of evaluation. Consider applying the boolean identity function to a boolean value: `((λx:Bool.x) true)`. We can build up a 'derivation tree' [38] that shows how the above rules apply to our term in Fig. 2.2.

At the bottom of this derivation tree is the application term and found type for it, while the derivation trees for its sub-terms show above the line with their own reasonings. This chains together the rules of the type

$$\dfrac{\dfrac{\dfrac{x\!:\!\text{Bool} \in (\emptyset, x\!:\!\text{Bool})}{\emptyset, x\!:\!\text{Bool} \vdash x\!:\!\text{Bool}}(Var)}{\emptyset \vdash \lambda x\!:\!\text{Bool}.x \ : \text{Bool} \rightarrow \text{Bool}}(Abs) \qquad \dfrac{}{\emptyset \vdash true : \text{Bool}}(T\!-\!true)}{\emptyset \vdash ((\lambda x\!:\!\text{Bool}.x)\ true) \ : \text{Bool}}(App)$$

Figure 2.2: Example derivation tree.

system, with only the unconditional axioms at the tops (leaves) of the derivation. This language is simple enough that there will be just one derivation possible; later on, we will see that this is not always the case. Syntax-directed 'algorithmic' typing rules will preserve this property of there being exactly one rule that we can apply in any particular situation. Just as parsers need unambiguous grammars to ensure they yield predictable and meaningful results, a typing implementation needs an unambiguous set of typing rules to reliably yield the same correct types.

2.1.3 System F_2

System F_2 [41], also called the polymorphic lambda calculus, expands on the simply typed lambda calculus by adding type abstractions and type applications to the term-space, and by adding type variables and universal types to type-space. This introduces abstractions of terms that depend on types, which may be understood as functions that accept a type parameter and result in a term. We expand on the definitions for terms `t`, types `T`, and environment Γ from the simply typed lambda calculus. `X` refers to a type variable.

```
t = ...  | λX.t | (t [T])
T = ...  | X | ∀X.T
Γ = ...  | X,Γ
```

We consider the new typing rules for System F_2's new constructs below. A type abstraction's body results in the $\forall$ universal type. For type applications we ensure that a universal-typed term is applied to a type, and the resulting substitution with that type argument is our resulting type.

$$\frac{\Gamma, X \vdash t_r : T_R}{\Gamma \vdash \lambda X.t_r \;: \forall X.T_R} \quad \texttt{(Ty-Abs)}$$

$$\frac{\Gamma \vdash t_f : \forall X.T}{\Gamma \vdash t_f[T_{arg}] : [X \mapsto T_{arg}]T} \quad \texttt{(Ty-App)}$$

These additions allow us to relax the identity function to solve our previous problem. Instead of creating separate functions at every type we'd use them, we could create functions like these:

```
id ::  ∀X.X→X
id = λX. λx:X.x

flip::∀A. ∀B. ∀C. (A→B→C) → B → A → C
flip = λA. λB. λC. λf:A→B→C. λb:B. λa:A. ((f a) b)
```

We see the new ∀ universal type constructs as well. We can use our new identity function with the new application. Though we've avoided defining evaluation's own set of derivation rules, we inspect the normalization (evaluation) of the polymorphic identity function `id` to understand how these features behave.

```
        ((id [Bool]) true)
⇒
        (((λX. λx:X.x) [Bool]) true)
⇒
        ((λx:Bool.x) true)
⇒
        true
```

System F_2 is impredicative, in that a quantified type may quantify over itself. For example, `((id [∀X.X→X]) id)::∀X.X→X`. Much like the value application and substitution of the simply typed

lambda calculus that we see in the third line, we can now apply type values to type abstractions to substitute for all occurrences of that type argument throughout the type abstraction's body. This is the only change that System F_2 introduces, but it is powerful. Assuming we have lists defined in our language as well, we can now write parametrically polymorphic list operations that will work on lists containing any particular type of values:

```
cons ::  ∀A. A → List A → List A
head ::  ∀A. List A → A
map ::  ∀A. ∀B. (A → B) → List A → List B
```

It is often the case that new language features such as the type abstraction and type application of System F_2 best showcase their abilities with the addition of other features such as Lists and other structured data. Notice that we could have defined map's type as `∀B. ∀A. (A→B)→List A→List B`, and it would have the same effect, but since we apply the types explicitly, usage would have to then supply the type arguments in this new order.

The explicit application of type arguments is System F_2's Achilles' Heel. When we rely heavily on polymorphic code, we find that type abstractions and type argument applications litter the code to the point of distraction and obfuscation. The requirement to write all these typing ascriptions ultimately turns programmers away from purely System F_2-based languages. It still can find use as the internal representation of languages. However, translation from an external language that does not require all the ascriptions to an internal language that explicitly maintains the ascriptions can be problematic, especially in the presence of sub-typing and other features.

2.1.4 System λ_ω

Although the simply typed lambda calculus provides types to organize our terms, there is nothing to organize our types. System λ_ω[2] extends the simply typed lambda calculus by providing `kinds` to group our types

[2] Barendregt [8] identifies System λ_ω as essentially the System F of Girard [19] (which is in French), but Pierce [38] offers a more accessible and modern introduction.

in much the same way that types group our terms. With kinds available, System λ_ω also provides type operators, type abstractions, and type applications. This provides, in effect, a copy of the simply typed lambda calculus at the type level instead of at the term level. We can create type-abstractions, type-applications, and we consider the 'type of a type', called a kind.

We re-define terms, types, and Γ, and introduce kinds:

```
t = x | λx:T.t | (t t)
T = T→T | X | λX::K.T | (T T)
Γ = ∅ | Γ,x:T | Γ,X::K
K = ⋆ | K ⇒ K
```

Now that we have an extra layer of classifications for our language via kinds, we need not just typing rules, but kinding rules also. We normally think of types as sets of values, but types that 'contain' values are actually called *proper types*, always of kind $\star$. Types whose kind involves the $\Rightarrow$ operator are called *type operators*, needing additional type arguments of particular kinds before they produce proper types. This implies that there are no terms whose type's kind is $\star\Rightarrow\star$. Following are the basic kinding rules for System λ_ω. Just as we have type rules to address all forms of terms, we have kinding rules for all forms of types.

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \quad \texttt{(K-TVar)}$$

$$\frac{\Gamma, X :: K_D \vdash T_R :: K_R}{\Gamma \vdash \lambda X :: K_D . T_R :: K_D \Rightarrow K_R} \quad \texttt{(K-Abs)}$$

$$\frac{\Gamma \vdash T_F :: K_D \Rightarrow K_R \quad \Gamma \vdash T_D :: K_D}{\Gamma \vdash T_F T_D :: K_R} \quad \texttt{(K-App)}$$

$$\frac{\Gamma \vdash T_A :: * \quad \Gamma \vdash T_B :: *}{\Gamma \vdash T_A \rightarrow T_B :: *} \quad \text{(K-Arrow)}$$

These rules truly look like the simply typed lambda calculus at the type level. The only addition is K-Arrow, ensuring that any types used in types of term-level abstractions truly are proper types with kind $\star$. This does introduce an issue of type equivalence–just as terms must be normalized to check for equivalence, our 'computable' type expressions must now be normalized prior to checking type equivalence. The type equivalence check is really just a closure of reflexive, symmetric, transitive, and eta-expanded equivalence over the structure of the types. The lack of a sub-typing relation also keeps this equivalence relation simple.

System λ_ω does not extend System F_2, so universal types ($\forall$) are not present. What System λ_ω provides is somewhat limited by itself: we can have terms depend on terms with basic term-level λ abstractions, and we can have types depend on types with type-level λ abstractions, but we cannot mix the two levels in any way. Any type ascriptions we have for terms will invariably be of kind $\star$. Our types are still islands of abstracted computation that may make things easier to phrase, but in the end we cannot link different terms' types together in any meaningful fashion, and certainly not with any parametrically polymorphic types in System λ_ω. The term-level type abstraction of System F_2 allows us to create terms that depend on types. But just as the addition of primitive list definitions aids in showcasing System F_2's power, System λ_ω becomes useful in confluence with other features, namely System F_2's universal types. The combination of System F_2 and System λ_ω is called System F_ω, and we look at it next.

2.1.5 System F_ω

System F_ω [19, 38, 41] combines the term-level type abstractions and applications of System F_2 with the type-level type-abstractions and applications of System λ_ω. This marriage of type-level type abstractions and term-level type abstractions allows for parametrization of types in type *abstractions*. Furthermore, term-level type abstractions provide kind information for the abstracted type (λA**::K**.t), meaning that we can create term-level type abstractions over any valid type-level syntax, including over type operators.

```
t = x | λx:T.t | (t t) | λT::K.t | (t [T])
T = T→T | X | λX::K.T | (T T) | ∀X::K.T
K = ⋆ | K⇒K
Γ = ∅ | Γ,x:T | Γ,X::K
```

This language introduces the ability to parametrize a term over type operators (kinds including ⇒), not just over proper types of kind ⋆.

```
id_{ty-op} ::∀A::(⋆⇒⋆).∀X::⋆.A X→A X
id_{ty-op} = λA::(⋆⇒⋆).λX::⋆.λx:(A X). x
```

We can now introduce concepts such as pairs to the language without having to make them primitive language constructs. A language unable to parameterize a term over a type would need to implement the pair type's polymorphism directly. If we assume the encoding for pairs as in the definition below, we can then define the `pair` constructor and the observers `fst` and `snd` (see [38] exercise 23.4.8, also §30.2).

```
Pair (X::⋆) (Y::⋆) ≝ ∀R::⋆.  (X→Y→R) →R

pair ::  ∀A::⋆.  ∀B::⋆.  A → B → Pair A B
pair = λA::⋆.  λB::⋆.  λa:A.λb:B.λR::⋆.λf:(A→B→R). f a b

fst ::  ∀A::⋆.  ∀B::⋆.  Pair A B → A
fst = λA::⋆.  λB::⋆.  λp:(∀R::⋆.(A→B→R)→R). p [A] (λf:A. λs:B. f)

snd ::  ∀A::⋆.  ∀B::⋆.  Pair A B → B
snd = λA::⋆.  λB::⋆.  λp:(∀R::⋆.(A→B→R)→R). p [B] (λf:A. λs:B. s)
```

The addition of existential types would allow even better enforcement by allowing us to package all the list operations together and abstract the interface. This could remove the motivation to create a type synonym for `Pair`, as the type would be brought into scope via the packaging of the existential type and scoped body.

If we introduce existentials and a direct way to create abstract data types[3], this type operator abstraction becomes much more powerful: we can then create polymorphic data structures such as lists, tuples, and trees directly within the language without primitive support for them by packaging all the constructors and observers of an ADT and abstracting the interface. In our pair example, we would no longer need a type synonym for `pair`, as the type would be brought into scope via the packaging of the existential type and scoped body.

Haskell's type classes can be seen as an extension of these features including existentials. A *type class* may be viewed as an abstraction over some type variables that provides a tuple of functions. Type class constraints such as `(Num t)=>` reduce down to an actual parameter that is implicitly applied, specifically a 'dictionary' of known implementations of the type class to cover the correct actual types where the functions of the type class are used. An `instance` of a type class is really a term that abstracts over the same types as the type class definition. This is strongly akin to the use of existentials to represent ADT's ([38], chapter 24). The same type class / existential type may be applied to different instances/existentially-typed implementations that may be viewed at that abstraction.

We have already seen all of the typing and kinding rules involved in System F_ω through the previous two sections. Even System F_ω has decidable typing, in its purest form. As with simpler languages, the addition of features such as sub-typing (System $F^{\omega}_{:>}$) can break this property.

2.1.6 *LF*, the Logical Framework

The last axis of the lambda cube introduces expressions where types depend on terms. These so-called dependent types can introduce undecidability in the typing relation. We will see the abilities and constraints of such languages and their type systems.

[3] often just a 'pack' and 'unpack' scheme – see §30.2 of [38]

The Edinburgh Logical Framework [7, 20] is a system that transforms a logical system definition into an editor and proof checker. We focus on the underlying language, where these logical systems are defined. It extends the simply typed lambda calculus with the dependent product Π, allowing type-level abstractions over terms. *LF* is predicative, meaning types cannot contain themselves or refer to themselves via type variables. This gives us some normalization properties vital to the usability of the language. The language defines terms, types, and kinds as follows, with syntax aligned to our other examples.

```
t = x | λx:T.t | (t t)
T = λx:T.T | (T T) | Πx:T.T
K = ⋆ | Πx:T.K

Γ = ∅ | Γ,x:T | Γ,x:K
```

The language is usually defined with term and type level constants, and a signature Σ separate from the environment Γ to hold the constants' typing and kinding information, but we'll ignore them and focus on the core language. The terms, types, and kinds are type checked ($\vdash_t$), kind checked ($\vdash_k$), and sort checked ($\vdash_s$), respectively. The following rules are adapted from Harper et al. [20].

`S-Star` and `S-Pi` are well-formedness checks over kinds, similar to checking for kind ⋆ in System λ_ω. Sorts are the 'types of kinds.'

$$\frac{\vdash_s \Gamma}{\Gamma \vdash_s *} \qquad \texttt{(S-Star)}$$

$$\frac{\Gamma, x :: A \vdash_s K}{\Gamma \vdash_s \Pi x : A.K} \qquad \texttt{(S-Pi)}$$

The kind-checking rules check for well-kindedness of types in *LF*. `K-Const` allows access to Γ, and `K-Conv` determines kind equality; the three other rules are more interesting. `K-Abs` and `K-App` are

straightforward, in that a type-lambda-abstraction is kinded with a Π kind, and a type application requires a substitution, just as term applications require substitution. Lastly, `K-Pi` indicates that the body `B` of the Π-abstraction may depend upon `x:A`, but still must be of kind `*`. The Π abstraction at the type level indicates that one type is dependent on some element of Γ, but it is overall a representation of a proper type. The body of the Π abstraction is the meaning of the abstraction, but its meaning may depend upon `x:A`. Type-lambda abstractions expect an actual parameter to be passed, and have the corresponding type application that is missing from Π abstractions.

$$\frac{\vdash_k \Gamma \quad c : K \in \Gamma}{\Gamma \vdash_k c : K} \qquad \texttt{(K-Const)}$$

$$\frac{\Gamma, x : A \vdash_k B : *}{\Gamma \vdash_k \Pi x : A.B : *} \qquad \texttt{(K-Pi)}$$

$$\frac{\Gamma, x : A \vdash_k B : K}{\Gamma \vdash_k \lambda x : A.B : \Pi x : A.K} \qquad \texttt{(K-Abs)}$$

$$\frac{\Gamma \vdash_k A : \Pi x : B.K \quad \Gamma \vdash_k M : B}{\Gamma \vdash_k A\,M : [M/x]K} \qquad \texttt{(K-App)}$$

$$\frac{\Gamma \vdash_k A : K \quad \Gamma \vdash_k K' \quad \Gamma \vdash_k K \equiv K'}{\Gamma \vdash_k A : K'} \qquad \texttt{(K-Conv)}$$

Type checking in *LF* is just as in $\lambda_{\rightarrow}$. There is a rule for looking up constants in the environment and a rule for type-conversion between equal types, as in kind-checking. Departing from $\lambda_{\rightarrow}$, instead of using arrow-types for functions such as `A → B`, we see Π `x:A.B`, allowing for `B`'s dependence on `x`. Some presentations will include the → type as well, indicating a type that cannot be dependent. Application is just substitution.

$$\frac{\vdash_t \Gamma \quad c : A \in \Gamma}{\Gamma \vdash_t c : A} \qquad \texttt{(Ty-Const)}$$

$$\frac{\vdash_t \Gamma \quad x : A \in \Gamma}{\Gamma \vdash_t x : A} \qquad \texttt{(Ty-Var)}$$

$$\frac{\Gamma, x : A \vdash_t M : B}{\Gamma \vdash_t \lambda x : A.M : \Pi x : A.B} \qquad \texttt{(Ty-Abs)}$$

$$\frac{\Gamma \vdash_t M : \Pi x : A.B \quad \Gamma \vdash_t N : A}{\Gamma \vdash_t M N : [N/x]B} \qquad \texttt{(Ty-App)}$$

$$\frac{\Gamma \vdash_t M : A \quad \Gamma \vdash_t A' : * \quad \Gamma \vdash_t A \equiv A'}{\Gamma \vdash_t M : A'} \qquad \texttt{(Ty-Conv)}$$

A type of the form $\Pi \texttt{x} : T_D \,.\, T_R$ describes a function from elements of type T_D (meaning T_D is a proper type and `x` is a term) and, allowing reference to `x` via the environment Γ, results in a type T_R (not in a term of type T_R). This allows a type function whose range depends on the *value* of the element of the domain. The typing rules rely on β-reduction in determining equality, meaning we must perform reduction-style evaluation of types or kinds in determining equality.

There is no notion of a quantifier in *LF*, so any term must have a monomorphic type, and any type must have a monomorphic kind. This lends to strong normalization, and so the language enjoys decidable typing. The introduction of a `fix` operator or some means of non-primitive recursion would of course break this, as would pollution of the normalization-as-equality check, e.g. via sub-typing. Even further, well-typed terms can undergo type erasure and their typing can be inferred in *LF*.

2.1.7 System λ_P

In Henk Barendregt's original formulation of the lambda cube [8], he used a simple syntax to encompass all the different languages. Instead of presenting *LF* for dependent typing, the language that extended the simply typed lambda calculus was called λ_P. The syntax Barendregt used for all the corners of the cube for term, type, and kinds merged into one expression grammar for λ_P:

```
e = x | λx:e.e | Πx:e.e | (e e)
```

and didn't include the striations into three levels as we see in *LF*. Nevertheless, the extension to allow type-level term abstractions via the dependent product Π is the key to the language's expressiveness, which Barendregt achieves through restrictions on the sorts allowed in the Π product's sub-expressions. This was the basis of a prototypical "Generalized Type System", the precursor to Pure Type Systems discussed in §2.1.9. System λ_P behaves essentially as System *LF*, so we do not discuss it in further detail.

2.1.8 Calculus of Constructions

The Calculus of Constructions [14] sits at the pinnacle of the lambda cube, combining the simply typed lambda calculus with the features of System F_2, System λ_ω, and System λ_P. Thus we have all four variations of abstraction that we've discussed — term-level abstractions over terms (simply typed lambda calculus), term-level abstractions over types (system F_2), type-level abstractions over types (system λ_ω), and type-level abstractions over terms (system λ_P).

Similar in nature to Pure Type Systems [43] (as we shall see next in §sec:pts), the language is typically defined in one grammar, instead of splitting it into terms, types, kinds, and sorts.

```
t = x | (t t) | λx:t.t | Πx:t.t | ⋆ | □
```

$\star$ and $\Box$ were called `Prop` and `Type`, respectively. Beyond the rules we'd expect according to the previously discussed languages, the calculus of constructions includes the axiom that $\overline{\vdash * : \Box}$. We have no universal quantifiers — although being the capstone of the lambda cube, the individual languages on the corners of

the lambda cube are really just examples of languages that exhibit the properties. So it is not quite correct to simply consider the calculus of constructions as the merging of system F_2, System λ_ω, and System λ_P. Since we do not have quantifiers, subtypes or other compromising features, we know that the language is strongly normalizing, impredicative, and must be written in the Church style (contain explicit type annotations).

The calculus of constructions has been at the center of many proof assistants, including Coq [15], LEGO [40], and NuPRL [23]. Coq is based on the Calculus of Inductive Constructions (CIC), meaning that inductive definitions are added as a language feature for ease in programming. LEGO implements various type systems, including System *LF* and the calculus of constructions. MetaPRL [22] is an updated implementation of NuPRL that allows for modularity. The requirement for so many ascriptions is apparently worth the effort in a proof environment, as opposed to a general purpose programming language such as ML [29] or Haskell [36], which opt out of many advanced features in the face of mandatory program-wide ascriptions.

2.1.9 Pure Type Systems

The lambda cube presents three key extensions to the simply typed lambda calculus. System F_2 adds term-level type abstractions and parametric polymorphism; System λ_ω adds type-level type abstractions, necessitating a kinding system to ensure 'type safety' of our types; lastly, System *LF* adds type-level term abstractions, introducing dependent types into the language. All of these different extensions can be generalized under one framework, itself able to generalize over more language features. This framework defines Pure Type Systems [43].

Pure Type Systems collapse the sense of layers in our languages: there is one grammar that defines terms, types, kinds, and sorts. We rely on Roorda and Jeuring [43], but maintain syntax closer to the previous languages that follow Pierce [38].

$$t = x \mid (t\ t) \mid \lambda x{:}t.t \mid \Pi x{:}t.t \mid \star \mid \square$$

$$\Gamma = \emptyset \mid \Gamma, x{:}t$$

One major change in this is to realize that, since types and terms all inhabit the same termspace, we see 'terms' where we might expect only types. For instance, the λ abstraction allows any `t` after the colon, where we would previously expect to find a type in the simply typed lambda calculus, or a kind in a type-level λ abstraction. $\star$ and $\square$ are both sorts, the type of kinds. The phrase 'type' is now a bit more flexible — we use it to mean 'type' when we say the type of terms, meaning 'kind' when we say the type of types, and meaning 'sort' when we say the type of kinds.

Roorda and Jeuring [43] define the following rules:

`(axiom)` $$\frac{}{[] \vdash * : \square}$$

`(start)` $$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$$

`(weak)` $$\frac{\Gamma \vdash A : B \;\; \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$$

`(abs)` $$\frac{\Gamma, x : A \vdash b : B \;\; \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}$$

`(pi)` $$\frac{\Gamma \vdash A : s \;\; \Gamma, x : A \vdash B : t}{\Gamma \vdash (\Pi x : A.B) : t}, \qquad (s,t) \in R$$

`(app)` $$\frac{\Gamma \vdash f : (\Pi x : A.B) \;\; \Gamma \vdash a : A}{\Gamma \vdash f\, a : B[x := a]}$$

`(conv)` $$\frac{\Gamma \vdash a : A \;\; \Gamma \vdash B : s \;\; A =_\beta B}{\Gamma \vdash a : B}$$

Given these simple rules, we can restrict the language to any point of the lambda cube by restricting the set R in rule (pi). R is defined as the relation $(\star,\star) \subseteq R \subseteq \{\star,\square\}\times\{\star,\square\}$. Consider the (pi) rule for Πx:A.B, defining the dependent product type for a term in the simply typed lambda calculus. We see that s is the type of our argument x's type A, and that t is the resulting type of our dependent product's body B. If $s = \star$, then A is a type whose type (kind) is $\star$, implying that x is a term whose type is A. If $t = \star$, then B's type (kind) is $\star$, and so an abstraction with this product as its type has a body of type B, meaning that the body of the abstraction is a term. By including $(\star,\star)$ in R, we are allowing abstractions over terms (left $\star$) at the term level (right $\star$), because the Π product representing such abstractions require exactly those sorts of kinds of types.

We can include the features of System F_2, System λ_ω, and System *LF* by adding the other three possible pairs to R. In order to allow System F_2 style expressions, we need term-level type abstractions; this corresponds to including $(\square,\star)$ in R. Again inspecting the (pi) rule defining the semantics for Π products, if $s=\square$, then A is of type (sort) $\square$, so x is of type (kind) $\star$, implying that x is a type; similarly, if $t=\star$, then just as before, this product characterizes an abstraction that yields a term. This is because B is of type $\star$, so an abstraction represented by this product must contain a body that is a term. In order to allow System λ_ω-style expressions, we need type-level type abstractions. We gain these by adding $(\square,\square)$ to R. Lastly, we allow System *LF*-style expressions — allowing type-level abstractions over terms — by including $(\star,\square)$ in R.

This provides a very convenient means of parametrizing a general structure to define different aspects of the lambda cube — there are eight corners to the cube, corresponding to the 2^3 elements in the power set of $\{(\square,\star),(\square,\square),(\star,\square)\}$ (recall that $(\star,\star)$ is always in R).

Pure Type Systems do more than just catalogue the lambda cube. By further parameterizing over the sorts rather than including $\star$ and $\square$ once and for all, we can allow more complex expressions. We must provide the set of axioms over these sorts in place of the single (axiom) rule above; lastly, we re-define our relation R in terms of these new sorts. A simple case is the identity function over types of any kind:

$$\lambda k{:}\square.\lambda t{:}k \;:\; (\Pi k{:}\square.k\to k)$$

In the lambda cube, this would be illegal, because $\Box$ is not given a type in any axiom — only sort $\star$ had a type. In a Pure Type System, we could extend our sorts and axioms to rectify this:

```
S = ⋆ | □ | □′
```

```
A = {(⋆,□),(□,□′)}
```

$$\texttt{(axiom):} \quad \overline{\vdash s_1 : s_2}, (s_1, s_2) \in A$$

```
R = {(⋆,⋆), (□,⋆), (□,□), (□′,□)}
```

Pure Type Systems introduce a further relaxation — `R` is a ternary relation S $\times$ S $\times$ S, reflecting that a Π product may result in a sort other than the sort of the body `B`:

$$\texttt{(pi)} \qquad \frac{\Gamma \vdash A : s_1 \;\; \gamma, x : A \vdash B : s_2,}{\Gamma \vdash (\Pi x : A.B) : s_3} \qquad (s_1, s_2, s_3) \in R$$

Of course to implement this, `R` should maintain the injective property that s_1 and s_2 uniquely define s_3. Roorda and Jeuring [43] suggest allowing (s_1, s_2) as shorthand for (s_1, s_2, s_2). They further extend this general framework of Pure Type Systems with a few key features to form the basis of a functional programming language. They add declarations via let-expressions, abstract data types of a more powerful form than those found in Haskell 98 due to their dependent nature involving parameters, and a case construct while maintaining decidability in typing. They adapt Barthe's Algorithm [10] for typing injective PTS's, which modifies the `(abs)` rule to use a classification over terms `t` and distributes the `(conv)` rule to achieve an algorithmic presentation of the type system.

Pure type systems collapse the hierarchy of terms, types, kinds, and sorts, and allow parameterization over the type system itself, though we study instances (given specific sorts, axioms, and allowed relations of sorts in the dependent product Π). The collapse both blurs the distinctions between levels of specification and allows for more flexible definitions. We see in general that pure type systems will need explicit type ascriptions, other than in degenerate cases such as representing the simply typed lambda calculus.

2.1.10 System $F_C(\chi)$

Haskell [36] is at its core a System F_2 language with extensions. Upon adding generalized abstract data types (GADT's), compiler implementers were finding transformations difficult or unavailable on these GADT's. Sulzmann et al. [46] introduced a language, System $F_C(\chi)$, which extends the functionality of System F_2 with type equality witnesses and open non-parametric type functions. Top-level axioms over types and equality must be checked for consistency via some decision procedure (called χ). This language allows for direct encoding of a host of features, including GADT's, functional dependencies, and associated types. Their goal was to update Haskell's core language to support these features, and in the end the language is now more expressive (in terms of what GADT's it can accept).

As a language definition for System $F_C(\chi)$, we see terms, types, kinds, and sorts. There are two sorts, types `TY` and coercions `CO`. Type equalities, represented as $\sigma_1 \sim \sigma_2$, are kinds. In order to support definitions used in execution of an expression, System $F_C(\chi)$ provides for defining programs as both declarations and an expression. In checking type safety for the language, there are many separate analyses that refer to each other. Kinds are checked to be of specific sorts. Types and coercions are kind-checked. Expressions are type-checked. Declarations are checked for consistency, and quite interestingly, result in a Γ describing the environment the declaration introduces. Programs collect these Γ's and use them in checking the type of the expression. Using separate analyses to define the environment Γ for a syntactic structure of a language to be used in other analyses seems like a way to separate these concerns the language definition. The authors of System $F_C(\chi)$ go on to show the operational semantics and soundness of the language and show some translations from a supposed source language to System $F_C(\chi)$. After type-erasure, System $F_C(\chi)$ programs should run the same, ensuring there is no run-time overhead.

We see in this language how explicitly dictating the sorts of a language can yield features such as GADT's. Directly adding features such as type coercions, providing separate analyses for them, and plugging them into the term-type-kind-sort hierarchy appropriately can help inject the complexity of a new feature such as open type functions and GADT's without sacrificing properties such as soundness and decidability of typing. System $F_C(\chi)$ is more than just a theoretical language, it is the basis of GHC [1], the most widely used Haskell implementation. It overcomes real issues in a manner that can participate in a full solution for a language, it is not just a prototype that showcases a new idea.

2.1.11 System $F_{<:}$

One feature that cross-cuts the concerns of all of the above languages is sub-typing. Whether we strictly refer to a hierarchy of base types that exhibit sub-typing such as naturals, integers, and reals, or instead refer to functions, parametrically polymorphic functions, record types and so on, adding the notion of subtypes to a language means re-examining every part of the language definition to see where sub-typing affects the definition.

System $F_{<:}$ extends System F_2 with sub-typing. In order to mix the term-level type abstractions with sub-typing, we add bounds to the quantifiers:

```
t = ...| λX<:T.t
T = ...| ∀X<:T.T
Γ = ...| X<:T
```

We introduce a sub-typing relation, including the rule for $\forall$ terms' sub-typing relation:

$$\frac{\Gamma, X <: U \vdash S_2 <: T_2}{\Gamma \vdash (\forall X <: U.S_2) \;<:\; (\forall X <: U.T_2)}$$

This simple version is based on the `KERNEL FUN` language of Cardelli and Wegner [11]. It restricts both types' `∀X` to be restricted by the same type `U`. This restriction is a bit severe — we can only use a polymorphic type when we supply a value quantified under the exact same bound, disallowing the use of terms with tighter bounds. However, if we adopt a more relaxed definition of sub-typing between these terms:

$$\frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash (\forall X <: S_1.S_2) \;<:\; (\forall X <: T_1.T_2)}$$

we lose decidability of type checking. There does not seem to be a succinct way to show this, however Pierce [37] goes into detail, defining 'positive' and 'negative' positions, relating typing to a two-counter Turing machine, and showing that there is a derivable sub-typing statement iff the Turing machine halts. Thus sub-typing owes its undecidability to logic along the lines of the halting problem's undecidability.

