Design and Implementation of an Optionally-Typed Functional Programming Language

by

Patrick S. Li

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Koushik Sen, Chair
Adjunct Professor Jonathan Bachrach
Professor George Necula
Professor Sara McMains

Fall 2017

Design and Implementation of an Optionally-Typed Functional Programming Language

Abstract

Design and Implementation of an Optionally-Typed Functional Programming Language

by

Patrick S. Li

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Koushik Sen, Chair

This thesis describes the motivation, design, and implementation of L.B. Stanza, an optionally-typed functional programming language aimed at helping programmers tackle the complexity of architecting large programs and increasing their productivity across the entire software development life cycle. Its design objectives arose out of our own frustrations with writing software, and we built Stanza to be a practical general-purpose language that resolves the problems we encounter in our daily work.

Stanza was designed to write programs of appreciable complexity; where development time is spent primarily on managing this complexity; and where better tools for managing complexity can significantly improve programmer productivity. In our experience with writing software, there are five primary activities that occupied the majority of our time: finding and fixing errors, coordinating multiple algorithms, architecting and maintaining a clean software infrastructure, minimizing and maintaining redundancies in code, and optimizing for performance.

Stanza consists of five orthogonal subsystems to address each of the previous issues: the optional type system, the targetable coroutine system, the multimethod object system, the programmatic macro system, and the LoStanza sublanguage. Each subsystem is responsible for a separate facet of software development – error detection, control flow and concurrency, architecture, syntactic abstraction, and low-level control – and work in concert to form a small but expressive language.

The optional type system allows programmers to transition freely between the dynamically-typed and statically-typed paradigms, and thus offers both productivity and flexibility in addition to early error detection capabilities. The targetable coroutine system allows for programmers to easily coordinate multiple algorithms, and also acts as a foundational control-flow operator. The multimethod object system is a novel *class-less* object system that unifies the functional programming with the object-oriented programming styles. The programmatic macro system allows for arbitrary syntactic abstractions in the style of Lisp while still retaining a natural and familiar syntax. Finally, the LoStanza sublanguage provides

programmers both direct access to low-level hardware details as well as the ability to easily communicate and interoperate with high-level code.

Stanza has been successfully applied to the development of a number of significant projects at U.C. Berkeley, including the development of a digital hardware design language, a just-in-time-compiled teaching language, as well as a full native-code optimizing compiler. By the later stages of the Stanza project, Stanza's productivity allowed for the optimizing compiler to be developed by a single student over the course of four months.

This thesis describes the language design goals we have identified as being important for the class of programs we write, and how these goals are realized in the design of Stanza and its subsystems. Stanza's design is compared against that of other languages, and we point out the advantages and disadvantages of each. We then explain the overall implementation of the Stanza compiler and runtime system; share our personal experiences with teaching and programming in Stanza; and end with a summary of future work.

To Mama and Baba,

for their simple unwavering belief in me.

Contents

List of Figures

Acknowledgments

Stanza was a deeply personal work of love for me, and I am profoundly grateful to all the following people that have supported me on this long journey.

My advisors Jonathan Bachrach, and George Necula allowed my imagination free-reign and provided me a blissful six years at Berkeley as a graduate student. George, my co-advisor, with his uncompromising taste in clarity and precision helped me develop the theoretical foundations of Stanza's type system, and passed onto me the appreciation of using clear notation to hone my own thinking. My relationship with Jonathan, my main advisor, was quite unlike all other advisor-advisee relationships. Instead of picking on minute details, Jonathan would ask me to reconsider wholesale the design of an entire subsystem; instead of design or code reviews he would immediately put the latest Stanza feature to use and send me the screenshots of his latest creation; and instead of asking me to focus and narrow my grand ambitions, he would dare me to dream about possibilities that I was too modest to consider. Instead of a teacher and student, we were two fanatic admirers of programming languages, and we built our working relationship and our friendship upon our mutual passion for this field. Thank you Jonathan for sharing with me your energy, enthusiasm, imagination, and boldness.

The early beginnings of Stanza were the most nerve-wracking. I spent many long hours sitting at my desk, with random scribbles on a page, lost in thought over seemingly nonsensical questions such as: "What exactly *is* an object?" and "What does it mean to *call* a function?". Kurt Keutzer, my entering advisor at Berkeley, was understanding and patient with me during this phase – a phase I retroactively termed "fundamentals hell" – and provided me one and a half years of complete and utter freedom to design the core of Stanza.

Adam Izraelevitz and I worked together on the implementation and specification of FIRRTL. Adam was a joy to work with and patiently and methodically learned to code in a brand new language – which at the time, didn't even have any documentation – and bravely championed its use in the rest of our laboratory.

James Martin, one of my closest friends, spent an enormous and equal amount of time both discussing Stanza with me and helping me put Stanza out of my mind. For the innumerable hours we spent wandering around Berkeley and San Francisco discussing Stanza's constructs, Haskell's type system, the merits of subtyping versus type classes, constructive logic, category theory, the latest movie and cartoon, and Joe Hisaishi's latest albums, among too many other topics to remember, thank you James.

After the public release of Stanza, Duncan Haldane, Austin Buchan, and Jonathan Bachrach were bold enough to co-found a company with me and adopt Stanza as our primary implementation language. For a young startup to be built around a brand new programming language is nearly unheard of. Thank you Duncan, Austin, and Jonathan, for your faith in Stanza, and in me.

Duncan Haldane, Jenny Huang, Johann Schleier-Smith, Marten Lohstroh, Austin Buchan, Danny Tang, Sumukh Sridhara, Ryan Orendorff, Howard Mao, Michael Driscoll, Cindy Rubio Gonzalez, David Biancolin, Colin Schmidt, Donggyu Kim, Richard Lin, Martin Maas, Chick Markley, Jim Lawson, Jack Koenig, Albert Magyar, and Palmer Dabbelt were attendees of the Stanza bootcamps and provided me with lots of useful early feedback on how to teach Stanza. Wontae Choi was very helpful during early discussions about Stanza's type system. Emina Torlak shared her expertise on constraint solvers as applied to solving subtyping and unification equations. Andy Keep offered very useful optimization advice for Stanza's compiler. Oleg Kiselyov, Koushik Sen, and Ras Bodik gave feedback and advice on Stanza's coroutine system. Michael Driscoll gave me solid advice on marketing and promoting Stanza.

Throughout the whole process, I was fortunate and grateful to be surrounded by the encouragement and love of my family. Three times every year, I fly back home to Calgary after four months of wrestling with type systems and compilers, and Luca and Emmy, my brother and sister, remind me about all the small things in life that I've put aside – like playing video games underneath the blanket on weekend mornings, and watching old movies (that we've already watched before) at night after supper, and reading and working beside each other by the dining room table. Thank you Luca and Emmy for keeping me a kid.

Lastly and most importantly, I want to thank my parents. Quite honestly, completing Stanza to my liking was a challenge for me. It wasn't clear that I would. There is this maxim that says that a Ph.D. is more similar to a marathon than to a sprint, that it is a better measure of endurance than it is of brilliance. And if that is true, then completing this dissertation is as much my parents' accomplishment as it is mine. Thank you Mama and Baba and for all the many nights you've picked me up and encouraged me to try again when my latest idea failed; for cheering me up each time I felt myself fade; and, throughout my whole life, for your simple unwavering belief in me.

Chapter 1

Introduction

L.B. Stanza (or Stanza for short) is an optionally-typed functional programming language designed to help programmers tackle the complexity of architecting large programs and significantly increase the productivity of application programmers across the entire software development life cycle. Towards this purpose, it features:

1. a flexible multimethod-based object system that unifies functional with object-oriented programming,
2. an optional type system that unifies dynamic- and static-typing,
3. a powerful coroutine system that acts as both a concurrency and foundational control flow operator,
4. a programmatic macro system that allows for natural syntax, and
5. a systems sublanguage for low-level hardware control.

These five orthogonal subsystems work in concert to form an expressive language that remains small and easy-to-learn.

From the onset, Stanza was designed to be a practical general-purpose programming language, suitable for writing any application that runs above the operating system layer. Only applications that have hard real-time constraints or that must run on severely resource-constrained platforms lie beyond the scope of the language. We have successfully used the language at our own lab to write a suite of substantial software, and it has also been used by others for a small number of projects beyond our university. To our knowledge, Stanza has so far been used to write compilers, debugging tools, computer games, virtual machines, image processing software, constraint solvers, hardware synthesis tools, 3D geometric design software, and web applications.

Stanza's overarching design objectives arose out of our own frustrations with writing software, and the decision to design our own programming language was made after attempting to adopt an existing language and finding none suitable for our purposes. For the class of programs we develop, the majority of time is spent on effectively managing *complexity*. Our programming challenges are rarely due to intricate algorithm design – most algorithms and datastructures in our programs are conceptually straightforward – but rather the sheer amount of detail in a large program. We have gradually learned, over time, that the most important factor affecting our success at managing these details is our choice of software architecture. In our experience, there are orders-of-magnitude productivity differences between working on a well-architected project versus a poorly-architected one – but no language provided us the required guidance or flexibility for designing a *good* architecture.

For our purposes, existing languages provide either next to no support for software architecture; or they impose an overly rigid structure that freezes the architecture too early into development – forcing upon programmers the near-impossible task of getting it right the first time. We wanted a simple language with constructs that inherently keeps the structure fluid and helps us *incrementally* arrive at a good software architecture.

This thesis describes the language design goals we have identified as being important for the class of programs we write, and how these goals are realized in the design of Stanza and its subsystems. Stanza's design is compared against that of other languages, and we point out the advantages and disadvantages of each. We then explain the overall implementation of the Stanza compiler and runtime system; share our personal experiences with teaching and programming in Stanza; and end with a summary of ongoing work.

Chapter 2

High-Level Design

Before discussing Stanza's design goals, it is first necessary to identify the class of programs we intend to write. The programs we are interested in share the following characteristics:

1. They run above an existing operating system. Simple interfaces are assumed to be provided for accessing the file and memory systems, and interacting with peripherals. Although Stanza can be used to write kernel-level software, such as device drivers, it is not intended to offer significant benefits over existing systems languages for this purpose.

2. The applications do not need to run on a resource-constrained platform. We require that the platform have at least 128 MB of memory, and either a 32-bit or 64-bit processor. We do not intend to program embedded systems using Stanza.

3. The application does not have any hard real-time constraints. Stanza's design presumes the existence of a garbage collector and consequently does not provide any firm guarantees on execution time. However, by incorporating an incremental garbage collector, Stanza *can* be suitable for applications with soft timing constraints, where specified execution times are strongly preferred but not required for correctness.

4. The program is non-trivial. We expect project timelines to be measured in weeks instead of hours. It is not our intention to use Stanza for writing short one-off scripts.

5. A substantial portion of development time is spent on ensuring correctness of the implementation, instead of on activities such as algorithm design, conducting user studies, or fine-tuning to achieve peak machine performance.

In short, we are interested in programs of appreciable complexity; where development time is spent primarily on managing this complexity; and where better tools for managing complexity can significantly improve programmer productivity. Programs that fall outside this domain are already adequately served by existing languages, or better served by other directions of language research.

Item 1 restricts Stanza's focus to application instead of infrastructure development. Kernel-level software is best served by languages such as C [33], that emphasize a transparent mapping to machine instructions and offers fine control over hardware resources – for example, the ability to reserve specific registers for holding a certain value.

Item 2 restricts Stanza to smartphone, desktop, and server application development. Development for resource-constrained platforms is best served by languages such as C [33] or assembly, that allow programmers to optimize for code size and memory footprints.

Item 3 is a consequence of garbage collection. Automatic garbage collection significantly improves productivity and reduces opportunity for error but it does make Stanza inappropriate for applications that have strict upper-bounds on execution time. Programs that communicate with external devices with strict timing requirements are better written in languages such as C [33] or assembly that have a straightforward performance model.

Item 4 reflects our lack of interest in using Stanza for writing simple scripts. At this scale, the optimal choice of programming language is dependent mostly upon on the ecosystem and available tools than upon any features of the language itself. This domain is already adequately served by existing scripting languages.

Item 5 reflects the primary difficulty we face during development. We are interested in complex programs for which achieving correctness is the primary obstacle. In fact, the typical program of our interest is of sufficient complexity that full correctness is not practically achievable and is merely an ideal to strive towards. There may be language-centric solutions for other aspects of software development, such as languages to aid algorithm design, or languages to aid in performance tuning, but it is not our focus.

Examples of programs that satisfy the above criteria include:

- the majority of user-facing desktop and smartphone applications, such as document, video, sound, and graphics editing applications,
- business management software,
- computer games and 3D modeling software, save perhaps for the core of their rendering engines,
- programming tools such as compilers, interpreters, and debuggers,
- database management software,
- and server-side web-applications,

among many others. Though not all-encompassing, substantial classes of programs still remain in our domain of interest.

2.1 High-Level Objectives

We summarize the high-level objectives of the language as follows. Stanza should:

1. help programmers organize their code,
2. help programmers detect and fix errors,
3. automatically manage tedious details,
4. allow for complex tasks to be effectively subdivided into simpler ones,
5. allow for common tasks to be implemented in a reusable fashion,
6. allow for sophisticated operations to be expressed concisely,
7. allow programmers to work at the level of abstraction most suited for their problem,
8. encourage the writing of maintainable and extensible software,
9. execute efficiently on current hardware,
10. interoperate with the surrounding software ecosystem, and
11. be easy to learn.

Item 1: Code Organization

Programmers rarely view a code base as a single monolithic block of text. Instead, it is split into divisions, where each division is responsible for a different logical task. These divisions are, in turn, split into further subdivisions, each responsible for handling some narrower subset of the task – and so on and so forth. The language should provide the tools necessary to subdivide the overall program to address the following concerns:

1. *File Organization:* Stanza, like the vast majority of programming languages, requires code to be stored as a set of plain text files. It is highly beneficial for all the code comprising a single logical task to be concentrated in one location in the file structure – ideally in a single section within a source file. Multiple programmers working on a project can be assigned separate logical tasks by assigning each programmer a different file.

2. *Separation of Concerns:* Divisions should be kept, as much as possible, independent from each other. The interface between each division should be simple, well-specified, and presume as few implementation details as possible. This allows separate divisions to be developed in-parallel by different programmers, and minimizes the change in other divisions necessary to accommodate changes in one division.

Item 2: Error Detection

As mentioned, full correctness for the typical program of our interest is not practically achievable. The language should provide as much assistance as possible for finding and fixing errors, which accounts for a large portion of development time. This feature is made up of two parts: dynamic error detection – which detects errors during the execution of the program – and static error detection – which detects errors before executing the program.

When a program attempts to execute an invalid operation – such as a write past the end of an array – the language should halt *immediately*, give a detailed report on what operation was attempted, why it is invalid, and which line of code caused it. This information gives the programmer an accurate starting point for identifying the source of the problem. We call this feature dynamic error detection.

In certain cases, we can determine that a program is likely incorrect even before executing it. For example, attempting to call a function with three arguments when it requires only two is almost assuredly a mistake. The language should do its best to analyze the program and report these highly suspicious lines of code. We call this feature static error detection.

Item 3: Automatic Management of Tedious Details

Much of the tedium of programming arises from having to cope with the physical limits of the machine – like the limited amount of memory, the clock frequency of the processor, the number of machine registers, and the division of memory into stack and heap memory, to name a few. If the performance cost is reasonable, programmers would prefer for the language to automatically manage as many of these details as possible. This increases both the productivity of the programmer by reducing work, and the reliability of the software by eliminating potential sources of error.

Item 4: Subdivision of Complex Tasks

In order to carry out a complex task, a programmer subdivides the task into smaller subproblems, solves each in isolation, and then combines their solutions. An intelligent subdivision of a problem can greatly improve productivity and reliability, and indeed one of the measures of a programmer's skill is in his or her ability to find such a subdivision. To aid the programmer, the language should provide constructs that allow tasks to be viewed from different perspectives and divided along a multitude of axes.

Item 5: Reusable Implementations of Common Tasks

To reduce work, a language should allow a programmer to reuse the implementations of common tasks – which requires for these tasks to be implemented in sufficient generality to be widely applicable. For example, a function for computing the sum of all integers in an array is not as widely useful as a function able to compute the sum of all numbers, whether integers or real numbers, in an array. Even more useful would be a function able to compute the sum of all numbers in any datastructure with a sequential ordering.

Item 6: Concise Expression of Sophisticated Operations

As a driving design principle, we hold the philosophy that the ease-of-use of a programming language is directly dependent upon how closely it can mirror the structure of natural language. The ultimate purpose of a programming language is to direct a machine to complete computational tasks. And since every computational task of interest to human beings is first conceived and communicated in natural language, programming difficulty is determined by how straightforward it is to translate a task from natural language to the programming language.

In practice, we adhere to this design principle by continuously conceiving reasonable tasks that can be described clearly and unambiguously in natural language, and asking whether they can be translated straightforwardly to Stanza. *Computational tasks with concise and unambiguous descriptions in natural language should translate to equally concise descriptions in Stanza.*

Item 7: Appropriate Level of Abstraction

Level of abstraction is an important axis along which programmers subdivide problems. Just as instructions on how to host a birthday party rely upon but are kept separate from the instructions on how to bake a cake, how to drive to the toy store, and how to write an invitation card; the code for accomplishing the high-level objectives of a task rely upon but are kept separate from the code spelling out the low-level details of accomplishing each objective.

The language should allow for a problem to be divided along different levels of abstraction. If the standard libraries that accompany the language are too rudimentary, the language should allow programmers to easily build their own abstractions to suit their problem.

Item 8: Maintainability and Extensibility

For a typical program of our interest, neither full correctness nor completion is achieved in practice. An application is continuously undergoing modifications and additions as it matures.

Design for maintainability is fundamentally linked to design for extensibility. An ideal software architecture should allow for new extensions to be implemented separately, without disturbing the existing code base. In contrast, poorly architected software requires extensions to be implemented as disorganized modifications that are scattered amongst the existing code. Core algorithms and datastructures steadily balloon in size to accommodate new concerns until the essence of the algorithms become obscured and unintelligible.

The language should allow for programmers to build software that can be extended according to deliberately designed interfaces. If the software must be extended in a way that is unsupported, then the language should allow for *easy* restructuring of the code base to support the extension.

Item 9: Efficiency

For developing a given application, whether or not a programming language is suitable is often determined by its execution efficiency. It is common for applications to impose hard requirements on performance – the program must serve a specific number of customers, or display sixty frames per second, or smoothly manage gigabyte-sized files, etc. Increasing the performance of the language implementation widens the applications for which it is suitable. We are targeting applications for which the speed of untuned C [33] is sufficient.

Item 10: Interoperation with Software Ecosystem

A practical programming language must be able to interoperate with the rest of the software ecosystem: the operating system, and existing code and applications written in other programming languages. This requires the ability to execute code not originally written in Stanza and manipulate datastructures not originally created within Stanza.

Item 11: Ease of Learning

One of the most practical concerns of language design is in whether it can be easily learned by potential users. A number of practical factors affect ease-of-learning such as the availability of learning material, or whether the language is taught in educational institutions; but the degree to which the learning curve is affected by the fundamental design of the language cannot be underestimated. C++ [55] is notorious, for instance, for being a difficult language to learn, while Python [47], in contrast, is often taught in introductory programming courses.

We do not want programmers to require formal training in computer science, mathematics, or engineering to use Stanza effectively. Completing the tutorials that accompany the language documentation should be sufficient for developing the working understanding of type theory and semantics necessary to be a productive programmer.

2.2 A Minimal Lisp

We start the design by beginning with a minimal Lisp-like [32] language that allows users to declare and call functions, create primitive values, declare and reference values and variables, and includes a set of built-in functions for constructing common datastructures. Here is an example of a small function:

```
(defn f (x, y)
  ;Performing arithmetic
  (defval z (plus 20 (plus x y)))
  (println z)

  ;A variable for storing a string
  (defvar greeting "hello")
  (set greeting (string-join greeting " world"))

  ;Creating and initializing an array
  (defval a (make-array 10))
  (defn loop (i)
    (if (less? i 10)
      (begin
        (array-set a i "Hi")
        (loop (plus i 1)))
      false))
  (loop 0)

  ;Returned value
  z)
```

The syntax `(defn f (x, y) body ...)` declares a new function `f` that takes two arguments, `x` and `y`, and executes the forms comprising `body`. The syntax `(defval z value)` declares a new value, `z`, initialized to the result of evaluating `value`. The syntax `(plus a b)` calls the `plus` function with the arguments `a` and `b`. We assume that `plus`, `println`, `string-join`, `make-array`, `less?`, and `array-set`, are included as built-in functions for respectively adding two integers, printing to the screen, concatenating two strings, creating a new array, determining whether one integer is strictly less than another, and storing into an array. Semicolons denote that the rest of the line is a comment, and commas are treated equivalently to white space and are used solely for readability.

After printing `z`, we use the syntax `(defvar greeting "hello")` to declare the variable `greeting` and initialize it to the string `"hello"`. The next line then computes the concatenation of `greeting` and `" world"` and assigns the result back to `greeting`. The syntax `(set x value)` assigns the result of evaluating `value` to the variable named `x`.

Next we create an array, `a`, of length 10 and begin a loop to initialize each slot in the array to the value `"Hi"`. As in the Scheme language [56], we assume that tail-calls are optimized and perform looping through tail recursion. Within the `loop` function, we use the `if` form to choose whether to perform another iteration. The syntax `(if pred conseq alt)` first evaluates `pred` and then, depending on whether it evaluates to `true` or `false`, returns the result of either evaluating `conseq` or `alt`. The syntax `(begin forms ...)` evaluates the given forms sequentially and returns the result of evaluating the last one.

Functions are assumed to automatically return the result of the last form in its body. Thus `z` is returned as the result of evaluating the above function.

Properties of the Minimal Lisp

Though not shown above, we can assume that our minimal Lisp supports the following types of values: integers, floating-point numbers, strings, characters, arrays, lists, true, and false. There is literal syntax for creating some of these values, such as integers and strings as shown above. Other values, such as arrays, are constructed and manipulated using the built-in functions.

Though the language is minimal it has the following properties:

1. It is *automatically garbage collected.* The `string-join` and `make-array` functions return values of varying size, depending on the inputs they are given, and hence must allocate these values in heap memory. There are two design choices we could have made for handling deallocation of these values: either require the programmer to explicitly deallocate these values using built-in `free-string` and `free-array` functions, or automatically deallocate these values when they can no longer be referenced. We chose the latter solution.

   Manually managing memory deallocation is exceptionally tedious and error-prone for programmers; and automatic garbage collection is a well-researched and performant solution. However, it does prevent us from using the language for strict real-time applications without considerable further research into garbage collector implementation. Since that is not our target domain, we feel it is an acceptable tradeoff. This property partially accomplishes items 3 and 11 of our high-level objectives.

2. It is *high-level.* The language does not expose any details of the machine architecture. Details such as the number of machine registers, the amount of memory, how memory is divided into stack and heap memory, how to address a location in memory, and how strings and arrays are represented in memory, are all concealed from the programmer. This partially accomplishes items 3 and 11 of our high-level objectives.

3. It is *safe*. During execution, the language checks all operations to ensure they are valid before executing them. For instance, the `string-join` function checks that its two arguments are indeed strings; the `make-array` function checks that its argument is an integer and non-negative; and the `array-set` function checks that the index argument is within the bounds of the array. If a check fails, then the program aborts immediately with a detailed report on the cause and source of the error. All programs are guaranteed to either execute until completion or fail with an error report. This accomplishes item 2 of our high-level objectives.

   A safe language also greatly increases ease-of-learning. Finding and fixing errors occurring in an unsafe language is somewhat of a black art. Errors do not immediately halt the program and instead corrupt vital areas of memory that are later read from. A large part of learning how to debug programs written in unsafe languages is on how to accurately postulate the original cause of errors. Thus dynamic error detection also accomplishes item 11 of our high-level objectives.

4. There is minimal *static error detection.* The language, as presented, has little support for detecting errors before execution. There is only a simple syntax checking phase for ensuring that the program is syntactically well-formed and that variables are declared before they are referenced. The checker would detect, for instance, that `(defval x)` is missing an initializing value, but it would not detect the obvious error in the following code:

   ```
   (defval a 0)
   (array-set a 1 "Hi")
   ```

   It is clear that the call to `array-set` will fail as `a` does not refer to an array.

5. It has a *minimal syntax.* Following Lisp [32] tradition, the language is represented using *s-expressions.* The frontend of an s-expression-based language starts with an extremely simple lexer for converting the program text into a tree datastructure called an s-expression. The rules are simple: a sequence of characters (e.g. `abc`) represents a *symbol*; integers are sequences of digits (e.g. `256`); strings are surrounded by double-quotes (e.g. `"hello world"`); and lists are surrounded by parentheses (e.g. `(a b c)`).

   Because the lexer is so transparent, programmers typically understand the syntax of the language in terms of s-expressions instead of in terms of characters. For instance, the proper syntax for defining a value is a three-element list starting with the `defval` symbol, followed by the name as a symbol, followed by another s-expression representing the initializing value.

   The minimalism of the s-expression syntax contributes to ease-of-learning, item 11 of our objectives, but it is somewhat counteracted by its unfamiliarity. Programmers typically do not have experience with s-expression syntax; the syntax for arithmetic expressions is plainly different than standard mathematical notation; and the cluttering of parentheses is aesthetically off-putting.

   The s-expression syntax is commonly described as *homoiconic* which means informally that the program structure is similar to its surface syntax. This property will be discussed in more detail later after we introduce macros.

6. It supports *nested functions.* In the example above, the function `loop` is declared within the body of `f`. Additionally, the body of `loop` references the value `a` which is not declared directly in `loop` but in the enclosing function `f`.

   From a design perspective, support for nested functions is essentially necessitated by our decision to model loops using tail recursion – which would otherwise be unmanageably cumbersome. The elimination of separate looping constructs keeps the language small and contributes to item 11 of our objectives.

   Nested functions are also convenient for finely subdividing a large function into a set of small orthogonal tasks, which contributes to items 4 and 7 of our objectives. Consider the following implementation of selection sort for an array of integers:

```
;Sort an array of integers
(defn sort (xs)
  ;Swap element i with element j
  (defn swap (i j)
    (if (not (equal? i j))
      (begin
        (defval xi (array-get xs i))
        (defval xj (array-get xs j))
        (array-set xs i xj)
        (array-set xs j xi))
      false))

  ;Find the index of the minimum element
  ;between index b (inclusive) and e (exclusive)
  (defn minimum (b e)
    (defn loop (min-i, min, i)
      (if (less? i e)
        (if (less? (array-get xs i) min)
          (loop i (array-get xs i) (plus i 1))
          (loop min-i min (plus i 1)))
        min-i))
    (loop b (array-get xs b) (plus b 1)))

  ;Loop
  (defn loop (i)
    (if (less? i (minus (array-length xs) 1))
      (begin
        (swap i (minimum i (array-length xs)))
        (loop (plus i 1)))
      false))
  (loop 0))
```

`swap` and `minimum` are factored out as two independent tasks. `swap` takes two indices and swaps the value at the first index with the value at the second index. `minimum` takes a starting and ending index and returns the index of the smallest value between them. With these two tasks factored out, the main sorting algorithm can then be described concisely at a high-level of abstraction.

2.3 Supporting Natural Syntax

The syntax for the Lisp-like [32] language is simple but has two major flaws:

1. it is unfamiliar and is aesthetically unappealing to many programmers, and
2. some common idioms (such as `if` expressions with no alternate clauses, and loops) are awkward.

We will resolve both issues by introducing a *macro* system.

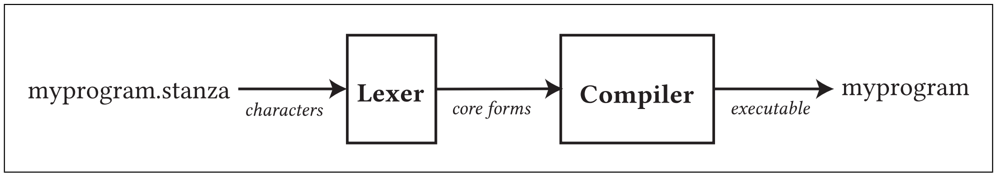

Figure 2.1: Simple Compilation Flow

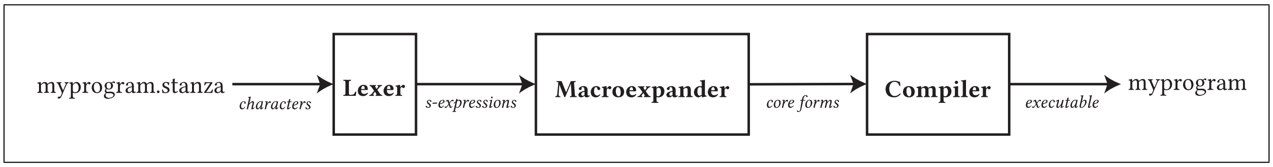

Figure 2.2: Macro Compilation Flow

As mentioned, an s-expression language starts with a simple lexer for converting the program text into a tree datastructure. Figure 2.1 shows a simplified flow of the language system. The program text is converted by the lexer into an s-expression, which the compiler then converts into a binary executable.

A *macroexpander* is an intermediate stage inserted between the lexer and the compiler that transforms the output of the lexer into a form that is acceptable to the compiler. Figure 2.2 shows the updated flow of the language system.

Let us suppose that we insert a macroexpander that performs the following transformations:

1. It provides a `when` shorthand for the `if` form such that:

   ```
   (when pred
     form1
     form2
     ...)
   ```

   is a shorthand for:

   ```
   (if pred
     (begin
       form1
       form2
       ...)
     false)
   ```

   This gives us a more concise syntax for `if` expressions without alternate clauses.

2. It provides a `for` shorthand for defining simple tail recursive functions such that:

   ```
   (for x in start to end do
     form1
   ```

```
  form2
  ...)
```

is a shorthand for:

```
(define loop (x end-index)
  (when (less? x end-index)
    form1
    form2
    ...
    (loop (plus x 1) end-index)))
(loop start end)
```

This gives us a more concise syntax for expressing a simple loop that iterates from some starting value to some ending value.

With these two transformations, our selection sort example can be expressed more naturally as:

```
;Sort an array of integers
(defn sort (xs)
  ;Swap element i with element j
  (defn swap (i j)
    (when (not (equal? i j))
      (defval xi (array-get xs i))
      (defval xj (array-get xs j))
      (array-set xs i xj)
      (array-set xs j xi)))

  ;Find the index of the minimum element
  ;between index b (inclusive) and e (exclusive)
  (defn minimum (b e)
    (defvar min-i b)
    (defvar min (array-get xs b))
    (for i in (plus b 1) to e do
      (when (less? (array-get xs i) min)
        (set min-i i)
        (set min (array-get xs i)))))

  ;Loop
  (for i in 0 to (minus (array-length xs) 1) do
    (swap i (minimum i (array-length xs)))))
```

Programmatic Macro Systems

The question now is how to specify the `when` and `for` transformations. Conceptually, a macro transformer is just a function that takes as input an s-expression and computes an output s-expression. Thus why not specify this function using our language itself? The following shows a possible implementation of the `when` macro transformer:

```
(defmacro (when forms)
  (defval pred (head forms))
  (defval body (tail forms))
  (defval conseq (cons ‘begin body))
  (cons ‘if (cons pred (cons conseq ‘(false)))))
```

The syntax `(defmacro (macro-name forms) body ...)` declares a new macro transformer that is evaluated during the macro expansion phase. Whenever the macroexpander encounters an s-expression of the form `(macro-name form1 form2 ...)`, it evaluates the transformer with `forms` bound to the list `(form1 form2 ...)` and substitutes the result in-place. The syntax `‘exp` denotes the literal s-expression `exp`. Thus `‘if` denotes the symbol `if`, and `‘(false)` denotes the single-element list containing `false`. The `head`, `tail`, and `cons` functions computes respectively the first element of a list, all elements after the first element of a list, and the result of attaching a new element to an existing list.

The next example shows the step-by-step behaviour of the macroexpander upon encountering the following form:

```
(when (less? (array-get xs i) min)
  (set min-i i)
  (set min (array-get xs i)))
```

Because the s-expression begins with `when`, the macroexpander will evaluate the `when` transformer in the following steps:

1. `forms` is bound to the list:

   ```
   ((less? (array-get xs i) min)
    (set min-i i)
    (set min (array-get xs i)))
   ```

2. `pred` is initialized to:

   ```
   (less? (array-get xs i) min)
   ```

3. `body` is initialized to:

   ```
   ((set min-i i)
    (set min (array-get xs i)))
   ```

4. `conseq` is computed to be:

   ```
   (begin
     (set min-i i)
     (set min (array-get xs i)))
   ```

5. Finally, the transformer returns:

   ```
   (if (less? (array-get xs i) min)
     (begin
       (set min-i i)
       (set min (array-get xs i)))
     false)
   ```

The value of having a homoiconic syntax is now clear. Since the macro writer is manipulating a datastructure that represents the syntax tree of the program, the program text must have a *straightforward* mapping to this datastructure.

A programmatic macro system accomplishes many of our high-level objectives:

1. It increases ease-of-learning (item 11) in two different ways. The first is that the language is made more similar to other popular languages and hence more familiar to programmers. The second is that it provides the language designer a method of including convenient constructs without introducing new semantics to the language. For example, the programmer can understand the `for` macro as just a syntactic shorthand for manually defining a tail-recursive function.

2. It allows common tasks to be implemented in a reusable way (item 5). As seen with the `for` and the `when` macro, not all tasks can be implemented as functions. Macros provides programmers with another tool for abstracting over common code patterns.

3. It allows users to define their own syntactic abstractions (item 7) to suit their application. In our example we recognized *looping from some starting value to some ending value* as a worthwhile abstraction to define a macro for. The `for` macro is suited for a broad range of applications but we can imagine macros that are more specialized in their purpose – for instance, a macro like the following for sending emails:

   ```
   (send-email
     (to "jonathan@berkeley.edu")
     (cc "patrick@berkeley.edu" "george@berkeley.edu")
     "Here is the revised draft of my thesis.")
   ```

   In the limit, a macro system can be used to define entire purpose-built languages designed to serve a niche domain – sometimes called *domain specific languages* (DSLs).

Decorated S-Expressions

With the addition of the `when` and `for` macro, the selection sort example is approaching how they would look in Python [47] or Ruby [22], but it is not quite there. Some aspects that stick out include:

1. the unfamiliar syntax for function calls,

2. the unfamiliar syntax for defining values and variables,

3. the cumbersome syntax for assigning to and retrieving from arrays,

4. and, notoriously, the abundance of parentheses.

To resolve these issues, we will *slightly* extend the lexer with some additional syntax. However, to keep the syntax homoiconic, it is important that these extensions do not obscure the mapping from the program text to the s-expression datastructure.

We will use the following strategy:

1. We will add a small number of shorthands that implicitly map to an equivalent s-expression.

2. We will add a structured-indentation feature to the lexer that will implicitly add parentheses around indented groups.

3. We will then convert these implicit s-expressions to core forms using macro transformers.

Here is a partial list of the shorthands that we add to the lexer:

`{a b c}`	is a shorthand for:	`(@afn a b c)`
`[a b c]`	is a shorthand for:	`(@tuple a b c)`
`a(b c)`	is a shorthand for:	`a (@do b c)`
`a<b c>`	is a shorthand for:	`a (@of b c)`
`a[b c]`	is a shorthand for:	`a (@get b c)`

The curly braces (`{}`) and square braces (`[]`) are shorthands for lists that begin with the symbols `@afn` and `@tuple` respectively. Parentheses (`()`), angle brackets (`<>`), and square braces (`[]`) that *immediately* follow (i.e. without intervening spaces) an s-expression are shorthands for lists that begin with the symbols `@do`, `@of`, and `@get` respectively.

For instance, the following decorated s-expression:

```
(begin
  val xi = xs[i]
  val xj = xs[j]
  xs[i] = xj
  xs[j] = xi)
```

is a shorthand for and completely equivalent to the following:

```
(begin
  val xi = xs (@get i)
  val xj = xs (@get j)
  xs (@get i) = xj
  xs (@get j) = xi)
```

Next we extend the lexer with the following rule to handle structured indentation: *any line-ending colon automatically surrounds the next indented block with parentheses.*

Thus the following decorated s-expressions:

```
defn f (x y) :
  if x == y :
    x + y
    x - y
```

are shorthands for and completely equivalent to the following:

```
defn f (x y) : (
  if x == y : (
    x + y
    x - y
  )
)
```

With these added shorthands, we can define the following macro transformers to arrive at a familiar natural syntax:

1. Macro for defining functions.

   ```
   defn f (x y z) : (form1 form2 ...)
   ```

   is a shorthand for:

   ```
   (defn f (x y z) form1 form2 ...)
   ```

2. Macros for defining values and variables, and assigning to variables.

   ```
   val x = v
   var x = v
   x = v
   ```

   are respectively shorthands for:

   ```
   (defval x v)
   (defvar x v)
   (set x v)
   ```

3. Macro for calling functions.

   ```
   f (@do x y z)
   ```

   is a shorthand for:

   ```
   (f x y z)
   ```

4. Macros for retrieving from and storing to arrays.

   ```
   x (@get i) = v
   x (@get i)
   ```

   are respectively shorthands for:

   ```
   (array-set x i v)
   (array-get x i)
   ```

5. Macros for binary operators.

   ```
   a + b
   a - b
   a < b
   a != b
   ```

are respectively shorthands for:

```
(plus a b)
(minus a b)
(less? a b)
(not (equal? a b))
```

6. Macro for if expressions.

```
if pred : (form1 form2 ...) else : (form3 form4 ...)
if pred : (form1 form2 ...)
```

are respectively shorthands for:

```
(if pred
  (begin form1 form2 ...)
  (begin form3 form4 ...))

(if pred
  (begin form1 form2 ...)
  false)
```

7. Macro for loops.

```
for i in start to end do : (form1 form2 ...)
```

is a shorthand for:

```
(define loop (x end-index)
  (when (less? x end-index)
    form1
    form2
    ...
    (loop (plus x 1) end-index)))
(loop start end)
```

With the added lexer shorthands, the structured-indentation feature, and the macro transformers described above, we can now rewrite the selection sort algorithm as:

```
;Sort an array of integers
defn sort (xs) :
  ;Swap element i with element j
  defn swap (i, j) :
    if i != j :
      val xi = xs[i]
      val xj = xs[j]
      xs[i] = xj
      xs[j] = xi

  ;Find the index of the minimum element
  ;between index b (inclusive) and e (exclusive)
  defn minimum (b, e) :
    var min-i = b
    var min = xs[b]
    for i in (b + 1) to e do :
      if xs[i] < min :
        min-i = i
        min = xs[i]

  ;Loop
  for i in 0 to array-length(xs) - 1 do :
    swap(i, minimum(i, array-length(xs)))
```

At this point, the language looks quite similar to the Python [47] programming language, a language well-known for its readability. Despite the pleasant syntax, there still exists a simple mapping from the program text to s-expressions, and programmatically manipulating code remains intuitive.