2.1.12 Relation to Current Work

How do all of these languages relate to the current work? We see how the simply typed lambda calculus is the basis for all of these languages, and that the addition of constructs to the language makes them more complex and able to express more programs. Rosetta is a language encompassing the lambda cube and Pure Type Systems. Understanding how these features interact, especially their implications on the decidability of typing those languages and the need for type ascriptions versus inferencability, is crucial in defining Rosetta's type system. Furthermore, we learn from them in identifying what can and cannot be done to present useful information to the programmer if we have certain features in the Rosetta language and implementations of Rosetta. More generally, to show that partial information can be realistically generated and presented for partial specifications, we must consider many of the same properties these languages provide, and the mechanisms by which they maintain properties sufficiently for a static typing analysis.

2.2 Type Inference

Type inference is the process of deriving types for terms inhabiting a language. Type inference allows us to omit the typing annotations that would otherwise be needed, but the expressions still have types — it allows for the absence of type *ascriptions* without degrading into an absence of types themselves. The inferred types must be the 'best' type; if there is not one best type, then this ambiguity cannot be resolved with type inference. The best type is often called the *principal type*.

Type inference has the key benefits of stronger type systems without the need to annotate types everywhere, being is both tedious and obfuscating. Unfortunately the requirement that best types exist is often denied by adding certain features. The classic example is subtyping, which can be a partial ordering. This means that, for instance, `Int→Int` and `Real→Real` may both be legitimate types for a function, but neither is a subtype of the other. This is due to contravariance in sub-typing the arguments and covariance in sub-typing the ranges of functions. When we add parametric polymorphism to a language, inference can be extended to supplement the type arguments as well, creating implicit parameters. Since inferring types is so valuable to users of a language, language designers go to great lengths, and avoid great features, to

maintain type inferencability. In this section, we explore some type inference capabilities and limitations, and the language features that support or deny the property.

2.2.1 Algorithm W

An early type inference algorithm, called `Algorithm W` [16], operates over a language called "mini-ML" [12] defined by extending Curry-style lambda calculus terms[4] with `let`-expressions. The algorithm works in bottom-up fashion to construct type schemes for all declarations. A type scheme may use universal quantifiers ($\forall$) to describe the declaration's type, but each usage of the declared item results in instantiations of those universally quantified variables, essentially requiring a monomorphic type for each parameter. This is not as powerful as System F_2. Algorithm W also manages α-conversions while constructing the set of constraints that must be met to unify the type variables and produce a principal type scheme. These constraints are solved via the 'most general unifier' algorithm from Robinson [42].

Damas and Milner [16] show it to be as expressive as the let-style polymorphism of ML. Let-style polymorphism allows us to instantiate a parametrically polymorphic type (one with a $\forall$ quantifier at its leftmost) in multiple places separately, so that we do not force the quantified variable to unify with each instance's usage. A naive implementation can just substitute during typing, but this mechanism finds the principal type scheme at the let binding site, manages names, and supplies fresh type variables at each usage to differentiate between the various uses.

There is a similar approach to Algorithm W that Lee and Yi [26] call 'Algorithm M', that is very similar in approach yet addresses the problem in top-down fashion, maintaining an environment. They show that it finds errors sooner than Algorithm W, specifically at the application of mismatched function and argument types rather than where the function or arguments are defined. They also argue that mixing the two directions can yield better implementations and error messages.

Nazareth and Nipkow [32] have formally verified Algorithm W's soundness and completeness mechanically, using Isabelle/HOL [34]. They restrict themselves to the monomorphic case, disallowing `let` constructs and polymorphic types. They show that maintaining the namespace, specifically managing new

[4] Curry-style lambda calculus terms contain no type annotations.

variables, is non-trivial. Naraschewski and Nipkow [31] extend the work to remove the previous work's restrictions, covering the entire mini-ML language.

2.2.2 Local Type Inference

Extending System F_2 with sub-typing creates a language where sub-typing is undecidable. However, this full System F_2 extension is called impredicative, meaning that a quantified type can be quantified over itself. Defining a parametrically polymorphic identity function, we can apply it to itself legally:

```
id ::  ∀X.X→X
id = λX.λx:X.x


(id [∀X.X→X] id) ::  ∀X.X→X
```

There are restrictions on System F_2 extended with sub-typing that can recover type checking decidability; the most common is ML-style let-polymorphism. This predicative parametric polymorphism dis-allows quantified types applying over other quantified types. This is achieved by only allowing quantified types to occur at let-bindings, and only allowing application of these quantified types with non-quantified types (called monotypes). The above identity function defined in a let-construct could not be applied to itself, though it could be applied to numerous monotypes in its let-body (for instance, a `map` function could be applied to lists with two different contained types).

Without discarding any of the power of System F_2, we can still infer some types. Pierce and Turner [39] show a system of type inference that allows for type parameters to be omitted when a best (principal) type exists. This inference occurs locally, in the sense that all the information needed to constrain and solve for the type is gathered from adjacent nodes in the abstract syntax tree (AST).

The procedures and operations they define for using this style of type inference are as follows:

- Define the internal and external language. The difference is simply that all type arguments must be supplied in the internal language, but the external language may omit them. The goal is to translate the external language (with omitted arguments) into the internal language (with all arguments present).

- Define the subtype relation. This is only defined over concrete types, and not over type variables.
- Define the least upper bound $\vee$ and greatest lower bound $\wedge$ between types. This relies on the subtype relation.
- Define typing over the internal language. Type arguments are present, so this is straightforward.
- Define the least super-type $\Uparrow$ and greatest subtype $\Downarrow$ relations. These operations push type variables out to the necessary extremes, giving the best type approximation while those type variables are unconstrained.
- Define constraint generation. Constraints are only generated where type arguments are missing, i.e. at the application of a parametrically polymorphic function without the type arguments. These constraints place upper and lower bounds on each type variable.
- Define position. Position relates an occurrence of a type variable to its location within a specific type expression R (the range of the applied polymorphic function), and is either constant, covariant, contravariant, or invariant.
- Define elaboration from the external to internal language. At omitted type arguments, elaboration elaborates sub-terms, types them, and uses that typing information to perform constraints generation and solving, based on the positions of each omitted type argument's locations.

2.2.3 Definitions and Typing Rules for Local Type Inference

We reproduce the rules found in Pierce and Turner [39], so that we can relate to them in the rules for Rosetta.

1. Internal Language

$\mathtt{T} \quad ::= \mathtt{X} \mid \mathtt{Top} \mid \mathtt{Bot} \mid \mathtt{All}(\overline{X}).\overline{T}\rightarrow\mathtt{T}$

$\mathtt{e} \quad ::= \mathtt{x} \mid \lambda[\mathtt{X}](\overline{x}:\overline{T}).\mathtt{e} \mid \mathtt{e}[\overline{T}]\rightarrow\mathtt{T}$

$\Gamma \quad ::= \{\} \mid \mathtt{x}:\mathtt{T},\Gamma \mid \mathtt{X},\Gamma$

2. Subtyping Relation

$$\frac{}{X <: X} \qquad \texttt{Sub-Refl}$$

$$\frac{}{X <: Top} \qquad \texttt{Sub-Top}$$

$$\frac{}{Bot <: X} \qquad \texttt{Sub-Bot}$$

$$\frac{\overline{T} <: \overline{R} \qquad S <: U}{All(\overline{X}).\overline{R} \to S \quad <: \quad All(\overline{X}).\overline{T} \to U} \qquad \texttt{Sub-Fun}$$

3. Least Upper Bound, Greatest Lower Bound

S∨T:

T	if S <: T
S	if T <: S
All($\overline{X}$)$\overline{M}$ →J	if S = All($\overline{X}$)$\overline{V}$ →P
	T = All($\overline{X}$)$\overline{W}$ →Q
	$\overline{V} \wedge \overline{W} = \overline{M}$
	P ∨ Q = J
Top	otherwise

S∧T:

S	if S <: T
T	if T <: S
All($\overline{X}$)$\overline{J}$ →M	if S = All($\overline{X}$)$\overline{V}$ →P
	T = All($\overline{X}$)$\overline{W}$ →Q
	$\overline{V} \vee \overline{W} = \overline{J}$
	P ∧ Q = M
Bot	otherwise

4. Typing the Internal Language

$$\frac{}{\Gamma \vdash x \in \Gamma(x)} \qquad \text{T-Var}$$

$$\frac{\Gamma, \overline{X}, \overline{x} : \overline{S} \vdash e \ : \ T}{\Gamma \vdash \lambda[\overline{X}]\overline{x} : \overline{S}.e : All(\overline{X})\overline{S} \to T} \qquad \text{T-Lam}$$

$$\frac{\Gamma \vdash f : All(\overline{X})\overline{S} \to R \qquad \Gamma \vdash \overline{e} <: [\overline{T}/\overline{X}]\overline{S}}{\Gamma \vdash f[\overline{T}](\overline{e}) : [\overline{T}\overline{X}]R} \qquad \text{T-App}$$

$$\frac{\Gamma \vdash f : Bot \qquad \Gamma \vdash \overline{e} \ : \ \overline{S}}{\Gamma \vdash f[\overline{T}](\overline{e}) : Bot} \qquad \text{T-AppBot}$$

$$\frac{\Gamma \vdash f : All(\overline{X})\overline{T} \to R \quad \Gamma \vdash \overline{e} : \overline{S} \qquad |\overline{X}| > 0 \quad \emptyset \vdash_{\overline{X}} \overline{S} <: \overline{T} \Rightarrow \overline{D} \qquad C = \wedge\overline{D} \quad \sigma = \sigma_{C_R}}{\Gamma \vdash f(\overline{e}) : \sigma R \Rightarrow f[\sigma\overline{X}](\overline{e})} \qquad \text{T-App-InfAlg}$$

5. Constraints Generation

$$\frac{}{V \vdash_{\overline{X}} T <: Top \Rightarrow \emptyset} \text{ CG-Top}$$

$$\frac{}{V \vdash_{\overline{X}} Bot <: T \Rightarrow \emptyset} \text{ CG-Bot} \qquad \frac{Y \notin \overline{X}}{V \vdash_{\overline{X}} Y <: Y \Rightarrow \emptyset} \text{ CG-Refl}$$

$$\frac{Y \in \overline{X} \quad S \Downarrow^V T \quad FV(S) \cap \overline{X} = \emptyset}{V \vdash_{\overline{X}} Y <: S \Rightarrow \{Bot <: Y <: T\}} \text{ CG-Upper}$$

$$\frac{Y \in \overline{X} \quad S \Uparrow^V T \quad FV(S) \cap \overline{X} = \emptyset}{V \vdash_{\overline{X}} S <: Y \Rightarrow \{T <: Y <: Top\}} \text{ CG-Lower}$$

$$\frac{V \cup \{\overline{Y}\} \vdash_{\overline{X}} \overline{T} <: \overline{R} \Rightarrow \overline{C} \quad V \cup \{\overline{Y}\} \vdash_{\overline{X}} S <: U \Rightarrow D \quad \overline{Y} \cap (V \cup \overline{X}) = \emptyset}{V \vdash_{\overline{X}} All(\overline{Y})\overline{R} \to S <: All(\overline{Y})\overline{T} \to U \Rightarrow (\wedge \overline{C}) \wedge D} \text{ CG-Fun}$$

6. Positions

`R is constant in X, if:`	`[S/X]R <: [T/X]R, for every S and T.`
`R is covariant in X, if:`	$\Gamma \vdash$ `[S/X]R <: [T/X]R` $\Leftrightarrow$ $\Gamma \vdash$ `S <: T.`
`R is contravariant in X, if:`	$\Gamma \vdash$ `[T/X]R <: [S/X]R` $\Leftrightarrow$ $\Gamma \vdash$ `S <: T.`
`R is invariant in X, if:`	$\Gamma \vdash$ `[S/X]R <: [T/X]R` $\Leftrightarrow$ `S = T.`

This procedure will not always be successful, but failure will occur at exactly the application where a most informative type, a principal type, is not available. This pinpoints the exact place where type arguments will have to be supplied. Thus this approach omits a specific group of ascriptions (inferrable type arguments) without losing decidability of type checking.

For our own rules in Rosetta, we will make claims that Γ can infer that a best choice for quantified parameters $\overline{Q}$ of a function `f` are $\overline{I}$ when given parameters $\overline{p:P}$, range R, and arguments $\overline{a:A}$, which we will represent with `infer`$(\overline{Q}, \overline{p:P}, \overline{R}, \overline{a:A}) \Rightarrow \overline{I}$.

2.2.4 Type Inference of System F_2

Type inference implies that we are omitting type ascriptions, but that specific types still define each expression in the terms of the language. In the lambda cube, Curry-style terms that can be typed belong to the simply typed lambda calculus, System F_2, or System F_ω. While it is known that the simply typed lambda calculus' typability was decidable, and that System F_ω's typability was *not* typable, System F_2 lies between the two and its typability was unknown.

Wells [48] put this open problem to rest, showing that System F_2 was not decidably typable after all. The basic sketch of the proof starts by showing that semi-unification SUP (known to be undecidable [25]) reduces to the type-checking TC problem. Next, type-checking TC is shown to be reducible to the is-typable problem TYP. It was already known that TYP was reducible to the problem of TC [9]. Thus typability and type checking are equivalent, and undecidable for Curry-style terms of System F_2. Typability only refers to the existence of a type that describes the term, but type inference — the task of assigning a type to a term without type ascriptions — clearly falls into this category, and is also an undecidable procedure.

Beyond answering the open problem, this work influences language design in a number of ways. As popular as System F_2 is as a core language, Wells' result indicates that full inference over this language will never be fully available, and thus efforts must focus on acceptable limitations on the language, on ascription requirements, or on another core language altogether. Along the lines of required ascriptions, if powers of further expression can be added to the language that direct type inferencing sufficiently, this extra expressiveness may be enough to regain decidable typing.

2.3 Dependent Types

In some of the languages of the previous section, we saw that types may depend on the values of terms. We've hinted at how dependent types lead to undecidability in a type system. When is this feature actually useful? Dependent types allow us to encode more information about a program into its types. We can create sized lists and matrices, or we can even encode properties such as sortedness directly into the type of a definition.

```
Vector Int a = Nil::Vector 0 a | Cons a (Vector n a) :: Vector (n+1) a
vhead :: Vector (n+1) a → a
vhead (Cons x xs) = x
vtail :: Vector (n+1) a → Vector n a
vtail (Cons x xs) = xs

matrixMult :: Matrix a b → Matrix b c → Matrix a c
```

Figure 2.3: Examples of dependent types.

Dependent types introduce the need to perform computation when determining types. Performing static type analysis is generally a decidable calculation, relying on the ability to compare types without evaluating a diverging computation. Types in languages are simple enough that divergent computations do not exist at the type level. But in allowing types to depend on values, any divergent value-expression can now cause type checking to also diverge. Completely separating evaluation from static analysis greatly simplifies the task, but precludes many interesting checks of code properties. Including lengths in list definitions or allowing predicate subtypes such as in SAL [4] allows for far more specification of the behavior of a program at the type-level. The cost for this expressiveness is a loss in ability to infer types, or indeed to even check given type ascriptions. SAL's type checker is incomplete – there are types in the language complex enough that the type checker will not be able to give an answer to type safety. Navigating the extra information of dependent types while still being able to answer the basic typing questions is at the forefront of implementing dependently-typed languages.

Coming back to pragmatic concerns from the viewpoint of type theory, a practical example arises in that systems level design is often ultimately interested in hardware. We would like design parameters that relate the bit-widths of different parameters. If we introduce a type such as `bitvector(n)` [5] and the type system is to ensure that two different bitvectors are of the same size, then we will inevitably end up requiring some basic operations to combine these bitvectors. Rather than allow arbitrary computation at the argument `n`, we are more likely to find a restriction such as Presburger arithmetic [21, 30], where we introduce the constants `0` and `1`, the + operation, and equality =. This sufficiently hobbles the arithmetic so that we can still define relationships between the size of things while maintaining decidability, were we to normalize a few and check for equality. Knowing what uses of dependent types are useful in systems design, and how to carefully introduce those uses, is a side goal of this work.

2.4 Refinement

A specification records the constraints and intentions of a system, but the process is not one large step – just as programming a large system is never a single edit-compile-run cycle. Specifications are written in high-

[5] or equivalently, `word(n)`

level abstract terms initially, and then more facts, constraints, and behaviors are added. The inclusion of information may simply respect the previous representation and simply give a more specific representation, or it may make decisions that introduce further constraints, choosing implementation paths and therefore excluding others. This process of adding information and narrowing down the possibilities of the specified system is the refinement process. Ideally, the specification can refine all the way down to an implementation. Just as hardware design must eventually bridge the gap between simulation and synthesis, refinement of a specification must bridge a similar gap between abstract system specification and eventual implementation. Accordingly, we refine a specification as close to implementation as possible or as needed, closing that gap and gaining the confidence that the specification matches the eventual implementation.

Throughout the refinement process, we are adding information to the specification, and we are using information in the specification. We expect the specification to infer what it can from the specification, but analysis is often defined over complete systems. Typing analysis is often phrased in terms of preservation of types — meaning that replacements of suitably typed elements will not change the typing information initially found. A function that accepts a real number and returns a boolean should work just fine if we instead supply an integer, due to the relation `integer ⊂ real`. What if we don't know anything about the type supplied to this function? Is it okay for another type to be used in its place? It may be that an unspecified sub-typing relation would answer in the affirmative, but the specification may only say that we have two undefined types. A type left entirely abstract is nothing more than a type variable, used to link the different uses. Typing analysis can then collect information about a type variable based on all these uses to create a composite picture of the type, checking for inconsistencies or ambiguity. Typing is generally used to answer with finality if given typing ascriptions are accurate, and similarly to answer what is "the" appropriate type of any expression. When there is no *best* answer, type inference and type checking will simply have to fail. Programming languages cannot usefully say that no errors were found if there are areas of incompleteness. Executable code cannot suffer such an ignorance and claim type safety, but in specification refinement, this ignorance is at least initially inherent.

Type checking in programming languages often cannot infer enough information to satisfy the compiler. This leads to the obligatory error message(s) along with compilation failure. Suppose instead of failure, we instead see some sort of flag marking unfulfilled obligations to guarantee type safety. The iterative

process of attempting compilation in a programming language implementation such as GHC [1] already gives instructive error messages, suggesting that certain language pragmas must be turned on to use certain features, or e.g. a type is "too polymorphic" and will inevitably require a (different) type ascription as used. In this way, the task of writing an executable program achieves its own partial specification refinement procedure, using the partial information available to guide the programmers' subsequent refinements.

PVS [35, 44] is a mature formal specification language with an integrated theorem prover. The language is based on higher order logic, and is often used for formal proofs of properties of systems. It includes predicate subtypes as a form of dependent types, allowing for very expressive specifications. This leads to type-correctness conditions (TCC's) that must be proven, though many are automatically discharged. PVS theorem proving is a user-guided process of inference and many tactics that are applied manually to discharge the TCC's and proof tasks. If one can get an appropriate model of a system in PVS, it can be a powerful tool for proving properties over a system. But the assumption is that you have enough of a system description in order to prove the properties needed over that system. The proposed work is focused on aiding the refinement process, not proving properties over a specified system.

Epigram [3, 47] is a dependently-typed programming language. The IDE introduces sheds encompassing undefined areas of the program in an interactive process where programmer and system both contribute to the codebase, based on type information available. The task in Epigram is to create a program, a different task than specification refinement, where we are more interested in the provable properties of a well-defined system than the eventual implementation, although obtaining a system specification that is or closely resembles an implementation is another goal.

In general, any time that typing inference fails, we have an opportunity to provide partial typing analysis. When there is not enough information to account for complete typing information, instead of failing to compile, we instead have reportable and recordable information to report that may be more than the specifier intended or understood, thereby increasing understanding about the specified system. In particular, type variables that arise from abstract or incomplete type definitions can be an organizational tool for relating information to the specification. We might report everything we know about a given type, and indicate pieces of information about it that contradict each other, or else state that needed/stated properties are not

known, and need further details or refinements. The point is to expect incomplete specifications and provide as much feedback as the analysis can.

Refinement is sometimes understood as the process of removing non-determinism from a system [49]. Here, non-determinism refers to multiple values being possible, and not to an instruction to select one of some set of actions or choices in order to introduce some variability. Non-determinism of systems is often contained 'below' the type level, such as by avoiding selection of specific values within a type, but nonetheless restricting possibilities to within a type. Again, partial analysis (beyond simple typing analysis) may further explicitly recover options within the non-determinism, pointing towards the possible further refinement and concretization of the specification.

2.5 Summary

Types systems formally define what terms are present in a language, and in conjunction with evaluation sufficiently define the syntax and semantics of a language. We've seen a series of languages, the different features they provide, and the different mechanisms by which they introduced and enforced those features. There is a constant strain between expressiveness and computability, between succinctness and decidable typing. In order to provide useful static analysis over partial specifications, there must be careful deliberation over the features that are useful to the Rosetta language, and the extent to which those features are implementable at this level of partial specification, so that this work provides a useful analysis that actually helps guide the specification process with useful information.

Chapter 3

Background Work

In order to provide the typing rules and typing analysis implementation, we discuss the Rosetta type system and earlier work towards a typing analysis. We first discuss the Rosetta language with a focus on features that concern the typing analysis, to understand the semantics we must address with typing rules. We then discuss previous work on a Rosetta typing analysis, showing the limitations that hindered the implementation. We last consider groundwork for the current imlementation, where the local type inference technique is applied to a reference graph-representation language prior to the Rosetta implementation.

3.1 The Rosetta Type System

This work provides formal typing rules for Rosetta's type system. To understand the type system we consider various features of Rosetta, discussing the issues each introduces for the type system. We saw in the related works how seemingly independent features can affect each other. The Rosetta type system itself is a static realization of Rosetta's semantics; any type system definition needs to successfully reconcile all these features. To that end, we explore some of Rosetta's features that are anticipated as the most pertinent in both language behavior and cross-cutting concerns in formally defining the type system.

3.1.1 Standard Types and Operators

In order to discuss the types of Rosetta, we need to first consider the ways in which basic values can be constructed and combined in Rosetta. Rosetta provides a rich set of primitive types, composite types, and a

standard library of functionality that operates over these basic types. Rosetta provides a semi-lattice of base types that encompasses numeric types and others (Fig. 3.1).

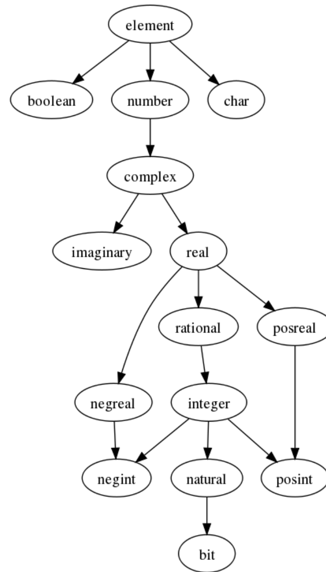

Figure 3.1: Rosetta Primitive Types

Rosetta provides many functions that operate over these base types. Functions themselves are first-class values. Combined with sub-typing, this complicates the typing analysis — the sub-typing relation is contravariant in function arguments, and covariant in its range, so strict type equality is no longer sufficient in admitting one function type as a subtype of another function type. Most numerical functions are defined at one point in the lattice of base types. For instance, `+`, `-`, `floor`, and `ceiling` are all defined over `complex` numbers. But their types are more complicated — if the arguments are sufficiently within subtype bounds, the result type is accordingly more constrained. For instance, addition (+) of two complex numbers yields a complex number, addition of two real numbers yields a real number, and addition of two integers yields an integer. This is a version of overloading that is baked into Rosetta's provided definitions, though users cannot create more overloaded functions themselves. Some functions can offer even tighter restrictions: if either argument to `max` is posreal, so is the resulting type.

These behaviors are particular to the basic numerical types, as well as to the function itself. Division (`/`) is not a closed operation as addition was — while dividing two real numbers yields a real number, dividing

two integers does not necessarily yield an integer. At least, since these types mirror actual mathematical numerical sets, we do not consider multiplication to be open due to some overflow criteria. There are separate functions, `div`, `rem`, and `mod`, for integer division. This behavior is inherent in these basic numeric types, but is not directly representable in the functions' types. Other languages such as Java [24] provide for explicit casting between types, provide overloading of functions at all possible signatures, or simply return less information in situations such as the `max` function's intelligent range considerations. Other languages provide for type-level representations of the behavior. Haskell provides type classes, giving a means for ad-hoc polymorphism by providing the implementation of a function at various types. If we define the `Ord` type class to include the `lessEq` function declaration, we can then create an `instance` of the `Ord` class for `String`s:

```
class Ord a where
   lessEq ::  a -> a -> Bool
   ...

instance Ord String where
   lessEq a b = a<=b

sort ::  (Ord a) => [a] -> [a]
```

It is then possible to use the `sort` function on lists of strings, because the compiler finds the `Ord String` instance and uses that implementation for `lessEq`. A type class constraint is semantically equivalent to passing a dictionary value of all available implementations of the type class as an argument and dispatching the correct version based on the types involved at each use of a function from the type class. The logic of the `sort` function's behavior is abstracted over the type class, and this gives a logical problem for the type system to solve in determining type safety and inferring types. The type system itself is not concerned with what addition over integers and reals means, it is concerned with type class requirements and finding the appropriate instance for the given types. Rosetta does not have a mechanism directly analogous to type

classes to describe constraints on the particular types in a function. Indeed, we could simulate something similar with `subtype` type declarations in Rosetta that we ascribe to different implementations fitting the desired function shapes, but this is both awkward, not an intended use in Rosetta, and ultimately is not a part of the typing of primitive numerical types and the provided functions over those types. The Rosetta type system itself will have to deal with these base types and the `Prelude` functions over them directly, or else their definitions will have to be refined appropriately.

3.1.2 Composite Types, Constructed Types

Rosetta provides some common mechanisms for building up the basic types into structured, larger values including sets, sequences, and multisets; for example:

```
1 setExample :: set(integer) is {1,3,5};
2 multisetExample :: multiset(integer) is {* 0,1,3,3,1 *};
3 sequenceExample :: sequence(natural) is [1,1,2,3,5,8,13];
```

These composite types allow us to collect values in the manner their names indicate. We can also create a constructed type definition, to generate constructor, observer, and recognizer functions over the new structure. For example,

```
1   Tree(a::type) :: type is data
2       leaf()::empty
3     | node(left::Tree(a);v::a;right::Tree(a))::nonempty
4     end data;
```

introduces the constructors `leaf` and `node`; the observers `left`, `v`, and `right`; and the recognizers `empty` and `nonEmpty`. This mechanism introduces abstract data types, which create sums across the constructors and products across the recognizers. This is a powerful way to construct values out of others. We can furthermore define types as subtypes of others by manually constructing new types, as below.

3.1.3 First-Class Types

Rosetta has first-class types — types can be constructed, passed as arguments, and can depend on other values in their construction. There is no semantic wall separating the world of values and the world of types. Instead of special type operators for defining functions over types, any Rosetta function may return a value or a type. The distinction is actually somewhat meaningless, since types *are* values in Rosetta. Thus there are no constraints on what parts of the language may be used to represent a type, and no limitation on the evaluation necessary to discover the type. We can create new types by constraining existing types, applying type functions such as the composite type operators, and we can also create new types without more information:

```
1 TeensInt::subtype(integer) is {13,14,15,16,17,18,19};
2 String::type is sequence(character);
3 Key::type;
```

the `subtype` type function creates a type that is known to be a subtype of the argument; when we actually supply a type value as above, we describe the type as a set. `Key::type` could have been written `Key::subtype (top)`, which is equivalent. This sets-as-types notation is useful for specifying new types explicitly, but may make for troublesome typing analysis. Consider the following Rosetta function definition:

```
1 addTI (x,y::TeensInt)::TeensInt is x+y;
```

clearly `x+y` will not return a `TeensInt` — addition is not closed over `TeensInts`. But for a type system to confirm this would require evaluating the expression. Beyond the point that 'evaluation' of a specification is generally not available (by virtue of being a specification, not an implementation), a typing analysis should shy away from performing possibly unbounded computation in providing type safety. This issue of value computation in typing also arises in our example from 2.3. In Rosetta, the package `rosetta.lang.prelude` provides the following definitions:

```
1  array (N::natural; T :: type) :: type;
2  sequence [N::natural] (T::type) :: type is array (N,T);
3  bitvector :: subtype (sequence) is sequence(bit);
4  word(n::natural)::subtype(bitvector) is
5      sel (b::bitvector | #b = n);
```

Arrays are sized compositions of `T`-typed values. `sequence` allows the size parameter `N` to be quantified/inferred. `bitvector` narrows down to sequences of `bits`, and `word` goes further, restricting the values to bitvectors of a particular size. Literal sequences may be simple enough to check their contained type `T` and length `N`, but results of functions and operators that yield a `word(n)` can quickly require calculation and synthesis of our modeled system in order to check that we actually get the `word` result of length `n` indicated. We must declare what properties the typing analysis can guarantee, and what proof-obligations it might be able to provide. Typing analysis can provide a theorem in some proof assistant that can be discharged manually, and recorded as evidence to the overall specification's correctness. In general since the typing analysis is a Rosetta-wide analysis, any dependent definition that is not built into the Rosetta language and default libraries cannot be checked. It may be possible for a typing implementation to provide some sort of hook into the typing mechanism to aid typing these dependent items, but they cannot be directly handled without some assistance.