2.4 Supporting First-Class Functions

In our examples thus far, every time we have referenced a declared function by name it has been in the function position of a function call expression. For example, the following code references the two functions `array-length` and `minimum`:

```
val n = array-length(xs)
minimum(i, n)
```

`array-length` is called with `xs`, and `minimum` is called with `i` and `n`.

Language support for first-class functions makes it valid to reference functions in all other contexts – essentially allowing functions to be treated identically to how we treat other values. Functions will be able to be assigned to values, stored in arrays, and passed as arguments. The following example stores `array-length` and `minimum` into an array, and is functionally equivalent to the code above:

```
val funcs = make-array(2)
funcs[0] = array-length
funcs[1] = minimum
val n = funcs[0](xs)
funcs[1](i, n)
```

We first store `array-length` and `minimum` into the first and second slot of the `funcs` array. Then they are retrieved from the array to be called with their original arguments.

Higher-Order Functions

First-class functions are a staple feature of functional programming languages and a suite of programming techniques have been developed around them – the most important of which is the use of *higher-order functions*. Consider the following functions, `array-sum` and `array-product`, for computing the sum and product of all numbers in an array:

```
defn array-sum (xs) :
  var accum = 0
  for i in 0 to array-length(xs) do :
    accum = plus(accum, xs[i])
  accum

defn array-product (xs) :
  var accum = 1
  for i in 0 to array-length(xs) do :
    accum = times(accum, xs[i])
  accum
```

The structure of the two algorithms are nearly identical – the only two differences being:

1. the initial value of `accum` (0 for `array-sum` and 1 for `array-product`), and

2. the two-argument function used to update the value of `accum` on each iteration (`plus` for `array-sum` and `times` for `array-product`).

By using a higher-order functions, we can abstract over the common pattern by accepting these differences as arguments. Here is a generalized implementation of a reduction over an array:

```
defn array-reduce (f, x0, xs) :
  var accum = x0
  for i in 0 to array-length(xs) do :
    accum = f(accum, xs[i])
  accum
```

where the initial value, `x0`, and reduction function, `f`, are passed in as arguments.

We can now use the generalized `array-reduce` function to compute either the sum or product of all numbers in an array by passing it different reduction functions and initial values:

```
val sum = array-reduce(plus, 0, xs)
val product = array-reduce(times, 1, xs)
```

Here's another example of using `array-reduce` to compute the result of concatenating all the lists contained within an array, `lists`:

```
val biglist = array-reduce(append, ‘(), lists)
```

where `append` is assumed to be a function for concatenating two lists.

Anonymous Functions

In functional programming style, many of the functions defined by a programmer are simply used as arguments to higher-order functions, and are never called directly. Consider the following example where we assume that `arrays` is an array of arrays, and we wish to compute the total number of elements in all the arrays:

```
defn add-length (len, xs) :
  len + length(xs)
val num-elements = array-reduce(add-length, 0, arrays)
```

The `add-length` function is used only as an argument to `array-reduce`, and never again referenced. *Anonymous functions* are provided specifically for these cases where a function is referenced only once. The following shows the above example rewritten using an anonymous function:

```
val num-elements = array-reduce(
                     fn (len, xs) : len + length(xs),
                     0,
                     arrays)
```

The syntax `fn (x, y) : body` denotes a function that takes two arguments, `x` and `y`, and returns the result of evaluating `body`.

We provide the following macro to make it even more convenient to define short anonymous functions. The syntax:

```
{_ + 3 * _}
```

which is a lexer shorthand for:

```
(@afn _ + 3 * _)
```

expands into the following during macro expansion:

```
(fn (x, y) : x + 3 * y)
```

An expression in curly braces (`{}`) denotes an anonymous function where underscores (`_`) become arguments to the function.

Using this shorthand, `num-elements` can be computed using a single concise line:

```
val num-elements = array-reduce({_ + length(_)}, 0, arrays)
```

Useful Higher-Order Functions

Here are some commonly-used higher-order functions:

- The `all?` function takes a single-argument predicate function and an array, and checks whether the predicate function returns `true` for all the items in the array:

```
defn all? (pred, xs) :
  defn loop (i) :
    if i < array-length(xs) :
      if pred(xs[i]) : loop(i + 1)
      else : false
    else : true
  loop(0)
```

- The functions `any?` and `none?` are implemented similarly to `all?` and checks whether the predicate returns `true` for any of the items in the array, or for none of the items in the array.
- The `find` function takes a single-argument predicate function and an array, and returns the first element in the array for which the predicate function returns `true`:

  ```
  defn find (pred, xs) :
    defn loop (i) :
      if i < array-length(xs) :
        if pred(xs[i]) : xs[i]
        else : loop(i + 1)
      else : false
    loop(0)
  ```

- The `array-map` function takes a single-argument function and an array, and returns a new array initialized with the results of calling the function on the elements in the original array:

  ```
  defn array-map (f, xs) :
    val ys = make-array(array-length(xs))
    for i in 0 to array-length(xs) do :
      ys[i] = f(xs[i])
    ys
  ```

Higher-order functions are enormously powerful and can be used to very concisely express sophisticated concepts (item 6 of our objectives). Here are some example tasks expressed first in natural language, followed by their translation to our programming language:

1. Find the first negative number in the array `xs`.

   ```
   find({_ < 0}, xs)
   ```

2. Determine whether the array `xs` contains the value `x` – i.e. determine if any value in `xs` is equal to `x`.

   ```
   any?({_ == x}, xs)
   ```

3. Determine whether the arrays `xs` and `ys` have any values in common – i.e. determine whether any value in `xs` is contained in `ys`.

   ```
   defn contains? (xs, x) : any?({_ == x}, xs)
   any?({contains?(ys, _)}, xs)
   ```

4. Find the spread of values in the array `xs` – i.e. determine the difference between the maximum and the minimum element in `xs`. Assume that all numbers are between 0 and 10.

```
defn max (x, y) : if x < y : y else : x
defn min (x, y) : if x < y : x else : y
array-reduce(max, 0, xs) - array-reduce(min, 10, xs)
```

5. Assume that the array xs represents an n-dimensional vector and find the square of its length.

```
defn square (x) : x * x
array-reduce(plus, 0, array-map(square, xs))
```

In addition to their conciseness, these examples also demonstrate how higher-order functions help us break down complex tasks into small reusable components (items 4 and 5 of our objectives). For example, the find function separates the task of looking for an element in an array from the task of specifying what to look for; and the array-map function separates the task of applying some operation to every element in an array from the task of specifying the operation.

Iteration and the Generalized For Macro

With the introduction of higher-order functions, we can now show the generalized implementation of the for macro.

First we define the do higher-order function as the following:

```
defn do (f, xs) :
  defn loop (i) :
    if i < array-length(xs) :
      f(xs[i])
      loop(i + 1)
  loop(0)
```

which abstracts over iterating through an array. It takes a single-argument function, f, and an array, and calls f on every element in the array. For example, we can use do to print the square of every number in xs like so:

```
do(fn (x) :
     println(x * x)
   xs)
```

We then define the generalized for macro as a shorthand for the above. The syntax:

```
for x in xs f :
  body
```

expands into the following:

```
f(fn (x) : body, xs)
```

which allows us to rewrite the above code as:

```
for x in xs do :
  println(x * x)
```

The `for` macro also generalizes over multiple collections. The syntax:

```
for (x in xs, y in ys, z in zs, ...) f :
  body
```

expands into the following:

```
f(fn (x, y, z, ...) : body, xs, ys, zs, ...)
```

This multiple collection form is used in conjunction with a corresponding `do` function for concurrently iterating through two arrays:

```
defn do (f, xs, ys) :
  defn loop (i) :
    if i < array-length(xs) and i < array-length(ys) :
      f(xs[i], ys[i])
      loop(i + 1)
  loop(0)
```

As an example, here's a function that computes the dot product of two vectors, each represented as an array of numbers:

```
defn dot-product (xs, ys) :
  var accum = 0
  for (x in xs, y in ys) do :
    accum = accum + x * y
  accum
```

To regain our original `start to end` syntax, we just need to introduce a macro to expand the `to` operator into a function call to `make-range`. The syntax:

```
i to j
```

expands into the following:

```
make-range(i, j)
```

where `make-range` is assumed to return an array containing all the integers between `i` (inclusive) and `j` (exclusive).

The generalized `for` macro can be similarly used with the `find`, `all?`, `any?`, `none?`, and `array-map` functions shown earlier. Here is an example showing how to search an array for the first number between some specified range:

```
defn find-in-range (start, end, xs) :
  for x in xs find :
    x >= start and x <= end
```

The `for` macro illustrates one of the core philosophies behind the design of Stanza. The number of concepts in the language is kept small by minimizing the number of kinds of *core forms*, i.e. the s-expressions remaining after macro expansion. Convenience constructs are then introduced as macro transformers, and can be understood purely as syntactic shorthands whose use is entirely optional. If macros are sufficiently general, then even just a few macros can significantly increase convenience. As shown, the `for` macro is not simply a looping construct. It can be used with many higher-order functions besides do, and nothing prevents the programmer from defining his or her own higher-order functions to be used with for.

2.5 Supporting Basic Objects

Many of the quantities that we manage in our daily lives occur together in groups. A person's name, age, height, and weight, for instance, are four separate quantities that often occur together. To help us manage these quantities jointly, we will extend our programming language with constructs for creating and manipulating *objects*.

A *struct* definition declares the structure of a class of objects: the names of all the *fields* that describe an object of that class. The following code defines a struct named Person, which declares that every Person object has a name, an age, a height, and a weight field.

```
defstruct Person: (name, age, height, weight)
```

The new-struct operator is used to create a Person object given the initial values for its four fields:

```
val p = new-struct Person{"Patrick", 29, 178, 162}
```

The dot (.) operator is used to retrieve the value of an object's field:

```
val name = p.name
val age = p.age
```

And we put the dot (.) on the left-hand side of the (=) operator to store a value into an object's field:

```
p.name = "Patrick Li"
p.age = 30
```

The following shows how these simple object constructs allow us to define a stack datastructure by using an integer for representing the length of the stack, and an array for holding its contents:

```
defstruct Stack: (length, array)

;Create a stack that can hold some maximum number of objects
defn make-stack (capacity) :
  new-struct Stack{capacity, make-array(capacity)}

;Push a new item to the end of the stack
defn stack-push (s, value) :
  val l = s.length
  s.array[l] = value
  s.length = l + 1

;Pop the last item pushed to the stack
defn stack-pop (s) :
  val l = s.length
  s.length = l - 1
  s.array[l - 1]
```

Given these functions, the programmer can use stacks in the same way they use the built-in array and list datastructures. The following shows the implementation of a reverse-polish-notation [10] (RPN) calculator that evaluates the operands and operators stored in the array `xs`:

```
defn rpn-calculator (xs) :
  val stack = make-stack(10)
  for x in xs do :
    if x == "+" :
      val a = stack-pop(stack)
      val b = stack-pop(stack)
      stack-push(a + b)
    else if x == "-" :
      val a = stack-pop(stack)
      val b = stack-pop(stack)
      stack-push(a - b)
    else :
      stack-push(x)
  stack-pop(stack)
```

The calculator keeps track of its state with a stack object and iterates through `xs`, evaluating each operator or operand it encounters. A `"+"` string indicates to replace the top two values from the stack with their sum; a `"-"` string indicates to replace the top two values from the stack with their difference; otherwise we assume it is an integer operand and push it onto the stack.

Adding support for objects accomplishes many of our high-level objectives:

1. It provides another means to subdivide complex tasks (item 4). The task of implementing an RPN calculator is split into two separate tasks: how to track a dynamically growing and shrinking list of values, and how to implement the behaviour of the operators and operands of an RPN calculator.

2. It provides another means to separate concerns (item 1). The details of how to represent the stack are isolated from the details of the RPN calculator. We can easily change the implementation of the stack to store its state as a list instead of in an array without affecting the implementation of `rpn-calculator`:

   ```
   defstruct Stack: (list)

   ;Create a stack that can hold an infinite number of objects
   defn make-stack (capacity) :
     new-struct Stack{‘()}

   ;Push a new item to the end of the stack
   defn stack-push (s, value) :
   ```

```
  s.list = cons(value, s.list)

;Pop the last item pushed to the stack
defn stack-pop (s) :
  val x = head(s.list)
  s.list = tail(s.list)
  x
```

3. It allows common components to be reused (item 5). Our stack functions are not tied specifically to the implementation of `rpn-calculator`. They are general and can be reused for any application requiring a stack datastructure.

4. It allows programmers to express algorithms at a higher level of abstraction (item 7). Even in natural language, the behaviour of an RPN calculator is almost always described in terms of operations performed on a stack. Thus the code in `rpn-calculator` is a direct translation of its natural language description.

Dynamic Dispatch

At this point, we will give a name to the property we have been using informally to categorize values: every value created in the language has a *type*. All integer values (e.g. `0`, `42`, `256`) are of type `Int`. All strings (e.g. `"hello"`, `"Patrick"`) are of type `String`. The values `true` and `false` are of type `True` and `False`. Arrays created with `make-array` have type `Array`, and lists created with `cons` have type `List`. All objects created with the syntax:

```
new-struct Person{a, b, c, d}
```

have type `Person`.

In this section, we will add a construct for dynamically testing the type of a value: the `match` expression. The syntax:

```
match(f(42)) :
  (x:Int) : body1
  (y:String) : body2
  (z:Person) : body3
  (w) : body4
```

first computes the result of evaluating `f(42)`. Then it tests whether this result has type `Int`, and if so, it binds the result to `x` and evaluates `body1`. If not, it proceeds to the next clause, where it tests whether the result has type `String`. If so, the result is bound to `y` and it evaluates `body2`. Similarly, the next clause binds the result to `z` and evaluates `body3` if the result is a `Person`. Finally, if the result is none of the above, the `match` expression evaluates `body4`.

The `match` expression can also accept and match upon multiple arguments. The syntax:

```
match(f(42), g(43)) :
  (x:Int, y:String) : body1
  (x, y:Person) : body2
  (x, y) : body3
```

first computes the results of evaluating `f(42)` and `g(43)`. The first clause tests whether the results have type `Int` and `String` respectively and, if they do, binds the results to `x` and `y` and evaluates `body1`. The second clause tests whether the second result has type `Person`, and evaluates `body2` if it does. Finally, if neither the first nor second clause matches, then it evaluates `body3`.

As an example of using the `match` expression, consider the following struct definitions for representing geometric shapes:

```
defstruct Rectangle: (x, y, width, height)
defstruct Square: (x, y, length)
defstruct Circle: (x, y, radius)
```

A `Rectangle` object is described by the `x` and `y` coordinates of its bottom-left corner and its `width` and `height`. A `Square` object is described by the `x` and `y` coordinates of its bottom-left corner and the `length` of its sides. A `Circle` object is described by the `x` and `y` coordinates of its center, and its `radius`.

Now consider the following function, `area`, which is able to compute the area of *any* geometric shape:

```
defn area (shape) :
  match(shape) :
    (r:Rectangle) : r.width * r.height
    (s:Square) : s.length * s.length
    (c:Circle) : 3.14f * c.radius * c.radius
```

Because different shapes have different formulas for computing their area, `area` begins by dynamically testing the type of its argument to branch to the appropriate code for each shape. If we mistakenly pass the value `42` to `area`, then the `match` expression will fail and halt the program, reporting that there is no clause that matches against the value `42`.

For the purpose of minimizing the number of core concepts in the language, note that the `match` expression subsumes the `if` expression. The following:

```
if x < 2 :
  f(42)
else :
  g(43)
```

is completely equivalent to:

```
match(x < 2) :
  (result:True) : f(42)
  (result:False) : g(43)
```

Thus we will now no longer consider the `if` expression a core form that is understood by the compiler. Instead we use a macro transformer to implement the `if` construct as a syntactic shorthand for the `match` expression.

2.6 Supporting Static Typing

Our language, thus far, offers very good dynamic error detection – invalid operations immediately halt the program and provide detailed error reports – but minimal static error detection. A human being can identify many programs as looking “wrong” even without executing the program. Consider the following program:

```
;Push a new item to the end of the stack s
defn stack-push (s, value) :
  s.list = cons(value, s.list)

defn main () :
  stack-push(42, "Hello")
```

The call to `stack-push` seems obviously “wrong” as `stack-push` is documented to require a `Stack` object for its first argument and the number `42` is not a `Stack` object.

Before we can extend our language with the ability to statically detect errors, we must first define what it means for a program to be “wrong”. Unfortunately, the most straightforward definition – *a program is wrong if it will halt with an error when executed* – is overly conservative. According to that definition it is not clear that the above example is wrong: after all, the `main` function may never be executed.

For our language, we will define “wrong” to mean that there is an inconsistency at the level of types. For instance, `stack-push` requires its first argument to be of type `Stack`, and therefore it is inconsistent to call it with a value of type `Int`.

Type Annotations

In order to detect type inconsistencies, we then have to consider how the compiler would know about the required argument types for `stack-push`. We make the design choice to require programmers to provide this information explicitly. We don’t consider this to be overly burdensome, as programmers already have a notion of what types the arguments should be – as evidenced by the comment above `stack-push`. We are simply asking them to write it down.

We extend the `defn` syntax to allow programmers to annotate the required types of the arguments:

```
defn stack-push (s:Stack, value:?) :
  s.list = cons(value, s.list)
```

where the syntax `s:Stack` indicates that the argument `s` must have type `Stack`.

Some examples of types include:

- The names of primitive types: e.g. `Int`, `String`, `True`

- The names of struct types: e.g. `Person`, `Stack`

- The unknown type: `?`

- A tuple type: e.g. `[Int,String]`, which denotes a two-arity tuple containing an `Int` followed by a `String`.
- A function type: e.g. `(Int, String) -> Int`, which denotes a two-argument function that takes an `Int` and a `String` and returns an `Int`.
- The union of a number of types: e.g. `Int|String`, which denotes either an `Int` or a `String`.

The unknown type, `?`, is special in that it denotes that we don't know or don't care about this argument's type. The language compiler will allow *any* value to be passed to an argument of type `?`, and also allow a value of type `?` to be used *anywhere*.

We will similarly extend the `defstruct` construct to accept type annotations for its fields:

```
defstruct Person:
  name: String
  age: Int
  height: Int
  weight: Int
```

Thus a person's name is represented using a `String`, but the person's age, height, and weight are represented using `Int` values.

These type annotations allow the language to statically detect many errors before executing the program (item 2 of our objectives) – a process called *typechecking*. Here are some examples of erroneous programs and the resulting errors reported by the compiler:

- Calling a function with arguments of the wrong type:

  ```
  stack-push(42, 42)
  ```

  Error: Cannot call `stack-push` which requires arguments of type `(Stack, ?)` with values of type `(Int, Int)`.

- Initializing an object with values of the wrong type:

  ```
  val p = new-struct Person{29, "Patrick", 178, 162}
  ```

  Error: Cannot initialize `Person` struct which requires fields of type `(String, Int, Int, Int)` with values of type `(Int, String, Int, Int)`.

- Assigning a value of the wrong type to a field:

  ```
  p.name = 40
  ```

  Error: Cannot assign a value of type `Int` to the `name` field of `Person` which requires type `String`.

Static error detection *greatly* increases programmer productivity. Each reported error typically requires only a few minutes at most to locate and fix, which is greatly reduced from the time required to manually write and maintain the tests for locating the same errors. It is considered uncommon for a programmer to write more than a page of code *without* making a mistake that is statically detected by the compiler.

Dynamic Error Detection

Because of the presence of the unknown type, `?`, an absence of statically detected errors in our language does not guarantee that the program will execute without errors. However, the type annotations do allow the system to dynamically detect *more* errors and *earlier* than without annotations.

Consider the following code, which assumes `xs` is an array containing a `Person` in its first slot and a `String` in its second slot:

```
val person = xs[0]
val name = xs[1]
person.name = name
```

Now examine what happens if the assumption is incorrect and that `xs[1]` contains an `Int`, `42`, instead of a `String`.

Prior to introducing type annotations, the `name` field of `Person` was unrestricted and could contain any value – so the system would have allowed `42` to be freely stored into the `name` field. This would have *hopefully* caused the program to crash at some later time, and the programmer would have had to track down exactly when the person object was first corrupted.

In contrast, the type annotations explicitly require for the `name` field of `Person` to be of type `String`. Thus *immediately* upon detecting the attempt to store `42` into a `String` field, the program will halt and give a detailed error report.

Function Overloading and Automatic Function Mixing

Beyond improving error detection, the extra information contained in the type annotations also allow us to introduce two new features – *function overloading* and *automatic function mixing* – for increasing the expressivity and ease-of-use of the language.

Consider, in addition to `stack-push` and `stack-pop`, defining the function `stack-get` for accessing an indexed slot within a stack datastructure.

```
defstruct Stack: (length:Int, array:Array)

defn stack-get (s:Stack, i:Int) :
  s.array[i]
```

`stack-get` has a similar purpose to `array-get`: it retrieves the value at some indexed slot within a datastructure. In fact, there are many datastructures that structurally consists of a series of slots – such as lists, vectors (a dynamically growing array), queues, etc. – and we can imagine an analogous `-get` function for each of them. Here's a listing of some of these functions along with their types:

```
array-get: (Array, Int) -> ?
stack-get: (Stack, Int) -> ?
list-get: (List, Int) -> ?
vector-get: (Vector, Int) -> ?
queue-get: (Queue, Int) -> ?
```

Function overloading allows for multiple functions to have identical names, as long as they require different argument types. Given a reference to a overloaded function, the system will deduce which function is meant based on the context in which it is used.

With support for function overloading, we can rename all of the above functions to `get` and rely upon the compiler to deduce which one we mean to call. The following code demonstrates three different calls to `get`, each to a different version:

```
defn main (xs:Vector) :
  val s = make-stack(10)
  val l = ‘(a b c)
  val a = get(s, 0)  ;Calls the Stack version of get
  val b = get(l, 0)  ;Calls the List version of get
  val c = get(xs, 0) ;Calls the Vector version of get
  false
```

We can now also reimplement our macro transformer to expand `xs[0]` to `get(xs,0)` instead of specifically to `array-get(xs,0)` – which will allow the use of the `[]` syntax for datastructures other than arrays. The above code can then be rewritten as follows:

```
defn main (xs:Vector) :
  val s = make-stack(10)
  val l = ‘(a b c)
  val a = s[0]  ;Calls the Stack version of get
  val b = l[0]  ;Calls the List version of get
  val c = xs[0] ;Calls the Vector version of get
  false
```

In some situations, the compiler cannot uniquely determine which overloaded version of a function is meant to be called. In the following example, `xs` is annotated with the unknown type `?`, and therefore any of the above `get` functions are applicable:

```
defn add-first-and-second (xs:?) :
  val a = xs[0]
  val b = xs[1]
  a + b
```

In such cases, the compiler will automatically insert a `match` expression to dispatch to the appropriate version at execution. The above example behaves equivalently to the following:

```
defn add-first-and-second (xs:?) :
  val a = match(xs) :
    (xs:Array) : get(xs, 0)  ;Dispatch to Array version
    (xs:Stack) : get(xs, 0)  ;Dispatch to Stack version
    (xs:List) : get(xs, 0)   ;Dispatch to List version
    (xs:Vector) : get(xs, 0) ;Dispatch to Vector version
    (xs:Queue) : get(xs, 0)  ;Dispatch to Queue version
  val b = match(xs) :
    (xs:Array) : get(xs, 1)  ;Dispatch to Array version
    (xs:Stack) : get(xs, 1)  ;Dispatch to Stack version
    (xs:List) : get(xs, 1)   ;Dispatch to List version
    (xs:Vector) : get(xs, 1) ;Dispatch to Vector version
    (xs:Queue) : get(xs, 1)  ;Dispatch to Queue version
  a + b
```

This feature is called *automatic function mixing*, and is important for allowing binders with unknown (`?`) types to be used conveniently. It is so-named because it is as if the compiler implicitly created a *mixed* version of `get` that accepts a union of all the relevant types and dispatches to the appropriate implementation.

As a matter of style, we now no longer preface function names with the type of the argument that they operate on. For instance, `array-length` will be renamed to simply `length`, and `array-map` will be renamed to simply `map`.

Nominal Subtyping and Type Hierarchies

Just as support for objects allows us to manage related quantities together as a group, different types also often occur in groups. For instance, the types introduced earlier – `Rectangle`, `Square`, and `Circle` – are all types for representing shapes and often occur together.

To help programmers manage related types, we introduce the `deftype` construct for specifying a *type hierarchy*. The following example uses the `deftype` construct to specify the relationships between the shape types:

```
deftype Shape
deftype BoxyShape <: Shape
deftype RoundShape <: Shape

defstruct Rectangle <: BoxyShape :
  x:Float,
  y:Float,
  width:Float,
  height:Float

defstruct Square <: BoxyShape :
  x:Float
  y:Float
  length:Float

defstruct Circle <: RoundShape :
  x:Float
  y:Float
  radius:Float
```

The `Shape` type represents all shape objects and is defined using the syntax:

```
deftype Shape
```

It is subdivided into two *subtypes*: the `BoxyShape` type, which represents shapes with straight lines and sharp corners; and the `RoundShape` type, which represents shapes with curved lines. The syntax:

```
deftype BoxyShape <: Shape
```

defines the type `BoxyShape` as a subtype of `Shape`.

We now update the `Rectangle` and `Square` types to be subtypes of `BoxyShape`, and `Circle` to be a subtype of `RoundShape`. The syntax:

```
defstruct Rectangle <: BoxyShape :
  ...
```

specifies the `Rectangle` struct to be a subtype of `BoxyShape`. Types specified using the `deftype` construct can be used within type annotations and `match` expressions.

The semantics of the language depends upon the type hierarchy in two situations:

1. It affects what constitutes a valid expression during typechecking. The `area` function we defined previously for computing the area of different shapes can be annotated as follows:

   ```
   defn area (shape:Shape) :
     match(shape) :
       (r:Rectangle) : r.width * r.height
       (s:Square) : s.length * s.length
       (c:Circle) : 3.14f * c.radius * c.radius
   ```

   which specifies that the argument `shape` must be some type of `Shape`. The compiler will now statically detect and flag a call to `area` with the value `42` as an error.

2. It affects the behaviour of the `match` expression. The following match clause:

   ```
   match(exp) :
     ...
     (s:BoxyShape) : body
     ...
   ```

   will test whether the match value is of type `BoxyShape` or any subtype of `BoxyShape` when determining whether to evaluate `body`.

   As an example, the following `pointy?` function returns `true` if its argument is some sort of `BoxyShape`, and `false` if it is some sort of `RoundShape`.

   ```
   defn pointy? (s:Shape) :
     match(s) :
       (s:BoxyShape) : true
       (s:RoundShape) : false
   ```

It will return `true` if called with `Rectangle` or `Square` objects, and `false` for `Circle` objects.

Parametric Types and Polymorphic Functions

While our type system can now statically detect many errors, it does have a remaining critical limitation with significant practical impact. Although the type annotations allow us to specify that an argument must be an array, they are not expressive enough for us to specify that it must be an array *of integers*. Thus the following incorrect code:

```
defn sum (xs:Array) :
  var accum = 0
  for x in xs do :
    accum = accum + x
  accum

defn main () :
  val str = make-array(3)
  str[0] = "Hello"
  str[1] = "Patrick"
  str[2] = "!!"
  sum(str)
```

which calls `sum` with an array of strings, will not be caught by the typechecker.

In this section, we improve the typechecker by extending it to support *parametric types* and *polymorphic functions*. To start, we will require the `Array` type to now always be accompanied with the type of its contents. An array containing values of type `T` is specified as `Array<T>`. So an `Array<Int>` is an array of integers; an `Array<String>` is an array of strings; and an `Array<Array<Int>>` is an array *of* array of integers.

The `make-array` function is also changed to require a *type argument* for specifying the type of the contents of the array. A call to `make-array<T>(n)` creates an array of `T` values of length `n`.

With these changes, the above code would be rewritten as follows:

```
defn sum (xs:Array<Int>) :
  var accum = 0
  for x in xs do :
    accum = accum + x
  accum

defn main () :
  val str = make-array<String>(3)
  str[0] = "Hello"
  str[1] = "Patrick"
  str[2] = "!!"
  sum(str)  ;ERROR
```

The typechecker, with the additional information, would now be able to detect the incorrect call to `sum`:

ERROR: Cannot call function `sum` which requires an argument of type `Array<Int>` with a value of type `Array<String>`.

The parametric type annotations also allow the typechecker to detect when the programmer attempts to put a value of the wrong type into an array, or incorrectly use a value from an array.

```
defn main () :
  val str = make-array<String>(3)
  str[0] = 42          ;ERROR
  val x = str[1] + 1   ;ERROR
  false
```

The code above contains two statically-detectable errors:

1. it is invalid to store the value `42` into an `Array<String>`, and
2. it is invalid to add `1` to the value `str[1]` which can be deduced to be of type `String`.

To declare a parametric type, the `deftype` and `defstruct` constructs now allows the type name to be optionally followed with the names of its *type parameters*. The following example parameterizes our `Stack` struct with the type of its contents:

```
defstruct Stack<T> :
  length:Int
  array:Array<T>
```

The above says that `Stack` is a parametric type where `T` is the type of its contents. Its `length` field, used for storing the length of the stack, is of type `Int`; and its `array` field, used for storing its contents, is of type `Array<T>`.

To create stacks, we will change `make-stack` into an polymorphic function, like `make-array`, and require an explicit type argument for specifying the stack content type.

```
defn make-stack<T> (capacity:Int) :
  new-struct Stack<T>{capacity, make-array<T>(capacity)}
```

The `push` and `pop` functions will also now be polymorphic functions:

```
defn push<?T> (s:Stack<?T>, value:T) :
  val l = s.length
  s.array[l] = value
  s.length = l + 1

defn pop<?T> (s:Stack<?T>) :
  val l = s.length
  s.length = l - 1
  s.array[l - 1]
```

The question mark prefacing the `?T` parameters in `push` and `pop` indicates that `T` are *captured* type parameters instead of explicit type parameters. Captured type parameters are implicitly provided by the compiler instead of explicitly given by the programmer, and will be discussed in further depth later.

The following example shows the deduction process of the typechecker:

```
val stack = make-stack<String>(10)
push(stack, x)
val y = pop(stack)
```

The call to `make-stack<String>` returns a `Stack<String>` value which is then bound to `stack`. In the call to `push`, the typechecker implicitly provides `String` for the `T` type parameter by inspecting the type of `stack`, and then verifies that `x` is indeed a `String`. Similarly, in the call to `pop`, `String` is implicitly provided for the `T` type parameter, and a value of type `String` is returned and bound to `y`.

Summary of Static Typing Extension

The static type system is the basis of how our language statically detects errors. Additional information about the types of arguments and fields – when they are known – are provided explicitly by the programmer through type annotations. Binders or fields with unknown types can be annotated with the special type `?`, in which case they will behave identically to before we added static typing. Based on the additional information, the typechecker detects and reports type inconsistencies in the program.

Related types can be managed through type hierarchies. Parametric types and polymorphic functions handle types that are parameterized by other types – such as arrays of integers, and stacks of strings.

In addition to improving error detection, the added information from the type annotations also allow for function overloading and automatic function mixing, two features that increase language expressivity and decrease verbosity. This section presents these two features as pleasant conveniences, but later we will show how they form an integral part of the object system.

2.7 Supporting Packages

Up to now, our examples have assumed that all code pertaining to a program is contained within a single file, which is far from true for a large program. In this section, we add support for *packages*, a simple feature for breaking up a code base into smaller isolated units.

One uncomplicated but tedious problem that arises often in day-to-day programming is the need to carefully manage the proliferation of names in a program. If a programmer chooses to use the `Bark` type to represent the sound made by dogs, then the name `Bark` can no longer be used for representing the outer sheath of a tree. Choosing simultaneously unique but succinct names is a conceptually uninteresting but essential skill for a working programmer. Packages provide programmers a tool for managing the namespace by keeping sets of names isolated from each other.

A package begins with the following declaration:

```
defpackage mypackage :
  import mydependency
```

which indicates that all following functions, variables, values, types, and structs belong to the `mypackage` package. A single file may contain multiple packages but a package may never span more than one file. The `import mydependency` clause will be explained later.

By default, all definitions have *private* visibility, which means that they can only be referenced from code within the current package. The programmer must explicitly preface a definition with the `public` keyword to indicate otherwise. Suppose that our code for creating and manipulating `Stack` objects is contained within the `stacks` package:

```
defpackage stacks :
  import core

public defstruct Stack<T> :
  length:Int
  array:Array<T>

public defn make-stack<T> (capacity:Int) :
  new-struct Stack<T>{capacity, make-array<T>(capacity)}

public defn push<?T> (s:Stack<?T>, value:T) :
  s.array[s.length] = value
  add-to-length(s, 1)

public defn pop<?T> (s:Stack<?T>) :
  add-to-length(s, -1)
  s.array[s.length]

defn add-to-length (s:Stack, delta:Int) :
  s.length = s.length + delta
```

The `Stack` struct, `make-stack`, `push`, and `pop` functions are declared `public` and can thus be referenced from outside the `stacks` package. The `add-to-length` function is private to the `stacks` package, and is for internal use only.

The code for the RPN calculator will be kept in a separate `rpn-calculator` package. Since the RPN calculator implementation requires the use of the stack functions, we must first *import* the `stacks` package into the `rpn-calculator` package.

```
defpackage rpn-calculator :
  import core
  import stacks

public defn rpn-calculator (xs:Array<Int|String>) :
  val stack = make-stack(10)
  for i in 0 to length(xs) do :
    if xs[i] == "+" :
      val a = pop(stack)
      val b = pop(stack)
      push(a + b)
    else if xs[i] == "-" :
      val a = pop(stack)
      val b = pop(stack)
      push(a - b)
    else :
      push(xs[i])
  pop(stack)
```

The `import stacks` clause makes the public functions in the `stacks` package available to be referenced from the `rpn-calculator` package.

Note that both the `stacks` and `rpn-calculator` packages import the `core` package, which is where commonly-used functions such as `make-array`, `get`, `length`, `plus`, and `minus` are defined.

The package system is simple but acts as the fundamental mechanism by which large programs are split into divisions (item 1 of our objectives). To minimize accidental dependencies between packages, programmers can use the visibility modifiers to conceal implementation details.

2.8 Supporting Multimethods

At this point, we will turn our focus towards the issues of maintenance and software extensibility. The introduction of the `match` expression allowed us to dynamically test the type of a value and evaluate different code depending on its type. One particularly effective use of this was in the implementation of the `area` function for computing the area of a shape:

```
defn area (shape:Shape) :
  match(shape) :
    (r:Rectangle) : r.width * r.height
    (s:Square) : s.length * s.length
    (c:Circle) : 3.14f * c.radius * c.radius
```

We can write a similar function for computing the perimeter of a shape:

```
defn perimeter (shape:Shape) :
  match(shape) :
    (r:Rectangle) : 2.0f * (r.width + r.height)
    (s:Square) : 4.0f * s.length
    (c:Circle) : 2.0f * 3.14f * c.radius
```

The powerful idea demonstrated by these implementations is that the programmer can regard `area` and `perimeter` as functions that can compute the area and perimeter of *any* shape. No matter what it is called with, `area` and `perimeter` will automatically determine, by inspecting the type of its argument, the correct algorithm for computing the shape's area or perimeter.

That idea allows us to write the following functions, `total-area` and `total-perimeter`, which computes the total area and perimeter of an array containing an assortment of different shapes:

```
defn total-area (shapes:Array<Shape>) :
  var total = 0
  for s in shapes do :
    total = total + area(s)
  total

defn total-perimeter (shapes:Array<Shape>) :
  var total = 0
  for s in shapes do :
    total = total + perimeter(s)
  total
```

Consider what we must change to extend the program to support another type of shape: triangles. The first step is to declare the struct for representing triangles, which we will represent using the `x` and `y` coordinates of its three vertices:

```
defstruct Triangle <: Shape :
  x1: Float
  y1: Float
  x2: Float
  y2: Float
  x3: Float
  y3: Float
```

The second step is to write extra match clauses for `area` and `perimeter` to handle the case of `Triangle` values:

```
defn area (shape:Shape) :
  match(shape) :
    ...
    (t:Triangle) :
      val d = t.x1 * (t.y2 - t.y3) +
              t.x2 * (t.y3 - t.y1) +
              t.x3 * (t.y1 - t.y2)
      abs(d / 2.0f)

defn perimeter (shape:Shape) :
  match(shape) :
    ...
    (t:Triangle) :
      defn dist (x1:Float, y1:Float, x2:Float, y2:Float) :
        sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
      dist(t.x1, t.y1, t.x2, t.y2) +
      dist(t.x1, t.y1, t.x3, t.y3) +
      dist(t.x2, t.y2, t.x3, t.y3)
```

The limitations of the above approach start to now come to light. The code for supporting triangles is written as a bunch of small changes made to existing functions that are scattered throughout the code base. Support for triangles required changes to only `area` and `perimeter`, but adding a new datatype to a larger program may require changes to tens or hundreds of functions, which will quickly become unmanageable. Ideally, we would like to keep all of the code for supporting triangles together in one location, and not have to edit existing code at all.