3.1.4 Building Blocks - Facets

The basic unit of a specification is called a *facet*. A facet typically represents a structural or logical component of a system. In Fig. 3.2, we see the `halfAdder` facet definition instantiated twice in defining the `fullAdder` facet. The facets contain parameter lists, a declarative region before the `begin` keyword, and a section for instantiating units and for behavioral boolean statements that must be true. Beyond facets, we can define `components` — essentially facets with explicit sections for assumptions and implications, to guide constraints on combined components.

On the whole, Rosetta specifications may largely be a collection of facets and components that instantiate each other to build up the overall structure of a system, and to record, collect, and propagate constraints on

```
facet halfAdder (x,y:: input bit; s,c::output bit)::state_based is
begin
  s' = x xor y;
  c' = x and y;
end facet halfAdder;

facet fullAdder (x,y,ci::input bit; s,co::output bit)::state_based is
  s1,c1,c2 :: bit;
begin
  ha1: halfAdder(x,y,s1,c1);
  ha2: halfAdder(s1,ci,s,c2);
  co = c1 or c2;
end facet fullAdder;
```

Figure 3.2: Sample Rosetta facets.

behavior through the behavioral statements in assumptions and implications. Equality(=) is true Liebniz equality.

In the above specification, we also see the domain ascription, `state-based`. This indicates that state-based models of computation are available, in particular the tick operator to indicate the value in the next state. `s' = x xor y` means that `s` in the next state must be equivalent to current the value of `x xor y`.

Rosetta contains far more than just the `state_based` domain, providing a lattice of domains (Fig. 3.3). The language allows for extensions of this domain hierarchy in the form of sub-domains that offer more concrete and specific semantics, facet-extensions of domains that crystallize usage of a particular domain, and even simple engineering domains that provide a set of common idioms within a domain specific to an engineering discipline.

3.1.5 Reflection and Interactions

Rosetta provides reflection capabilities. We can manipulate the Rosetta language as data in Rosetta itself. This provides a meta-programming environment in which we can write a specification that defines a specification. This of course will complicate the typing analysis — reflection can break strong typing by defining badly-typed expressions that cannot be spotted until evaluation. We've already discussed how mixing evaluation and typing is a dicey affair, and reflection directly forces any analysis to interleave with evaluation,

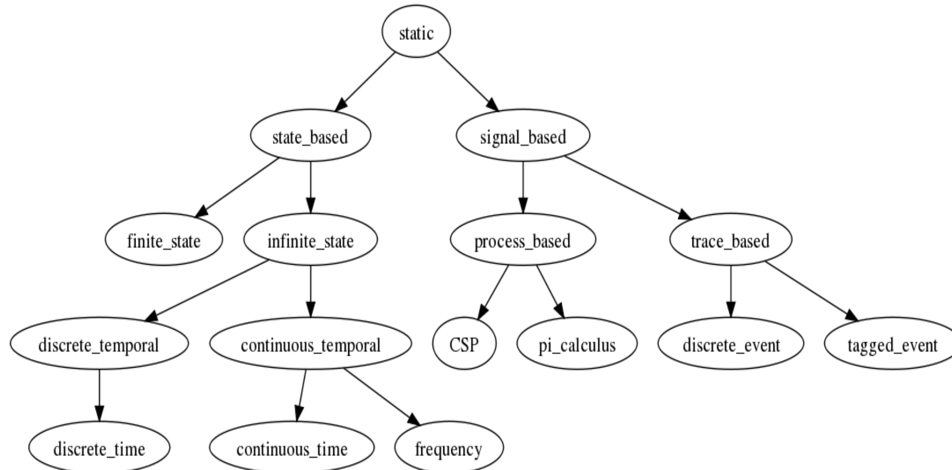

Figure 3.3: Rosetta Domain Lattice.

via the `denotes` function. While we can analyze the reflective terms themselves for type safety, we cannot provide guarantees of type safety in the code 'generated' by the reflective Rosetta code.

One of the primary goals of Rosetta is to semantically define how specifications in different domains interact. When a system is specified utilizing multiple domains, decisions in one domain affect the design in other domains. Rosetta allows for defining these inter-domain interactions via projection functions, functors, and combinators. Projection functions allow us to 'cast' a facet into a value of another domain, essentially defining what the view of one domain's definitions looks like from another domain. These projections are defined pair-wise between domains. Functors are projections that transform a facet of one domain into a facet of another domain, a sort of higher-order transformation between domains, perhaps to bring two domains to a common super-domain for analysis. Combinators provide means for combining multiple facets into one, usually by product or sum operations defining the sharing that should be evident between the combined facets. Combinators are defined in terms of projection functions and functors. Interactions are an advanced Rosetta feature that have not been extensively used at present. Accordingly, this work does not specifically address interactions.

3.1.6 Rosetta Type System Summary

Having seen the majority of the most interesting features of Rosetta from a typing perspective, it is clear that writing specifications in Rosetta may begin in, and for some time occupy, a design space that cannot by fully analyzed for type safety. And yet the earlier we can find a problem the better; we still need typing information at these early stages. This work provides a partial typing analysis of Rosetta to alleviate this lack of information for high-level, partial specifications.

3.2 Rosetta Alpha Type Checker

There has already been substantial effort in developing a typing analysis for Rosetta. The first analysis implementation employed modular monadic semantics [27], along with InterpreterLib [45], to break up the process into a series of sequenced algebras that collectively teased out a global set of constraints that it would attempt to unify. While some real progress was made, this codebase highlighted some of the issues in typing the Rosetta language. Some of those issues made the implementation fatally flawed. We turn our attention to the implementation in greater detail, pointing out the strengths and weaknesses along the way.

This initial typing work and its overall approach is heavily based upon InterpreterLib[45], a Haskell library aiding in defining interpreters and analyses. With InterpreterLib we can modularly define the syntax and semantics of a language. It uses many generic programming techniques to provide a modular monadic semantics. InterpreterLib provides modularity of syntax by supporting the use of sum types and injections/projections into and out of those sums. In this way, different pieces of syntax can be combined to define the language. InterpreterLib is used to define semantics via algebras and the catamorphism (a general fold), allowing us to inject the recursion of the analysis at the last moment, further modularizing the process. Semantics are modularized both in utilizing the same functor-sum structures as the underlying syntax, but also by composing entire algebra-analyses in various ways. Monads are used to simulate side effects in the pure language Haskell.

I briefly summarize InterpreterLib usage, as it pertains to the first Rosetta type checker. An intermediate representation for the Rosetta language constitutes the Rosetta AST, being a set of functors each representing different features of the language such as facets, expressions, types, and functions. By creating the sum of all

of these functors, we have a representation of the Rosetta language that is extensible by simply adding more functors. We next write algebras, defining the results of an analysis. These algebras are written per-functor, further maintaining the modularity of the design. We split the entire typing analysis into a series of algebras, much like the separate passes of a compiler. We connect these analyses together by sequencing the algebras, providing results of prior algebras to any further-sequenced algebras that want to use the information. The algebras are as follows:

- The identity algebra paramorphically allows us to inspect the original AST's structure; normally algebras only see their sub-terms' results.
- We generate type variables for each node in the AST, report all scope-items created per-node.
- We propagate scope-items' definitions, build (and result in) a symbol table for each node.
- We use that symbol table information to generate constraints on the node's type, return representation of the node's type
- We unify constraints to solve for type variables, generating a set of substitutions for all the type variables.
- substitute for all type variables (with a generic traversal) in symbol tables, which can be serialized on a per-design-unit basis.
- substitute for type variables (with a generic traversal) in nodes' type results, yielding a fully type-annotated abstract syntax tree.

Notice that almost half of the typing analysis is concerned with name management, in propagating the original structure and constructing symbol tables. Rosetta has a rich library, package, and facet structure where names arise from many places. All of the original tools over Rosetta had to duplicate this name-chasing, which was both an error-prone and redundant effort. Mutually recursive elements also proved difficult — while defining values in terms of each other may seem perfectly suited to Haskell, we still must be able to terminate the calculations based on this dependency. Using the `mdo` notation in Haskell provided a workable but highly brittle and unsatisfactory work-around to the issue.

The unification of the constraints was arguably the weakest link in the process. It was based on a unification algorithm [16] that did not account for sub-typing, trying to adapt it to sub-typing. Rosetta is more expressive than System F_2 in that we can have type parameters that are inferred. As we saw [48], this is equivalent to semi-unification and is undecidable. Different iterations of the unification process were at times a flawed greedy algorithm, at other times a non-terminating process but for a hard limit to the iterations of the algorithm's tactics.

Added to this difficulty is the typing for elements of Rosetta's prelude such as the + and = operators. As mentioned in §3.1.1, Rosetta does not use types to indicate the overloaded nature of these operators. In determining a best type for these operators, the typing analysis would create a series of bookkeeping constraints that turned out to be semantically similar to the type schemes of Algorithm W [16] – there would be a series of universally quantified parameters and a series of constraints to be added, all of which would be instantiated at each usage of the offending operator. In this way, the typing analysis could handle uses of the equality operator = for booleans, integers, and any other types, as long as they all obeyed the constraints over some new type variable α that `(=)::`$\alpha \to \alpha \to \alpha$. More complex constraints and bookkeeping would be necessary for operators such as +. We can reduce the possible typings of + to a few closed versions, such as `(+)::integer→integer→integer` or `(+)::real→real→real`, but not just any type is acceptable as it was in `(=)`'s case. Constraints could be strict equalities, subtype restrictions, set-membership for a given type, even a disjunction of further constraint sets. In this way, we could require e.g. that division (/) either satisfy one set of constraints or some other, without the various legitimate typings all being enforced at once. Using set-membership, = was actually only allowed over the base types by including the constraints that `(=)::`$\alpha \to \alpha \to \alpha$, but also that $\alpha \in${`boolean, character, bit, natural, integer,...complex` }. Overall, while this expanded the portion of Rosetta that could be type-checked, it introduced some very expensive backtracking.

Global unification in this way also makes for poorer error messages. Bad typings can propagate far away from their original locations, and at best the typing analysis could say, "I couldn't combine these two constraints that came from here and here", or more frustratingly, "I had to give up on unification, here are the remaining constraints." The former is at least targeting what might be the location of the problem, but the latter simply exposes the lack-of-progress check required to halt the undecidable cases of typing. More

localized constraints and unification could much more precisely pinpoint the exact location of an error, as well as what the error truly is.

Overall, names management got in the way of the typing analysis; global unification was not quite the right approach to ferret out needed ascriptions and the backtracking efforts were sound but horribly inefficient. There was no sound basis for the decisions made in this implementation — no formal system to answer questions of proper behavior.

3.3 Preparatory Typing Work

Resuming typing analysis efforts that learn from the previous implementation's experience, names management in Rosetta has now become its own analysis [33], and we've implemented approaches similar to local type inference as in §2.2.2.

The naming analysis turns the original Rosetta syntax into the same AST, but further resolves all names, constructing an abstract syntax graph (ASG) with all names resolved. This involved process even β-reduces facet instantiations to create nodes representing named items exposed in that instantiation, among many other operations to provide a graph with no more need for a naming environment. The end result is that any analysis — not just type-checking — can operate on a *scopeless abstract syntax* [33], reaping the benefits without duplicating the work as our original tools did. The graph presents the nodes as strongly-connected components (SCC's). Instead of the algebras of InterpreterLib, we instead can write node decorators for the acyclic and cyclic components. This maintains the synthetic information flow from sub-terms upwards, but cuts off the inheritance information flow from the top of the AST downwards, as the graph may have multiple paths from the focus or 'root' of the graph to a particular node.

3.3.1 Local Type Inference on ASG's

The generated graphs originally began as AST's, but some reference nodes were pointed back into the tree, creating the graph. For this reason, it makes sense to view the graph as a tree with back-edges. If we maintain annotations of which sub-term edges are references, we can regain the top-down flow of information, which is the way researchers often think about type systems and typing environments. In doing so, we no longer

have the identity of cycles automatically detected (to be handled via separate cyclic and acyclic definitions of node decoration), but we can still manually handle these situations. Overall, this modified approach to ASG's places slightly more responsibility on the programmer (identifying cycles and satisfying them), but provides more information than was available in InterpreterLib (the 'was-a-reference' annotations) and does not restrict the flow of information either up or down the structure.

We explored local type inference such as in §2.2.2 on an example language utilizing a graph structure, demonstrating that it is still amenable to a graph-based structure. The tree-with-back-edges approach allows us to enjoy the benefits of naming analysis performed by the graphing analysis, while still performing an analysis defined in top-down fashion such as local type inference.

3.3.2 Error Reporting

One deficiency of the original typing analysis was poor support for error reporting. In the Haskell code of this iteration of typing analysis, a Writer monad is used to collect errors. An error type is reported at the failing node, and analysis can continue, with the possibility that the error will not spread too far away. The original typing analysis would quit at the first sign of a problem, which occurred at the stage of global constraints unification. One hard-to-parse error message would sometimes be all that was reported. Now, we can report a series of typing errors. Some of these may of course be the result of a propagated error, but more information is generally a good thing. This approach is much better suited to the intent of providing type information in the face of missing or wrong typing information in a specification.

3.4 Summary

Overall, the previous work and related work guide the way forward for this work. The type system is formally defined relying on the insight into features' interactions; splitting the typing rules into various phases such as in System $F_C(X)$ simplifies the rules. We know where sources of undecidability are likely to come from (such as the System F_2 expressibility, unhindered dependent typing), indicating what partial analyses are possible in an implementation. We have practical issues such as naming conventions handled in the graph representation, and tactics such as local type inference that are viable in this approach.

Chapter 4

Methodology

4.1 Introduction

We discuss the specific tasks and approaches used in developing the typing analysis for Rosetta. This typing analysis is defined to operate over partial Rosetta specifications. The analysis attempts to infer types, collect constraints, and check for properties and information that would be useful to the specification refinement process.

In order to implement the analysis, I use the graph analysis techniques discussed in order to handle naming conventions in Rosetta. By leveraging the scopeless abstract syntax work, the typing analysis gets name resolution 'for free', which also keeps the tool aligned with other research efforts into the Rosetta language in the SLDG laboratory.

Given the output graph of the naming analysis, I use the Reference Algebra work to treat this graph structure as a tree with identified back-edges. This approach recovers the top-down traversal strategy of many typing analyses, departing from the topologically-sorted traversals of scopeless abstract syntax in favor of the more familiar top-down traversals common to many type analyses. This traversal is essential for both identifying and handling cycles as well as mutual recursion that arises in specifications. This trades for the ability to order the traversal over nodes, at the expense of manually ensuring that typing information is available for particular nodes when needed.

Within this reference algebra, I first apply a basic typing analysis, and then scale up to analysis emphasizing multiple-error recovery, partial typing analysis, and local type inference. The goal is to provide

analysis over under-specified specifications, and so all the techniques should cater to (or at least deal with) the expected lack of information in the specifications. There should be opportunities related to abstract types in Rosetta, types that are defined but not given a specific set of values. The analysis should result in either complete concrete typing information, specific typing errors, or all typing information observed and inferred for given types in the specification, indicating what constraints must be met or what conflicts may arise.

The tree-with-back-edges view will expand on the graph structure to allow implementation of features such as local type inference, indicating that the graph structure will not create further problems 'downstream'. The graph structure has already been tested with a tree-with-back-edges style catamorphism on a toy language, yielding positive results in a narrower scope. This approach also yields more localized and better error reporting than previous typing efforts in Rosetta. In this way, the reference algebra approach stays in the familiar top-down and bottom-up flow of information while benefiting from the scopeless abstract syntax work at the same time.

4.2 Defining Rosetta Typing Rules

I use the formalisms as in the language definitions above in providing a formal definition of the Rosetta type system. In particular, splitting the rules into various phases such as type-checking and kind-checking further serves to codify the semantics of the Rosetta language. Similar to System $F_C(X)$, we can separate environment extension from typing analysis with separate rules for generating the Γ related to a Rosetta construct such as a facet or package.

We now formalize rules to define the Rosetta typing semantics embodied by the typing analysis. It is important to realize that this is not intended to fully define the semantics of the Rosetta language, nor prove any properties ove the type system. That would be a separate dissertation unto itself. The point of these rules is to formalize the semantics that the typing analysis utilizes, providing a basis for discussing the typing semantics of the analysis. Furthermore, we are not trying to present these rules as being an algorithmic set of rules. Again, we are just formalizing the logic inherent in the typing analysis.

In order to define rules of the typing system, we define the notion of an environment Γ. An environment is a relation between names and types that gives meaning to some expression in the language. As we define

typing rules over constructs that introduce names through parameters and definitions, we would then need to extend Γ to include this new name and ascribed type in order to reason about the body within the construct.

Γ is simple: it contains labels with types, a label with a Γ and list of parameter types and another Γ, or it contains more Γ's via concatenation.

$$
\begin{array}{rcl}
\Gamma & = & \{\} \\
& | & label : T, \Gamma \\
& | & <label:(\overline{p} : \overline{P})|\Gamma>, \Gamma \\
& | & \Gamma, \Gamma
\end{array}
$$

A label is just a name, possibly qualified (e.g., `x.y.f`). Since the typing analysis is defined after the graph analysis and names resolution, we can assume that all labels are fully qualified, with no chance for name shadowing.

In order to understand how we use Γ, we need to think of type analysis as a mix of two phases–defining $\vdash_\gamma$ through rules that generate the Γ associated with some design unit (e.g. facet, component, package, domain), and defining $\vdash_\tau$ through rules that find a typing for any expression in the language. The two are mutually recursive: when finding Γ for a facet, we may need to reason via $\vdash_\tau$ to decide what Γ to generate. Likewise, when finding something's type, we may need to find Γ for a used package or something else via $\vdash_\gamma$ to add to the context when typing it. They are intermingled, utilizing each other. This practice is seen in other language definitions, notably System F_c(X).

In an effort to make these rules somewhat more algorithmic (though not provably so), I am assuming that $\vdash_\gamma$ does not perform type checking except where necessary to find Γ. Rather, it assumes the unit is type-safe, and it is the burden of other rules to enforce this type-safety. This is exactly how letrec is often handled: accept its ascribed type, put that in Γ, type check to find out if it was valid. This lends to the reasoning that the two analyses ($\vdash_\gamma$,$\vdash_\tau$) achieve termination. Even though both may refer to each other, $\vdash_\gamma$ usually does not need to refer to $\vdash_\tau$, instead it just records named entities such as parameters and definitions with their types.

We adopt certain syntactic conventions and make certain organizational choices to make the rules more compact and readable. The syntax for function types is written not as types in Rosetta, instead favoring the style found in many functional languages; thus, we prefer $\overline{p} \rightarrow r$ to $<\!*(\overline{p}) : r*\!>$. We present the $\vdash_\gamma$ and $\vdash_\tau$ relations together for each construct in turn, rather than all $\vdash_\gamma$ relations followed by all $\vdash_\tau$ relations. We also make another simplification in types representation: normally, representations of non-dependent types use $\rightarrow$ while dependent types use Π. But in Rosetta, there is no restriction on the value-dependence of types, and therefore little to be gained in making the distinction beyond perhaps duplicating rules for purely syntactic reasons. Therefore, we just use $\rightarrow$, understanding that a type on the left of an arrow is likely to be a type-ascribed name where that name may show up on the right side of the arrow in a dependent fashion. We similarly allow for a multi-parameter situation to be represented in uncurried tuple fashion, much like the signatures of Rosetta themselves.

We organize the analysis as a top-down tree walk, so we can consider an environment to exist as the set of node type annotations that must be present in the results graph. It is the task of the analysis to ensure that some type annotation is present for each node that corresponds to a label in the environments of these rules. In this way, although we do not explicitly generate the environments described in these rules, those very environments are embedded in our results graph. At the same time, the names analysis has already checked that each named entity is a legal reference to a name, so we are guaranteed that our typing analyis should not be looking anywhere outside of the actual allowed scope at any time. This provides an upper bound (all accessible nodes post-names resolution) and a lower bound (all previously visited nodes) on what may be in the environment. As long as the analysis actually visits all nodes named in an environment (or records a delayed type), we can informally argue that the environment is present, complete, and will not incur any interference from ‘outside’ information akin to arbitrary extensions to the environment from extraneous results that have been recorded in the results graph.

4.2.1 A Note on Lists

Throughout these rules, we use the familiar vector notation $\overline{p}$ to represent a list of items $p_1, p_2, ...p_n$. Parameters are ordered, meaning that a specific parameter is visible in all subsequent parameters. This fact is not readily representable in the rules, and is therefore assumed, for parameters only. When we see $\Gamma \vdash_\tau \overline{p} : \overline{P}$,

the assumption is that we actually mean the chain of claims:

$\Gamma \vdash_{\tau} p_1 : P_1$

$\Gamma, p_1 : P_1 \vdash_{\tau} p_2 : P_2$

...

$\Gamma, (p_1 : P_1), \ldots (P_{n-1} : P_{n-1}) \vdash_{\tau} p_n : P_n$

$\Gamma \vdash_{\tau} \overline{p} : \overline{P}$

where the final claim is all that we normally show, assuming the interplay between parameters and their environments.

Other vector representations, such as declarations and definitions, are mutually defined, and do not have these orderings. Fortunately, Rosetta in general requires type ascriptions for these batches of items, and so we can instead record the name-ascription pairs in bulk in order to then check for the type of them, regardless of order.

4.2.2 Qualified Names

Although the graph analysis resolves all names, we can reason about the name resolution it performs. When we construct the environment $\Gamma_{a.b}$ for a qualified name `a.b` in some Γ, we must reason in terms of extending that local environment Γ with the environment Γ_a of `a` in order to construct the environment for `a.b`.

- **G-Qualified**.

  $\Gamma \vdash_{\gamma} a : \Gamma_a$

  $\Gamma_a, \Gamma \vdash_{\gamma} b : \Gamma_b$

  $\overline{\Gamma \vdash_{\gamma} a.b : \Gamma_b}$

- **T-Qualified**.

  $\Gamma \vdash_{\gamma} a : \Gamma_a$

  $\Gamma_a \vdash_{\tau} b : T_b$

  $\overline{\Gamma \vdash_{\tau} a.b : T_b}$

4.2.3 Design Units

A facet definition has a domain, parameters, definitions, and declarations. The facet is an extension of that domain, so all domain definitions are present and visible all throughout the facet. Declarations are not ordered: they are mutually defined. All parameters are visible in the declarations. Definitions (the 'body' of the facet) are also mutually defined between themselves; all parameters and declarations are visible.

- **G-Facet**

$$\frac{\begin{array}{l}\Gamma \vdash_\gamma dom : \Gamma_{dom} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P} \vdash_\tau \overline{decl} : \overline{Decl} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P}, \overline{decl} : \overline{Decl} \vdash_\tau \overline{defn} : \overline{Defn} \\ \Gamma_d = \Gamma_{dom} \cup \{\overline{p} : \overline{P}, \overline{def} : \overline{Def}, \overline{decl} : \overline{Decl}\}\end{array}}{\Gamma \vdash_\gamma facet\ f(\overline{p} :: \overline{P}) :: dom\ is\ \overline{decl}\ begin\ \overline{defn}\ end\ facet\ f : \Gamma_d}$$

- **T-Facet**

$$\frac{\begin{array}{l}\Gamma \vdash_\gamma dom : \Gamma_{dom} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P} \vdash_\tau \overline{decl} : \overline{Decl} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P}, \overline{decl} : \overline{Decl} \vdash_\tau \overline{defn} : \overline{Defn}\end{array}}{\Gamma \vdash_\tau facet\ f(\overline{p} :: \overline{P}) :: dom\ is\ \overline{decl}\ begin\ \overline{defn}\ end\ facet\ f : dom}$$

A facet instantiation is a fully-applied facet definition, meaning that all parameters are supplied. (A partially-applied facet definition may be considered a function from the arguments to a facet instance). These parameters must be substituted all throughout the facet. The resulting type of a facet instantiation is this substituted version of the facet definition's type, which will be the domain specified.

- **G-FacetInstantiation**

$$\frac{\begin{array}{l}\Gamma \vdash_\tau \overline{a} : \overline{P} \\ <f:(\overline{p} : \overline{P})|\Gamma_f> \in \Gamma\end{array}}{\Gamma \vdash_\gamma f(\overline{a}) :: [\overline{p} \mapsto \overline{a}]\Gamma_f}$$

- **T-FacetInstantiation**

  $\Gamma \vdash_\tau \overline{a} : \overline{P}$

  $<f :(\overline{p} : \overline{P})|\Gamma_f> \in \Gamma$

  $\overline{\Gamma \vdash_\tau f(\overline{a}) :: [\overline{p} \mapsto \overline{a}]f}$

We define packages, components, and domains with similar typing rules. We leave the actual rules to the appendix (A.1).

4.2.4 Use Clauses

A use clause allows a package to be imported, such that all definitions within it are visible for a given design unit such as as facet, domain, component, or other package. It behaves just like a giant, special `let` binding.

- **G-Use**

  $\Gamma \vdash_\gamma p : \Gamma_p$

  $\Gamma, \Gamma_p \vdash_\gamma decl : \Gamma_{decl}$

  $\overline{\Gamma \vdash_\gamma use\ p\ in\ decl : \Gamma_{decl}}$

- **T-Use**

  $\Gamma \vdash_\gamma p : \Gamma_p$

  $\Gamma, \Gamma_p \vdash_\tau decl : T_{decl}$

  $\overline{\Gamma \vdash_\tau use\ p\ in\ decl : T_{decl}}$

4.2.5 Quantified Parameters

As mentioned in §2.2.3, when local type inference is capable of identifying a principal type for a type parameter, we will capture the claim that local type inference can find a principal type for a particular omitted type parameter, through the `infer` relation. The required information is an application of a function with inferrable parameters $\overline{Q}$ and explicit parameters $\overline{p}$, and supplied arguments $\overline{a}$. We can then represent the `T-App-InfAlg` rule's applicability thus:

$$\Gamma \vdash_\tau f : [\overline{Q}](\overline{p} : \overline{P}) : R$$

$$\Gamma \vdash_\tau \overline{a} : \overline{A}$$

$$\{\} \vdash_\tau \overline{A} <: \overline{P} \Rightarrow \overline{C}$$

$$\overline{I} = \sigma \overline{C}_R$$

$$\overline{\text{infer}(\overline{Q}, \overline{p} : \overline{P}, R, \overline{a} : \overline{A}) \Rightarrow \overline{I}}$$

If we're given the function signature `f[Q]`$(\overline{p}:\overline{P})$`:R`, and arguments $\overline{a}:\overline{A}$, and if ensuring that $\overline{A} <: \overline{P}$ implies the constraint set `C` upon the inferrable parameters $\overline{Q}$, then we can define $\overline{I}$ as the best set of substitutions for inferrable types $\overline{Q}$ when solving the constraint set `C` with respect to the type `R`). `infer` doesn't refer to f directly, instead relating its quantified parameters, explicit parameters, range type, and arguments to the best set of type substitutions for the inferrable quantified parameters.

Because this `infer` relation only shows up in a few places in the Rosetta rules, we do not re-write all of the typing rules from local type inference for Rosetta directly. They would be identical up to the language syntax differences. Lambdas and applications become any parameterizable definitions and function calls/instantiations; `Top` and `Bottom` also exist in Rosetta; and Rosetta also defines the subtyping relation `<:`.

Of course, in Rosetta we are only allowed to infer quantified parameters; explicit parameters must be supplied, even if they are inferrable. Quantified parameters may seem to be conflated with inferrable parameters, but the distinction between the two is made by the specifier by choosing whether to place a type parameter in the quantified parameters list of the explicit parameters list.

4.2.6 Applications

Applications occur whenever we supply parameters. This can apply to facet definitions, functions, anything with parameters. The dependent nature of Rosetta means that we must perform substitutions for our parameters. As the orderedness of parameters affects their environments, so too does the orderdness affect the substitution of parameters. Since a parameter is visible in all subsequent parameters, we substitute parameters one at a time, through both the remaining parameters and the range of the function.

- **G-Application**

  $\Gamma \vdash_\tau f : (\overline{p : P}) \to R$

  $\Gamma \vdash_\tau \overline{a : P}$

  $\Gamma, \overline{p : P} \vdash_\gamma R : \Gamma_R$

  $\overline{\Gamma \vdash_\gamma f(\overline{a}) : \Gamma_R}$

- **T-Application**

  $\Gamma \vdash_\tau f : \overline{p : P} \to T$

  $\Gamma \vdash_\tau \overline{a : P}$

  $\overline{\Gamma \vdash_\tau f(\overline{a}) : T}$

- **G-Application-Infer**

  $\Gamma \vdash_\tau f : [\overline{Q}](\overline{p : P}) \to R$

  $\Gamma \vdash_\tau \overline{a : A}$

  $\text{infer}(\overline{Q}, \overline{p : P}, R, \overline{a : A}) \Rightarrow \overline{I}$

  $\underline{\Gamma, \overline{p : P} \vdash_\gamma [\overline{Q} \mapsto \overline{I}]R : \Gamma_R}$

  $\Gamma \vdash_\gamma f(\overline{a}) : \Gamma_R$

- **T-Application-Infer**

  $\Gamma \vdash_\tau f : [\overline{Q}](\overline{p : P}) \to R$

  $\Gamma \vdash_\tau \overline{a : A}$

  $\text{infer}(\overline{Q}, \overline{p : P}, R, \overline{a : A}) \Rightarrow \overline{I}$

  $\overline{\Gamma \vdash_\tau f(\overline{a}) : [\overline{Q} \mapsto \overline{I}, \overline{p} \mapsto \overline{a}]R}$

4.2.7 Variables

A variable's type is just a lookup into the environment: the variable must be in the environment for any well formed expression, and thus the lookup will not fail if the scope-introducing construct correctly extended the environment as we require.