The other fundamental limitation is that, since supporting new shapes requires editing the existing functions, it is impossible for users of the shapes library to define their own shapes. Only the library writers can extend the library.

Multis and Methods

To overcome the previous limitations, we will introduce two constructs, `defmulti` and `defmethod`, that conceptually allow us to separate the declaration of the function from the implementation of each match clause.

We begin by declaring `area` as a *multi* with the following syntax:

```
defmulti area (shape:Shape) -> Float
```

which declares `area` to be a function that accepts a `Shape` and returns a `Float`.

Next, we use the following syntax to implement each of the match clauses as a *method* for `area`:

```
defmethod area (r:Rectangle) :
  r.width * r.height

defmethod area (s:Square) :
  s.length * s.length

defmethod area (c:Circle) :
  3.14f * c.radius * c.radius

defmethod area (t:Triangle) :
  val d = t.x1 * (t.y2 - t.y3) +
          t.x2 * (t.y3 - t.y1) +
          t.x3 * (t.y1 - t.y2)
  abs(d / 2.0f)
```

These methods provide the implementations for the `area` function for the cases where the argument is a `Rectangle`, a `Square`, a `Circle`, and a `Triangle` respectively. The `area` multi will automatically dispatch to the appropriate implementation based on the types of its arguments, and will behave identically to the previous version.

The multimethod system allows us now to keep all of the code for supporting triangles in one location. Support for any shape can be added by first defining its struct, and then providing methods for the `area` and `perimeter` multis. It is no longer necessary to edit an existing function.

```
defstruct Triangle <: Shape :
  x1: Float
  y1: Float
  x2: Float
  y2: Float
  x3: Float
  y3: Float

defmethod area (t:Triangle) :
  val d = t.x1 * (t.y2 - t.y3) +
          t.x2 * (t.y3 - t.y1) +
          t.x3 * (t.y1 - t.y2)
  abs(d / 2.0f)

defmethod perimeter (t:Triangle) :
  defn dist (x1:Float, y1:Float, x2:Float, y2:Float) :
    sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
  dist(t.x1, t.y1, t.x2, t.y2) +
  dist(t.x1, t.y1, t.x3, t.y3) +
  dist(t.x2, t.y2, t.x3, t.y3)
```

The code for supporting triangles is not required to be in the same package as the rest of the shapes library – it can be in its own package, e.g. `triangles`. Additionally, in the same way as with triangles, the user can freely extend the shapes library to support custom shapes, as long as the algorithms for computing their areas and perimeters are provided. The rest of the shapes library, `total-area` and `total-perimeter`, will automatically be able to handle the new shapes.

The multimethod system is the basic tool in the language for architecting extensible and maintainable programs (item 8 of our objectives). The set of multis that operate on a type defines the interface by which the program interacts with values of that type. A program is extended in a systematic way by defining new types and implementing new methods for them.

Symmetric Multiple Dispatch

The `area` and `perimeter` multis only accept a single argument, but the multimethod system does generalize in a straightforward way to multiple arguments. In this case, the types of *all* the arguments are checked to dispatch to the appropriate method, a feature known as *multiple dispatch*.

A good application of this feature for the shapes library would be an `intersects?` function that checks whether two shapes are intersecting. Here is the declaration of the multi:

```
defmulti intersects? (a:Shape, b:Shape) -> True|False
```

It accepts two shapes as arguments and returns `true` if they are intersecting, or `false` otherwise.

Here is the method specifically for checking whether two circles intersect:

```
defmethod intersects? (a:Circle, b:Circle) :
  val x = a.x - b.x
  val y = a.y - b.y
  val r = a.radius + b.radius
  x * x + y * y <= r * r
```

Here is the method for checking whether two rectangles intersect:

```
defmethod intersects? (a:Rectangle, b:Rectangle) :
  defn overlap? (a1:Float, a2:Float, b1:Float, b2:Float) :
    a1 <= b2 and b1 <= a2
  overlap?(a.x, a.x + a.width, b.x, b.x + b.width) and
  overlap?(a.y, a.y + a.height, b.y, b.y + b.height)
```

Here is the method for checking whether a rectangle intersects with a square, where we take advantage of the fact that a square is just a rectangle with equal width and height:

```
defmethod intersects? (a:Rectangle, b:Square) :
  intersects?(a, new-struct Rectangle{b.x, b.y, b.length, b.length})
```

And continuing in this fashion, we can define a separate method to test for the intersection between every possible pair of types. The multi will automatically dispatch to the appropriate method based on the types of both of its arguments.

Instance Methods

The `defmulti` and `defmethod` constructs, as shown thus far, are restricted to only be valid as top-level expressions. It is not valid to nest these constructs within a function. In this section, we will lift this restriction by adding the `new` construct which allows us to create an object accompanied by a set of *instance methods*.

Suppose we have a shape with a known area and perimeter: 10.0 and 30.0 respectively. We can use the `new` syntax to directly create an object representing such a shape like so:

```
defn make-my-shape () :
  val s = new Shape :
    defmethod area (this) :
      10.0f
    defmethod perimeter (this) :
      30.0f
  s
```

The `make-my-shape` function creates a new `Shape` object, binds it to `s`, and then returns `s`. The `defmethod` constructs following the `new` construct specify the *instance methods* for the object that is created. The syntax for instance methods is nearly identical to top-level methods save for one aspect: there must be *exactly* one argument that is named `this`.

The `this` argument is special and refers to the object that is created by `new`. Thus the `area` method should be interpreted to mean: when the `area` multi is called specifically with the `s` shape, then `area` should return `10.0f`. Similarly, when the `perimeter` multi is called with the `s` shape, then `perimeter` should return `30.0f`.

Here is an example of computing the total area of three shapes: one square, and two shapes created using `make-my-shape`:

```
defn main () :
  val shapes = make-array<Shape>(3)
  shapes[0] = new-struct Square{0.0f, 0.0f, 1.0f}
  shapes[1] = make-my-shape()
  shapes[2] = make-my-shape()
  total-area(shapes)
```

`total-area` iterates through the array and calls `area` on each shape within it. The first shape is a square of length 1.0, and so `area` returns 1.0 × 1.0 = 1.0. The second and third shapes are the shapes created by `make-my-shape` and so `area` returns 10.0 for each of them. Hence the total area returned is 21.0.

Instance methods are allowed to refer to any values or variables defined in its surrounding scope. Let us revise `make-my-shape` to accept an argument for specifying the area of the shape to create:

```
defn make-my-shape (a:Float) :
  new Shape :
    defmethod area (this) :
      a
    defmethod perimeter (this) :
      30.0f
```

Now when `area` is called on the shape, it will return `a`, the argument passed to `make-my-shape`. We also no longer bother assigning the shape to the value `s`, and just return the created shape directly.

Here's the previous example revised such that `10.0f` and `20.0f` are passed as arguments in the calls to `make-my-shape`:

```
defn main () :
  val shapes = make-array<Shape>(3)
  shapes[0] = new-struct Square{0.0f, 0.0f, 1.0f}
  shapes[1] = make-my-shape(20.0f)
  shapes[2] = make-my-shape(30.0f)
  total-area(shapes)
```

Again, `total-area` calls `area` on each shape within the array. The first call returns 1.0 for the square as before. The second call now returns 20.0 for the shape created using `make-my-shape(20.0f)`, and the third call returns 30.0 for the shape created using `make-my-shape(30.0f)`. As can be seen, the values of the relevant binders referred to by the instance methods are stored with the object in order to be retrieved later. This is a conceptually simple but important feature that forms the basic mechanism of Stanza's multimethod object system.

2.9 The Multimethod Object System

Our basic object system so far consists of four constructs:

1. The `defstruct` construct for specifying the structure of objects as a collection of named fields. For example:

```
   defstruct Circle <: Shape :
     x:Float
     y:Float
     radius:Float
   ```

2. The `new-struct` construct for creating an object given the initial values of its fields. For example:

   ```
   new-struct Circle{1.0f, 2.0f, 3.0f}
   ```

3. The dot operator (`.`) for retrieving a field from an object. For example:

   ```
   val r = c.radius
   ```

4. The dot operator (`.`) for assigning to a field in an object. For example:

   ```
   c.radius = r
   ```

While simple, the object system does have a number of deficiencies:

1. All fields are always visible. The object system allows any code to retrieve from and store into any field in any object. But many fields are meant for internal use only – such as a flag to track some status, or a cache of some result. Programmers need these fields to stay concealed in order to be able to change their implementation freely without affecting the rest of the program.

   This is a widely-recognized issue and is solved by many languages, such as Java [2] and C++ [55], by accompanying fields with a visibility annotation.

2. All fields are mutable. When using the objects defined by a library, it is unclear whether the library allows for a field to be assigned to, and under what conditions it is allowed to be assigned to. The inability to declare immutable fields also makes it impossible to define pure immutable datastructures, a functional programming technique that can greatly reduce defects.

3. The dot operator leads to multiple syntaxes for retrieving data from an object. For instance, we can write a function for retrieving the diameter of a circle, which is simply twice its radius:

   ```
   defn diameter (c:Circle) :
     2.0f * c.radius
   ```

   But to the user of the library, there is a seemingly arbitrary difference between retrieving a circle's radius (`c.radius`) versus retrieving its diameter (`diameter(c)`).

4. The `new-struct` construct leads to multiple syntaxes for creating objects. For instance, we can write the following convenience functions for creating a unit circle – a circle with radius 1.0 – and for creating a unit circle centered at the origin:

```
defn make-Circle (x:Float, y:Float) :
  new-struct Circle{x, y, 1.0f}
defn make-Circle () :
  new-struct Circle{0.0f, 0.0f, 1.0f}
```

But to the user of the library, there is again seemingly arbitrary differences between the different ways of creating a circle. A unit circle at the origin is created with `make-Circle()`; a unit circle at some point is created with `make-Circle(1.0f, 3.0f)`; but a circle at some point with radius 2.0 is created with:

```
new-struct Circle{1.0f, 3.0f, 2.0f}
```

5. The objects are not abstract. Suppose we wish to create a circle whose radius is expensive to compute, in which case we would like to avoid computing it all-together if it is never used. This can be accomplished by delaying the computation until the first time the circle's radius is requested. This is not achievable currently as the `defstruct` construct specifies concretely that the radius must be stored as a `Float`.

The solution for eliminating these deficiencies happens to be quite simple and follows from recognizing that the multimethod system, by itself, is *already sufficient* for defining the object system. The `defstruct`, `new-struct`, and dot operator (`.`), are redundant and unnecessary.

Defining a Circle Datastructure

To demonstrate the multimethod object system, we will define the circle datastructure without the use of the basic object constructs. The first step is to define the type for representing circles:

```
deftype Circle <: Shape
```

which indicates that a circle is a type of shape.

The second step is to declare the multis that constitute the interface of a circle:

```
defmulti x (c:Circle) -> Float
defmulti y (c:Circle) -> Float
defmulti radius (c:Circle) -> Float
```

These multis declare that a circle is fundamentally defined by three operations: an operation each for retrieving the x and y coordinates of the circle, and an operation for retrieving the radius of the circle.

The last step is to define a function for creating circles:

```
defn make-Circle (x-pos:Float, y-pos:Float, r:Float) :
  new Circle :
    defmethod x (this) : x-pos
    defmethod y (this) : y-pos
    defmethod radius (this) : r
```

This function creates a circle from three floating point values. The object is created using the `new` construct, and instance methods are used to retrieve the appropriate values. For the purposes of clarity, we named the argument values `x-pos` and `y-pos`, but we could have also named them `x` and `y` with no issues.

Note that `make-Circle` is just a simple function. There is no distinguished concept of a *constructor* in our language, and we are free to define additional functions for creating circles. As an answer to limitation 5, the following example creates a circle that delays computing its radius:

```
defn make-Circle (x:Float, y:Float, compute-r: () -> Float) :
  var r-cache = false
  new Circle :
    defmethod x (this) : x
    defmethod y (this) : y
    defmethod radius (this) :
      match(r-cache) :
        (r-cache:Float) :
          r-cache
        (r-cache:False) :
          val r = compute-r()
          r-cache = r
          r
```

The radius of the circle is provided via a function – which is evaluated once when the radius of the circle is first requested.

To complete the definition of the circle datastructure, we have to rewrite our previous `area` and `perimeter` methods without the use of the dot operator:

```
defmethod area (c:Circle) :
  3.14f * radius(c) * radius(c)

defmethod perimeter (c:Circle) :
  2.0f * 3.14f * radius(c)
```

Note that because we no longer require the use of the `new-struct` or dot operator constructs, there is now one and only one way to create objects and retrieve their fields: through function calls. This removes limitations 4 and 3 of our basic object system.

Additionally, because all object operations are now done through function calls, the object system combines elegantly with other functional programming techniques – such as higher-order functions. For instance, if `cs` is a list of circles, `List<Circle>`, we can retrieve the list of all of their x coordinates using `map(x,cs)`.

Mutable “Fields”

The circle datastructure we defined is immutable by virtue of not supporting any functions for mutating it, but we can easily define a mutable circle datastructure by providing mutation functions. Suppose that a circle has a fixed location that cannot be changed, but that its radius can be changed. This can be accomplished by adding the following setter function to its interface:

```
defmulti set-radius (c:Circle, r:Float) -> False
```

We then change `make-Circle` to use a variable, `r`, to keep track of its current radius:

```
defn make-Circle (x-pos:Float, y-pos:Float, initial-r:Float) :
  var r = initial-r
  new Circle :
    defmethod x (this) : x-pos
    defmethod y (this) : y-pos
    defmethod radius (this) : r
    defmethod set-radius (this, radius:Float) : r = radius
```

The variable is initialized to the argument, `initial-r`, passed to `make-Circle` but it can be changed through calls to `set-radius`. The `radius` method simply returns the current value of this variable.

Thus we have the ability to control the mutability of individual fields without having to introduce any additional mechanisms to the language. This removes limitation 2 from our basic object system.

“Field” Visibility

Since “fields” are no longer a distinct concept and are instead modeled using multis, we can use the package visibility modifiers to control their visibility – thus removing limitation 1 from our basic object system.

Currently, the `Circle` type, the four interface functions `x`, `y`, `radius`, `set-radius`, and the `make-Circle` function, are all private to the package. As an example, we can declare the type, getter, and constructor functions as public, but leave `set-radius` as it is:

```
public deftype Circle <: Shape
public defmulti x (c:Circle) -> Float
public defmulti y (c:Circle) -> Float
public defmulti radius (c:Circle) -> Float
defmulti set-radius (c:Circle, r:Float) -> False

public defn make-Circle (x-pos:Float, y-pos:Float, initial-r:Float) :
  ...
```

With these changes, code from other packages will be able to create circles and retrieve their properties. As the library writer, we are allowed to mutate the radius of a circle, but this ability is not exposed to users.

The Defstruct Macro

Now that we’ve shown that the `defstruct`, `new-struct`, and dot operator (`.`) are unnecessary, we can remove them from our language. However, since `defstruct` provides a concise way of expressing such a commonly used pattern, we will reimplement it as the following macro transformer:

```
defstruct Circle <: Shape :
  x: Float
  y: Float
  radius: Float
```

expands to and is equivalent to:

```
deftype Circle <: Shape
defmulti x (c:Circle) -> Float
defmulti y (c:Circle) -> Float
defmulti radius (c:Circle) -> Float

defn Circle (x:Float, y:Float, radius:Float) :
  new Circle :
    defmethod x (this) : x
    defmethod y (this) : y
    defmethod radius (this) : radius
```

Hence the `defstruct` construct is now just a syntactic shorthand for declaring a new type, a set of multis, and a basic constructor function.

2.10 Supporting Non-Local Control Flow

Philosophically, we have purposefully kept the number of core concepts in the language to a minimum. There is one construct, in particular, that is commonly found in other languages and noticeably missing from ours: there is no construct for returning early from a function. A function currently returns and can only return the result of evaluating its last form.

The Label Construct

To address this deficiency, we will introduce a general non-local control flow operator: the `label` construct. As an example of its use, the following searches through an array and returns the first negative number it encounters:

```
defn first-negative (xs:Array<Int>) :
  label<False|Int> return :
    for x in xs do :
      if x < 0 :
        return(x)
```

The `label` construct above is specified to return a value of type `False|Int`, and to use `return` as the name of its *exit function*. The block passed to it executes to completion and returns `false` if `return` is never called. If `return` is called, however, then execution of the block *immediately* halts, and the argument passed to `return` becomes the result of the `label` construct.

Because Stanza uses higher-order functions to abstract over iteration constructs, it is important that the concept of exiting a block of code be kept orthogonal from the concept of functions – as is accomplished by the `label` construct. This is more clearly seen if we express `first-negative` without the use of the `for` macro:

```
defn first-negative (xs:Array<Int>) :
  label<False|Int> return :
    defn loop-body (x:Int) :
      if x < 0 :
        return(x)
    do(loop-body, xs)
```

The loop is expressed as calling the `do` higher-order function with the `loop-body` nested function. The design of the `label` construct allows us to specify clearly that `return` causes execution to exit from the entire loop rather than from just the `loop-body` function. From a design perspective, the `label` construct is key to making it practical to express iteration through higher-order functions.

The Generate Construct

The next control flow operator we will introduce is the `generate` construct, which is used to generate a sequence of items by concurrently executing a block of code. As an example of its use, the following demonstrates generating an infinite sequence containing all points in the first quadrant of the 2D cartesian plane:

```
defn all-2d-points () :
  generate<[Int,Int]> :
    for y in 0 to false do :
      for x in 0 through y do :
        yield([x, y - x])
```

The `generate` construct above is specified to return a sequence containing `[Int,Int]` tuples. The block passed to it is executed concurrently to generate tuples *as needed*. Each time the next tuple in the sequence is requested, the block executes until the next call to `yield` which returns the next tuple and also saves the state of the computation so that it may be resumed later.

The following requests and prints out the first three tuples in the sequence:

```
val points = all-2d-points()
println("The first point is: %_" % [next(points)])
println("The second point is: %_" % [next(points)])
println("The third point is: %_" % [next(points)])
```

where the `next` function is used to request the next tuple. When executed, the above prints out:

```
The first point is: [0, 0]
The second point is: [0, 1]
The third point is: [1, 0]
```

Targetable Coroutines

Both the `label` construct and the `generate` construct are just different usage patterns of the same control flow mechanism: Stanza's targetable coroutine system. Other usage patterns of the coroutine system include *consumers* – the dual of the `generate` construct – and throwing and catching exceptions.

The language is kept easy-to-learn (item 11 of our objectives) by the isolation of control flow handling to one orthogonal mechanism, and coroutines provide yet another tool for subdividing a complex task (item 4 of our objectives). As demonstrated, the `generate` construct allows us to separate the algorithm for generating a sequence of items from the code that operates on the items.

The general targetable coroutine system will be discussed in greater detail in a later chapter.

2.11 Supporting Low-Level Hardware Operations

Our language as described is a high-level language, and its semantics are independent of the underlying hardware. Details such as managing memory, representing values, and interacting with the operating system have been either completely automated (as in the case of memory-management) or concealed behind the implementation of library functions (such as `make-array` and `println`).

If the language were to be used only for a specific narrow domain, then this strategy would be sufficient. It would require some effort to ensure that all operations relevant to the domain are provided in the standard library, but it is not an impossible task. For a general-purpose language, however, it is impractical to foresee and include every operation that a user may require.

We will take the approach of providing a *sublanguage* that provides the low-level constructs necessary for the user to interact directly with the underlying hardware and operating system. LoStanza has the benefit that it allows for low-level control while also being designed to easily interoperate with code written using the high-level constructs. For purposes of clarity, we will call our current language HiStanza, and call the low-level language LoStanza.

Low-Level Functions

LoStanza constructs are kept isolated from HiStanza code, and can only be used from within a LoStanza function. The following shows an example of a LoStanza function taken from the implementation of Stanza's garbage collector:

```
lostanza defn scan-map-word (map:long, n:long, refs:ptr<long>) -> int :
  var ref-ptr:ptr<long> = refs
  for (var i:long = 0, i < n, i = i + 1) :
    val b = (map >> i) & 1
    if b : [ref-ptr] = post-gc-object([ref-ptr])
    ref-ptr = ref-ptr + sizeof(long)
  return 0
```

The `lostanza` prefix denotes that `scan-map-word` is a LoStanza function. The function demonstrates the use of a number of low-level constructs that are not available in HiStanza, such as:

- LoStanza types: e.g. `int`, `long`, and `ptr<long>`.
- Pointers: The `refs` argument is a pointer to 64-bit integers.
- Memory Loads/Stores: The `[ref-ptr]` expressions denote raw load and store operations to memory.

LoStanza code is able to freely call functions defined in HiStanza, and HiStanza code can call a subset of LoStanza functions.

Low-Level Objects

Unlike HiStanza, LoStanza allows objects to be defined with an explicitly specified memory layout. As an example, the following LoStanza type, `TimePoint`, represents a coordinate in four-dimensional space-time, where the three space axes are stored as 32-bit floating-point numbers, and the time axis is stored as a 64-bit integer.

```
lostanza deftype TimePoint :
  x: float
  y: float
  z: float
  t: long
```

Within a LoStanza function, the LoStanza type `TimePoint` can now be used to refer to a 192-bit bit pattern, where bits 0-31, 32-63, and 64-95 represent the x, y, and z space coordinates, and bits 128-191 represents the time coordinate. Bits 96-127 are skipped in order to keep the `t` field 64-bit aligned.

The following LoStanza code demonstrates creating a struct, retrieving a field, and storing a field:

```
val p = TimePoint{1.0f, 2.0f, 3.0f, 1987L}
val x = p.x
p.z = 4.0f
```

Managing Memory

The pointer type explicitly exposes low-level control of memory to the programmer. Assuming that `p` is a pointer to a 32-bit integer, `ptr<int>`, then we can load an integer from the address specified by `p` using the following syntax:

```
val i = [p]
```

and an integer can be stored into the address specified by `p` using the following syntax:

```
[p] = 32
```

Pointers can also be manipulated through arithmetic operations. As an example, the following loop clears a block of memory by assigning 0 to the 80 bytes of memory following `p`:

```
val end = p + 80
while p < end :
  [p] = 0
  p = p + 4
```

To allocate an object in heap memory, we can use the LoStanza `new` operator. The following code allocates a `TimePoint` value in the heap:

```
val p = new TimePoint{1.0f, 2.0f, 3.0f, 1987L}
```

Foreign Code

To interoperate with the surrounding software ecosystem, LoStanza can call functions written in other languages, and other languages can also call functions written in LoStanza.

Suppose we wanted to call a C [33] function with the following declaration:

```
float my_c_function (int a, float b);
```

From LoStanza, we use the `extern` keyword to declare the existence of `my_c_function` with the appropriate type signature:

```
extern my_c_function: (int, float) -> float
```

Then, within a LoStanza function, we call `my_c_function` with the following syntax:

```
lostanza defn f () -> float :
  val x = call-c my_c_function(42, 32.0f)
  return x
```

where the `call-c` operator is used to indicate that the C calling convention should be used for the call.

To define a LoStanza function, `my_stanza_function`, to be called from C, we use the following syntax:

```
extern defn my_stanza_function (a:int, b:float) -> float :
  val r = (a as float) + b
  return r
```

which will correspond to the following C declaration:

```
float my_stanza_function (int a, float b);
```

Summary of LoStanza

LoStanza provides the low-level constructs necessary to interact directly with the hardware and operating system, thereby allowing us to easily interoperate with foreign code (item 10 of our objectives). The sublanguage has a simple mapping to hardware instructions, so programmers that require fine control for performance-tuning can choose to implement their compute-heavy kernels in LoStanza (item 9). The sublanguage is also kept separate from the main language, and thus keeps the language easy-to-learn for the majority of programmers who do not require such control (item 11).

With the introduction of LoStanza, it is no longer necessary for the language to include built-in functions. Functions such as `make-array` and `println`, which previously could not be expressed in Stanza, can now be implemented using LoStanza constructs.

2.12 Interactions Between Subsystems

This chapter attempted to provide a rational motivation for the design of each of Stanza's key language features. Starting with a minimal Lisp [32] language, we incrementally added features to eliminate language deficiencies, until we arrived at the final language design, a union of the five following subsystems: the macro system, the optional type system, the multimethod object system, the targetable coroutine system, and the LoStanza sublanguage.

In reality however, our actual design process was less structured than presented, and the five subsystems were not designed independently. The interactions between the subsystems are nuanced and contribute to the operation of the language as a cohesive whole. None of the above subsystems can be removed from the language without significantly lowering the overall expressivity of the language.

The design of Stanza starts with Lisp [32] as a base, which offers an executable top-level, an s-expression-based programmatic macro system, and support for functional programming. From the Scheme [56] dialect, Stanza inherits its minimalism philosophy: the language is defined by a small number of core forms, a single namespace is used for both functions and variables, and loops are expressed using tail-recursion. Similar to Smalltalk [24] and Ruby [22], iteration constructs are abstracted as calls to higher-order functions, which is made practical through the use of coroutines as a general control-flow operator. Stanza deviates from Scheme's reliance upon built-in functions for interfacing with the software ecosystem and creating core datastructures, and instead provides the LoStanza sublanguage to enable users to execute foreign code and manipulate memory themselves.

The programmatic macro system makes three primary contributions to the language: it enables Stanza to have a natural syntax; it allows programmers to define their own syntactic abstractions and domain specific languages; and it allows the majority of Stanza's constructs to be implemented as syntactic shorthands. This last contribution makes it possible for Stanza to both be easy-to-learn – because of its small number of core forms – yet still be convenient for daily programming. The `for` macro, in particular, is necessary in order for iteration to be sensibly expressed as calls to higher-order functions.

A simple package system allows users to manage the namespaces of a large project. The package visibility modifiers allows definitions to be concealed within a package, or to be available for use by external code. In a later chapter, we will show that packages also double as the unit of compilation for Stanza's separate compiler, which outputs one `.pkg` file per package.

The type system and type annotations allows Stanza to statically detect errors before execution, but also plays an integral role in Stanza's `match` expression, multimethod dispatch, function overloading, and automatic function mixing features. The `match` expression dynamically dispatches to different code based on the runtime types of its arguments, and serves as Stanza's *only* branching construct. Multimethod dispatch relies upon the method argument type annotations for determining the target method, and is an important part of the object system. Type annotations are used to disambiguate references to overloaded functions – both statically, in the case of function overloading, and dynamically, in the case of automatic function mixing.

The multimethod system serves dual purpose as both a software architecting mechanism and as part of the object system. Object creation is handled through extending the multimethod system with the `new` construct, and object state is handled through instance methods that close over the lexical environment. The same mechanism is used for implementing both objects and closures. The package visibility modifiers used for controlling visibility of package definitions also serves to conceal the details of an object's concrete implementation. The emulation of object "fields" using getter and setter functions is made convenient enough for practical use through the function overloading and automatic function mixing mechanisms.

Stanza's targetable coroutine system doubles as both a concurrency operator and general control-flow operator. Constructs for returning early from functions, breaking from loops, generating sequences, and handling exceptions are all implemented as syntactic shorthands using the programmatic macro system.

Chapter 3

The Stanza Macro System

Stanza supports a programmatic macro system that operates on a homoiconic s-expression-based syntax. Macros are expressed as a set of transformations organized within a *parsing expression grammar* [23] (PEG) framework. After macro expansion, a program consists of only *core forms*, and the final executable results from compiling these core forms.

3.1 Decorated S-Expressions

The surface syntax of Stanza programs are expressed in terms of simple recursive tree datstructures known as *s-expressions*. An s-expression is either an *atom* – which can be a character, a string, a symbol, a number, the boolean values true or false – or a list of nested s-expressions. Stanza allows five different types of numbers: bytes, ints, longs, floats, and doubles.

```
sexp = byte
     | int
     | long
     | float
     | double
     | char
     | string
     | symbol
     | true
     | false
     | list of sexp
```

The mapping from characters to the s-expression datastructure is straightforward. The following listing shows some examples of the syntax:

```
bytes: 10Y, 24Y, 1Y
ints: 42, 128, -3
longs: 42L, 128L, -3L
floats: 1.0f, 128.3f, 12.0f
doubles: 1.0, 128.3, 12.0
```

characters: `’a’, ’b’, ’A’`
strings: `"hello world", "patrick"`
symbols: `hello, world, >=`
true: `true`
false: `false`
lists of s-expressions: `(a 53L "c"), (), (1 (2))`

In addition to the above, the lexer also supports a small number of shorthands to permit a more natural syntax:

`{x}`	is a shorthand for:	`(@afn x)`
`[x]`	is a shorthand for:	`(@tuple x)`
`f(x)`	is a shorthand for:	`f (@do x)`
`f{x}`	is a shorthand for:	`f (@do-afn x)`
`f[x]`	is a shorthand for:	`f (@get x)`
`f<x>`	is a shorthand for:	`f (@of x)`
`?x`	is a shorthand for:	`(@cap x)`
`‘sexp`	is a shorthand for:	`(@quote sexp)`

`a b c :`
`  d e f`
	is a shorthand for:	`a b c : (d e f)`

Curly brackets (`{}`) expand to a list with the `@afn` symbol as its first item. Square braces (`[]`) expand to a list with the `@tuple` symbol as its first item. An s-expression followed *immediately* by an opening parenthesis (`(`) inserts the `@do` symbol as the first item in the following list. An s-expression followed immediately by an opening curly bracket (`{`) inserts the `@do-afn` symbol as the first item in the following list. An s-expression followed immediately by a square brace (`[`) inserts the `@get` symbol as the first item in the following list. An s-expression followed immediately by an opening angle bracket (`<`) inserts the `@of` symbol as the first item in the following list. A question mark followed immediately by a symbol expands to a list with the `@cap` symbol as its first item. A backquote followed by an s-expression expands to a list with the `@quote` symbol as its first item. A line ending colon automatically wraps the next indented block in a list.

The following example shows the surface syntax and the underlying s-expression after all lexer shorthands have been expanded.

```
defn map<?T,?R> (f: T -> ?R, xs:List<?T>) -> List<R> :
   if empty?(xs) : List()
   else : cons(f(head(xs)), map(f, tail(xs)))
```

expands to:

```
defn map (@of (@cap T) (@cap R)) (f : T -> (@cap R)
                                  xs : List (@of (@cap T))) ->
                                  List (@of R) : (
  if empty? (@do xs) : List (@do)
  else : cons (@do f (@do head (@do xs)) map (@do f tail (@do xs))))
```

3.2 Core Forms

As is typical for Lisp-inspired [32] languages, Stanza programs are expressed primarily in terms of macros that expand the surface s-expressions to s-expressions that are understood by the compiler – also known as *core forms*. After macro expansion, a program is expressed entirely in terms of core forms. Here is a complete listing of the Stanza core forms:

```
Top Level Forms:
  ($package name imports ...)            (Package Definition)
  ($import name prefixes ...)            (Import Package)
  ($prefix-of (names ...) p)             (Assign Prefix to Names)
  ($prefix p)                            (Assign Prefix to All)
  ($public es ...)                       (Public Visibility)
  ($protected es ...)                    (Protected Visibility)
  ($deftype name parent children ...)    (Type Definition)
  ($defchild name parent)                (Child Definition)
  ($defmulti name (a1 ...) a2)           (Define Multi)

Expression Forms:
  ($def name type value)                                    (Define Value)
  ($defvar name type value)                                 (Define Variable)
  ($defn name (args ...) (a1 ...) a2 body ...)              (Define Function)
  ($defmethod name (args ...) (a1 ...) a2 body ...)         (Define Method)
  ($fn (args ...) (a1 ...) a2 body ...)                     (Anonymous Function)
  ($multi fs ...)                                           (Multi-arity Function)
  ($begin es ...)                                           (Grouped Expression)
  ($let e)                                                  (New Scope)
  ($match (es ...) branches ...)                            (Match Expression)
  ($branch (args ...) (ts ...) body ...)                    (Match Branch)
  ($new type methods ...)                                   (New Object)
  ($as exp type)                                            (Downcast)
  ($as? exp type)                                           (Upcast)
  ($set name exp)                                           (Assignment)
  ($do f args ...)                                          (Function Call)
  ($prim f args ...)                                        (Primitive Call)
  ($tuple es ...)                                           (Tuple Expression)
  ($quote v)                                                (Literal S-Expression)

Type Forms:
  ($of name args ...)   (Parametric Type)
  ($and a b)            (Intersection Type)
  ($or a b)             (Union Type)
  ($-> (a1 ...) a2)     (Function Type)
  ($cap x)              (Capture Variable)
  ($void)               (Void Type)
  ($?)                  (Unknown Type)

Miscellaneous Forms:
  ($none)   (Unspecified)
```

3.3 Syntax Packages

Macros are expressed as a set of transformations organized within a *parsing expression grammar* [23] (PEG) framework. A *syntax package* is defined using the `defsyntax` construct. The following example defines a syntax package with the name `my-syntax-package`.

```
defsyntax my-syntax-package :
  body ...
```

All of the macros that implement the standard constructs of the Stanza language are in the `core` syntax package.

Every syntax package defines a set of *productions*, each consisting of a number of *transformation rules*. The following syntax:

```
defproduction my-production: ProductionType
```

defines a production named `my-production` that returns a value of type `ProductionType` when matched.

To define a transformation rule for a production, we use the `defrule` construct:

```
defrule my-production = (pattern ...) :
  body
```

The above code defines a transformation rule for the `my-production` production indicating to return the result of evaluating `body` if the input matches the pattern given by `pattern`. Note that the value returned by `body` must agree with the type specified by the production definition – which, in this case, is `ProductionType`.

A *failure rule* is a special transformation rule, and is defined using the `fail-if` construct:

```
fail-if my-production = (pattern ...) :
  body
```

The above specifies that the `my-production` production should *never* match the given pattern. If it does, then the input is badly formed, and `body` is evaluated to obtain an `Exception` value that describes the error. Failure rules are used to produce descriptive error messages for badly formed input.

To refer to productions defined in other syntax packages, we use the `import` construct.

```
import (type, exp) from core
```

The above code imports the `type` and `exp` productions from the `core` syntax package.

3.4 Pattern Syntax

Patterns are expressed in a variant of Backus-Naur form [5] (BNF) extended to describe s-expressions instead of just flat sequences of tokens. Here we will show some examples of the syntax used for defining patterns.

A symbol pattern matches against a symbol if they are the same symbol.

```
hello world matches against:
  hello world
```

An underscore pattern matches against any s-expression.

```
hello _ world matches against:
  hello x world
  hello 42 world
  hello (a b c) world
```

A list pattern matches against a list if the patterns within the list pattern match the contents of the list.

```
a (hello _ world) b matches against:
  a (hello x world) b
  a (hello (a b c) world) b
```

An ellipsis pattern matches against zero or more repeated occurrences of the pattern.

```
a (x ...) b matches against:
  a () b
  a (x) b
  a (x x x x) b

a ((x _) ...) b matches against:
  a () b
  a ((x 2)) b
  a ((x 2) (x hello) (x (y z))) b
```

A splice-ellipsis pattern matches against zero or more repeated occurrences of the *contents* of a list pattern.

```
a ((x _) @...) b matches against:
  a () b
  a (x 2) b
  a (x 2 x hello x (y z)) b
```

Ellipsis patterns can be nested.

```
a ((x (y z) @...) ...) b matches against:
  a () b
  a ((x) (x y z)) b
  a ((x) () (x y z y z y z)) b
```

A binder pattern matches against the given pattern, and then binds the result of matching against that pattern to the given binder. This binding can then be referred to in the rule body.

```
a ?xs:((x (y z) @...) ...) b matches against:
  a () b
    with xs bound to: ()
  a ((x) (x y z)) b
    with xs bound to: ((x) (x y z))
  a ((x) () (x y z y z y z)) b
```

```
    with xs bound to: ((x) () (x y z y z y z))

(do ?x:_ ?y:_) ... matches against:
  (do a b) (do c d)
    with x bound to: (a c)
    and y bound to: (b d)
  (do a b) (do (1 2 3) z) (do x y)
    with x bound to: (a (1 2 3) x)
    and y bound to: (b z y)
```

A production pattern matches if any of its transformation rules match. The result of the match is the result of evaluating the matched transformation rule.

```
a (#exp) b matches against:
  a (d x) b
    if the exp production matches against: d x
```

3.5 Stanza's Core Macros

Stanza's standard library comes included with a large collection of macros defined in the `core` syntax package for implementing the standard constructs. The important productions defined in `core` are `type` and `exp`:

1. The `type` production matches against s-expressions for expressing Stanza types. The following lists some examples:

   ```
   Int
   Int|String
   Int -> String
   (Int, Array<Int>) -> String|False
   ```

2. The `exp` production matches against s-expressions for expressing Stanza expressions. The following lists some examples:

   ```
   x + y
   f(x, y)
   match(x, y) :
     (x:Int, y:String) : body1
     (x:String, y:Int) : body2
   defn f (x, y) :
     body
   ```

   For handling operator precedence, the `exp` production internally relies upon the `exp0`, `exp1`, `exp2`, `exp3`, and `exp4` productions that match against expressions at the 0th through 4th precedence levels.

3.6 Example

The while construct allows programmers to repeatedly execute some body as long as a given predicate evaluates to true.

```
while not empty?(xs) :
  val x = next(xs)
  println(x)
```

The above code repeatedly checks whether empty?(xs) returns true. If it does not, then it prints the result of next(xs).

The following example shows the implementation of the while macro:

```
defrule exp4 = (while ?pred:#exp : ?body:#exp) :
  val template = ‘(
    let :
      defn* loop () :
        if (pred upcast-as core/True|core/False) :
          body
          loop()
      loop())
  parse-syntax[core / #exp](
    fill-template(template, [
      ‘loop => gensym(‘loop)
      ‘pred => pred
      ‘body => body]))
```

The transformation rule is defined for the exp4 production, indicating that it is a Stanza expression at the 4th precedence level. In order to match, the pattern requires a while symbol, followed by an exp production representing the predicate, followed by the : symbol, followed by another exp production representing the body.