- **T-Var**

  $$\frac{t : T \in \Gamma}{\Gamma \vdash_\tau t : T}$$

4.2.8 Functions

Rosetta provides multiple ways of defining functions. The body of a function may be omitted, may be defined, or may be declared as `constant` without giving a specific value. A function definition also may include a `where` clause, placing constraints upon the values via `boolean` expressions that must be true, without directly specifying the value of the function. These various syntactic forms all obey the same typing rules: the body of the function must be of the type of the range of the function, and `where`-clauses must contain a boolean expression.

- **T-Function-Anonymous**

  $$\frac{\Gamma, \overline{p : P} \vdash_\tau t : T}{\Gamma \vdash_\tau \; <*(\overline{p :: P}) :: T \text{ is } t*> : \; (\overline{p :: P}) \rightarrow T}$$

- **T-FunctionApplication**

  $$\frac{\Gamma \vdash_\tau \overline{a : P} \quad < f(\overline{p : P}) :: T > \in \Gamma}{\Gamma \vdash_\tau f(\overline{a}) \; : \; (\overline{p : P}) \rightarrow T}$$

4.2.9 Direct Functions

- **T-FunctionInterpretable**

  $$\frac{\Gamma, \overline{p : P} \vdash_\tau t : T}{\Gamma \vdash_\tau f(\overline{p :: P}) :: T \text{ is } t; \; : \; (\overline{p : P}) \rightarrow T}$$

- **T-FunctionUninterpretable**

  $$\frac{}{\Gamma \vdash_\tau f(\overline{p :: P}) :: T \text{ is constant}; \; : \; (\overline{p : P}) \rightarrow T}$$

- **T-FunctionQualifiedInterpretable**

  $$\frac{\Gamma, \overline{p : P} \vdash_\tau t : T \quad \Gamma, \overline{p : P} \vdash_\tau t_w : \textbf{boolean}}{\Gamma \vdash_\tau f(\overline{p :: P}) :: T \text{ is } t \text{ where } t_w; \; : \; (\overline{p : P}) \rightarrow T}$$

- **T-FunctionQualifiedUninterpretable**

  $$\frac{\Gamma, \overline{p : P} \vdash_\tau t_w : \textbf{boolean}}{\Gamma \vdash_\tau f(\overline{p :: P}) :: T \text{ is constant where } t_w; :\ (\overline{p : P}) \to T}$$

- **T-FunctionVariable**

  $$\frac{}{\Gamma \vdash_\tau f(\overline{p :: P}) :: T; :\ (\overline{p : P}) \to T}$$

- **T-FunctionQualifiedVariable**

  $$\frac{\Gamma, \overline{p : P} \vdash_\tau t_w : \textbf{boolean}}{\Gamma \vdash_\tau f(\overline{p :: P}) :: T \text{ where } t_w; :\ (\overline{p :: P}) \to T}$$

4.2.10 Anonymous Functions

- **T-FunctionFormer**

  $$\frac{\Gamma, \overline{p : P} \vdash_\tau t : T}{\Gamma \vdash_\tau \ <*\overline{p :: P}*>\ \text{ is } t\ :\ (\overline{p : P}) \to T}$$

- **T-FunctionValue**

  $$\frac{\Gamma, \overline{p : P} \vdash_\tau t_{is} : T \quad \Gamma, \overline{p : P} \vdash_\tau t_W : \textbf{boolean}}{\Gamma \vdash_\tau \ <*(\overline{p :: P}) :: T \text{ is } t_{is}*>\ \text{ where } t_w; :\ (\overline{p : P}) \to T}$$

4.2.11 Further Binding Sites: Functions and Lets

We consider some other locations where names may be bound. The main consideration is that parameters are in the environment of the body of the defined scope. `Let`-expressions do not have ascriptions on them, and so we do not see quantified parameters in this rule itself; a function definition bound by a `let`-expression may itself introduce quantified parameters, but that would therefore concern the `function` typing rules instead.

- **G-Function**

  $$\frac{\Gamma, \overline{q : Q}, \overline{p : P} \vdash_\tau v : T \quad \Gamma \vdash_\gamma v : \Gamma_v \qquad \Gamma, \overline{q : Q}, \overline{p : P} \vdash_\tau b : \textbf{boolean}}{\Gamma \vdash_\gamma \ f[\overline{q :: Q}](\overline{p :: P}) : T \text{ is } v \text{ where } b; :\ \{f : [\overline{q : Q}]((\overline{p : P}) \to T) | \Gamma_v\}}$$

- **T-Function**

  $\Gamma, \overline{q : Q}, \overline{p : P} \vdash_\tau v : T$

  $\Gamma, \overline{q : Q}, \overline{p : P} \vdash_\tau b : \textbf{boolean}$

  $$\frac{}{\Gamma \vdash_\tau f[\overline{q :: Q}](\overline{p :: P}) :: T \text{ is } v \text{ where } b; : [\overline{q : Q}](\overline{p : P}) \rightarrow T}$$

- **G-Let**

  $\Gamma, \overline{x : X} \vdash_\tau \overline{t : X}$

  $\Gamma, \overline{x : X} \vdash_\gamma e : \Gamma_e$

  $$\frac{}{\Gamma \vdash_\gamma \text{let } \overline{x :: X \text{ be } t} \text{ in } e \text{ end let}; : \Gamma_e}$$

- **T-Let**

  $\Gamma, \overline{x : X} \vdash_\tau \overline{t : X}$

  $\Gamma, \overline{x : X} \vdash_\tau e : T_e$

  $$\frac{}{\Gamma \vdash_\tau \text{let } \overline{x :: X \text{ be } t} \text{ in } e \text{ end let}; : T_e}$$

4.2.12 Sequence Predicates and Operators

Rosetta provides some predicate functionality that can be applied across sequences. `forall` allows us to express as a boolean expression that all elements of a set, and therefore all elements of a type, obey some predicate `pred`. Of course, this boolean expression may turn out to be false – this does not make an assertion, it is an encoding of the $\forall$ symbol.

- **G-Forall**

  $$\frac{}{\Gamma \vdash_\gamma \text{forall}(x : T \mid pred) : \{\}}$$

- **T-Forall**

  $\Gamma, x : T \vdash_\tau pred : T \rightarrow \textbf{boolean}$

  $$\frac{}{\Gamma \vdash_\tau \text{forall}(x : T \mid pred) : \textbf{boolean}}$$

`filter` allows us to filter a sequence with a predicate, resulting in the previous sequence with only elements that upheld the predicate.

- **G-Filter**

  $\Gamma \vdash_\gamma \; seq : \Gamma_s$

  $\overline{\Gamma \vdash_\gamma \text{filter}(pred, seq) : \Gamma_s \setminus \{x : T | pred(x) = \textbf{false}\}}$

- **T-Filter**

  $\Gamma \vdash_\tau pred : T \rightarrow \textbf{boolean}$

  $\Gamma \vdash_\tau s : \textbf{sequence}(T)$

  $\overline{\Gamma \vdash_\tau \text{filter}(pred, s) : \textbf{sequence}(T)}$

4.2.13 Control-Flow Expressions

- **G-If**

  $\Gamma \vdash_\gamma t : \Gamma_t$

  $\Gamma \vdash_\gamma f : \Gamma_f$

  $\Gamma_R = \Gamma_t \cap \Gamma_f$

  $\overline{\Gamma \vdash_\gamma \text{if } b \text{ then } t \text{ else } f \text{ end if; } : \Gamma_R}$

- **T-If**

  $\Gamma \vdash_\tau b : \textbf{boolean}$

  $\Gamma \vdash_\tau t : T$

  $\Gamma \vdash_\tau f : F$

  $R = T \sqcap F$

  $\overline{\Gamma \vdash_\tau \text{if } b \text{ then } t \text{ else } f \text{ end if; } : \; R}$

- **G-Case**

  $\Gamma \vdash_\tau \overline{p : P}$

  $\Gamma, \overline{p : P} \vdash_\gamma \overline{e : \Gamma_e}$

  $\Gamma_R = \bigcap \overline{\Gamma_e}$

  $\overline{\Gamma \vdash_\gamma \text{case } e \text{ of } \overline{p \longrightarrow e} \text{ end case; } : \; \Gamma_R}$

- **T-Case**

  $\Gamma \vdash_\tau e : E \quad \sqcap \overline{P} = E$

  $\Gamma, \overline{p : P} \vdash_\tau \overline{e : T_e} \quad T = \sqcap \overline{T_e}$

  $\overline{\Gamma \vdash_\tau \text{case } e \text{ of } \overline{p \longrightarrow e} \text{ end case; } : \ T}$

4.2.14 Ascriptions

Ascriptions do not introduce new scoped items, so the `G-Ascription` rule only passes the generated environment through. Also, note that we do not perform any type checking that is not required to formulate the resulting environment in a $\vdash_\gamma$ rule. That check is performed in the $\vdash_\tau$ rule. The syntax for a Rosetta ascription is *expr* :: *typeexpr*.

- **G-Ascription**

  $\Gamma \vdash_\gamma t : \Gamma_T$

  $\overline{\Gamma \vdash_\gamma t :: T : \Gamma_T}$

- **T-Ascription**

  $\Gamma \vdash_\tau t : S$

  $S <: T$

  $\overline{\Gamma \vdash_\tau t :: T : T}$

4.2.15 Constructed Types

User-defined constructed types allow for a parameterized definition of a collection of constructors, observers, and recognizers. The parameters play an important role in the typing rules: although explicit in a fully instantiated representation of the constructed type, these type parameters become quantified (inferrable) parameters for all of the constructors, observers, and recognizers.

- **G-Constructed-Type**

$$\frac{\begin{array}{l}\Gamma_{constr} = \cup \overline{constructor} : [\overline{p:P}](\overline{obsp:Obs}) \rightarrow adt(\overline{p:P}) \\ \Gamma_{obser} = \cup \overline{obs} : [\overline{p:P}](\overline{obsp:Obs}) \rightarrow \overline{Obs} \\ \Gamma_{recog} = \cup \overline{recog} : [\overline{p:P}](\overline{adt(\overline{p:P})}) \rightarrow \textbf{boolean} \\ \Gamma_d = \{adt : (\overline{p::P}) \rightarrow type\} \cup \Gamma_{constr} \cup \Gamma_{obser} \cup \Gamma_{recog}\end{array}}{\Gamma \vdash_\gamma \text{adt } (\overline{p::P}) :: \text{type is data } \overline{constructor(\overline{obsp::Obs}) :: recognizer} \text{ end data; } : \Gamma_d}$$

The Rosetta typing analysis relies on well-formedness checking through the graph analysis. Since a constructed type is just a signature, there is no typechecking to be done beyond requiring types at type locations in the signature structure.

- **T-Constructed-Type**

$$\frac{\begin{array}{l}\Gamma \vdash_\tau \overline{p::P} :: \text{type} \\ \Gamma, \overline{p:P} \vdash_\tau \overline{Obs} :: \text{type}\end{array}}{\Gamma \vdash_\tau \text{adt } (\overline{p::P}) :: \text{type is data } \overline{constructor(\overline{obsp:Obs}) :: recognizer} \text{ end data; } : (\overline{p:P}) \rightarrow \text{type}}$$

4.2.16 Mathematical Operators

Rosetta provides many built-in mathematical operators. They are not surprising in their semantics, but we nonetheless provide the related typing rules, primarily in the appendix (§ A.18). Given the rich set of numerical types in Rosetta, there is a lot of ad-hoc polymorphism in the typing behavior of an operation such as addition. We could simply claim a type for addition such as `+::complex→complex→complex`, but this loses a lot of information. We'd much prefer `4+5` to be of type `posint`, instead of `complex`. And some operations such as division are not even conveniently closed for a type representation such as `(/)::a→a→a` for `a∈{natural, posint, negint, integer, rational, real, posreal, negreal, imaginary, complex}`, which we could try to reason for an operation like addition. We present a small number of rules here to grasp the feel of these rules, but leave the exhaustive set of rules for the appendix (§ A.18).

For operators such as addition, we define a relation R that captures the three types in an valid statement "a value of type A plus a value of type B yields a value of typer C", and then state the rule generally.

Let the relation R be {

(complex,complex,complex),	(imaginary,real,complex),	(imaginary,imaginary,imaginary),
(real,real,real),	(posreal,posreal,posreal),	(negreal,negreal,negreal),
(rational,rational,rational),	(integer,integer,integer),	(posint,natural,posint),
(posint,posint,posint),	(negint,negint,negint),	(natural,natural,natural)

}. Then:

T-Addition

$$\frac{(A,B,C) \in R \ or \ (B,A,C) \in R \qquad \Gamma \vdash_\tau x : A \ \ \Gamma \vdash_\tau y : B}{\Gamma \vdash_\tau \ x + y : C}$$

We capture the commutativity of addition with the *or* clause. We define other operators such as subtraction, multiplication, division, and exponentiation in similar fashion in the appendix (§ A.18).

4.2.17 Top and Bottom Literals

- **T-Top**

$$\frac{}{\Gamma \vdash \textbf{top} : \mathit{top}}$$

- **T-Bottom**

$$\frac{}{\Gamma \vdash \textbf{bottom} : \mathit{bottom}}$$

4.3 Subtyping Relationship

The subtype relation `<:` in Rosetta is defined between types, indicating that all values of the subtype are also values of the supertype. It is reflexive, transitive, and antisymmetric. The relation shows up in a few places. The basic numeric types such as `integer`, `posint`, and `real` exhibit the expected subtype relationships. Composite types, such as sequences and sets, exhibit a subtyping relationship when the type parameters exhibit the appropriate subtyping relationship. Functions have a more interesting subtype relationship, being contravariant in the parameters' types and covariant in the range. User defined types can also be parameterized over types, and can offer some subtyping relations accordingly. Lastly, Rosetta allows for defining types specifically as subtypes of other types with the `subtype` keyword. We explore all of these subtype relations as rules.

4.3.1 Subtyping for Primitive Types

We revisit the lattice of primitives in Rosetta:

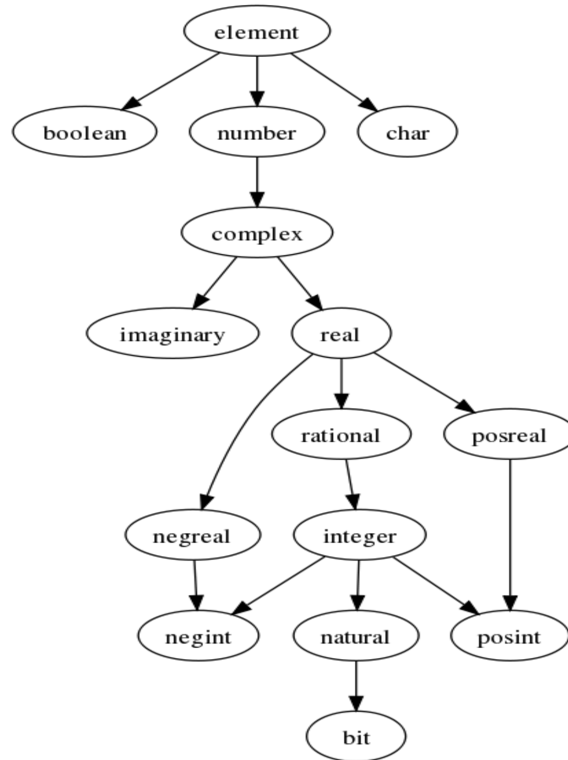

Figure 4.1: Lattice of primitive types in Rosetta.

Each linkage in the graph (Fig. 4.3.1) relates the lower and higher nodes with the subtype relation, as `lower <: higher`. For instance, we realise the following axioms:

$$\frac{}{\textit{posint} <: \textbf{integer}} \qquad \frac{}{\textbf{integer} <: \textbf{rational}} \qquad \frac{}{\textbf{rational} <: \textit{real}}$$

and we could then chain these together to show that `posint <: real`. Rosetta provides basic mathematical operators that are themselves cognizant of these subtype relations, returning the most specific types allowed by the relations. For instance, the addition operator + can be typed as `integer→integer→integer` or `real→real→real`. All of the other rules are implicitly defined rather than listed out manually. These operators have their own specialized typing rules (in § A.18).

In practice, the typing analysis wants to find the most specific type available. When seeking the most specific type when this lattice of base types is involved (with the `bottom` type occupying the, er, bottom), this means finding the least upper bound, or meet $\sqcap$. By tracing upwards from any two types along all possible paths, and finding the 'lowest' path collision, we can find the most specific (i.e., constrained) type that is a supertype of both types. It is quite normal for one of those supertype relations to be the reflexive subtyping relation. As the lattice of base types does not change, it is easily implemented as a recursive function.

As a simplification to the typing rules, we include the following subtyping rule:

$$\frac{\Gamma \vdash x : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash x : T}$$

This allows the subtyping relation to dictate when a subtype relationship leads to the more general typing claim. This allows the majority of the rules to use the type-of operator **:** instead of inserting the subtype operator <**:** in rules where subtyping is not the major concern.

4.3.2 Subtyping for Composite (Structural) Types

The three composite container types provided in Rosetta enforce the subtyping relation through their contained types.

- **Sub-Set**

  $$\frac{\Gamma \vdash_\tau t : \text{set}(S) \;\; S <: T}{\text{set}(S) \; <: \; \text{set}(T)}$$

- **Sub-Multiset**

  $$\frac{\Gamma \vdash_\tau t : \text{multiset}(S) \;\; S <: T}{\text{multiset}(S) \; <: \; \text{multiset}(T)}$$

- **Sub-Sequence**

  $$\frac{\Gamma \vdash_\tau t : \text{sequence}(S) \;\; S <: T}{\text{sequence}(S) \; <: \; \text{sequence}(T)}$$

Some common types are defined via these container types, and thus do not require their own rules.

```
string :: type is sequence(character);
bitvector :: type is sequence(bit);
```

Values of type `word(n)` are defined slightly differently, as `word` is a type function that requires the bitvector values that inhabit it to have length `n`.

4.3.3 Subtyping for Functions

Functions are mappings from types of parameters to a range type. In order for one function to be a subtype of another function, we cannot simply check for subtyping between the parameters and the ranges as we did for the contained types with the structural types above. Given f_i:`integer`→`integer` and f_r:`real`→`real`, we might ask if f_i<:f_r. We are asking more specific questions than twice asking whether `integer<:real`. Since we want to use f_i where f_r is expected, can f_i accept as its arguments all of the values that f_r would have accepted? No, because that would require f_i to accept any real value, and it only accepts integers. But we also can't flip the relation around to claim that f_r<:f_i. Since that would be asking to use f_r where f_i was expected, we can only allow f_r to result in values that would be possible for f_i to result in, and yet all the non-integer real values that f_r may return will not 'fit' in the `integer` return type of f_i. This shows us that the subtyping relationship is a *partial* ordering.

Here is the subtyping rule for functions.

$$\frac{\Gamma \vdash_\tau f_s : \overline{P_s} \rightarrow R_s \quad \Gamma \vdash_\tau f_t : \overline{P_t} \rightarrow R_t \quad \Gamma \vdash_\tau P_t <: P_s \quad \Gamma \vdash_\tau R_s <: R_t}{\Gamma \vdash_\tau f_s <: f_t}$$

4.3.4 Subtyping for User-Defined Datatypes

Is `Tree(integer)` a subtype of `Tree(real)`? When dealing with user defined datatypes, it is tempting to treat them as structural entities and just check the subtyping relationship between their type arguments. And in cases where the type arguments are only used structurally, that would admit a logical meaning of a subtyping relationship between the two. But consider the following datatype and two uses of it:

```
1 FunctionMaker (a::type; b::type) :: type is
2     data MakeFun (func :: a -> b)::isFun? end data;
3
4 funcInt  :: FunctionMaker (integer, integer);
5 funcReal :: FunctionMaker (real, real);
```

This clearly represents the same issue of subtyping functions. While a tree of `integers` can safely be used where a tree of `reals` was expected, the same does not hold in general. In order to properly check a subtyping relationship between two instances of type-parameterized datatype definition requires inspecting the definition itself, identifying all uses of the type parameters with regards to their positions. It's common to consider positions as positive or negative, corresponding to the contravariance/covariance at the site of usage.

4.3.5 Explicit Subtype Definitions

In Rosetta, we can explicitly define a subtype. These subtypes rely on the sets-as-types view, allowing us to create a new type as a further restriction on the values of some other type. As with the rest of the language, we can choose to provide or not provide a value.

```
1 Evens :: subtype(integer);
2 Odds  :: subtype(integer) is sel (x::integer | x mod 2 = 1);
```

The `Evens` definition does not tell us what restrictions on the `integer` set are additionally implied, whereas the `Odds` definition uses the `sel` function to explicitly construct the set-value to use. The set expression that defines the value of the type can use any Rosetta features. The typing analysis does not

attempt to perform any and all evaluation, so initially we do not support any reasoning about the values of the subtype. But there may be future work in finding useful evaluations to incorporate into the typing analysis to give better results. We can only check that values are members of the supertype, which is of course insufficient. This will still exclude cases where values do not belong to the supertype, which is the most appropriate approximation. We generate a warning for the user that the tool is only checking for membership in the supertype.

4.4 Developing the Typing Analysis

The typing analysis obtains a naming analysis via the scopeless abstract syntax [33] work. This naming analysis turns the original Rosetta syntax into the same Rosetta AST as previous type checking efforts, but further resolves all names, constructing an abstract syntax graph (ASG) with all names resolved. This involved process even β-reduces facet instantiations to create nodes representing named items exposed in that instantiation, among many other operations to provide a graph with no more need for a naming environment. The end result is that any analysis — not just type-checking — can operate on a 'scopeless abstract syntax' [33], reaping the benefits without duplicating the work as our original tools did. The graph presents the nodes as strongly-connected components (SCC's). Instead of the algebras of InterpreterLib, we could instead write node decorators for the acyclic and cyclic components. This maintains the synthetic information flow from sub-terms upwards, but cuts off the inheritance information flow from the top of the AST downwards.

Local type inference such as in §2.2.2 has been explored on an example language utilizing a graph structure, demonstrating that it is still amenable to a graph-based structure. The tree-with-back-edges approach allows us to enjoy the benefits of naming analysis performed by the graphing analysis, while still performing an analysis defined in top-down fashion such as local type inference.

4.4.1 Reference Algebras

We must address an issue that arises when using the scopeless abstract syntax work: we are forced to visit the nodes in a topologically sorted order, and there is no means to pass information such as an environment

from a term to its subterms. Typing analyses are concerned with relating sub-terms and their parent terms in order to calculate types, so such a restriction would cause some significant redesign of algorithms that do not lend towards accomplishing our goals. In order to recapture this behavior of downward information flow, I've created Reference Algebras, allowing the graph to be viewed as a tree with back-edges occuring at variable references.

Typing Over Graphs

When we describe our computation over a graph in terms of the acyclic and cyclic portions of the graph, there is no 'best' way to model information flow downwards. Typically we structure our analysis code in top-down fashion over the AST, monadically tracking the environment, counters, results, and so on, with the recursive calls to subterms inheriting this information and synthesizing more information in their returned values. But by losing the notion of 'top-down' when we switch to the topological sort of nodes to order our analysis, we cannot reason about the path to our node from the root. We can still synthesize information upwards, because topological sorting preserves this ordering for acyclic units, but in both directions the cyclic components cause problems.

There is no restriction on the size of a cyclic portion of a Rosetta specification. Therefore, any arbitrary size of a specification may be in a cyclic strongly connected component. All nodes that are dependent upon these cyclic nodes may also be pulled into the strongly connected component. The user must break up the cycle in some fashion, at which point we have the chance to remove an edge and topologically sort the now-acyclic nodes. Identifying the 'breakpoint' of the cycle is challenging. Each way of introducing a cycle in the graph must be identified, with no indication from the subgraph of nodes in the cycle where to look.

If we must explicitly handle each source of cycles in the graph, the benefit of topologically sorting the nodes is lost while adding the issue of finding and breaking cycles. A typing analysis necessarily defines the type of one node by reasoning about the types of the nodes around it, so a cycle in the graph cannot be ignored. These sources of cycles are directly comparable to the points in an AST where new definitions are brought into scope, raising the possibility that their types will be referenced before we have a chance to calculate the type itself. One simple approach often available is to create some sort of placeholder for the type, and then allow the actual type to be figured out afterwards, in delayed fashion. Knowing which name

or node we are referring to does not remove the problem of mutually recursively defined types, and so the ability to traverse the graph in a top-down fashion is still valuable.

The graphing analysis creates the graph through the following process: It transforms an abstract syntax tree's nodes into graph nodes, and then resolves names to replace edges to variable references with edges to the appropriate node of the item to which the reference referred. These reference replacements may introduce cycles, causing the result to be a graph. For this reason, it makes sense to view the graph as a tree with added back-edges to references. If we maintain annotations of which sub-terms are references, we can regain the top-down flow of information, which is how we typically reason about type systems, typing environments, and so on. In doing so, we no longer have the identity of cycles automatically detected, but we can still manually handle these situations. Overall, this modified approach to ASG's places slightly more responsibility on the programmer, but provides more information than was available in InterpreterLib and does not restrict the flow of information either up or down the structure. In short, the task of breaking down cycles is replaced with the task of 'priming' the set of node-type answers with placeholders for all entities whose values could be used before calculation.

The Solution

By viewing a graph as a tree with back-edges to references, we can recover the usual tree-walking structure. Since the underlying representation is still a graph, we can store our answer for each subterm by annotating each node in the graph. The only special case is to track and identify references so that we look up the result rather than calculating it. Only the 'parent' of a node is allowed to evaluate it, ensuring that only the parent may determine the environment in which it is evaluated. This is the Reference Algebra approach.

For our purposes, the reference algebra will be used to construct a catamorphism that operates over the graph representation of a tree with reference back edges, while respecting the lookup-only status of references. Each node in the graph is labeled with a unique identifier (just an integer), and contains a functor constructor where each hole refers to the unique identifier of the intended subterm. Each hole also is annotated with a boolean value representing whether it is a reference or not.

The state we maintain for a reference algebra is minimal – a graph of the results (represented by an `IntMap`). We can then define a type for a reference algebra, called `RefAlg` below:

```haskell
data RefSt a = RefSt { results :: IntMap.IntMap a }

type RefAlg f m a = (MonadStateX RC (RefSt a) m) =>
                        Int -> f (m a,Int,Bool) -> m a
```

A `RefAlg` type relates some functor `f`, some monad `m`, and some type `a`. For our purposes, `f` is our language in functor form, `m` can be extended with whatever monadic properties the target analysis uses. `RefAlg` only constrains `m` to include stateful properties embodied by `MonadStateX RC (RefSt a)` [28], which means that we can access and update the `RefSt a` state. lastly, `a` will be our representation of types. Reference algebras could be defined for other analyses than typing, so the definition is not typing specific.

The signature of `RefAlg` indicates that, given an `Int` representing the unique identifier of the current node and the functor of that node where each hole consists of a triplet of the computation to perform for that node, the unique identifier of the subterm as an `Int`, and a boolean representing whether this subterm is a reference instead of an owned subterm, it will give back `m a`, a monadic computation representing the `a` value for this node.

A `RefAlg` is analogous to an algebra of InterpreterLib, in that we are defining a large-step semantics, where the recursion is left to be explicitly constructed from the algebra itself with a catamorphism or other appropriate fold. All subterms already contain (a monadic computation for) their own results of the analysis, and we do not perform the recursion ourselves. This occurs when we construct a catamorphism over such an algebra:

Implementing Reference Algebras

In Figure 4.4.1, we define a catamorphism for `RefAlg`. To understand what is happening, we compare it to a simpler catamorphism, such as found in InterpreterLib:

```haskell
-- InterpreterLib-style catamorphism:
cata phi tm = phi . fmap cata $ tm
```

```
1  refcata :: (MonadReader Int m, MonadStateX RC (RefSt a) m,
2              Functor f, FunctorZip f
3             )
4             => [(Int,f Int)]       {- functors          -}
5             -> [(Int,f Bool)]      {- was-a-reference info -}
6             -> RefAlg f m a        {- phi               -}
7             -> Int                 {- key               -}
8             -> m a                 {-- monadic result.  --}
9  refcata functors wars phi key =
10     let Just functorAtKey = lookup key functors
11         Just warsAtKey    = lookup key wars
12         withSubtermsDone  = fmap (refcata functors wars phi) functorAtKey
13         wsd'              = fmap (recordResult) $
14                               fzip functorAtKey withSubtermsDone
15     in
16         phi key (fzip3 wsd' functorAtKey warsAtKey)
17
18  where   recordResult :: (MonadStateX RC (RefSt a) m) => (Int,m a) -> m a
19          recordResult (i,ma) = do
20            a <- ma
21            putAlso [(i,a)]
22            return a
```

Figure 4.2: catamorphism defined for RefAlg algebras.