The template value shows the transformed code that the macro should expand into. The loop is implemented by defining and then calling a tail-recursive function. The fill-template utility function is used to substitute pred with the predicate expression, and body with the body expression. The loop symbol is substituted with a unique symbol in order to avoid accidental capture or shadowing.

Note that the transformed code is, itself, expressed in terms of more macros – such as let and if. The parse-syntax[core / #exp](...) syntax indicates that the resulting substituted s-expression should continue to be expanded using the exp production in the core package.

Note that the body of the macro is written using arbitrary Stanza code. The while macro is a straightforward syntactic shorthand and the implementation is simple, but macros can be much more involved. The implementation of some macros resemble miniature compilers for a small language. The defsyntax, defproduction, and defrule constructs are macros themselves, for instance, and take nearly two thousand lines to implement.

3.7 Staged Compilation

One fundamental design problem that arises in languages that support programmatic macros is the order and time at which macros execute. This is especially complicated for languages with interpreter semantics that execute a program line-by-line. The following lists just a few of the issues that need to be carefully considered:

- If the evaluation of a macro contains side-effects (such as printing a message to the screen), when should these side-effects take place?
- If interpreting a program line-by-line, it seems natural to first expand all the macros in the line, and then execute the resulting core forms. But then how would the program behave when compiled? If we expanded all macros in the program before compilation, then the same program would behave differently when compiled versus when interpreted.
- Suppose we choose to expand all macros before compilation. Assume that `x` is a global value:

  ```
  val x = f(42)
  ```

  and that `x` is referenced from within the body of our macro transformer. When should `x` be computed? The value of `x` is needed for proper functioning of the macro. But `f(42)` may be an operation that should not execute until runtime!

Common Lisp [32] works around the above issues by pushing the responsibility to the programmer. The `eval-when` form provides the programmers fine control over when forms are evaluated, but it is a notoriously hard-to-use construct.

Stanza follows a different philosophy for its macro system. Stanza is, at its heart, a compiled language, and adopts a staged compilation approach for handling macro expansion. Macro definitions can *never* be declared and used immediately in the same compilation step. Instead, the standard Stanza compiler can be *extended* with additional syntax packages to conceptually produce a *new compiler* that is then able to understand the new constructs. Hence, code that requires new syntactic constructs are explicitly separated into two phases: a set of syntax packages that extend the standard compiler, and the code that then relies upon the new syntax provided by the extended compiler.

Any side-effects that occur due to expansion of a macro happens during the compilation phase, and the resulting compiled program executes only the core forms that remain after macro expansion.

3.8 Relationship to Other Syntax Frameworks

The goal of Stanza's macro system is to provide programmers a natural readable syntax along with the tools to easily implement their own syntactic abstractions.

The first property to note is that our system is intentionally limited in its flexibility for supporting different syntaxes. For example, no amount of macros will allow users to program Stanza using Java's [2] syntax. The lexer rules – which controls the syntax for comments, for literals, for indentation, etc. – are fixed and cannot be changed through macros. Our objective is to give the *minimum* amount of flexibility to programmers such that they can design a *reasonable syntax* for a useful construct. Towards this goal, Stanza differs greatly from parser generator frameworks such as ANTLR [44] and Bison [15] that provide maximal flexibility in order to parse a wide variety of *existing* syntaxes. We make the deliberate decision to trade flexibility for better composability and ease-of-use.

The engine that drives macro expansion is heavily inspired by parser generators based upon parsing expression grammars [23] (PEGs). The macro engine is differentiated from PEGs by one critical aspect: the patterns and inputs are in terms of s-expressions instead of in terms of a flat sequence of lexed tokens or sequence of characters. This imposes an additional structure to the design of syntaxes. Macros naturally respect the scoping rules that fall out of the hierarchical s-expression datastructure. Consider the following example in which the `while` construct appears within the consequent clause of the `if` construct:

```
if x < 3 :
  while x < 10 :
    println(x)
    x = f(x)
else :
  println(x)
```

The `while` construct exists within its own list. Unlike a general parsing system, Stanza guarantees that the `while` macro is localized to within the `if` construct. The parsing of the `if` macro is completely shielded from the contents of the consequent and the alternate clauses.

One other major advantage of operating on s-expressions instead of on flat sequences of tokens is the resulting ease of implementing accurate error detection and recovery algorithms. In order to detect more than a single error at a time, a parser must be able to skip badly formed input and resume parsing at a "safe" position – a challenging practical problem. By operating on s-expressions, the parser can simply give up parsing within the current list, and resume from another.

Compared to traditional Lisp [32] macros, Stanza's macro system has the advantage that multiple s-expressions can map to a single core form, whereas Lisp makes the assumption that there is a one-to-one mapping between input s-expressions and core forms. This is the key distinguishing feature that allows Stanza to easily implement infix operators. Consider the following list of s-expressions:

```
3           expands to:   3
3 + 4       expands to:   ($do plus 3 4)
3 + 4 - 5   expands to:   ($do minus ($do plus 3 4) 5)
```

Although each line contains a different number of s-expressions – the first contains one, the second contains three, and the third contains five – they all expand into a single core form. The combination of this feature and the small set of lexer shorthands allows Stanza to provide a natural and familiar syntax to programmers, whereas Lisp has always been criticised for its unappealing syntax.

Chapter 4

The Stanza Type System

Stanza supports an optional type system that offers the advantages of both dynamically-typed and statically-typed languages. During the prototyping stage of development, optionally-typed languages provide flexibility and productivity on par with dynamically-typed languages; and also offer the early error detection capabilities of statically-typed languages. Programmers *incrementally* add type annotations to their untyped code in order to gradually increase the number of errors that can be statically caught by the compiler.

4.1 The Promises of Optional Typing

The intuition behind our optional type system is straightforward, and begins with the observation that the semantics of static and dynamic typing do not actually logically contradict each other. It is possible to design a single semantics that can exhibit both sets of behaviours.

In an optionally-typed language, a user would begin a new project by programming without providing any type annotations. In this usage mode, the language would both look and behave like a dynamically-typed language. The language is at its least restrictive, but errors are not detected until execution.

As the design stabilizes, the user is free to incrementally add type annotations to mature interfaces. This mixed-typed mode is the most common usage mode of an optionally-typed language. Some, but not all, of the binders have explicit type annotations that indicate their intended usage. For the annotated binders, the compiler ensures that they are used in a way that is consistent with their annotation and issues a typechecking error if not. For the unannotated binders, errors are left to be detected during execution as before.

Finally, as the project nears completion, the user can provide type annotations for all binders in the program, in which case the language behaves identically to a fully statically-typed language.

When working in an optionally-typed language, the transition from a dynamically-typed to a statically-typed paradigm is a smooth and continuous one, and, most importantly, is not monotonic. Users are able to freely add *and remove* as many or as few type annotations as is appropriate for the project. Even on top of a fully statically-typed code base, users are free to develop new functionality using a completely dynamically-typed coding style. It is all governed by the same semantics underneath.

4.2 Desired Characteristics

There has been many attempts at hybrid dynamic-static type systems in the past, though sometimes with different goals. For enabling the specific development style outlined above, where programmers freely and fluidly move between paradigms to suit their needs, there are three important characteristics that we consider desirable from an optional type system:

1. The type system must allow typed and untyped (a.k.a. dynamically-typed) code to be mixed freely and with fine granularity. We want users to be able to freely choose in which paradigm to write different functions in the same package, write different sections in the same function, and even annotate different binders in the same section.

2. Both paradigms must be equally regarded, and no paradigm should dominate over the other. One problem that plagued earlier attempts at hybrid systems, such as that by Abadi et al. [1], was that either the static or dynamic paradigm would be *invasive* and gradually take over the code base. For example, users would prototype the program in the dynamic paradigm, but then immediately after adding the first static type annotation, the system would issue an error indicating that it is illegal to pass an untrusted value from the dynamic paradigm into the trusted context of the static paradigm.

   The user would then have to insert additional type annotations to prove that the value can safely be passed into the trusted context, but this would simply lead to more errors indicating that untrusted values are now being passed into these newly annotated static contexts. Eventually, the user is forced to annotate significant portions of their program before it passes the typechecker. To avoid this phenomenon, it is important that both the dynamic and the static paradigms be *non-invasive*.

3. Users should be able to migrate code from an untyped to typed paradigm simply by adding type annotations, with no *structural* changes required. As an example, consider porting a program from Python [47] to Java [2]. Such a port *cannot* typically be done by merely adding type annotations and changing the syntax. Often the code must be rewritten to conform to a different logical structure. This is because many Python idioms cannot be expressed under the restrictions of Java's static type system.

   For example, Python allows variables to take on values of two different types, and Java does not. In Python, to call a function that requires an array of integers, it is sufficient to ensure that the given array contains only integers at the time of the call. In Java, the same function must be called with an array that was *originally* created to store only integers.

> We must minimize the need for structural changes if users are to be able to seamlessly transition between paradigms as envisioned. To achieve this, the type system must be carefully designed to be expressive enough to type the idioms that are prevalent in dynamically-typed programming languages.

4.3 Example Interaction

Before diving into the technical details of the type system, we first demonstrate here an example interaction that shows the development style made possible by Stanza.

Untyped Code

The following shows an example of untyped Stanza code:

```
defstruct Clothes : (sort, clean?)
defstruct Yarn : (length)
defstruct Person : (name, clothes)

defn wash (c) :
  println("Washing clothes")
  println("It’s %_" % [sort(c)])
  println("Br...")
  Clothes(sort(c), true)

defn knit (y) :
  for i in 0 to 10 do :
    if i < length(y) :
      println("Knitting (%_ cm used up so far)" % [i])
    else :
      println("Out of yarn!")
  Clothes("a tshirt", true)

defn wear (p, c) :
  println("I am %_" % [name(p)])
  println("I am wearing %_" % [sort(clothes(p))])
  println("Now I’m wearing %_" % [sort(c)])
  Person(name(p), c)

val tshirt = knit(Yarn(100))
val patrick = Person("Patrick", Clothes("nothing", false))
wash(tshirt)
wear(patrick, tshirt)
```

A `Clothes` object contains two fields, `sort`, and `clean?`; a `Yarn` object contains a single `length` field; and a `Person` object contains a `name` and a `clothes` field. The `wash` function takes a single argument, `c`, which is assumed to be a `Clothes` object, and prints out some messages along with the sort of clothing it is. `wash` returns a new `Clothes` object of the same sort but with clean? set to true. The knit function takes a single argument, y, assumed to be a Yarn object, prints out a message in a loop, and finally returns a Clothes object representing a clean t-shirt. The wear function takes two arguments, p and c, assumed to be a Person and a Clothes object, and prints out the name of the person, the sort of clothing he is currently wearing, and the sort of clothing represented by c. It returns a new Person object with the same name but now wearing c. At the top-level, the tshirt value is initialized to the result of calling knit with 100 units of Yarn. The patrick value is initialized to a Person named "Patrick" that is currently wearing "nothing". We then call wash on tshirt, and later, wear, on patrick and tshirt.

Note that there is not a single type declaration in the code. In this usage mode, Stanza looks and behaves like a dynamically-typed language. The following shows the output of running the code:

```
Knitting (0 cm used up so far)
Knitting (1 cm used up so far)
Knitting (2 cm used up so far)
Knitting (3 cm used up so far)
Knitting (4 cm used up so far)
Knitting (5 cm used up so far)
Knitting (6 cm used up so far)
Knitting (7 cm used up so far)
Knitting (8 cm used up so far)
Knitting (9 cm used up so far)
Washing clothes
It’s a tshirt
Br...
I am Patrick
I am wearing nothing
Now I’m wearing a tshirt
```

Runtime Errors

When being used as a dynamically-typed language, Stanza does not detect errors until execution. Suppose that the programmer makes the following mistake: instead of calling wash on tshirt, the programmer calls wash on the integer 42:

```
val tshirt = knit(Yarn(100))
val patrick = Person("Patrick", Clothes("nothing", false))
wash(42)
wear(patrick, tshirt)
```

The following shows the output of executing the incorrect code:

```
Knitting (0 cm used up so far)
Knitting (1 cm used up so far)
Knitting (2 cm used up so far)
Knitting (3 cm used up so far)
Knitting (4 cm used up so far)
Knitting (5 cm used up so far)
Knitting (6 cm used up so far)
Knitting (7 cm used up so far)
Knitting (8 cm used up so far)
Knitting (9 cm used up so far)
Washing clothes
FATAL ERROR: Expected argument of type Clothes but got Int.
   at trycode.stanza:1.21
   at trycode.stanza:7.23
   at trycode.stanza:27.0
```

By inspecting the stack trace, we can determine that the error occured at the call to `sort(c)` in the `wash` function. The error message indicates that the function `sort` is expecting a `Clothes` object, but it was incorrectly called with an `Int` object (namely the value `42`).

Mixed-Typed Code

Upon seeing the error message, the programmer may have the following thought: "`c` should be a `Clothes` object, because `wash` is expected to be called with a `Clothes` object. So let us make this expectation explicit with a type annotation." Here is the `wash` function with the additional annotation:

```
defn wash (c:Clothes) :
  println("Washing clothes")
  println("It's %_" % [sort(c)])
  println("Br...")
  Clothes(sort(c), true)
```

With this type annotation, the program now becomes an example of using Stanza as a mixed-typed language. Attempting to compile the code now results in the following error:

```
trycode.stanza:27.0: Cannot call function
  wash
of type
  Clothes -> Clothes
with arguments of type
  (Int).
```

The error indicates that the call to `wash(42)` is incorrect. The `wash` function expects an argument of type `Clothes`, but it was called with an argument of type `Int`. Based on this error message, the programmer can then quickly locate and fix the source of the mistake.

Typed Core Libraries

It is worth pointing out mixed-typed code is the most common sort of Stanza code. There are extremely few Stanza programs that are completely untyped. Even though the type annotation on `wash` is the sole *user* type annotation in our previous example, note that Stanza's core libraries are mature and fully statically-typed.

Suppose that the programmer made a mistake in the implementation of `knit` and called `length` on `y` instead of on `i`:

```
defn knit (y) :
  for i in 0 to 10 do :
    if length(i) < y :
      println("Knitting (%_ cm used up so far)" % [i])
    else :
      println("Out of yarn!")
  Clothes("a tshirt", true)
```

Despite the lack of type annotations in the `knit` function, Stanza is still able to detect the mistake using the type annotations in its core library. Here is the resulting compilation error:

```
trycode.stanza:13.7: No appropriate function
  length
for arguments of type
  (Int).

Possibilities are:
  length: Lengthable -> Int
    at core/core.stanza:1467.16
  length: RandomAccessFile -> Long
    at core/core.stanza:2037.21
  length: Yarn -> ?
    at trycode.stanza:2.18
```

The error says that there are three overloaded definitions of the `length` function. The first two are defined in the `core` library and accept arguments of types `Lengthable` and `RandomAccessFile` respectively. The last one is the `length` getter function for the `Yarn` object. None of those definitions can be appropriately called with an argument of type `Int`, which is the type that Stanza inferred for `i`.

This example demonstrates a particularly exciting characteristic of optionally-typed languages which hints that optional typing may be more productive than both fully dynamically-typed and fully statically-typed languages. During the prototyping stage, Stanza offers flexibility on par with dynamically-typed languages for the *user's* code. However, incorrect usages of *trusted* code are still detected automatically.

Typed Code

Finally, as the program matures and stabilizes, the programmer may choose to insert explicit type annotations for all binders. In this mode, Stanza acts as a fully statically-typed language.

```
defstruct Clothes : (sort:String, clean?:True|False)
defstruct Yarn : (length:Int)
defstruct Person : (name:String, clothes:Clothes)
```

```
defn wash (c:Clothes) :
  println("Washing clothes")
  println("It's %_" % [sort(c)])
  println("Br...")
  Clothes(sort(c), true)

defn knit (y:Yarn) :
  for i in 0 to 10 do :
    if i < length(y) :
      println("Knitting (%_ cm used up so far)" % [i])
    else :
      println("Out of yarn!")
  Clothes("a tshirt", true)

defn wear (p:Person, c:Clothes) :
  println("I am %_" % [name(p)])
  println("I am wearing %_" % [sort(clothes(p))])
  println("Now I'm wearing %_" % [sort(c)])
  Person(name(p), c)

val tshirt = knit(Yarn(100))
val patrick = Person("Patrick", Clothes("nothing", false))
wash(tshirt)
wear(patrick, tshirt)
```

The above code is fully statically-typed, where explicit type annotations have been inserted for all field declarations and all function arguments. (The `tshirt` and `patrick` values have their types inferred automatically and do not need annotations.)

Prototyping Additional Features

The next interaction demonstrates the fluidity with which users can move between paradigms in Stanza. The program is mature and fully statically-typed, and we now consider prototyping new features on top of a typed code base.

The following example adds the additional function `patch` to the existing program. `patch` is coded in a dynamically-typed style without any explicit type annotations.

```
defstruct Clothes : (sort:String, clean?:True|False)
defstruct Yarn : (length:Int)
defstruct Person : (name:String, clothes:Clothes)

defn wash (c:Clothes) :
  ...

defn knit (y:Yarn) :
  ...

defn wear (p:Person, c:Clothes) :
  ...
```

```
defn patch (c, y) :
  println("Patching %_ with %_ cm of yarn." % [
    sort(c), length(y)])

val tshirt = knit(Yarn(100))
val patrick = Person("Patrick", Clothes("nothing", false))
wash(tshirt)
patch(tshirt, Yarn(10))
wear(patrick, tshirt)
```

Notice that no effort is required to enable the new function to interact with the rest of the typed code base.

When the `patch` function stabilizes, the programmer may then insert type annotations to bring it to the same level of reliability as the rest of the code base. Stanza allows users to freely move from the untyped to typed paradigm *and vice versa* to suit the project needs.

4.4 Overview of the Type System

There are four key components to the design of our type system:

1. the nominal subtyping framework,
2. the `?` type,
3. support for parametric types, and
4. type inference.

We chose the nominal subtyping framework as the foundation upon which to build our type system. In this framework, the programmer specifies the set of types in the program and the set of subtyping relations between the types. As an example the following table shows a partial categorization of the animal kingdom and their subtyping relations.

Types	Subtyping Relations
`Animal`	`Mammal <: Animal`
`Mammal`	`Fish <: Animal`
`Fish`	`Salmon <: Fish`
`Salmon`	`Tuna <: Fish`
`Tuna`	`Dog <: Mammal`
`Dog`	`Cat <: Mammal`
`Cat`	`Tabby <: Cat`
`Tabby`	

The subtype operator (`<:`) can be informally read as "is a type of." So the statement `Salmon <: Fish` can be read as "a `Salmon` is a type of `Fish`". The most important use for the subtype relation is in checking the legality of function calls. Given a function that is annotated to require an argument of type `Fish`, a subtyping framework would allow the function to be called with any value of type `Fish` or subtype of `Fish`.