We first map `cata` over all subterms. This recursively 'walks' down the abstract syntax tree's nodes all the way down to the leaves. We then call `phi` on the nodes in the tree, first on the leaves, then up one level at a time, all the way to the root. `phi` operates over the functors with some particular type of value in the holes. In a term, the holes would be filled with the fixed point of the terms as well; in a `phi` for type checking, the types of the subterms would be in the holes; in a `phi` for evaluation, the values of the subterms would be in the holes rather than the terms themselves. The `cata` function embodies the recursion style and means of 'folding' the `phi` function over the term `tm`, in order to perform the analysis and collapse the term down to a specific value. The operational semantics of this mean that we are first building up our recursive calls to `cata`, and then working our way up from the leaves of the AST with the calls to `phi` that were laced at each node through the `cata` calls.

Now we can consider `refcata` from Figure 4.4.1 above, beginning with the function signature. The `functors` parameter is a graph representation of just the nodes and their subterms. `wars`, meaning "Was-A-Reference" list, is the graph again, with boolean values instead of the unique identifiers. The third

argument is the `phi` function itself, the `RefAlg`. The last argument is just the unique identifier for the 'root' node. The result is (a monadic computation of) the value of the algebra at the 'root' node.

The implementation of `refcata` first uses a `let` block to first look up the functor and was-a-reference information at the current key. `withSubtermsDone` performs the recursive calls to `refcata`, while `wsd'` records the results to our `RefSt` state. The body of the `let` calls the `phi` algebra on this recursion-completed result-recording functor filled with triplets of the subterms' (result, identifier,was-a-reference) values.

Defining `refcata` is more natural in terms of the graph structure; therefore, we also provide a graph-based version, `refcataGraph`, accepting the `RefAlg` and the graph.

```
refcataGraph :: ( MonadStateX RC (RefSt a) m, Functor f, FunctorZip f)
                 => RefAlg f m a -> Focus (FCompose f WasRef) -> m a

data WasRef x = WR Bool x
```

The details of this function delve more into the graph representation and are not germane to the current work, so we include only its type signature. The type `Focus (FCompose f WasRef)` repersents a graph with a root (a 'focus') annotated with 'was-reference' annotations by composing the `WasRef` functor, allowing it to wrap a `WR` constructor around each hole of the `f` functor.

Reference Algebra Usage

Since the decision of whether to look up a reference's result or to compute a node's result directly is based on the information that is exposed by a reference algebra, it would be possible for a user to abuse this and still disobey by trying to evaluate a node's expression despite not being the parent node. The `handle` function embodies the logic of deciding whether to look up a reference's result or simply bind the computation for the subterm itself. We will need to bind subterms' monadic computations to use them anyways, so by wrapping calls to `handle` around each binding we ensure that references are always looked up instead of evaluated.

As a sample usage, in a simple language we might see usage for the `If` functor constructor as below. It first handles all subterms, and whether they were references to pre-defined terms or expressions built in-

place, we will appropriately find their analysis results and name them `guard'`, `tru'`, and `fls'`. We can then use them to check if the guard is typed as a Boolean, if the branches' types agree, and so on.

```
phi (If guard tru fls) = do
  guard' <- handle guard
  tru'   <- handle tru
  fls'   <- handle fls
  if (guard'==TyBool) && (tru'==fls')
     then return tru'
     else error ...
```

Anticipating Cycles

The last hurdle to using reference algebras is to anticipate and account for the cycles that would have risen in the topological sorting approach to ordering our node visitation. As we stated before, each cycle has a cause that relates to a definition being used by either itself or in some mutually recursive way. The solution is to put a placeholder into our results that can later be substituted once the result has been calculated.

In order to manually add results, we simply need to append a list of new 'results' to our `results` stored in the `StateX RC` monad:

```
putAlso :: (MonadStateX RC (RefSt a) m) => [(Int,a)] -> m ()
putAlso as = do
  rec <- getx RC
  let newresult = foldr (uncurry IntMap.insert) (results rec) as
  putx RC $ rec {results = newresult}
```

When must we manually add results? It is sufficient to identify all named items and put a placeholder in for each. So each let-binding, each parameter, each declaration in a facet will have a placeholder added.

For all named entities, we still are benefitting tremendously from the scopeless abstract syntax work. This approach does not have to determine what a name actually refers to, because we already have the node. We are only putting in placeholders to account for the situation where a node's result is needed before it can be calculated, no matter how the order of node visitation was decided.

Reference Algebras within the Rosetta Typing Analysis

Reference algebras were first developed and tested over a small evaluation language. This language contains features that are relevant for reference algebras, Rosetta, and local type inference. Its definition is presented in Fig. 4.4.1. It has functions with type ascriptions in their signatures. It also has multiple numerical types that exhibit a subtype relationship. Functions also have inferrable type parameters, represented by the `String` synonym `TyName`.

```
1 data F x = If x x x | Tru | Fls
2            | TmNat Int | TmInt Int | TmReal Float
3            | Mult x x | Minus x x | Plus x x | Eq x x
4            | Fun Name [TyName] [(x,Ty)] x Ty | Param Name Ty | App x [Ty] [x]
5
6 data Ty = TyBool | TyNat | TyInt | TyReal | TyFun [TyName] [Ty] Ty
7
8 type TyName = String
```

Figure 4.3: Test Language Definition for Reference Algebras.

As a functor, we can directly represent terms of this language in graph form. To represent `example = If Tru (TmInt 3) (TmInt 5)`, we identify all constructors as nodes, and build up the graph by assigning a unique identifier to each to represent subterms. The graph for our `example` is:

```
1 example = Graph  [(0,FCompose (If (WR False 1) (WR False 2) (WR False 3)))
2                  ,(1,FCompose Tru)
3                  ,(2,FCompose (TmInt 3))
4                  ,(3,FCompose (TmInt 5))
5                  ]
```

Rather than implement a parser and tree-to-graph conversion, examples were hand-coded into this graph representation. There was no added benefit to writing these tools over the test language.

Next I defined a typing analysis that uses `refcataGraph`, `handle`, and `putAlso` as discussed above. This typing analysis performs local type inference as described in [39], inferring the type parameters of the `Fun` constructor where possible. The local type inference code only interacts with the rest of this typing analysis at the site of `App` nodes that have been applied. In the full Rosetta typing analysis, this is

very useful — it is an orthogonal concern that does not limit other aspects of the typing analysis, having only localized implications.

This evaluation language served the purpose of testing reference algebras as well as implementing local type inference in a Rosetta-centric but simplified language. The code in this typing analysis was then ported to operate over the actual graph representation used by the rosetta-graph tool. This forms the core of the typing analysis, though many more cases must be handled, and many other features can be added onto it.

4.4.2 Tasks for Basic Type Checking

With structure in place for performing a typing analysis, we then proceed to perform simple type checks on Rosetta specifications. In order to do so, we need certain core functionalities such as checking if one type is a subtype of another type or simplifying a type expression. We also must organize these computations to work around the issue of mutually recursive definitions; the analysis creates placeholders for the types of nodes, but we require these functionalities to operate over concrete types. We see why and how we deal with this concreteness requirement.

Checking For Subtypes

`isSubtype` is a function that determines whether one type is a subtype of another type. It operates over concrete types, meaning that there is always an answer whether one type is truly capable of being interpreted as a constrained version of another type. `isSubtype` respects the lattice of base types found in Rosetta (see Fig. 4.3.1). Most of the structure of types is directly evident in the subtyping relationship. For example, given the subtype relation `<:`, we say that `set(A) <: set(B)` if `A <: B`.

There are some chances for further detailed information about typability. For instance, inferrable parameters may be considered in any order.

```
1 apply1 [A,B](f::A->B; a::A) :: B is ...;
2 apply2 [X,Y](f::Y->X; a::Y) :: X is ...;
3
4 apply (f::S->T;s::S) :: T is ...;
```

Both of these `applyN` functions can be specialized to the type of `apply`, given `[S→A,T→B]` and `[S→Y,T→X]`. This requires allowing for all permutations of these inferrable parameters to be considered, though, which can be costly. This is currently not done, as the intended order has meaning the specifier may not want to be inverted in search of an alternate view of a type that happens to match another type's structure.

Simplifying Types

There are many places in the typing analysis where we need a normalized form of a type. One such need is to allow us to check for equality. Other concerns may be more pragmatic — as we learn more about the types in a specification, we may facilitate further simplifications that were not readily available. Simple forms of evaluation of dependently typed fragments also merit simplifying a type in order to learn more about it.

Calls to the `simplifyType` function are pervasive throughout the typing analysis. This function performs any currently available reductions on types. Some simplifications are simple, such as:

```
   -- applying zero arguments
TyApp f [] ⇒ f

-- applying 'native' functions
TyApp f args = if (f `isNodeFor` 'and') && all (==TyBool) args
then ...
```

The analysis has a representation of Rosetta values, primarily for numerical values. There is a function `simplifyValue` that is dispatched within `simplifyType` whenever a `RosettaValue` is encountered within a type. We also simplify all sub-expressions within a type with recursive calls to the function.

As we will see below, there are times when we must introduce placeholder type expressions for some nodes, which are later made concrete by looking up the result for that node after it has been found. Before a concrete type has been found, we represent type applications with the `TyApp` constructor, e.g. `TyApp (Ty`

i) *args*. Later calls to `simplifyType` may look up the result at node *i*, leading to further simplification of this expression. Because these applications may involve dependent types, we perform a substitution while simplifying these `TyApp` expressions. This is described in more detail in § 4.5.3.

Using Delayed Results

More complicated simplifications involve looking up the type of a node in our results. We sometimes delay storing an actual result in our results graph, in order to break cycles of dependence in the graph. These delayed results can 'leak' to other results, and must be handled appropriately.

At the core of this 'delay' approach is the `(TyAt Int)` type constructor. It is a representation of the type at a particular node, which is given the unique identifier (`Int`) for that node. It is opaque, in the sense that we cannot inspect this type or anything about it beyond the node whose type is referenced. Once we have visited the node behind this opaque type, however, its result can then safely be looked up and most likely affords further type simplification.

Looking up `TyAt` results within a type can ostensibly be repeated every time that type is inspected, so at key moments in the typing analysis we systematically simplify all results and store these newer results. Those key moments are specifically at points where we have (potentially) introduced some delayed types and also have traversed through those nodes. In this way we avoid some repeated type lookups. In the source code, this is the `crunchGraph` function.

Rather than defining `crunchGraph` as a fixed point that iterates repeatedly until no changes occur, we rely on the fact that `crunchGraph` is called frequently enough that any situation where multiple passes would be beneficial will instead be handled by these subsequent calls. Execution time has never exceeded the order of seconds during testing on small testing examples, so we are satisfied with runtime behavior. We could only trade time spent looking up and simplifying types with time spent repeatedly calling `crunchGraph`. The biggest concern is that this decision does not affect the semantics of our analysis, and so we are free to decide for or against this optimization. This simplification and storage of the simplified results simply operates as an efficiency improvement without changing the meaning of the represented types.

Dealing with Concreteness

One major design choice is to require a concrete type as our result. This means that we do not create type variables to represent a node's type and then eventually solve a set of constraints to determine its type. When we evaluate a node's type, we must immediately determine a representation for the node's type. We do allow for placeholders, as described above, in order to give us a name for nodes that have not been evaluated yet, but when we evaluate that node itself, the placeholder then refers to the result of that evaluation. Although this placeholder is opaque, it is not abstracted out, or represented by a variable or some constraint.

This concreteness of types is a requirement of local type inference. The generated constraints are always between concete types and the omitted type parameters. It is a choice that we can work with: as we are avoiding any global collection of constraints in favor of more localized approaches, this implies that there is no further global stage in which we would be solving for type variables introduced in such a fashion.

When we implement functionality such as the is-subtype relation (as the `isSubtype` function), we must only allow ourselves to check this relation on concrete types. Cyclic dependencies are broken by introducing 'placeholder' types, which we need to look up in the results or else delay the current computation until we have a concrete representation for that node. Indeed, the only instance of `TyAt` within the `isSubtype` function involves a lookup of the result, because there should already be an answer there.

4.5 Partial Information Approaches

One goal of this work is to track or expose any partial information of a specification in order to learn more about its types. We consider different ways in which we track information to distinguish more about types. We track the value that 'witnesses' a type when it is given. We interpret a node in the ASG as representing a Rosetta type directly. We perform limited substitution, which is complicated by the dependent nature of Rosetta. We use witnesses to compare `word` sizes by calling out to a decision procedure.

4.5.1 Witnesses

Whenever we supply a parameter to a function, we have both a value and its type. In non-dependent languages, knowing the type is sufficient for substitutions. However, Rosetta is dependent — and so we must keep track of the values as well in order to successfully perform substitutions.

In order to obtain the correct values for substitution, we need to record them. To achieve this, we use the following infix type constructor:

```
data Ty = ... | Int ::: Ty | ...
```

The `:::`, or 'witness' type as it is called, represents the type `Ty` on the right-hand side of the `:::` constructor. It also records the node representing the value of this type. For instance, if we had `120:::integer` where the Rosetta value at node 120 happened to be the number 5, then we are representing the `integer` type while at the same time really representing the set $\{5\}$ as a type whose value can only be `5`. This is related to 'singleton types' [17], where we would say that a value $\texttt{v}:\{\texttt{v}\}$. By treating the set $\{\texttt{v}\}$ as a type, we are stating that the value `v` is an element of that type.

Witnesses allow us to track values as necessary, and allows us to perform some evaluation over types that normally wouldn't be directly allowed in a non-dependent language. For instance, if we create the user-defined type in Rosetta as follows:

```
  Pair(pa::type;pb::type)::type is
      data PairV (v1::pa;v2::pb)::IsPairV end data;
```

we can then use `Pairs` both at the value level as well as at the type level:

```
valuepair :: Pair(integer,boolean) is PairV(5,true);
typepair  :: Pair(type,type) is PairV (integer, boolean);
```

Things get more interesting when we observe these `Pair` values:

```
x :: integer is v1(valuepair);
y :: (v1(typepair)) is x;
z :: integer is y;
```

Observing valuepair with v1 is straightforward. But simply extracting the left part of typepair in similar fashion would only be type, meaning that we cannot further guarantee that y is an integer value, and therefore we cannot confirm that z is well-typed. In order for this to work, we need to have the *witness* to typepair — integer — in order to safely conclude that z is well-typed. By recording the type of typepair as Pair(intNode:::type,boolNode:::type), we can see that v1(typepair) is not just an element of type, but we can further reduce this down to state that y is an element of integer. (We assume intNode is the identity of the graph node representing the integer type — even base types are not hard-coded into the graph).

4.5.2 Nodes as Types

The lack of separation between values and types leads to another complication in type-checking. If our analysis is supposed to find the type of each node, then what is the type of a type? In type theory, we understand the type of a type to be a *kind*. In Rosetta, without all these separated levels of terms, types, kinds, sorts, and so on, we represent the type of a type with a special keyword, **type**, which can be considered to be equivalent to the $\star$ kind discussed in section 2.1.4. An $\Rightarrow$ kind from System F_ω would become a Rosetta function of type `<* (_::type)::type *>`.

Some nodes in the graph represent typing annotations within Rosetta, e.g. for function signatures, ascriptions, and let-bindings. If we were to just look up the results for these nodes, we would not be viewing the type *at* that node, but actually the type *of* that node. While that might be useful for a 'kind well-formedness' check, it means something fundamentally different. Indeed, some early bugs and limitations during development were due to a lack of respect for this distinction.

As far as the typing analysis goes, in order to extract typing annotations from the Rosetta source, we must elaborate from the nodes in the graph to a type in the internal type representation. The nodeAsType function performs this elaboration. The graph representation is tightly coupled with the original source code, and this means that each type annotation in the source will correspond to a single node in the graph representation. Of course that one node can refer to other nodes, but for the purposes of the nodeAstype function, a single node as the 'root' of the type is sufficient.

This does not amount to performing type checking all over again — instead, we see that `nodeAsType` performs pattern-matching over the allowed representations of types in Rosetta, and then simplifies the resulting type as in the general case with `simplifyType`. Without this capacity to both get the type *represented* by a node as well as the type *of* a node, examples such as `typepair`, above, would not be available for type checking.

4.5.3 Substitutions

From a typing perspective, a key concern arising with dependent types is substitution. We need to have enough information available to correctly substitute through types when inspecting applications at the type level. We must be aware of what evaluation is allowed during this substitution in order to perform this evaluation. And we must ensure that this evaluation terminates. We will inspect some of the concerns of substitution in Rosetta types, consider a restricted case of evaluation that we can support, and the approach to actually performing that evaluation.

As Rosetta has first-class types, and therefore dependent types, we must perform substitutions to capture that dependency. We discuss some of the situations in which substitution is necessary, and the extent that it is possible to perform these substitutions.

In System F_2-style terms where a type parameter is part of the range in some type, we must substitute the type variable with the supplied argument. Each type parameter can show up in subsequent parameters, so we must perform substitutions throughout the remaining parameter list and result type. This operation is O(n·lg n) for *n* parameters, but parameter lists are rarely particularly long.

In System F_ω-style terms, where term parameters appear in following parameters and the resulting type, we must take care to correctly replace these as well. The typing analysis cannot handle all replacements, as this requires reasoning about whether two arbitrary expressions in the term language were equivalent, in turn requiring arbitrary computation. Instead, this analysis supports a restricted set of numerical expressions expressible within Presburger arithmetic [30]. This roughly translates to addition, subtraction, and multiplication by a constant. In general, as we identify further values that are representable within types, we can add them to the `Value` definition (discussed below), and then reason about their equality and substitute them within types as well.

One effect of using the graph is that the type substitution problem simplifies to substituting for a unique node.

```
problematicSubTy :: Name -> Ty -> Ty
betterSubTy :: UniqueInteger -> Ty -> Ty
```

Normally, a substitution algorithm is highly concerned with name capture. That is to say, substituting for a type variable based on its name must be careful to observe scoping rules and identify whether a particular instance of a type variable refers to a distinct type variable defined coincidentally with the same name. The graph analysis has already alpha-converted the entire specification to a structure where each named entity exists with a unique 'name' — its node number — and thus there are no direct name clashes possible. The only concern is to do with multiple references to a node that involves some parameter type variable. But as these multiple occurrences will show up only within our own internal representation of types, they will be structurally separated within the tree-like structure of a type, due to being defined as an abstract syntax **tree**.

Why do we need substitution at all, then? Type variables can still exist, they are just not introduced by the typing analysis itself. Every inferrable parameter, though it has its own node, can show up as a type variable in the rest of the type. This occurrence will be a reference back to the inferrable parameter, and once an inferred type is found we will substitute a reference to this node with the appropriate substitute type. The same is true for non-inferrable type parameters.

When we are simplifying types, we sometimes find an application at the type level, and must therefore perform substitution. The `checkAndSubAnApp` function, however uninventively named, accepts a list of formal parameters, actual arguments, function range, and returns the result of applying the actual arguments for the formal parameters, one at a time. This function performs limited checks such as enforcing a subtyping relationship between the supplied argument and the formal parameter. It then substitutes throughout the remaining parameters and range, one argument at a time. It utilises the witnesses (section 4.5.1) to further inform the substitution in order to propagate the witnesses as necessary throughout the remainder of the type.

4.5.4 Size control

In order to control the values that may appear and be involved in typing a Rosetta specification, we introduce the `Value` datatype, as well as an internal type representation allowing these values to show up in types:

```
data Value = Numeric Int
        | NumericF Double       -- Float
        | NumVar Int            -- node that refers to the value.
        | SomeNum String        -- a number we can't specifically handle
        | NumberAt Int          -- delayed lookup of the number at this node.
        | NumAdd Value Value    -- an addition  of two Values.
        | NumSub Value Value    -- a subtraction of two Values.

data Ty = ... | RosettaValue Value | ...
```

Rosetta has a much richer set of numbers than just integers and doubles (floating-point numbers), but we are restricting ourselves to these representations. Presburger arithmetic is actually only defined over integers. But as we find more values that can be handled in some controlled fashion, we can add them to this `Value` structure, effectively trapping them within a few isolated points in the code.

Analagous to `nodeAsType`, we also have the `nodeAsValue` function to elaborate from a Rosetta representation of a value to the internal `Value` representation. When we are checking the type of an application (with no indication whether it is a term-level application or a type-level application), we can check via `nodeAsValue` whether this node is representable in our value space in order to get a more specific type representation — as the value it truly is — rather than just recording the type of the value represented by that node. As before, it is crucial to not confuse the type of the node for the type at the node.

4.5.5 Presburger Arithmetic

Presburger arithmetic [21, 30] makes reasoning about a restricted set of equations a decidable problem. We use this restricted set of arithmetic and relations to bring these restricted values into the Rosetta types, specifically in allowing finer-grained reasoning about the `word` type. Cooper's decision procedure [13] for Presburger arithmetic is a decidable algorithm that inputs Presburger formulæand outputs whether the

formula is valid. It transforms the formula to disjunctive normal form, constructs the sets of truth values (for variables) that make the various relations consistent, and then checks if any of these sets of truth values make the formula itself true. Conveniently, there is a Haskell library (Diatchki [18]) that implements Cooper's algorithm for Haskell. There are simple extensions to the basic grammar of natural numbers, addition, equality, and quantifiers that are incorporated in Fig. 4.4.

```
1  data Formula  = Formula :/\: Formula
2                | Formula :\/: Formula
3                | Formula :=>: Formula
4                | Not Formula
5                | Exists (Term -> Formula)
6                | Forall (Term -> Formula)
7                | TRUE
8                | FALSE
9                | Term :<:   Term
10               | Term :>:   Term
11               | Term :<=:  Term
12               | Term :>=:  Term
13               | Term :=:   Term
14               | Term :/=:  Term
15               | Integer :| Term
```

Figure 4.4: Presburger Formula Definition.

Boolean relations consist of and (`:/\:`), or (`:\/:`), implies (`:=>:`), and not (`Not`). Boolean values `TRUE` and `FALSE` are present. We extend beyond simple equality (`:=:`) to include inequality (`:/=:`), less-than (`:<:`), greater-than (`:>:`), less-than-or-equal (`:<=:`), greater-than-or-equal (`:>=:`), and divides (`:|`). A `Term` is represented by a special structure that overloads the `Num` type class so that we can input integers directly, e.g. `3:<:5 :/\: x:>: 10`. Multiplication by a constant is also possible, by 'unfolding' the constant `k` from (`k*x`) to ($x_1+x_2+...+x_k$).

The task is now to find questions about Rosetta's types that we can phrase in terms of Presburger formulæ. We inspect the sized bitvector type, `word(n)`, defining a sequence of bits of length `n`. First, we should be able to inspect explicit sizes:

```
1  w1 :: word(6) is b"001101";
2
3  w2 :: (x::word(8); y::word(4)) :: word(12) is x & y;
```

In the first case of `w1`, the analysis can represent the value `6` as a part of the type `word(6)`, as well as calculate the type of `b"001101"` to be `word(6).` We can then *equate* these numbers while checking if the latter is a subtype of the former. This is a dispatch of the Presburger formula `6:=:6`. In the case of `w2`, we see the & operator perform concatenation. The related Presburger formula we generate is `8+4 :=: 12`. Again, this is a simple dispatch to the library, and we are returned `True` as the result, allowing us to see that the body of `w2`'s type is appropriate.

The more interesting cases include variables in the resulting Presburger formulæ. Consider `w3`:

```
w3 :: (a,b::natural; x::word(a); y::word(b)) :: word(b+a+1) is x & "1" & y;
```

The Presburger formula for `w3` is `Forall(\b-> Forall (\a-> b+a+1 :=: a+1+b))`. This show that we can use the analysis to test for size restrictions involving specific but unknown quantities, as variables.

We can also use less information than is present, and similarly track more information than is locally necessary.

```
w4 :: sequence(integer) is [] & [1,2,3];
w5 :: sequence(bit) is b"101";
w6 :: (a::natural; x::word(a)) :: word(3+a) is w5 & x;
```

`w4` does not need to be sized — in order to be well typed, it only needs to be a `sequence` of integers, of any length. This is also the case for `w5`. However, when `w5` is used within `w6`, the length of `w5` is important. The analysis finds that `w5` is indeed of type `word(3)`, allowing for `w6` to check the Presburger formula `Forall (\a-> 3+a :=: 3+a)`.

These checks require special cases to handle operators such as `&`, as well as parameters. We can also — very cursorily — step inside functions slightly, to try to learn even more about our functions.

```
repeatbits (n::natural; b::bit) :: word(n) is
  if n=0
  then []
  else ([b] & repeatbits (n-1,b))
  end if;
```

This situation is tricky; the value of `n` is cased-over for different values. the then-branch will be of type `word(0)`, but the else-branch will be of type `word(1+n-1)`. The issue is that the then-branch is actually of type `word(n)` precisely when `n=0`, yet `word(0)` is recorded. We'd end up checking the Presburger formula `Forall (\n-> 0 :=: 1+n-1)`, but of course the two branches have been merged without concern for their 'environments.' One soft check we can do, however, is to instead check `Exists (\n-> 0 :=: 1+n-1)` and then check `Exists (\n-> n :=: 1+n-1)`, which ensures that there is a value for which `repeatbits` can legally be called. This is not entirely satisfactory, as it does not check that all values that may be used will fit. It should instead be possible to try to keep track of what we know about `n` in each branch. Then we would be able to check `Forall (\n-> (n:=:0) :=>: (0 :=: n))` and `Forall (\n-> (Not (n:=:0) :=>: (1+n-1) :=: n))`. Both of these are true formulæ. But we do not currently dig this deep — attempts were too fragile, breaking other previously checkable examples.

The constructed Presburger formulæ are checkable via the `equate` function, returning a boolean value. The `valueFit` function is little more than a wrapper around `equate`, returning a yes-no-maybe value semantically equivalent to a `Maybe Bool`, where we return 'maybe' when we cannot be sure that they are not equivalent. `valueFit` is used by the `isSubtype` function to determine the relationship between numerical Rosetta values, specifically for checking whether two `word` sizes relate correctly to each other.

4.5.6 Local Type Inference

We have already discussed what local type inference is. In Rosetta, some uses for local type inference arise in code generated during the graph analysis. Other uses of local type inference are for inferring the quantified parameters (those in square brackets). Here, we describe these specific cases of applying local type inference to Rosetta.

Generated Nodes and Chances for Inference

Some definitions introduce functions that are only named, though their implementation is understood. The creation and usage of user-defined types introduces functions that are only represented syntactically, with their implementations understood from the context. The graphing analysis will create nodes to represent the

implied definition of these functions in the graph representation. As an example, we define a user-defined constructed type using the `data` keyword:

```
1  IntPair :: type is
2      data
3          IntPairVal (leftval::integer; rightval :: integer)::isIntPairVal
4      end data;
```

`IntPairVal` is defined with type `IntPairVal :: integer -> integer -> IntPair`, but we do not see its definition. In the graph, however, any call to this constructor function needs a node for reference; so nodes are generated in the graph analysis as if the following function was present:

```
1  IntPairVal (x::integer; y::integer)::IntPair is ...;
```

Similarly, we also have 'inserted' definitions for `leftval`, `rightval`, and `isIntPair`. We might consider the above `IntPair` definition to be:

```
1  IntPair :: type;
2  IntPairVal (_::integer; _::integer)::IntPair;
3
4  leftval (x::IntPair)::integer is
5      case x of     IntPairVal(lft,_) -> lft     end case;
6  rightval(x::IntPair)::integer is
7      case x of     IntPairVal(_, rt) -> rt      end case;
8
9  isIntPairVal(x::IntPair)::boolean is
10     case x of IntPairVal(_) -> true
11               | _           -> false
12     end case;
```

There are of course far more nodes apparent in this second presentation — instead of just a parameter named `leftval`, we have another top-level definition of a function with its own parameter.

So where is the inference? We see inference arise when the user-defined type is itself parameterized by types. Instead of an `IntPair` definition, let's consider a pair of any two particular types, where we must keep track of the type of the left and right values.

```
1 Pair(a::type; b::type) :: type is data
2         PairVal (first::a; second::b)::isPairVal
3     end data;
```

Now the automatically generated functions have more interesting types:

```
1  PairVal[a,b](x::a; y::b)::Pair(a,b) ;
2
3  first [a,b](x::Pair(a,b))::a is
4      case x of PairVal(lft,rt) -> lft end case;
5
6  second[a,b](x::Pair(a,b))::b is
7      case x of PairVal(lft,rt) -> rt  end case;
8
9  isPairVal[a,b](x::Pair(a,b))::boolean is
10     case x of
11         PairVal(_,_) -> true
12         | _          -> false
13     end case;
```

Not only have we generated functions with inferrable parameters (`[a,b]` throughout), but we also have generated *usage* of these inferrable parameters. `PairVal` is a constructor with two inferrable type parameters and two required parameters (that use the inferrable types). We see that `PairVal` is used in the `case` expressions of all three other functions. In order to type check these nodes, we must be able to infer proper values for `a` and `b`.

Consider the definition of `first`. We `case` over `x`, which we know is of type `Pair(a,b)` (whatever `a` and `b` happen to be — we can't see any further than the formal inferrable parameters). We must use this information to assume that `PairVal(lft,rt)` is of type `Pair(a,b)`, to in turn imply that `lft` is of type `a` to determine that the (only) pattern's body is of type `a` as well, matching the entire function's

range. This follows normal application of local type inference — we have an application site that omits type parameters; we collect constraints over allowable values, and solve; finally, we record the types used in the substitution.

User-generated Datatype Usage

We see the same chance to infer parameters for user-generated usage of datatypes. If we continue with the `Pair` type but define our own functions over it, we again can leave the inferrable parameters out:

```
1 collapse [a,b,c::type](p :: Pair(a,b); f ::<*(_::a;_::b)::c*> ) :: c is
2     case p is PairV (x,y) -> f (x,y) end case;
```

This is the same usage of the inferrable parameters as before. We also can write `case` expressions over expressions not directly ascribed types, and then figure out what values for inferrable parameters are best, given the pattern matches:

```
1 noAscriptionExample :: integer is
2   case PairVal(true,6) of PairVal (x,y) ->
3     if x then y else 5 end if
4   end case;
```

The outcome is the same: there is an application of a function with inferrable parameters, and those omitted parameters are found based on the surrounding usage of the type parameters when compared to the other supplied arguments.

4.6 Error Reporting

The typing analysis generates errors, warnings, and information messages as necessary when analysing a specification. We explore the design choices in constructing these messages, including efforts towards offering fine-grained debugging of the analysis itself. The main goals are to provide for error recovery and reporting multiple errors.

The initial parsing records source positions of each token, which are in turn recorded in the AST. The graph structure preserves this source position information, copying what makes the most sense for generated

nodes and partially evaluated nodes. We are able to reference these source positions when generating error or warning messages by simply looking up the source position information for the current node (or any other specific node, if it is clear where else to report).

There exists a set of analysis tools beyond this dissertation's typing analysis that is integrated with an Eclipse [2] plugin. All of these tools may generate Info, Warning, and Error messages. This is directly usable in a typing analysis. We can generate Error messages for each problem found, we can create Warning messages for things that cannot be checked but that are not necessarily wrong, and we can generate Info messages for any other communications.

While we try to connect error messages to the original source as much as we can, the fact of using the graph analysis and its partial evaluation makes for an effectively irreversible transformation, and we cannot always pinpoint a meaningful location e.g. for nodes that do not directly relate to any syntax in the source.

Once the analysis is completed, the graphing analysis generates a visualization of the graph representation as a dot file, and a table of the graph nodes and the found types is generated as an HTML page. Given the following simple definition of a function:

```
1 func (x::natural)::integer is x+1;
```

we get the sample graph in Fig. 4.5. The top node is the focus (root) of the graph, and to find a particular named entity we trace from that top node down named arrows exactly according to the design units and named entities that define it. In this simple sample, we have a top-level definition for `func`, so we directly follow the arrow from the root to node 403. The seemingly high numbers occur because definitions in the Rosetta prelude are given their own unique nodes in the representation, but are only shown as dotted-border nodes when used, and are otherwise hidden to keep the graph representation minimal.

Fig. 4.6 shows a part of the table of node typing annotations that the analysis generates. the table lists the node number, represents the type both in raw and more readable form, and then displays the raw graph representation of the node. Between the well-organized and named graph display and the HTML output, the specifier can track down the type annotation for any sub-expression of the specification.

Throughout the analysis, a Writer monad is present that allows the `tell`ing of new messages, that are then collected and reported to the Eclipse IDE or command line as appropriate. But one major hurdle to

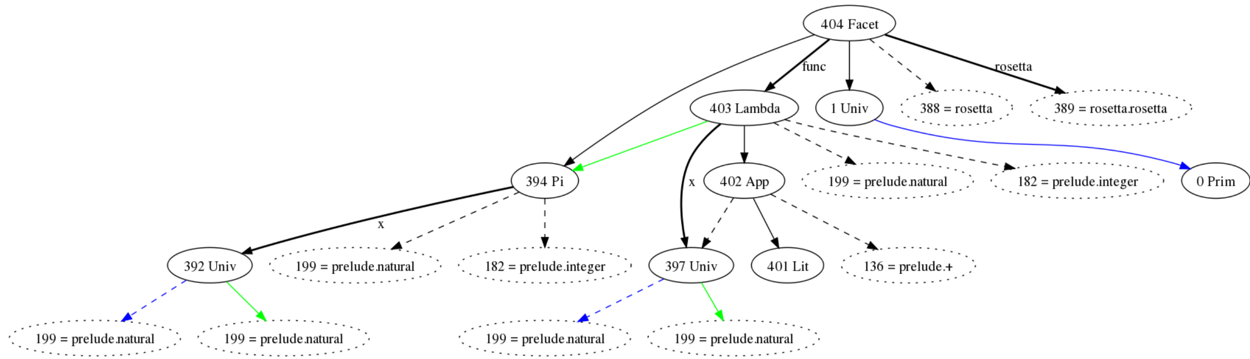

Figure 4.5: Sample Graph Output.

259	Star 259 star 259	Primitive "type"
261	Ty 261 static	Primitive "static"
275	Ty 275 set	Primitive "set"
305	Ty 305 top	Primitive "top"
333	Ty 333 and	Lambda (Parameters []) (Parameters [(WR True 180,Just (x,WR False 324)),(WR True 180,Just (y,WR False 327))]) (Just (WR True 180)) (WR False 332)
349	Ty 349 or	Lambda (Parameters []) (Parameters [(WR True 180,Just (x,WR False 340)),(WR True 180,Just (y,WR False 343))]) (Just (WR True 180)) (WR False 348)
365	Ty 365 =>	Lambda (Parameters []) (Parameters [(WR True 180,Just (x,WR False 356)),(WR True 180,Just (y,WR False 359))]) (Just (WR True 180)) (WR False 364)
392	Ty 199 natural	Universal Nothing (WR True 199)
394	TyArrow [392 ::: Ty 199] (Ty 182) (392:::natural) -> integer	DepProduct (Parameters []) (Parameters [(WR True 199,Just (x,WR False 392))]) (WR True 182)
397	Ty 199 natural	Universal Nothing (WR True 199)
401	Ty 255 bit	Lit (Bit 1)
402	397 ::: Ty 199 (397:::natural)	App (WR True 136) [] [WR True 397,WR False 401]
403	TyArrow [397 ::: Ty 199] (Ty 182) (397:::natural) -> integer	Lambda (Parameters []) (Parameters [(WR True 199,Just (x,WR False 397))]) (Just (WR True 182)) (WR False 402)
404	Ty 0 type	Facet Package (WR False 1) (Parameters []) (FacetParameters []) (Declarations [(rosetta,(False,[]),WR True 388,WR False 389),(func,(False,[rosetta.lang.prelude,rosetta.lang]),WR False 394,WR False 403)]) (Constraints [] [] []) (Exports [rosetta,func])

Figure 4.6: Sample Type Annotations.

overcome is to make the text meaningful to the reader. Not only do we have to deal with the fact that the graph does not represent the original specification in one-to-one correspondence of syntax to nodes, but we have also replaced basic definitions of the language with nodes. `integer` is no longer handled specially — we have a node that represents the `integer` type, and when we check if something is a subtype of `integer`, we are checking if it is a subtype of the type represented by the node that happens to refer to

`integer`. We must be careful to track and extract all of these 'nameable' things so that we see the latter of these two type representations in our messages:

```
TyArrow [TyData 410 [Ty 182,Ty 199]] (Ty 199)

Pair(integer ,natural) -> natural
```

We replace not only base types and primitive definitions of the language, but also user-defined types. In order to do this, we must record enough information at the time of message generation so that we can later look up all of the related things from some generated context. To this end, instead of building up the string for a message, we build up a list of string and type elements that should be displayed in order to generate the message. We use the following data structure in the Haskell implementation:

```
data Showable = STy Ty
              | STys [Ty]
              | SStr String
              | DebugTy Ty
              | DebugStr String
```

We see at the same time the inclusion of `Debug*` versions. While creating the analysis and debugging, it proved useful to add extra information through these elements in the message list that could all be selectively stripped out by a simple check. In this way, a debugging mode could be turned on or off without actually editing the messages that are generated.

Turning debugging on or off is not the only level of control available. If the debugging flag is set to true, we can also create and set multiple flags, so that we are actually checking things such as : "if we are debugging, and the Subtyping flag is present, then generate this message."

```
data Flag =
    FlagWalkParameters  -- for debugging how parameters are handled.
  | FlagInferencedNode  -- for inferenced node tracking.
  | FlagApp             -- for messages relating to the App constructor.
  | FlagLambda          -- for messages relating to the Lambda constructor.
  | FlagSimplify        -- for type simplification messages.
```

```
7  | FlagPutAlso          -- for values manually entered into the results graph.
8  | FlagMain             -- messages in the main function.
9  | FlagLiterals         -- shows the literals.
10 | FlagASGNodes         -- print out all the nodes (near the top of printout).
11 | FlagNodeEntry        -- print the "entering node i" messages.
12 | FlagInfer            -- for inference messages.
13 | FlagFinalResults     -- prints results per-node after typechecking.
14 | FlagTyRep            -- for messages related to TyRep.
15 | FlagData             -- for datatypes.
16 | FlagConstraints      -- for constraints.
17 | FlagCase             -- for cases (and their ascription issues)
18 | FlagTraversals       -- show traversal steps (print per node entry).
19 | FlagSubtyping        -- for subtyping issues.
20 | FlagVoid             -- (unused).
```

While it's not the most beautiful coding solution to edit the data structure every time we want to create a new flag, it is direct, descriptive, and allows us to turn on batches of extra messages in a meaningful way. Many messages will check if any or all of a set of flags are set before generating an Info message, and support functionality allows for checking that certain flags *and* some boolean condition are both met before performing some action. Yet all of this is conditional on if `debug = true`, so it is simple to remove all debugging information not just by groups of messages, or by conditions met, or even by parts of messages, but also all at once as well. Each release for a demonstration was a simple as turning off debugging.

4.7 Summary of Methodology

Through this work, we have defined the typing semantics that the typing analysis observes through the above typing rules. We have provided a tool that performs basic typing checks on Rosetta specifications, announcing errors as they are found. We also have implemented some more advanced typing analyses in the tool that attempt to extract more information from the specification and use that information in order to make stronger claims about the Rosetta specifications. By providing all of this information to the user,

more typing information is automatically inspected, and more typing information is available for manual inspection, all in a language for which complete type checking is undecidable.

We have presented typing rules that codify the semantics of the Rosetta type system sufficiently to address the issues of the typing analysis we've created. These rules are designed through some co-related relations, primarily $\vdash_\tau$ and $\vdash_\gamma$. These rules do not intend to be algorithmic, as a complete specification of the Rosetta type system would itself be undecidable. Instead, these rules offer tactics that may lead to a complete typing judgement, and the typing analysis may be seen as a limited approach at applying these rules to a specification with some degree of automation.

We have also presented a typing analysis that performs some basic type checking on Rosetta specifications. This analysis builds on the semantics described in the typing rules. This analysis automates the application of the simpler, non value-dependent typing rules, providing a basic type checking analysis of those non-dependent features of Rosetta. It presents errors and warnings to the specifier as output, and also generates a graph representation of the specification with typing annotations available. This provides a simple framework for type checking simple specifications, allowing us to perform more detailed checks when applicable.

This typing analysis also explores ways to perform further typing judgements, focusing on controlled ways of introducing dependent features or opportunistically adding typing features not always available. We have introduced local type inference to Rosetta, despite the significant differences between the original language example of System $F_{<:}$ and Rosetta. This proved useful both in user generated and automatically generated instances of inferrable quantified parameters. We also introduced some value dependence through the tracking of witnesses, where the specific value inhabiting a type at a particular usage of the type is preserved, allowing further inspection. One such check enabled by witnesses was in tracking the widths of bit vectors. This was accomplished by tracking the numerical expressions via witnesses and embedding the values in the internal representation of Rosetta types, and then dispatching the equality check between different numerical expressions to an implementation of Cooper's decision procedure for solving Presburger arithmetic formulæ.

This work allows us to see what effect we gain by providing more detailed information to the specification writer, and performing type correctness checks when applicable in a best-effort approach rather than

only providing analyses that can always be performed. This allows for Rosetta specifications that utilize the full spectrum of Rosetta's rich type system, without forfeiting all type analysis. Not only do we attempt to type check usage of features that are sometimes feasible to check, but we provide details and annotations of the typing analysis over the entire specification even when one part of the specification utilizes features for which we cannot perform type checking.

Chapter 5

Evaluation

We now evaluate the work completed and the claims we have made. We have provided type rules to address the features of the Rosetta type system that the typing analysis addresses. These rules provide a formal view of the Rosetta type system in the traditional presentation format, and allow us to compare any implementation against it. We implemented a typing analysis for the Rosetta specification language that handles the expressions and structures of the language. It also performs more sophisticated analysis, tracking witness values and checking `word` sizes when it can, as well as performing local type inference for the quantified parameters. Emphasis on concrete type representations helps in error recovery. The analysis provides type annotations for each node in the graph representation of the specification, allowing the specification writer to inspect the analysis' results with a fine-grained level of detail.

Overall, the analysis reconsiders from the beginning what the purpose of a typing analysis is for a specification language. Given that Rosetta and other specification languages tend towards expressiveness to the point of undecidable type checking, the purpose of the analysis transitions away from completely checking for type safety in a yes/no answer and towards annotating type information, tracking dependent typing features as it can even when complete coverage isn't possible, and generally giving a best effort in the expectation that the specification will be further revised and expanded based on the information the analysis can provide.

The typing analysis approaches Rosetta with the understanding that a complete typing analysis is unachievable and instead provides assistance when it can, always maintaining decidability and termination

of the analysis. The strategy of operating over concrete types and providing localized analyses leaves the analysis extensible to incorporate further opportunistic analysis of dependent typing features.

5.1 The Type System

We have presented typing rules for the Rosetta specification language. They define typing as the mutually recursive $\vdash_\tau$ and $\vdash_\gamma$ relations, relying on the subtyping relationship $<:$ and the rules of local type inference. These rules fully define the implemented Rosetta semantics found in the typing analysis, but do not attempt to provide a complete set of typing rules for the entire Rosetta language. Other dissertations or research efforts will surely formalize the language's semantics explicitly in the future, and the current work may perhaps serve as a basis or starting point. The current work is concerned with developing a typing analysis implementation over Rosetta. As such, the rules provided are intended to formalize the aspects of Rosetta that the analysis covers. Features that are notably missing from the typing rules include facet composition in its various forms, as well as interactions.

Conversely, some of the primitive types are not supported by the graph analysis, and therefore cannot be supported by the typing analysis until then. The graph analysis supports the `bit`, `natural`, `integer`, and `real` numerical types but does not support `posint`, `negint`, `rational`, `posreal`, `negreal`,`imaginary`, and `complex` numerical types. The typing rules however may describe the semantics of these numerical types, and operators over them, in anticipation of inclusion. The requirement for these rules is that the typing analysis' behavior is fully described by the rules.

The typing analysis itself is defined over the core ASG of the graphing analysis, in turn defined over the Rosetta AST after parsing the original source. This means that there is not a one-to-one relationship between syntax in Rosetta and syntax in the typing analysis. These differences are not a hindrance to defining typing semantics. All of the design units — facets, components, domains, packages — have similar areas of definition and parameters, and the collapse into a single structure in the ASG does not confuse the already-highly-similar typing rules. These rules are defined over Rosetta syntax and are applied over the graph ASG, but there is little to no cause of ambiguity in the elaboration between the two.

Through these typing rules, we gain a more formal view of the semantics of Rosetta. We have a formal foundation for discussing what the typing analysis *should* do, allowing us to define and identify correct or incorrect behavior for the analysis. Perhaps the biggest finding is in recognizing that the ad-hoc semantics of some primitive operators in Rosetta are not well-represented in Rosetta themselves. The addition operator + is a simple example. Addition in other languages can be made simpler by only defining it for integers and floats, with implicit but well-defined casts as seen in Java. It can be made more explicit, with type classes of Haskell providing the ad-hoc mechanism for defining multiple implementations of addition via the `Num` type class and relying on the algorithm for instance selection to choose the most appropriate (i.e. most specific) instance to apply. But in Rosetta, we have more numerical types than a 32-bit integer representation and one or two floating-point representations. These numerical representations are also reflections of the mathematical types of integers and reals, not implementations with all the associated details such as overflow, two's complement representation, and the IEEE 754 standard for floating point representation. And type inference alone is not a sufficient substitute for the type classes of Haskell. It may be valid Rosetta to embed a computation for the range of a function that is quite dependent upon the likely inferred types of the formal parameters, but it would certainly not be good practice, as the issue of mixing evaluation with typing is still present, and must be addressed in some fashion. Therefore the typing rules explicitly handle these primitive types and provided operators.

In general, development of these rules was not expected to greatly redirect the specification of the Rosetta language itself, and it did not lead to any new developments in the standards procedure or the design of the language, and it indeed did not cause such changes.

5.2 Basic Typing

The second task of this work is to provide basic typing analysis for the simpler aspects of the Rosetta language. This is characterized by simple expressions and scoping constructs, and does not include type inference or anything of a dependent nature. The analysis supports the following kinds of expressions:

Expression	Example
atomic (literal)	-1 5.2 'c' true ...
operators	+ - * / # ...
function definition	f ($\overline{x :: T}$)::T_r is e;
function application	f($\overline{a}$)
if expression	if b then e_t else e_f end if
case expression	case e of $\overline{p \rightarrow e}$ end case
let expression	let $\overline{x :: T \text{ be } e}$ in e_{body} end let
primitive types	bit natural integer real boolean character string top bottom
composite values	{sets} {* multisets *} [sequences] bitvectors word(n)

The analysis also reasons about the following Rosetta functions that are defined in the prelude:

numerical operators	+ - *
bit/boolean operators	& and or xor
relational operators	< > >= =< = ==
bitvector/word operator	sequence # word replace

Within the implementation of the analysis, the abstract syntax graph admits a smaller core of language constructs. The various supported expressions follow.

- Literals – Various numerical values, booleans, characters, and strings are directly represented in the ASG. Their typing rules are straightforward: they are of the respective type. Some of the numerical types require a check on the specific value, e.g. if an integer value is 0 or 1, and would return `bit`; if instead we can show the value is greater than zero, we report `posint`; and so on.

- Lambda – Functions are represented with the more familiar lambda expression. Rosetta lambdas are ascribed with a type, containing both quantified parameters (which may be possibly inferred) and

explicit parameters. Being Rosetta, the ascribed range may depend on the values of these parameters. We do not differentiate between Π values and λ values. Typechecking a lambda consists of visiting the parameter nodes (to ensure their usage will see the ascriptions as typing results), visiting the body of the lambda, and checking that the body of the lambda is a subtype of the ascribed range.

- Application – To typecheck an application, we first visit the function and all arguments. We then check if there are any quantified parameters that were omitted, and attempt to infer them. We then check that all parameters are subtypes of the respective formal parameters of the function being applied to the arguments.

- Conditional – To typecheck if-expressions we first visit each subterm. The guard is restricted to be a subtype of `boolean`. We then calculate the least upper bound of the two branches. We do not check for one to be a subtype of the other, as that would exclude cases such as `if true then -5 else 5 end if`; we cannot claim that `posint` or `negint` is a subtype of the other, and yet there is an appropriate type to return. One unfortunate consequence of allowing the least upper bound to be returned is that we may find a least upper bound of `top`. But this is technically valid Rosetta, and is not an error. The directly surrounding expression most likely expects to know more about the value's type than `top` betrays, and a mistaken `top`-typed value will most likely trigger a type error when used.

- Constructor Types – Rosetta constructed type definitions and constructors are represented in the ASG. In order to typecheck a constructor usage, we can use the (provided) node of the constructed type's definition to build the type "type of node *i*". A constructed type definition itself is of type `type`. It would be tempting to report more here, but violating the value-type-kind level, especially when the distinction isn't syntactically or semantically preserved, causes trouble.

- Case expression – Case expressions require us to inspect a scrutinee expression, and then combine several patterns over an expression to define a type that contains that scrutinized expression. We must consider inferrable parameters when defining that type. Next, we visit each pattern, taking care to visit the variable matches within to prepare for visiting the matching result expression. We find the

least upper bound of all of these result expressions, for similar reasons to the conditional expression, and report this as the type of the entire case expression.

- Let expression – A let expression introduces named expressions that scope over the body of the let expression. In the ASG, we simply visit the declarations to associate typing annotations with the named items' nodes themselves, and then visit the body, returning the body's type as the entire let expression's type.

- Ascription – We view the type ascription's node as a type, to convert from a Rosetta-syntax version of a type to our own internal representation of a type, visit the annotated expression, and enforce the subtyping relation between the two. If the subtyping relation does not hold, we generate an error, but still report the annotated type. This seems to capture the specifier's intent the best.

The ASG also supports representations of Rosetta types.

- Dependent Product – this represents a Π type. Rosetta does not explicitly have Π in its syntax, but the abstract syntax graph uses them to represent function types, as they are semantically the same as the dependent product Π we saw in chapter 2. Internally, the analysis represents this with our arrow type `TyArrow`, since all typing may be dependent.

- SubtypeOf – use of the `subtype` keyword results in this. It is represented directly in our internal representation of types.

- Constructed – this represents a constructed type, such as our `Tree` and `List` examples. It is represented directly in our internal representation of types.

All of the supported Rosetta design units in the ASG are placed in a single functor, called `Facet`. We visit its domain expression, and then the quantified and explicit parameters sequentially to include type results for all items that may be referenced through the environment. We then visit the declarations and manually add placeholders for these declarations as they can be mutually recursively defined. We then visit the assumptions, terms, and implications. The assumptions and implications must all be boolean expressions while the terms must be either boolean expressions or facet instantiations.

One aspect of the graphing analysis is that for a variable to refer to a parameter's value, there must be a node for the *value* of the parameter. Since we cannot in general know what form the value will have, the ASG represents it with a `Universal`. A `Universal` contains the node that is its type, and maybe an instance, essentially containing the value that the node itself represents.

Types that are completely unspecified can also be distinguished, by their node in the graph:

```
1 Foo::type;
2 Bar::type;
3 func (a::Foo; b::Bar) :: Bar is b;
```

although we know nothing about `Foo` or `Bar`, the typing analysis will distinguish between values of each. We know absolutely nothing else about the types. We cannot do anything to unbox these values, or inspect them in any way. In fact, the only known value of type `Foo` is the error-value `bottom`.

The analysis also addresses `subtype` declarations to a small degree. In the following, `val` should at a minimum be an `integer` value:

```
1 Bazz::subtype(integer);
2 val::Bazz is 5;
```

We do not know how `Bazz` constrains the `integer` type, if at all, and so there is nothing to enforce.

We have achieved the goals of basic type checking. If a specification uses no dependent typing, then the basic typing aspect of the analysis should completely analyze the specification. The only limitations are to what extent the primitive types are supported, and therefore whether their operators are available. This is just a matter of the degree to which support has been added to the graphing analysis and cases added to the type checking analysis to support these additional types and operators.

This analysis maintains decidability of type checking. When none of the more complex features of Rosetta type system are present, the analysis will return the equivalent of a "yes" or "no" to whether the specification is well-typed or not. Of course, it also provides the ASG graph representation of the specification, along with typing annotations on a per-node basis.

5.3 Partial Typing

The third and most important task of this work is to explore ways to provide typing information to the specifier, and to perform localized checks that we find feasible. This allows us to consider the thesis of the work–that providing typing analysis and type information improves the specification refinement process.

5.3.1 Local Type Inference

The first major task was to adapt the local type inference approach of Pierce and Turner [39] to the Rosetta language, exhibiting a much richer type system than System $F_{<:}$ of the original implementation. This approach allows parametric type parameters to be omitted and thus inferred when a principal (best) type can be determined. Rosetta provides for such parameters directly in the language itself, called quantified parameters. These parameters are found in square brackets, distinct from the explicitly required parameters.

Local type inference allows the Rosetta typing analysis to type check strictly more terms than it previously was capable of doing. A previous iteration of the typing analysis used a more traditional approach, where global constraints were collected and unified as one set. This previous approach had two major issues. First, with no notion of the 'closeness' of constraints over particular areas of the specification, the constraints that were the cause of unification failure were often not the specific constraints that eventually clashed. By limiting constraints unification to a specific application site, it is clearer with local type inference where the source of the error is. Second, when the constraints clearly could not be unified, there was no sense of continuing with the typing analysis. Which constraints should be removed in order to continue unification in a meaningful fashion, just the ones that clashed? There is no good recovery strategy in this situation and the localized technique is more resilient in this case.

The clearest set of examples that become type checkable via local type inference are user-defined constructed types parameterized over type parameters, such as the `List` and `Tree` types. They generate constructor, observer, and recognizer functions with the `data` definition's type parameters designated as inferrable parameters. Usage of these functions, as well as case expressions over these constructed types, all require some type to be supplied for these type parameters. Through local type inference, these can be omitted and inferred when a principal type exists. Because there is not always a principal type, we could

not guarantee to always infer type parameters. A global constraint solving approach also would not be able to always infer type parameters, and has no recourse in case of failure. But the local approach targets each individual application with omitted parameters and attempts to infer them, much in the spirit of our best effort approach that expects incompleteness and failures, and expects to continue the analysis.

5.3.2 Witnesses

The second approach to locally extrapolating or extracting information out of a specification through the typing analysis is through the tracking of witness values. The Rosetta typing analysis has a type representation that represents a type along with a specific value of that type. Thus, when an expression receives a typing annotation we can keep track of the value itself with the type.

This work tracks these value ‘witnesses’ and attempts to allow a very simplified form of evaluation arise in type expressions. We gave an example in §4.5.1, where a constructed type expression was used to represent a type. This allows more Rosetta types to be included in type checking. In general, the approach of tracking witnesses allows for more information to flow through the typing analysis, leading to more chances to reason about the specification. The witness information can always be safely ignored when it is not needed, but provides more information that has been shown useful. Because Rosetta does not differentiate between values, types, and kinds syntactically, constructed types themselves are available in type expressions, and witnesses allow us to analyse such types.

5.3.3 Tracking Sizes

One particular usage of witness types that we pursued further was in tracking the sizes of bitvector `words`. This work tracks numerical parameters that are used in a dependently typed fashion, primarily for examples such as `double(n::natural;x::word(n))::word(n+n)`. As we discussed in §2.3, general dependent typing leads to undecidable type checking, and thus we seek limited forms that retain decidability of type checking. The limitation in this case is to only allow numerical expressions that are representable in Presburger arithmetic (§ 4.5.5, [21, 30]) for which decision procedures exist [13]. We use a minimal representation of Rosetta values that can be wrapped into the internal representation of types, and then encode Presburger formulæ and dispatch them to an implementation of Cooper’s algorithm to reliably perform this

limited form of evaluation at the type level. As is the recurring theme in this work, whenever we cannot phrase the check in a sufficient fashion, we can always simply drop the check, generate a warning, and continue the analysis over the rest of the specification.

5.3.4 Error Reporting and Graph Annotations

Error recovery is often a goal in static analysis tools. Given our focus on localized, best-effort analysis that highly expects portions of a specification to be under-specified, we seek error recovery to avoid breaking the standard typing analysis at the first error or underspecified portion of a specification. This work pushes most reasoning about types down to concrete type representations, helping in both reporting and recovering from errors. There is no global type variable constraints unification, only localized constraints unification, where the source of the error is well defined. Rosetta does offer ascriptions alongside most every declaration, so we can use these ascriptions to help contain typing errors from crossing scoping boundaries.

The analysis generates the abstract syntax graph of the specification and annotates a type for every single node in the graph, meaning that each sub-expression is both type-checked and annotated for the specifier to review. While we do not pretend to offer a polished interface in providing this information at this time, the presence of this information allows any found typing errors to be analyzed in greater detail. We provide both the internal type representation and a more human-readable type representation, where primitive types and constructed types' names are included with syntax much closer to Rosetta syntax. Compilers and static analysis tools cannot read the programmer's or specifier's intent, and errors in language usage often appear away from the error in semantic intent, but tools usually only provide information where the error was found, and not where errors were not found — even though these other locations may indeed be legal but against the writer's intent.

When a specification cannot be successfully typed by the basic checks that suffice for more restricted languages and the more opportunistic checks that attempt to extend into the dependent typing realm selectively, the specifier will have to be involved. The approach of compartmentalizing errors and providing extensive annotated analysis results allows for the specifier to interact not only with the end result of the analysis, but with the information that was generated throughout the whole analysis.

5.4 Driving Specification Refinement

In the end, what can we say about a best-effort, possibly incomplete approach to a typing analysis and its effects on the process of specification refinement? When failure becomes an option — in the sense of unfulfilled analysis tasks, not in the sense of type incorrectness — we find there are ways to bring more information about the specification to the specifier, and we find we can apply techniques not always intended for a language consisting of quite the same language features as our target.

We have argued the position that specification languages should not be conflated with programming languages, because there are many opportunities to serve the interests of the system specifier. These arise both by relaxing executability restrictions that are not in place and by enhancing other efforts through controlled analysis extensions that smell of dependent typing.

We have shown that we can perform opportunistic analysis without having to guarantee that it can always be done. Indeed, most of the localized and controlled dependently typed checks with which we experimented can be overwhelmed by legitimate examples in the language. But we can still perform these checks and enhance the specifier's ability to handle the successful cases.

We have shown that we can expose information from throughout the analysis through the graph annotations that are a byproduct of the analysis. As argued previously, this allows the specifier to analyse not just the location where an error was discovered, but all areas down to the sub-expression where the specifier's intent can be checked against the automatic, intent-agnostic results of the static typing analysis. This is useful both in solving typing errors as well as checking that the specifier's intent has been preserved.

This work has provided a foundation of typing semantics in the form of typing rules. It provides for basic type checking, applicable as an executable static analysis and integrated through the Rosetta Eclipse plugin. Further opportunistic approaches to typing were implemented to attempt to provide extra checks and safeguards on the code. This analysis provides a detailed typing annotation over the graph representation of the specification for detailed inspection of the typing analysis' results. And the approach of localizing these different approaches through a focus on concrete types allows for decent error recovery and pinpointing the source of errors.

By considering the purpose of types in a specification language and defining what a typing analysis should be, we have identified an approach to testing and preserving typing semantics in a language such as Rosetta that focuses on the process of generating a well typed specification rather than the end test of whether a specification obeys the language's typing rules. This end check is still possible with our tool. In our view, the only appropriate way to approach the type system of a language with undecidable typing is to involve the specifier in the process. Whether we are pinpointing where more information is needed, or eventually going further to state exactly what further information is needed to prove the claims made on the types in a specification, the specifier will need to be involved in the process. Specifications are written to explore what a system is and how it behaves rather than to create the artifact of the system itself, and thus this purpose pervades the static analysis tools written over the specification language.

Appendix A

Rosetta Typing Rules

Here we collect all the type rules that define the Rosetta type system. There is more in-depth discussion in §4.2, but any rules presented there are also included here for a complete record of the typing rules.

A.1 Design Units

A facet definition has a domain, parameters, definitions, and declarations. The facet is an extension of that domain, so all domain definitions are present and visible all throughout the facet. Declarations are not ordered: they are mutually defined. All parameters are visible in the declarations. Definitions (the 'body' of the facet) are also mutually defined between themselves; all parameters and declarations are visible.

- **G-Facet**

$$\frac{\begin{array}{l}\Gamma \vdash_{\gamma} dom : \Gamma_{dom} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P} \vdash_{\tau} \overline{decl} : \overline{Decl} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P}, \overline{decl} : \overline{Decl} \vdash_{\tau} \overline{defn} : \overline{Defn} \\ \Gamma_d = \Gamma_{dom} \cup \{\overline{p} : \overline{P}, \overline{def} : \overline{Def}, \overline{decl} : \overline{Decl}\}\end{array}}{\Gamma \vdash_{\gamma} facet\ f(\overline{p} :: \overline{P}) :: dom\ is\ \overline{decl}\ begin\ \overline{defn}\ end\ facet\ f : \Gamma_d}$$

- **T-Facet**

$$\frac{\Gamma \vdash_\gamma dom : \Gamma_{dom} \quad \Gamma, \Gamma_{dom}, \overline{p} : \overline{P} \vdash_\tau \overline{decl} : \overline{Decl} \quad \Gamma, \Gamma_{dom}, \overline{p} : \overline{P}, \overline{decl} : \overline{Decl} \vdash_\tau \overline{defn} : \overline{Defn}}{\Gamma \vdash_\tau facet\ f(\overline{p} :: \overline{P}) :: dom\ is\ \overline{decl}\ begin\ \overline{defn}\ end\ facet\ f : dom}$$

A facet instantiation is a fully-applied facet definition, meaning that all parameters are supplied. (A partially-applied facet definition may be considered a function from the arguments to a facet instance). These parameters must be substituted all throughout the facet. The resulting type of a facet instantiation is this substituted version of the facet definition's type, which will be the domain specified.

- **G-FacetInstantiation**

$$\frac{\Gamma \vdash_\tau \overline{a} : \overline{P} \quad <f :(\overline{p} : \overline{P}) | \Gamma_f> \in \Gamma}{\Gamma \vdash_\gamma f(\overline{a}) :: [\overline{p} \mapsto \overline{a}]\Gamma_f}$$

- **T-FacetInstantiation**

$$\frac{\Gamma \vdash_\tau \overline{a} : \overline{P} \quad <f :(\overline{p} : \overline{P}) | \Gamma_f> \in \Gamma}{\Gamma \vdash_\tau f(\overline{a}) :: [\overline{p} \mapsto \overline{a}]f}$$

A package is a collection of definitions that may be imported through `use`-clauses. Packages can themselves be parameterized.

- **G-Package**

$$\frac{\Gamma \vdash_\gamma dom : \Gamma_{dom} \quad \Gamma, \Gamma_{dom}, \overline{p} : \overline{P} \vdash_\tau \overline{def} : \overline{Def} \quad \Gamma, \Gamma_{dom}, \overline{p} : \overline{P}, \overline{def} : \overline{Def} \vdash_\gamma \overline{def} : \Gamma_{\overline{def}}}{\Gamma \vdash_\gamma package\ pack(\overline{p} : \overline{P}) : dom\ is\ \overline{def}\ end\ package\ pack; \ : \Gamma_{dom}, \overline{def} : \overline{Def}, <pack :(\overline{p} : \overline{P}) | \Gamma_{dom}, \overline{def} : \overline{Def}>}$$

- **T-Package**

$$\frac{\begin{array}{l}\Gamma \vdash_{\gamma} dom : \Gamma_{dom} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P} \vdash_{\tau} \overline{def} : \overline{Def}\end{array}}{\Gamma \vdash_{\gamma} package\ pack(\overline{p} : \overline{P}) : dom\ is\ \overline{def}\ end\ package\ pack; : dom}$$

A component combines the declarative aspects of a facet with sets of assumptions and implications about its usage. These are pre- and post-conditions.

- **G-Component**

$$\frac{\begin{array}{l}\Gamma \vdash_{\gamma} dom : \Gamma_{dom} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P} \vdash_{\tau} \overline{defn} : \overline{Defn} \\ \Gamma_{d} = \Gamma_{dom} \cup \{\overline{defn} : \overline{Defn}\}\end{array}}{\begin{array}{l}\Gamma \vdash_{\gamma} component\ c(\overline{p} : \overline{P}) : dom\ is\ \overline{a}\ \overline{defn}\ \overline{imp}\ end\ component\ c \\ \quad : \Gamma_{dom}, \overline{defn} : \overline{Defn}, <c : (\overline{p} : \overline{P}) | \Gamma_{dom}, \overline{defn} : \overline{Defn}>\end{array}}$$

- **T-Component**

$$\frac{\begin{array}{l}\Gamma \vdash_{\gamma} dom : \Gamma_{dom} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P} \vdash_{\tau} \overline{defn} : \overline{Defn} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P}, \overline{defn} : Defn \vdash_{\tau} \overline{a} : \textbf{boolean} \\ \Gamma, \Gamma_{dom}, \overline{p} : \overline{P}, \overline{defn} : Defn \vdash_{\tau} \overline{imp} : \textbf{boolean}\end{array}}{\Gamma \vdash_{\tau} component\ c(\overline{p} : \overline{P}) : dom\ is\ \overline{a}\ \overline{defn}\ \overline{imp}\ end\ component\ c : dom}$$

- **G-Component-Instantiation**

$$\frac{\begin{array}{l}\Gamma \vdash_{\tau} \overline{a}^{n} : \overline{P}^{n} \\ <c : (\overline{p}^{n} : \overline{P}^{n}) | \Gamma_{f}> \in \Gamma\end{array}}{\Gamma \vdash_{\gamma} c(\overline{a}) : [\overline{p} \mapsto \overline{a}]\Gamma_{f}}$$

- **T-Component-Instantiation**

$$\frac{\begin{array}{l}\Gamma \vdash_{\tau} \overline{a}^{n} : \overline{P}^{n} \\ <c : (\overline{p} : \overline{P}^{n}) | \Gamma_{f}> \in \Gamma\end{array}}{\Gamma \vdash_{\tau} c(\overline{a}^{n}) : [\overline{p} \mapsto \overline{a}]c}$$

Rosetta domains have already been discussed to some length. A domain definition must extend a domain, and may introduce its own definitions.

- **G-Domain**

  $\Gamma \vdash_\gamma S_{dom} : \Gamma_S$

  $\Gamma, \Gamma_S, \overline{p} : \overline{P} \vdash_\tau \overline{defn} : \overline{Defn}$

  $\Gamma, \Gamma_S, \overline{p} : \overline{P}, \overline{defn} : \overline{Defn} \vdash_\tau \overline{tm} : \overline{Tm}$

  $\Gamma_D = \Gamma_{S_{dom}}, \overline{defn} : \overline{Defn}, \overline{tm} : \overline{Tm}$

  $\overline{\Gamma \vdash_\gamma domain\ dom(\overline{p} : \overline{P}) : S_{dom}\ \overline{decl}\ \overline{tm}}$

  $: \Gamma_D, < c : (\overline{p} : \overline{P}) | \Gamma_D >$

- **T-Domain**

  $\Gamma \vdash_\gamma S_{dom} : \Gamma_S \quad \Gamma \vdash_\tau S_{dom} : T_S$

  $\Gamma, \Gamma_S, \overline{p} : \overline{P} \vdash_\tau \overline{defn} : \overline{Defn}$

  $\Gamma, \Gamma_S, \overline{p} : \overline{P}, \overline{defn} : Defn \vdash_\tau \overline{tm} : \overline{Tm}$

  $\overline{\Gamma \vdash_\tau domain\ dom(\overline{p} : \overline{P}) : S_{dom}\ \overline{decl}\ \overline{tm} : S_{dom}}$

A.2 Use Clauses

A use clause allows a package to be imported, such that all definitions within it are visible for a given design unit such as as facet, domain, component, or other package. It behaves just like a giant, special `let` binding.

- **G-Use**

  $\Gamma \vdash_\gamma p : \Gamma_p$

  $\Gamma, \Gamma_p \vdash_\gamma decl : \Gamma_{decl}$

  $\overline{\Gamma \vdash_\gamma use\ p\ in\ decl : \Gamma_{decl}}$

- **T-Use**

  $\Gamma \vdash_\gamma p : \Gamma_p$

  $\Gamma, \Gamma_p \vdash_\tau decl : T_{decl}$

  $\overline{\Gamma \vdash_\tau use\ p\ in\ decl : T_{decl}}$

A.3 Quantified Parameters

As mentioned in §2.2.3, when local type inference is capable of identifying a principal type for a type parameter, we will capture the claim that local type inference can find a principal type for a particular omitted type parameter, through the `infer` relation. The required information is an application of a function with inferrable parameters $\overline{Q}$ and explicit parameters $\overline{p}$, and supplied arguments $\overline{a}$. We can then represent the `T-App-InfAlg` rule's applicability thus:

$$\frac{\begin{array}{l}\Gamma \vdash_\tau f : [\overline{Q}](\overline{p} : \overline{P}) : R \\ \Gamma \vdash_\tau \overline{a} : \overline{A} \\ \{\} \vdash_\tau \overline{A} <: \overline{P} \Rightarrow \overline{C} \\ \overline{I} = \sigma \overline{C}_R\end{array}}{\text{infer}(\overline{Q}, \overline{p} : \overline{P}, R, \overline{a} : \overline{A}) \Rightarrow \overline{I}}$$

If we're given the function signature `f[Q]`$(\overline{p}:\overline{P})$`:R`, and arguments $\overline{a}:\overline{A}$, and if ensuring that $\overline{A} <: \overline{P}$ implies the constraint set `C` upon the inferrable parameters $\overline{Q}$, then we can define $\overline{I}$ as the best set of substitutions for inferrable types $\overline{Q}$ when solving the constraint set `C` with respect to the type `R`. `infer` doesn't refer to f directly, instead relating its quantified parameters, explicit parameters, range type, and arguments to the best set of type substitutions for the inferrable quantified parameters.

Because this `infer` relation only shows up in a few places in the Rosetta rules, we do not re-write all of the typing rules from local type inference for Rosetta directly. They would be identical up to the language syntax differences. Lambdas and applications become any parameterizable definitions and function calls/instantiations; `top` and `bottom` also exist in Rosetta; and Rosetta also defines the subtyping relation `<:`.

Of course, in Rosetta we are only allowed to infer quantified parameters; explicit parameters must be supplied, even if they are inferrable. Quantified parameters may seem to be conflated with inferrable parameters, but the distinction between the two is made by the specifier by choosing whether to place a type parameter in the quantified parameters list of the explicit parameters list.

A.4 Qualified Names

- **G-Qualified**.

  $\Gamma \vdash_\gamma a : \Gamma_a$

  $\Gamma_a, \Gamma \vdash_\gamma b : \Gamma_b$

  $\overline{\Gamma \vdash_\gamma a.b : \Gamma_b}$

- **T-Qualified**.

  $\Gamma \vdash_\gamma a : \Gamma_a$

  $\Gamma_a \vdash_\tau b : T_b$

  $\overline{\Gamma \vdash_\tau a.b : T_b}$

A.5 Variables

A variable's type is just a lookup into the environment: the variable must be in the environment for any well formed expression, and thus the lookup will not fail if the scope-introducing construct correctly extended the environment as we require.

- **T-Var**

  $t : T \in \Gamma$

  $\overline{\Gamma \vdash_\tau t : T}$

A.6 Applications

Applications occur whenever we supply parameters. This can apply to facet definitions, functions, anything with parameters. The dependent nature of Rosetta means that we must perform substitutions for our parameters. As the orderedness of parameters affects their environments, so too does the orderedness affect the substitution of parameters. Since a parameter is visible in all subsequent parameters, we substitute parameters one at a time, through both the remaining parameters and the range of the function.

- **G-Application**

$$\frac{\begin{array}{l}\Gamma \vdash_\tau f : (\overline{p} : \overline{P}) \rightarrow R \\ \Gamma \vdash_\tau \overline{a} : \overline{P} \\ \Gamma, \overline{p} : \overline{P} \vdash_\gamma R : \Gamma_R\end{array}}{\Gamma \vdash_\gamma f(\overline{a}) : \Gamma_R}$$

- **T-Application**

$$\frac{\begin{array}{l}\Gamma \vdash_\tau f : \overline{P} \rightarrow T \\ \Gamma \vdash_\tau \overline{p} : \overline{P}\end{array}}{\Gamma \vdash_\tau f(\overline{p}) : T}$$

- **G-Application-Infer**

$$\frac{\begin{array}{l}\Gamma \vdash_\tau f : [\overline{Q}](\overline{p} : \overline{P}) \rightarrow R \\ \Gamma \vdash_\tau \overline{a} : \overline{A} \\ \text{infer}(\overline{Q}, \overline{p} : \overline{P}, R, \overline{a} : \overline{A}) \Rightarrow \overline{I} \\ \Gamma, \overline{p} : \overline{P} \vdash_\gamma [\overline{Q} \mapsto \overline{I}]R : \Gamma_R\end{array}}{\Gamma \vdash_\gamma f(\overline{a}) : \Gamma_R}$$

- **T-Application-Infer**

$$\frac{\begin{array}{l}\Gamma \vdash_\tau f : [\overline{Q}](\overline{p} : \overline{P}) \rightarrow R \\ \Gamma \vdash_\tau \overline{a} : \overline{A} \\ \text{infer}(\overline{Q}, \overline{p} : \overline{P}, R, \overline{a} : \overline{A}) \Rightarrow \overline{I}\end{array}}{\Gamma \vdash_\tau f(\overline{a}) : [\overline{Q} \mapsto \overline{I}, \overline{p} \mapsto \overline{a}]R}$$

A.7 Functions

Rosetta provides multiple ways of defining functions. The body of a function may be omitted, may be defined, or may be declared as `constant` without giving a specific value. A function definition also may include a `where` clause, placing constraints upon the values via `boolean` expressions that must be true, without directly specifying the value of the function. These various syntactic forms all obey the same typing

rules: the body of the function must be of the type of the range of the function, and `where`-clauses must contain a boolean expression.

- **T-Function-Anonymous**

$$\frac{\Gamma \vdash_\tau \overline{p:P} \qquad \Gamma \vdash_\tau t:T}{\Gamma \vdash_\tau \; <*(\overline{p:P}) :: T \text{ is } t*>: \; \overline{p:P} \rightarrow T}$$

- **T-FunctionApplication**

$$\frac{\Gamma \vdash_\tau \overline{a:P} \qquad \Gamma \vdash_\tau f:(\overline{p:P}) \rightarrow T_R}{\Gamma \vdash_\tau f(\overline{a}) \;:\; [p \mapsto a]T_R}$$

A.7.1 Direct Functions

- **T-FunctionInterpretable**

$$\frac{\Gamma \vdash_\tau \overline{p:P} \qquad \Gamma \vdash_\tau t:T}{\Gamma \vdash_\tau f(\overline{p::P}) :: T \text{ is } t; \; : \; \overline{p:P} \rightarrow T}$$

- **T-FunctionUninterpretable**

$$\frac{\Gamma \vdash_\tau \overline{p:P}}{\gamma \vdash_\tau f(\overline{p::P}) :: T \text{ is constant; } : \; \overline{p:P} \rightarrow T}$$

- **T-FunctionQualifiedInterpretable**

$$\frac{\Gamma \vdash_\tau \overline{p:P} \qquad \Gamma \vdash_\tau t:T \qquad \Gamma \vdash_\tau t_w : \textbf{boolean}}{\Gamma \vdash_\tau f(\overline{p::P}) :: T \text{ is } t \text{ where } t_w; \; : \; (\overline{p:P}) \rightarrow T}$$

- **T-FunctionQualifiedUninterpretable**

$$\frac{\Gamma \vdash_\tau \overline{p:P} \qquad \Gamma \vdash_\tau t_w : \textbf{boolean}}{\Gamma \vdash_\tau f(\overline{p::P}) :: T \text{ is constant where } t_w; \; : \; \overline{p:P} \rightarrow T}$$

- **T-FunctionVariable**

$$\frac{\Gamma \vdash_\tau \overline{p : P}}{\Gamma \vdash_\tau f(\overline{p :: P}) :: T;\ :\ \overline{p : P} \to T}$$

- **T-FunctionQualifiedVariable**

$$\frac{\Gamma \vdash_\tau \overline{p : P} \qquad \Gamma \vdash_\tau t_w : \mathbf{boolean}}{\Gamma \vdash_\tau f(\overline{p :: P}) :: T \text{ where } t_w;\ :\ \overline{p : P} \to T}$$

A.7.2 Anonymous Functions

- **T-FunctionFormer**

$$\frac{\Gamma \vdash_\tau \overline{t : T} \qquad \Gamma \vdash_\tau t_e : T_e}{\Gamma \vdash_\tau\ <\!*\overline{t :: T}*\!>[\text{ is } t_e]\ :\ \overline{t : T} \to T_e}$$

- **T-FunctionValue**

$$\frac{\Gamma \vdash_\tau \overline{p : P} \qquad \Gamma \vdash_\tau t : T \qquad \Gamma \vdash_\tau t_w : \mathbf{boolean}}{\Gamma \vdash_\tau\ <\!*(\overline{p :: P}) :: T[\text{ is } (t|\text{ constant })]*\!>[\text{ where } t_w]\ :\ \overline{p : P} \to T}$$

A.8 Let-expressions

- **G-Let**

  $\Gamma \vdash_\tau \bar{t} : \overline{T}$

  $\Gamma, \bar{x} : \overline{T} \vdash_\gamma e : \Gamma_e$

  $\overline{\Gamma \vdash_\gamma \text{let } \bar{x} \text{ be } \bar{t} \text{ in } e \text{ end let;} : \Gamma_e}$

- **T-Let**

  $\Gamma \vdash_\tau \bar{t} : \overline{T}$

  $\Gamma, \bar{x} : \overline{T} \vdash_\tau e : T_e$

  $\overline{\Gamma \vdash_\tau \text{let } \bar{x} \text{ be } \bar{t} \text{ in } e \text{ end let;} : T_e}$

A.9 Sequence Predicates

Rosetta provides some predicate functionality that can be applied across sequences.

- **G-Forall**

  $$\frac{}{\Gamma \vdash_\gamma \text{forall}(x : T \mid pred) : \{\}}$$

- **T-Forall**

  $$\frac{\Gamma, x : T \vdash_\tau pred : \textbf{boolean}}{\Gamma \vdash_\tau \text{forall}(x : T \mid pred) : \textbf{boolean}}$$

- **G-Filter**

  $$\frac{\Gamma \vdash_\gamma \; seq : \Gamma_s}{\Gamma \vdash_\gamma \text{filter}(pred, seq) : \Gamma_s \setminus \{x : T | pred(x) = \textbf{false}\}}$$

- **T-Filter**

  $$\frac{\Gamma \vdash_\tau pred : T \to \textbf{boolean} \qquad \Gamma \vdash_\tau s : \textbf{sequence}(T)}{\Gamma \vdash_\tau \text{filter}(pred, s) : \textbf{sequence}(T)}$$

A.10 Control-Flow Expressions

- **G-If**

  $$\frac{\Gamma \vdash_\gamma t : \Gamma_t \;\; \Gamma \vdash_\gamma f : \Gamma_f \qquad \Gamma_R = \Gamma_t \cap \Gamma_f}{\Gamma \vdash_\gamma \text{if } b \text{ then } t \text{ else } f \text{ end if; } : \Gamma_R}$$

- **T-If**

  $$\frac{\Gamma \vdash_\tau b : \textbf{boolean} \qquad \Gamma \vdash_\tau t : T \;\; \Gamma \vdash_\tau f : F \qquad R = T \sqcap F}{\Gamma \vdash_\tau \text{if } b \text{ then } t \text{ else } f \text{ end if; } : \; R}$$

- **G-Case**

  $\Gamma \vdash_\tau \overline{p} : \overline{P}$

  $\Gamma, p_i : P_i \vdash_\gamma e_i : \Gamma_{e_i}$ for each i

  $\Gamma_R = \bigcap \overline{\Gamma_e}$

  $\overline{\Gamma \vdash_\gamma \text{case } e \text{ of } \overline{p} \rightarrow \overline{e} \text{ end case; } : \ \Gamma_R}$

- **T-Case**

  $\Gamma \vdash_\tau e : E$

  $\Gamma \vdash_\tau \overline{p} : \overline{P}, \ \ \sqcap \overline{P} = E$

  $\Gamma, p_i : P_i \vdash_\tau e_i : T_{e_i}$ for each i

  $T = \sqcap \overline{T_e}$

  $\overline{\Gamma \vdash_\tau \text{case } e \text{ of } \overline{p} \rightarrow \overline{e} \text{ end case; } : \ T}$

A.11 Ascriptions

Ascriptions do not introduce new scoped items, so the `G-Ascription` rule only passes the generated environment through. Also, note that we do not perform any type checking that is not required to formulate the resulting environment in a $\vdash_\gamma$ rule. That check is performed in the $\vdash_\tau$ rule.

- **G-Ascription**

  $\Gamma \vdash_\gamma t : \Gamma_T$

  $\overline{\Gamma \vdash_\gamma t : T : \Gamma_T}$

- **T-Ascription**

  $\Gamma \vdash_\tau t : S$

  $S <: T$

  $\overline{\Gamma \vdash_\tau t : T : T}$

A.12 Constructed Types

User-defined constructed types allow for a parameterized definition of a collection of constructors, observers, and recognizers. The parameters play an important role in the typing rules: although explicit in a fully instantiated representation of the constructed type, these type parameters become quantified (inferrable) parameters for all of the constructors, observers, and recognizers.

- **G-Constructed-Type**

  $\Gamma_{constr} = \cup \overline{constructor} : [\overline{p}](\overline{observer}) \rightarrow adt(\overline{p})$

  $\Gamma_{obser} = \cup \overline{observer} : [\overline{p}](\overline{observer} :: \overline{Observer}) \rightarrow \overline{observer}$

  $\Gamma_{recog} = \cup \overline{recog} : [\overline{p}](\overline{adt(\overline{p})}) \rightarrow \textbf{boolean}$

  $\Gamma_d = \{adt : (\overline{p} :: \overline{P}) \rightarrow type\} \cup \Gamma_{constr} \cup \Gamma_{obser} \cup \Gamma_{recog}$

  $\Gamma \vdash_\gamma \text{adt } (\overline{p} :: \overline{P}) :: \text{type is data } \overline{constructor}(\overline{\overline{observer} :: \overline{Observer}}) :: \overline{recognizer} \text{ end data; } : \Gamma_d$

- **T-Constructed-Type**

  $\Gamma \vdash_\tau \overline{P} :: \text{type}$

  $\Gamma, \overline{p} :: \overline{P} \vdash_\tau \overline{Observer} :: \text{type}$

  $\Gamma \vdash_\tau \text{adt } (\overline{p} :: \overline{P}) :: \text{type is data } \overline{constructor}(\overline{\overline{observer} :: \overline{Observer}}) :: \overline{recognizer} \text{ end data;}$

  $: (\overline{p} :: \overline{P}) \rightarrow \text{type}$

A.13 Rosetta-Standard Operators

Rosetta has many functions built into the language through the `prelude`. Many of these are defined over the composite types and over the primitive numerical types of Rosetta. As the semantics of these operators are built into Rosetta directly, we present some typing rules to give meaning to these operators.

A.13.1 Composite Types

Composite types are ways of structuring values of some particular type. The three composite types offered in Rosetta are sets, multisets, and sequences. As these composite types are all concerned with constructing values and not with introducing any scope, we do not provide $\vdash_\gamma$ rules.

- **T-Set**

$$\frac{}{\Gamma \vdash label :: \textbf{set}(T) : \{T\}}$$

- **T-Multiset**

$$\frac{}{\Gamma \vdash label :: \textbf{multiset}(T) : \{*T*\}}$$

- **T-Sequence**

$$\frac{}{\Gamma \vdash label :: \textbf{sequence}(T) : [T]}$$

- **T-SetFormer**

$$\frac{\forall i \ \Gamma \vdash t_i : T}{\Gamma \vdash \{t_0, t_1, \ldots, t_n\} : \textbf{set}(T)}$$

- **T-MultisetFormer**

$$\frac{\forall i \ \Gamma \vdash t_i : T}{\Gamma \vdash \{*t_0, t_1, \ldots, t_n*\} : \textbf{multiset}(T)}$$

- **T-SequenceFormer**

$$\frac{\forall i \ \Gamma \vdash t_i : T}{\Gamma \vdash [t_0, t_1, \ldots, t_n] : \textbf{sequence}(T)}$$

A.14 Set Operations

- **T-SetRelations**

  Let op be one of: $< \mid =< \mid >= \mid > \mid ==$

$$\frac{\Gamma \vdash t_1 : \textbf{set}(T) \ \Gamma \vdash t_2 : \textbf{set}(T)}{\Gamma \vdash t_1 op t_2 : \textbf{boolean}}$$

- **T-SetUnion**

$$\frac{\Gamma \vdash t_1 : \textbf{set}(T) \ \Gamma \vdash t_2 : \textbf{set}(T)}{\Gamma \vdash t_1 + t_2 : \textbf{set}(T)}$$

- **T-SetIntersection**

$$\frac{\Gamma \vdash t_1 : \textbf{set}(T) \ \Gamma \vdash t_2 : \textbf{set}(T)}{\Gamma \vdash t_1 * t_2 : \textbf{set}(T)}$$

- **T-SetDifference**

$$\frac{\Gamma \vdash t_1 : \textbf{ set}(T) \; \Gamma \vdash t_2 : \textbf{ set}(T)}{\Gamma \vdash t_1 - t_2 : \textbf{ set}(T)}$$

- **T-SetCardinality**

$$\frac{\Gamma \vdash t : \textbf{ set}(T)}{\Gamma \vdash \#t : \textbf{ natural}}$$

- **T-SetContainment**

$$\frac{\Gamma \vdash t_v <: T \; \Gamma \vdash t_s : \textbf{ set}(T)}{\Gamma \vdash t_v \, \textbf{in} \, t_s : \textbf{ boolean}}$$

- **T-SetContents**

$$\frac{\Gamma \vdash t : \textbf{ set}(T)}{\Gamma \vdash \sim t : \textbf{ set}(T)}$$

- **T-SetChoose**

$$\frac{\Gamma \vdash t : \textbf{ set}(T)}{\Gamma \vdash \textbf{ choose}(t) : T}$$

- **T-SetImage**

$$\frac{\begin{array}{cc} \Gamma \vdash_\tau Sub_1 : T_1 & \Gamma \vdash_\tau T_2 : Sup_2 \\ \Gamma \vdash_\tau f : Sub_1 \rightarrow Sup_2 & \Gamma \vdash_\tau t_1 : \textbf{set}(T_1) \end{array}}{\Gamma \vdash_\tau \textbf{ image}(f, t_1) \; : \; \textbf{set}(T_2)}$$

- **T-SetFilter**

$$\frac{\Gamma \vdash t : \textbf{ set}(T) \; \Gamma \vdash S <: T \; \Gamma \vdash f : S \rightarrow \textbf{boolean}}{\Gamma \vdash \textbf{ filter}(f, t) : \textbf{ set}(T)}$$

A.15 Multiset Operations

- **T-MultisetUnion**

$$\frac{\Gamma \vdash t_1 : \textbf{ multiset}(T) \; \Gamma \vdash t_2 : \textbf{ multiset}(T)}{\Gamma \vdash t_1 + t_2 : \textbf{ multiset}(T)}$$

- **T-MultisetIntersection**

$$\frac{\Gamma \vdash t_1 : \textbf{ multiset}(T) \; \Gamma \vdash t_2 : \textbf{ multiset}(T)}{\Gamma \vdash t_1 * t_2 : \textbf{ multiset}(T)}$$

- **T-MultisetDifference**

$$\frac{\Gamma \vdash t_1 : \textbf{multiset}(T) \;\; \Gamma \vdash t_2 : \textbf{multiset}(T)}{\Gamma \vdash t_1 - t_2 : \textbf{multiset}(T)}$$

- **T-MultisetCardinality**

$$\frac{\Gamma \vdash t : \textbf{multiset}(T)}{\Gamma \vdash \#t : \textbf{natural}}$$

- **T-MultisetElementCardinality**

$$\frac{\Gamma \vdash t_m : \textbf{multiset}(T) \;\; \Gamma \vdash t_e <: T}{\Gamma \vdash t_e \# t_m : \textbf{natural}}$$

- **T-MultisetContainment**

$$\frac{\Gamma \vdash t_v <: T \;\; \Gamma \vdash t_s : \textbf{multiset}(T)}{\Gamma \vdash t_v \,\textbf{in}\, t_s : \textbf{boolean}}$$

- **T-MultisetContents**

$$\frac{\Gamma \vdash t : \textbf{multiset}(T)}{\Gamma \vdash \sim t : \textbf{set}(T)}$$

- **T-MultisetChoose**

$$\frac{\Gamma \vdash t : \textbf{multiset}(T)}{\Gamma \vdash \textbf{choose}(t) : T}$$

- **T-MultisetImage**

$$\frac{\begin{array}{c}\Gamma \vdash Sub_1 <: T_1 \;\; \Gamma \vdash T_2 <: Sup_2 \\ \Gamma \vdash f : Sub_1 \rightarrow Sup_2 \;\; \Gamma \vdash t_1 : \textbf{multiset}(T_1)\end{array}}{\Gamma \vdash \textbf{image}(f, t_1) : \textbf{multiset}(T_2)}$$

- **T-MultisetFilter**

$$\frac{\Gamma \vdash S <: T \;\; \Gamma \vdash t : \textbf{multiset}(S) \;\; \Gamma \vdash f : T \rightarrow \textbf{boolean}}{\Gamma \vdash \textbf{filter}(f, t) : \textbf{multiset}(T)}$$

- **T-Set2Multiset**

$$\frac{\Gamma \vdash t : \textbf{set}(T)}{\Gamma \vdash \textbf{set2multiset}(t) : \textbf{multiset}}$$

A.16 Sequence Operations

- **T-SeqeuenceConcatenation**

$$\frac{\Gamma \vdash t_1 : \textbf{sequence}\,(T) \;\; \Gamma \vdash t_2 : \textbf{sequence}\,(T)}{\Gamma \vdash t_1 \& t_2 : \textbf{sequence}\,(T)}$$

- **T-SequenceRandomAccess**

$$\frac{\Gamma \vdash t : \textbf{sequence}\,(T) \;\; \Gamma \vdash t_n : \textbf{natural}}{\Gamma \vdash t(t_n) : T}$$

- **T-SequenceSubset**

$$\frac{\Gamma \vdash t : \textbf{sequence}\,(T) \forall i \;\; \Gamma \vdash t_i : \textbf{natural}}{\Gamma \vdash t\, \textbf{sub}\,[t_0, t_1, \ldots, t_n] : \textbf{sequence}\,(T)}$$

- **T-SequenceRelations** (for subsequence testing)

$$Let\, op = < \mid =< \mid >= \mid >$$

$$\frac{\Gamma \vdash t_1 : \textbf{sequence}\,(T) \;\; \Gamma \vdash t_2 : \textbf{sequence}\,(T)}{\Gamma \vdash t_1\, op\, t_2 : \textbf{boolean}}$$

- **T-SequenceMin**

$$\frac{\Gamma \vdash t_1 : \textbf{sequence}\,(T) \;\; \Gamma \vdash t_2 : \textbf{sequence}\,(T)}{\Gamma \vdash t_1\, \textbf{min}\, t_2 : \textbf{sequence}\,(T)}$$

- **T-SequenceMax**

$$\frac{\Gamma \vdash t_1 : \textbf{sequence}\,(T) \;\; \Gamma \vdash t_2 : \textbf{sequence}\,(T)}{\Gamma \vdash t_1\, \textbf{max}\, t_2 : \textbf{sequence}\,(T)}$$

- **T-SequenceCardinality**

$$\frac{\Gamma \vdash t : \textbf{sequence}\,(T)}{\Gamma \vdash \# t : \textbf{natural}}$$

- **T-Contents**

$$\frac{\Gamma \vdash t : \textbf{sequence}(T)}{\Gamma \vdash \sim t : \textbf{multiset}(T)}$$

- **T-SequenceHead**

$$\frac{\Gamma \vdash t : \textbf{sequence}(T)}{\Gamma \vdash \textbf{head}(t) : T}$$

- **T-SequenceTail**

$$\frac{\Gamma \vdash t : \textbf{sequence}(T)}{\Gamma \vdash \textbf{tail}(t) : \textbf{sequence}(T)}$$

- **T-SequenceLast**

$$\frac{\Gamma \vdash t : \textbf{sequence}(T)}{\Gamma \vdash \textbf{last}(t) : T}$$

- **T-SequenceCons**

$$\frac{\Gamma \vdash t : \textbf{sequence}(T) \;\; \Gamma \vdash t_e <: T}{\Gamma \vdash \textbf{cons}(t_e, t) : \textbf{sequence}(T)}$$

- **T-SequenceReverse**

$$\frac{\Gamma \vdash t : \textbf{sequence}(T)}{\Gamma \vdash \textbf{reverse}(t) : \textbf{sequence}(T)}$$

- **T-SequenceImage**

$$\frac{\begin{array}{c}\Gamma \vdash Sub_1 <: T_1 \;\; \Gamma \vdash T_2 <: Sup_2 \\ \Gamma \vdash f : Sub_1 \rightarrow Sup_2 \;\; \Gamma \vdash t : \textbf{sequence}(T_1)\end{array}}{\Gamma \vdash \textbf{image}(f, t) : \textbf{sequence}(T_2)}$$

- **T-SequenceFilter**

$$\frac{\Gamma \vdash t :\ \textbf{sequence}\,(T) \qquad \Gamma \vdash T <: Sup \qquad \Gamma \vdash f : Sup \rightarrow \textbf{boolean}}{\Gamma \vdash \textbf{filter}\,(f, t) :\ \textbf{sequence}\,(T)}$$

- **T-SequenceReduce** (fold left)

$$\frac{\Gamma \vdash f : T \rightarrow T \qquad \Gamma \vdash t :\ \textbf{sequence}\,(T) \qquad \Gamma \vdash t_0 : T}{\Gamma \vdash \textbf{reduce}\,(f, t_0, t) : T}$$

- **T-SequenceReduceTail** (fold right)

$$\frac{\Gamma \vdash f : T \rightarrow T \qquad \Gamma \vdash t :\ \textbf{sequence}\,(T) \qquad \Gamma \vdash t_0 : T}{\Gamma \vdash \textbf{reduce_tail}\,(f, t_0, t) : T}$$

- **T-SequenceZip**

$$\frac{\Gamma \vdash t_1 :\ \textbf{sequence}\,(T_1) \qquad \Gamma \vdash t_2 :\ \textbf{sequence}\,(T_2) \qquad \Gamma \vdash f : T_1 \rightarrow T_2 \rightarrow T_r}{\Gamma \vdash \textbf{zip}\,(f, t_1, t_2) :\ \textbf{sequence}\,(T_r)}$$

- **T-SequenceReplace**

$$\frac{\Gamma \vdash t :\ \textbf{sequence}\,(T)\ \Gamma \vdash t_{nat} :\ \textbf{natural}\ \ \Gamma \vdash t_v : T}{\Gamma \vdash \textbf{replace}\,(t, t_{nat}, t_v :\ \textbf{sequence}\,(T)}$$

A.17 Bitvector Operations

A bitvector is simply a sequence of bits; thus common operations over bitvectors are provided.

- **T-BitvectorPadl**

$$\frac{\Gamma \vdash t: \textbf{ bitvector } \Gamma \vdash t_{length}: \textbf{ natural } \Gamma \vdash t_{fill}: \textbf{ bit}}{\Gamma \vdash \textbf{ padl}\,(t, t_{length}, t_{fill}): \textbf{ bitvector}}$$

- **T-BitvectorPadr**

$$\frac{\Gamma \vdash t: \textbf{ bitvector } \Gamma \vdash t_{length}: \textbf{ natural } \Gamma \vdash t_{fill}: \textbf{ bit}}{\Gamma \vdash \textbf{ padr}\,(t, t_{length}, t_{fill}): \textbf{ bitvector}}$$

- **T-Bitvector2Natural**

$$\frac{\Gamma \vdash t: \textbf{ bitvector}}{\Gamma \vdash \textbf{ bv2nat}\,(t): \textbf{ natural}}$$

- **T-Natural2Bitvector**

$$\frac{\Gamma \vdash t_n: \textbf{ natural}}{\Gamma \vdash \textbf{ bv2nat}\,(t_n): \textbf{ natural}}$$

- **T-BitvectorOps**

$$Let\, op = \textbf{ twos}|\textbf{lshl}|\textbf{lshr}|\textbf{ashl}|\textbf{ashr}|\textbf{rotl}|\textbf{rotr}|\textbf{not}$$

$$\frac{\Gamma \vdash t: \textbf{ bitvector}}{\Gamma \vdash op(t): \textbf{ bitvector}}$$

- **T-BitvectorSext**

$$\frac{\Gamma \vdash t: \textbf{ bitvector } \Gamma \vdash t_n: \textbf{ natural}}{\Gamma \vdash \textbf{ sext}\,(t, t_n): \textbf{ bitvector}}$$

- **T-WordFormer**

$$\frac{\Gamma \vdash t_{nat}:\ \textbf{natural}}{\Gamma \vdash\ \textbf{word}\ (t_{nat}):\ \textbf{word}\,(t_{nat})}$$

- **T-BitvectorBooleanOperations**

$$Let\,op\ =\ \textbf{and}|\textbf{or}|\textbf{nand}|\textbf{nor}|\textbf{xor}|\textbf{xnor}$$

$$\frac{\Gamma \vdash t_1, t_2:\ \textbf{bitvector}\ \Gamma \vdash t_1.length == t_2.length == n}{\Gamma \vdash t_1\,op\,t_2:\ \textbf{bitvector}\,with\,length\,n}$$

A.18 Mathematical Operators

Rosetta provides many built-in mathematical operators. They are not surprising in their semantics, but we nonetheless provide the related typing rules. Given the rich set of numerical types in Rosetta, there is a lot of ad-hoc polymorphism in the typing behavior of an operation such as addition. We could simply claim a type for addition such as `+::complex→complex→complex`, but this loses a lot of information. We'd much prefer `4+5` to be of type `posint`, instead of `complex`. And some operations such as division are not even conveniently closed for a type representation such as `(/)::a→a→a` for `a∈{natural, posint, negint, integer, rational, real, posreal, negreal, imaginary, complex}`, which we could try to reason for an operation like addition.

A.19 Addition

Let the relation R be {

(complex,complex,complex),	(imaginary,real,complex),	(imaginary,imaginary,imaginary),
(real,real,real),	(posreal,posreal,posreal),	(negreal,negreal,negreal),
(rational,rational,rational),	(integer,integer,integer),	(posint,natural,posint),
(posint,posint,posint),	(negint,negint,negint),	(natural,natural,natural)

}. Then:

T-Addition

$$\frac{(A,B,C) \in R\ or\ (B,A,C) \in R\ \ \Gamma \vdash_{\tau} x : A\ \ \Gamma \vdash_{\tau} y : B}{\Gamma \vdash_{\tau}\ \ x + y : C}$$

A.20 Subtraction

Let the relation R be defined as {

(complex,complex,complex)	(imaginary,imaginary,imaginary)	(real,real,real)
(posreal,negreal,posreal)	(negreal,posreal,negreal)	(posint,negreal,posreal)
(natural,negreal,posreal)	(negreal,posint,negreal)	(posreal,negint,posreal)
(negreal,natural,negreal)	(rational,rational,rational)	(integer,integer,integer)
(posint,negint,posint)	(negint,posint,negint)	(natural,negint,posint)
(negint,natural,negint)		

}. Then:

T-Subtraction

$$\frac{(A,B,C) \in R \quad \Gamma \vdash_\tau x : A \quad \Gamma \vdash_\tau y : B}{\Gamma \vdash_\tau \quad x - y : C}$$

A.21 Multiplication

Let the relation R be defined as: {

(complex,complex,complex)	(imaginary,imaginary,imaginary)	(imaginary,real,imaginary)
(real,real,real)	(posreal,posreal,posreal)	(posreal,negreal,negreal)
(negreal,negreal,posreal)	(posreal,posint,posreal)	(posreal,negint,negreal)
(negreal,posint,negreal)	(negreal,negint,posreal)	(rational,rational,rational)
(integer,integer,integer)	(posint,posint,posint)	(posint,negint,negint)
(negint,negint,posint)	(natural,natural,natural)	(bit,bit,bit)

}. Then:

T-Multiplication

$$\frac{(A,B,C) \in R \quad \Gamma \vdash_\tau x : A \quad \Gamma \vdash_\tau y : B}{\Gamma \vdash_\tau \quad x * y : C}$$

A.22 Division

Let the relation R be defined as: {

(complex,complex,complex)	(imaginary,imaginary,imaginary)	(imaginary,real,imaginary)
(real,imaginary,imaginary)	(real,real,real	(posreal,posreal,posreal)
(posreal,negreal,negreal)	(negreal,posreal,negreal)	(negreal,negreal,posreal)
(posreal,posint,posreal)	(posreal,negint,negreal)	(negreal,posint,negreal)
(negreal,negint,posreal)	(posint,posreal,posreal)	(posint,negreal,negreal)
(negint,posreal,negreal)	(negint,negreal,posreal)	(posreal,natural,posreal)
(negreal,natural,negreal)	rational,rational,rational)	(integer,bit, integer)

}. Then:

T-Division

$$\frac{(A,B,C) \in R \quad \Gamma \vdash_\tau x : A \quad \Gamma \vdash_\tau y : B}{\Gamma \vdash_\tau \quad x/y : C}$$

A.23 Exponentiation

Let the relation R be defined as: {

(complex,complex,complex)	(imaginary,imaginary,real)	(real,real,real)
(real,negreal,posreal)	(posreal,real,posreal)	(negreal,posreal,negreal)
(negreal,negreal,negreal)	(negreal,posint,negreal)	(negreal,negint,negreal)
(integer,integer,integer)	(posint,real,posreal)	(posint,posreal,posreal)
(posint,negreal,posreal)	(posint,rational,posreal)	(posint,inetger,posreal)
(posint,posint,posint)	(posint,negint,posreal)	(posint,natural,posint)
(negint,negreal,posreal)	(negint,posint,integer)	(negint,negint,posreal)
(negint,natural,integer)	(natural,real,posreal)	(natural,posreal,posreal)
(natural,negreal,posreal)	(natural,rational,posreal)	(natural,integer,posreal)
(natural,posint,natural)	(natural,negint,posreal)	(natural,natural,natural)
(bit,real,bit)		

}. Then:

T-Power

$$\frac{(A,B,C) \in R \quad \Gamma \vdash_\tau x : A \quad \Gamma \vdash_\tau y : B}{\Gamma \vdash_\tau \quad x\hat{\ } y : C}$$

A.24 Exponentials

- **T-ExpComplex**

$$\frac{\Gamma \vdash t <: \textbf{complex}}{\Gamma \vdash e^t : \textbf{complex}}$$

- **T-ExpReal**

$$\frac{\Gamma \vdash t <: \textbf{real}}{\Gamma \vdash e^t : \textbf{posreal}}$$

A.25 Negation and Identity for Numbers

Let the relation R be defined as {

(complex,complex),	(imaginary,imaginary),	(real,real),
(posreal,negreal),	(negreal,posreal),	(rational,rational),
(integer,integer),	(posint,negint),	(negint,posint),
(natural,integer),	(bit,integer)	

}. Then:

T-Negation

$$\frac{(A,B) \in R \;\; \Gamma \vdash_\tau x : A \;\; \Gamma \vdash_\tau y : B}{\Gamma \vdash_\tau -x : B}$$

- **T-NumberIdentity**

$$\frac{\Gamma \vdash t : T \;\; \Gamma \vdash T <: \textbf{complex}}{\Gamma \vdash +t : T}$$

A.26 Bit and Boolean Operators

$$Let\, op = \textbf{and}|\textbf{or}|\textbf{nand}|\textbf{nor}|\textbf{xor}|\textbf{xnor}\,|{=}{>}|{<}{=}|\,\textbf{implies}$$

- **T-BinaryBitOps**

$$\frac{\Gamma \vdash t_1 : \textbf{bit} \;\; \Gamma \vdash t_2 : \textbf{bit}}{\Gamma \vdash t_1\, op\, t_2 : \textbf{bit}}$$

- **T-BinaryBooleanOps**

$$\frac{\Gamma \vdash t_1 : \textbf{boolean} \;\; \Gamma \vdash t_2 : \textbf{boolean}}{\Gamma \vdash t_1 \, op \, t_2 : \textbf{boolean}}$$

- **T-BitNot**

$$\frac{\Gamma \vdash t : \textbf{bit}}{\Gamma \vdash \textbf{not}\, t : \textbf{bit}}$$

- **T-BooleanNot**

$$\frac{\Gamma \vdash t : \textbf{boolean}}{\Gamma \vdash \textbf{not}\, t : \textbf{boolean}}$$

- **T-Bit2Boolean**

$$\frac{\Gamma \vdash t : \textbf{bit}}{\Gamma \vdash \%t : \textbf{boolean}}$$

- **T-Boolean2Bit**

$$\frac{\Gamma \vdash t : \textbf{boolean}}{\Gamma \vdash \%t : \textbf{bit}}$$

A.27 Functions over Complex Numbers

- **T-Complex-Function**

  Let op = `re | im | abs | arg | conj`:

$$\frac{\Gamma \vdash t : \textbf{complex}}{\Gamma \vdash op(t) : \textbf{real}}$$

A.28 Trigonometry and More with Complex Numbers

- **T-Complex-Operation-Complex**

Let $op =$ `sin | cos | tan | arcsin | arccos | arctan`
`| sinh | cosh | tanh | arcsinh | arccosh | arctanh`
`| exp | sqrt | log | log10 | log2 :`

$$\frac{\Gamma \vdash t : \textbf{complex}}{\Gamma \vdash op(t) : \textbf{complex}}$$

- **T-Complex-Operation-Integer**

$$\textit{Let } op = \text{floor} \mid \text{ceiling} \mid \text{trunc} \mid \text{round} \mid \text{sgn}$$

$$\frac{\Gamma \vdash t : \textbf{complex}}{\Gamma \vdash op(t) : \textbf{integer}}$$

A.29 Mathematical Constants

- **T-j**

$$\frac{}{\Gamma \vdash j : \textbf{imaginary}}$$

- **T-e**

$$\frac{}{\Gamma \vdash e : \textbf{real}}$$

- **T-pi**

$$\frac{}{\Gamma \vdash pi : \textbf{real}}$$

A.30 Real, Imaginary Math Ops

- **T-MinReal**

$$\frac{\Gamma \vdash t_1 : \textbf{ real } \; \Gamma \vdash t_2 : \textbf{ real }}{\Gamma \vdash t_1 \, \textbf{min} \, t_2 : \textbf{ real }}$$

- **T-MinImaginary**

$$\frac{\Gamma \vdash t_1 : \textbf{ imaginary } \; \Gamma \vdash t_2 : \textbf{ imaginary }}{\Gamma \vdash t_1 \, \textbf{min} \, t_2 : \textbf{ imaginary }}$$

- **T-MaxReal**

$$\frac{\Gamma \vdash t_1 : \textbf{ real } \; \Gamma \vdash t_2 : \textbf{ real }}{\Gamma \vdash t_1 \, \textbf{max} \, t_2 : \textbf{ real }}$$

- **T-MaxImaginary**

$$\frac{\Gamma \vdash t_1 : \textbf{ imaginary } \; \Gamma \vdash t_2 : \textbf{ imaginary }}{\Gamma \vdash t_1 \, \textbf{max} \, t_2 : \textbf{ imaginary }}$$

- **T-RelationReal**

$$Let \; op = < \mid =< \mid > \mid >=$$

$$\frac{\Gamma \vdash t_1 : \textbf{ real } \; \Gamma \vdash t_2 : \textbf{ real }}{\Gamma \vdash t_1 \, op \, t_2 : \textbf{ real }}$$

- **T-RelationImaginary**

$$Let \; op = < \mid =< \mid > \mid >=$$

$$\frac{\Gamma \vdash t_1 : \textbf{ imaginary } \; \Gamma \vdash t_2 : \textbf{ imaginary }}{\Gamma \vdash t_1 \, op \, t_2 : \textbf{ imaginary }}$$

A.31 Rational Number Operations

- **T-DenominatorRational**

$$\frac{\Gamma \vdash t : \textbf{rational}}{\Gamma \vdash \mathrm{den}(t) : \textbf{integer}}$$

- **T-NumeratorRational**

$$\frac{\Gamma \vdash t : \textbf{rational}}{\Gamma \vdash \mathrm{num}(t) : \textbf{integer}}$$

- **T-MinRational**

$$\frac{\Gamma \vdash t_1 : \textbf{rational} \;\; \Gamma \vdash t_2 : \textbf{rational}}{\Gamma \vdash \min(t_1, t_2) : \textbf{rational}}$$

- **T-MaxRational**

$$\frac{\Gamma \vdash t_1 : \textbf{rational} \;\; \Gamma \vdash t_2 : \textbf{rational}}{\Gamma \vdash \max(t_1, t_2) : \textbf{rational}}$$

A.32 Integer and Natural Number Operations

- **T-IntegerTruncationDivision**

$$\frac{\Gamma \vdash t_1 : \textbf{integer} \;\; \Gamma \vdash t_2 : \textbf{integer}}{\Gamma \vdash t_1 \mathrm{div} t_2 : \textbf{integer}}$$

- **T-IntegerRemainderDivision**

$$\frac{\Gamma \vdash t_1 : \textbf{integer} \;\; \Gamma \vdash t_2 : \textbf{integer}}{\Gamma \vdash t_1 \mathrm{rem} t_2 : \textbf{integer}}$$

- **T-IntegerModularDivision**

$$\frac{\Gamma \vdash t_1 : \textbf{integer} \;\; \Gamma \vdash t_2 : \textbf{integer}}{\Gamma \vdash t_1 \mathrm{mod} t_2 : \textbf{integer}}$$

- **T-ClosedNaturalOperations**

$$Let\, op \, = \, +|*|\wedge|\mathrm{div}|\mathrm{rem}|\mathrm{mod}$$

$$\frac{\Gamma \vdash t_1 : \textbf{natural} \;\; \Gamma \vdash t_2 : \textbf{natural}}{\Gamma \vdash t_1 \, op \, t_2 : \textbf{natural}}$$

A.33 Character Operations

- **T-Char2Unicode(Ord)**

$$\frac{\Gamma \vdash t : \textbf{character}}{\Gamma \vdash \textbf{ord}\,(t) : \textbf{natural}}$$

- **T-Unicode2Char(Char)**

$$\frac{\Gamma \vdash t : \textbf{natural}}{\Gamma \vdash \textbf{char}\,(t) : \textbf{character}}$$

- **T-CharacterRelations**

$$Let\ op\ =<|=<|>=|>$$

$$\frac{\Gamma \vdash t_1 : \textbf{character} \;\; \Gamma \vdash t_2 : \textbf{character}}{\Gamma \vdash t_1 \, op \, t_2 : \textbf{boolean}}$$

- **T-UpperCase**

$$\frac{\Gamma \vdash t : \textbf{character}}{\Gamma \vdash \mathrm{uc}(t) : \textbf{character}}$$

- **T-LowerCase**

$$\frac{\Gamma \vdash t : \textbf{character}}{\Gamma \vdash \mathrm{dc}(t) : \textbf{character}}$$

A.34 Top and Bottom Literals

- **T-Top**

$$\frac{}{\Gamma \vdash \textbf{top} : top}$$

- **T-Bottom**

$$\frac{}{\Gamma \vdash \textbf{bottom} : bottom}$$

References

[1] The Glorious Glasgow Haskell Compiler. URL `http://haskell.org/ghc/`.

[2] The Eclipse Project. URL `http://www.eclipse.org/`.

[3] The Epigram Project. URL `http://www.e-pig.org/`.

[4] The SAL Language Manual. Technical report, 2003.

[5] P. Alexander. *System-Level Design with Rosetta*. Morgan Kaufmann Publishers, Inc., 2006.

[6] P. Alexander, D. Barton, and C. Kong. *Rosetta Usage Guide*. The University of Kansas / ITTC, 2335 Irving Hill Rd, Lawrence, KS, 2000.

[7] A. Avron, F. Honsell, and I. Mason. An Overview of the Edinburgh Logical Framework. In *In Current Trends in Hardware Verification and Automated Theorem Proving, G. Birtwistle*, pages 323–240. Springer-Verlag, 1989.

[8] H. Barendregt. Introduction to Generalized Type Systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[9] H. Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[10] G. Barthe. Type-checking Injective Pure Type Systems. *J. Funct. Program.*, 9(6):675–698, 1999. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796899003573.

[11] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[12] D. Clément, T. Despeyroux, G. Kahn, and J. Despeyroux. A Simple Applicative Language: mini-ML. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 13–27, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: http://doi.acm.org/10.1145/319838.319847.

[13] D. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, 1972.

[14] T. Coquand and G. Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2-3):95–120, 1988. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/0890-5401(88)90005-3.

[15] C. Cornes, J. Courant, J. C. Filliâtre, G. Huet, P. Manoury, C. P. Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual*. Institut National de Recherche en Informatique et en Automatique, 1995.

[16] L. Damas and R. Milner. Principal Type-schemes for Functional Programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM. ISBN 0-89791-065-6. doi: http://doi.acm.org/10.1145/582153.582176.

[17] David Aspinall. Subtyping with Singleton Types. In *Eighth International Workshop on Computer Science Logic*, pages 1–15. Springer-Verlag, 1995.

[18] I. S. Diatchki. Presburger: Cooper's Decision Procedure for Presburger Arithmetic. `http://hackage.haskell.org/package/presburger-0.3`, April 2009.

[19] J.-Y. Girard. Interprétation fonctionnelle et Élimination des coupures de l'arithmétique d'ordre supérieur. In J. Fenstad, editor, *Summary in Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, Université Paris VII, 1972. North-Holland.

[20] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40:194–204, 1987.

[21] Herbert Enderton. *A Mathematical Introduction to Logic*. Academic Press, Boston, MA, 2nd edition, 2001.

[22] J. Hickey, A. Nogin, R. L. Constable, B. E. Aydemir, E. Barzilay, Y. Bryukhov, R. Eaton, A. Granicz, A. Kopylov, C. Kreitz, V. N. Krupski, L. Lorigo, S. Schmitt, C. Witty, and X. Yu. MetaPRL - A Modular Logical Environment, 2003.

[23] P. Jackson. *The Nuprl Proof Development System, Version 4.1: Reference Manual and User's Guide*. Cornell University. Department of Computer Science, 1993.

[24] Ken Arnold, Tim Lindholm, Frank Yellin, The Java Team, Mary Campione, Kathy Walrath, Patrick Chan, Rosanna Lee, Jonni Kanerva, James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification - Second Edition, 2000.

[25] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The Undecidability of the Semi-Unification Problem. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 468–476, New York, NY, USA, 1990. ACM. ISBN 0-89791-361-2. doi: http://doi.acm.org/10.1145/100216.100279.

[26] O. Lee and K. Yi. A Generalized Let-Polymorphic Type Inference Algorithm. Technical report, Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, 2000.

[27] S. Liang, P. Hudak, and M. Jones. Monad Transformers and Modular Interpreters. In ACM, editor, *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[28] Mark Snyder and Perry Alexander. Monad Factory: Type-Indexed Monads. In *Post-Proceedings of Trends in Functional Programming*, pages 106–120, May 2010.

[29] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[30] Mojżesz Presburger. Über die Vollstndigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. pages 92–101, 1929.

[31] W. Naraschewski and T. Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23:3–4, 1999.

[32] D. Nazareth and T. Nipkow. Formal Verification of Algorithm W: The Monomorphic Case, 1996.

[33] Nicolas Frisby. Scopeless Abstract Syntax, 2009.

[34] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[35] V. S. Owre, S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-calvert. PVS Language Reference.

[36] S. Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. 2003.

[37] B. C. Pierce. Bounded Quantification is Undecidable. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 305–315, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: http://doi.acm.org/10.1145/143165.143228.

[38] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[39] B. C. Pierce and D. N. Turner. Local Type Inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/345099.345100.

[40] R. Pollack. The Theory of LEGO - A Proof Checker for the Extended Calculus of Constructions. Technical report, 1994.

[41] J. C. Reynolds. Towards a Theory of Type Structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag. ISBN 3-540-06859-7.

[42] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321250.321253.

[43] J. Roorda and J. Jeuring. Pure Type Systems for Functional Programming (Extended Abstract).

[44] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-calvert. PVS Prover Guide - Version 2.2.

[45] M. Snyder, N. Frisby, G. Kimmell, and P. Alexander. Writing Composable Software with InterpreterLib. In *SC '09: Proceedings of the 8th International Conference on Software Composition*, pages 160–176, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02654-6.

[46] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System F with type equality coercions, 2007.

[47] Thorsten Altenkirch, Conor McBride, and James McKinna. Why Dependent Types Matter. In *In preparation, http://www.e-pig.org/downloads/ydtm.pdf*, 2005.

[48] J. B. Wells. Typability and Type Checking in System F Are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1998.

[49] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-948472-8.