We chose to build our type system on top of the nominal subtyping framework for two major reasons. The first is that it is the framework that is used by the type systems of C++ [55], C# [27], and Java [2], three of the most popular statically-typed languages used in industry. Thus a programmer experienced with any of the three languages should feel at ease with our system.

The second reason concerns the coding style prevalent in untyped languages. Python [47], Ruby [22], and Javascript [18] are three of the most popular dynamically-typed languages used in industry. Despite having no static type systems, we found that programmers code and architect their programs as if they were in a nominal subtyping framework. For instance, Python code tends to resemble Java code more than it resembles OCaml [35] or Haskell [31] code. This second reason is important because it means that typical untyped code can be more easily migrated to a nominal subtyping framework than to, for example, a Hindley-Milner [29] style type system.

The `?` type is the key mechanism underlying our optional type system, and is what allows us to model the semantics of dynamically-typed languages. It is governed by two rules:

1. An expression of type `?` is allowed to be passed to any context.

2. An expression of any type is allowed to be passed to a context expecting a `?` type.

Python can be thought of as having our optional type system but where every binder has been annotated with the `?` type.

Parametric types allow us to express types that are parameterized by other types – like an array *of* integers, or a list *of* strings. To support this, we allow for types to optionally accept type parameters, and for functions to optionally accept type arguments.

Parametric types are crucial for increasing the range of errors that can be statically caught by the typechecker. Generic collections, such as arrays, lists, and tables, are prevalently used in daily programming. Parametric types allow the type checker to check whether values to be stored into a collection are of the correct type, and whether values retrieved from a collection are used appropriately.

Finally, to make the system convenient to use, Stanza's inference algorithm allows the programmer to elide type annotations for a number of constructs.

4.5 The Nominal Subtyping Framework

The following lists every type that is supported by Stanza:

```
Named Types: A<T1, ..., Tn>
Tuple Types: [T1, ..., Tn]
Union Type: T1|T2
Intersection Type: T1&T2
Arrow Type: (T1, ..., Tn) -> Tr
Bottom Type: Void
Unknown Type: ?
Type Variable: T
```

Named types refer to a type that has been declared using the `deftype` construct. It is the most frequent type seen in daily programming. Some common examples are `Int`, `String`, `True`, `False`, and `Array<Int>`. Note that if the type parameters for a named type are not given, then they are assumed to be `?` by default. Thus the `List` type is equivalent to `List<?>`.

A tuple type denotes tuple values where the arity and type of each component is statically known. For instance, the type `[Int, String]` denotes all 2-arity tuples containing an `Int` followed by a `String`. Tuple types are often used to denote the return types for functions that return multiple values. The following is an example of a function, `quad-root`, that returns a tuple containing the two solutions to a quadratic equation:

```
defn quad-root (a:Float, b:Float, c:Float) -> [Float, Float] :
  val determinant = b * b - 4 * a * c
  [((- b) + sqrt(determinant)) / (2 * a),
   ((- b) - sqrt(determinant)) / (2 * a)]
```

Stanza provides special support for destructuring a tuple to make it convenient to call the above function:

```
val [r1 r2] = quad-root(2.0f, 10.0f, 1.0f)
```

Union types arise naturally as the resulting type of a `match` expression. Consider the following code:

```
val x = match(y) :
  (y:Int) : 42
  (y:String) : "Hello"
  (y:Char) : ’z’
```

where the value `x` is initialized to either the integer `42`, the string `"Hello"`, or the character `’z’`, depending on whether the type of `y` is an `Int`, `String`, or `Char`. The inferred type of `x` will be `Int|String|Char` to indicate that it may take on values of any of those types.

The intersection type indicates that a value must be *simultaneously* of two types. For example, a `Collection<Int>` represents an abstract collection containing `Int` values; and a `Lengthable` represents a value with a length property. The intersection type `Collection<Int> & Lengthable` represents a value that is *both* a collection of `Int` values and also has a length property. The following definition of an integer array type:

```
deftype IntArray <: Collection<Int> & Lengthable
```

states that `IntArray` can be such a value.

The arrow type indicates that a value must be a function. The type (Int, String) -> Int represents a two-arity function that accepts an Int and a String and returns an Int.

One other common situation where the intersection type arises are functions that take a variable number of arguments. The following function takes either one or three arguments:

```
multifn :
  (x:Int) : x
  (x:Int, y:String, z:Int) : append(to-string(x + z), y)
```

The inferred type for the above would be:

```
(Int -> Int) & ((Int String Int) -> String)
```

to reflect that it can either be called with an Int, in which case it will return an Int; or it can be called with an Int, a String, and an Int, in which case it will return a String.

The Void type is special in that there are *no* values of this type. Consider the following function, which sets a flag and then throws an exception:

```
defn flag-error () :
  ERROR-OCCURRED = true
  throw(Exception("Flagged Error"))
```

What is the type of the values returned by flag-error? Well, from inspecting the function body, we see that flag-error? *doesn't return* any values. Thus flag-error is declared to have a return type of Void.

The unknown type, ?, is the basic mechanism used by Stanza to model untyped code. A binder annotated with the ? type indicates to the compiler that there is no static information about the type of that binder, and to allow it to be used freely.

Type variables occur in the definition of parametric polymorphic functions, and will be discussed in greater depth later.

Defining the Type Hierarchy

Named types are defined using the deftype construct. The general syntax is as follows:

```
deftype MyType<T1, ..., Tn> <: Parent1 & Parent2 & ... & Parentm
```

The name of the type is given as MyType, and is parameterized by n type parameters: T1 through Tn. The angle brackets (<>) can be omitted for types without any type parameters. MyType is explicitly stated to be a subtype of m parent types: Parent1 through Parentm. The type parameters can be referenced by the parent types. The <: is omitted for types that are not subtypes of any other type.

The following shows an example of the deftype expressions necessary to represent the type hierarchy representing our animal kingdom:

```
deftype Animal
deftype Mammal <: Animal
deftype Dog <: Mammal
deftype Cat <: Mammal
deftype Tabby <: Cat
deftype Fish <: Animal
deftype Salmon <: Fish
deftype Tuna <: Fish
```

All animals fall under the `Animal` type, which has two direct subtypes: `Mammal` and `Fish`. `Mammal` can be further classified as `Dog` or `Cat`, and `Cat` has one direct subtype: `Tabby`. `Fish` can be further classified as `Salmon` or `Tuna`. None of the animal types are parameterized.

The following shows an example of a parametric type:

```
deftype Array<T> <: Collection<T> & Lengthable
```

An array has one type parameter, `T`, for indicating the type of its contents. It is explicitly stated to be a subtype of `Collection<T>` and `Lengthable`, indicating that an array is an abstract collection and has a length.

Subtyping Relation

Figure 4.1 shows the inference rules defining Stanza's subtyping relation. The relation:

```
X <: Y
```

means that the type `X` is a subtype of the type `Y`. The subtype relation is the most important relation underlying Stanza's typechecker. A value of type `X` is allowed to be passed to a location expecting a value of type `Y` if and only if `X <: Y`.

Named$_1$ and Named$_2$ define the subtyping rules for named types. Named$_1$ defines a named type to be a subtype of another named type if they refer to the same type, and if the type parameters of the first are respectively subtypes of the type parameters of the second. Note that this means that all parametric types in Stanza are *covariant.* This is an instance of where we deliberately chose to sacrifice some static type safety for ease-of-use.

Named$_2$ specifies how the subtyping relation is defined relative to the type hierarchy. A named type is a subtype of another type, `X`, if its parent is a subtype of `X`. To compute its parent we assume that there is an entry in the type hierarchy of the form:

```
deftype A<T1, ..., Tn> <: P
```

The parent is computed by replacing the type variables `T1, ..., Tn` with the respective type parameters `S1, ..., Sn`.

Tuple$_1$ and Tuple$_2$ define the subtyping rules for tuple types. Tuple$_1$ specifies that a tuple type is a subtype of another tuple if they have the same arity, and if the elements of the first are respectively subtypes of the elements of the second.

Tuple$_2$ allows a tuple of known arity to be interpreted as a tuple of unknown arity. The core Stanza library defines the `Tuple` type to be a subtype of, among others, `Collection` and `Lengthable`. Tuple$_2$ allows Stanza to derive, for example, that `[Int, String]` is a subtype of `Collection<Int|String>`.

Union$_1$, Union$_2$, and Union$_3$ define the subtyping rules for the union type. Union$_1$ specifies that in order for a union type `T1|T2` to be a subtype of another type, `X`, *both* `T1`

`X <: Y` means that `X` is a subtype of `Y`.

$$\frac{\texttt{T1 <: S1} \quad \ldots \quad \texttt{Tn <: Sn}}{\texttt{A<T1, ..., Tn> <: A<S1, ..., Sn>}}\ \textsc{Named}_1 \qquad \frac{\begin{array}{c}\texttt{deftype A<T1, ..., Tn> <: P} \\ \texttt{P[T1 := S1, ..., Tn := Sn] <: X}\end{array}}{\texttt{A<S1, ..., Sn> <: X}}\ \textsc{Named}_2$$

$$\frac{\texttt{T1 <: S1} \quad \ldots \quad \texttt{Tn <: Sn}}{\texttt{[T1, ..., Tn] <: [S1, ..., Sn]}}\ \textsc{Tuple}_1 \qquad \frac{\texttt{Tuple<T1|...|Tn> <: X}}{\texttt{[T1, ..., Tn] <: X}}\ \textsc{Tuple}_2$$

$$\frac{\texttt{T1 <: X} \quad \texttt{T2 <: X}}{\texttt{T1|T2 <: X}}\ \textsc{Union}_1 \qquad \frac{\texttt{X <: T1}}{\texttt{X <: T1|T2}}\ \textsc{Union}_2 \qquad \frac{\texttt{X <: T2}}{\texttt{X <: T1|T2}}\ \textsc{Union}_3$$

$$\frac{\texttt{T1 <: X}}{\texttt{T1\&T2 <: X}}\ \textsc{Intersection}_1 \qquad \frac{\texttt{T2 <: X}}{\texttt{T1\&T2 <: X}}\ \textsc{Intersection}_2 \qquad \frac{\texttt{X <: T1} \quad \texttt{X <: T2}}{\texttt{X <: T1\&T2}}\ \textsc{Intersection}_3$$

$$\frac{\texttt{S1 <: T1} \quad \ldots \quad \texttt{Sn <: Tn} \quad \texttt{Tr <: Sr}}{\texttt{(T1, ..., Tn) -> Tr <: (S1, ..., Sn) -> Sr}}\ \textsc{Arrow} \qquad \frac{}{\texttt{Void <: X}}\ \textsc{Void}$$

$$\frac{}{\texttt{T <: T}}\ \textsc{TypeVar} \qquad \frac{}{\texttt{? <: X}}\ \textsc{Unknown}_1 \qquad \frac{}{\texttt{X <: ?}}\ \textsc{Unknown}_2$$

Figure 4.1: Stanza Subtyping Relation

and `T2` need to be a subtype of `X`. UNION$_2$ and UNION$_3$ together say that in order for a type `X` to be a subtype of a union type `T1|T2`, it is sufficient for `X` to be a subtype of either `T1` or `T2`.

INTERSECTION$_1$, INTERSECTION$_2$, and INTERSECTION$_3$ define the subtyping rules for the intersection type. INTERSECTION$_1$ and INTERSECTION$_2$ together say that an intersection type, `T1&T2`, is a subtype of another type, `X`, if either `T1` or `T2` is a subtype of `X`. INTERSECTION$_3$ says that in order for a type `X` to be a subtype of an intersection type `T1&T2`, `X` needs to be a subtype of both `T1` and `T2`.

The UNION and INTERSECTION rules are taken directly from the textbook *Types and Programming Languages* [45].

The ARROW rule defines the subtyping relation between two arrow types and is standard. An arrow type is a subtype of another if they have the same arity, the argument types of the second are subtypes of the argument types of the first, and if the return type of the first is a subtype of the return type of the second.

The VOID rule says that the bottom type, `Void`, is a subtype of all other types.

The TYPEVAR rule says that the subtyping relation for type variables is reflexive, and a type variable is a subtype of itself.

The UNKNOWN$_1$ and UNKNOWN$_2$ rules define the subtyping relation for the unknown type. The ? type is a subtype of all types, and all types are a subtype of the ? type.

4.6 Polymorphic Functions and Captured Type Parameters

Stanza supports second-class polymorphic parametric functions by allowing named functions to accept type parameters. The following example shows the declaration of a general reduction function, myreduce, that is able to accept lists of any type:

```
defn myreduce<T> (f: (T, T) -> T, xs: List<T>) -> T :
  if empty?(tail(xs)) : head(xs)
  else : f(head(xs), myreduce(f, tail(xs)))
```

The myreduce function takes one type parameter, T, and two normal arguments, f and xs. f is a function that computes a new T given two T values, and xs is a list of T values. The function merges all the values in xs into one by calling f repeatedly on each element in xs.

By parameterizing myreduce over the type T, users can call myreduce to reduce lists of *any* type. The following examples shows using myreduce to implement a sum and append-all function:

```
defn my-sum (xs: List<Int>) -> Int :
  myreduce<Int>(plus, xs)

defn my-append-all (xs: List<List<Int>>) -> List<Int> :
  myreduce<List<Int>>(append, xs)
```

my-sum computes the sum of a list of integers by calling myreduce with the type argument Int and the merging function plus. my-append-all computes the concatenation of a list containing lists of integers by calling myreduce with the type argument List<Int> and the merging function append.

Type Parameter Inference and the Array Store Problem

Note that the previous example explicitly provided the type arguments Int and List<Int> in the calls to myreduce. It is an error to call a polymorphic function without the exact number of required type arguments.

From looking at the definition of my-sum and my-append-all, however, it is easy to deduce that xs in my-sum has type List<Int>, and that xs in my-append-all has type List<List<Int>>. Why is it that the compiler cannot automatically infer the type arguments in the call to myreduce by inspecting the types of the arguments it is called with? It seems straightforward to infer the necessary type argument to make the program typecheck successfully.

The reason is due to Stanza's subtyping relation, which states that all parametric types are *covariant* in their type parameters. Due to this rule, Stanza cannot make use of existing type argument inference algorithms. To illustrate the critical problem, let us assume that the type `Dog` is a subtype of `Animal`, and consider the following function for storing a value to the fifth slot in an array:

```
defn store-5th<T> (xs:Array<T>, v:T) :
  xs[4] = v
```

What type argument should be inferred in a call to `store-5th` with arguments of type `Array<Animal>` and `Dog`?

```
val animals : Array<Animal> = ...
val dog : Dog = ...
store-5th<???>(animals, dog)
```

The answer seems trivially to be `T` = `Animal`. The first argument, `Array<Animal>`, can trivially be passed to a location expecting an `Array<Animal>`, and due to subtyping, the second argument, `Dog`, can be passed to a location expecting an `Animal`. This behaviour is consistent with our expectations and the program typechecks successfully. A dog *should* be able to be stored to the fifth slot of an array of animals.

The problem arises when `store-5th` is called with arguments of type `Array<Dog>` and `Animal`:

```
val dogs : Array<Dog> = ...
val animal : Animal = ...
store-5th<???>(dogs, animal)
```

What type should be inferred for `T` to make the program typecheck successfully? The answer *remains* `T` = `Animal`. The first argument, `Array<Dog>`, can be passed to a location expecting an `Array<Animal>` because Stanza's parametric types are covariant. The second argument, `Animal`, can trivially be passed to a location expecting an `Animal`. The program continues to typecheck successfully, which is *inconsistent* with our expectations. An arbitrary animal *should not* be able to be stored to the fifth slot of an array of dogs. We will refer to this as the *array store problem*.

Existing Solutions to the Array Store Problem

The array store problem described previously extends beyond just type parameter inference and is a general problem that arises from allowing mutable types (such as arrays) to be covariant. Stanza, along with Eiffel [39], Java [2], C# [27], and Dart [6], among others, deliberately chose for arrays to be covariant for a simple reason: it is intuitive and the behaviour that is expected by users.

Nonetheless, the array store problem needs to be addressed somehow. A type system that allows an animal to be stored in an array of dogs is not a useful one. There have been three different solutions employed by existing languages:

1. Provide the user a syntax for explicitly annotating the variance of parametric types. This approach, employed by Scala [42], Kotlin [9], and partially by Java [2] and C#

[27], provides the most flexibility to the user while also providing the strictest safety guarantees. The disadvantage is that such a system adds significant complexity to the language and greatly steepens the learning curve. Since Stanza is targeted towards users without extensive type system experience, we felt this solution to be inappropriate for our audience.

2. Avoid introducing the concept of subtyping altogether from the language. OCaml's [35] module system and Haskell [31] and Rust's [17] typeclass systems are examples of how a language can be designed to sidestep the concept of subtyping while still remaining useful. This is easier for users to learn than a variance annotation system, but is also significantly less flexible. For example, creating a heterogenous collection in Haskell, which is a trivial operation in Java [2], requires the use of existential types, a non-standard extension to the base language.

3. Limit covariance to special system-provided types, such as arrays, and provide a special-cased typing rule for handing stores to arrays. This is the approach employed by Eiffel [39], Java [2], and C# [27] for its arrays.

We like the third approach and feel that it provides the most intuitive behaviour for our target users, but do not like how arrays behave differently from user-defined types. Stanza generalizes this solution in the form of its captured type parameters and allows arrays and user-defined parametric types to be handled uniformly.

Captured Type Parameters

Stanza offers two categories of type parameters: *explicit* and *captured*. Explicit type parameters are the ones that have already been discussed. When calling a function with explicit type parameters, the type arguments must be provided by the user. In contrast, when calling a function with captured type parameters, the type arguments are provided automatically by Stanza's capturing system based upon the annotated *capturing locations* in the function definition. Here is how `myreduce` can be declared using captured instead of explicit type parameters:

```
defn myreduce<?T> (f: (T, T) -> T, xs: List<?T>) -> T :
  if empty?(tail(xs)) : head(xs)
  else : f(head(xs), myreduce(f, tail(xs)))
```

The `?` prefix in front of the `?T` type parameter indicates to Stanza that `T` is a captured type parameter. The `List<?T>` annotation for `xs` denotes the capturing location for `T`. Roughly, it says that `xs` must be a list and `T` should be captured from the element type of the list. The following code demonstrates calling this second definition of `myreduce`:

```
defn my-sum (xs: List<Int>) -> Int :
  myreduce(plus, xs)

defn my-append-all (xs: List<List<Int>>) -> List<Int> :
```

```
myreduce(append, xs)
```

Note that the type arguments are now captured from the argument types instead of explicitly given by the user. In the call to `myreduce(plus, xs)`, the captured type for `T` is `Int` as `xs` has type `List<Int>`. In the call to `myreduce(append, xs)`, the captured type for `T` is `List<Int>` as `xs` has type `List<List<Int>>`.

The Flow Relation

The key relation that governs Stanza's capturing mechanism is the *flow* relation, whose inference rules are shown in Figure 4.2.

The flow relation:

$$\mathtt{Z} \in \mathtt{X} \underset{\mathtt{?T}}{\Rightarrow} \mathtt{Y}$$

is read "the type `Z` exists in the set of types that flow into type parameter `T` when type `X` is passed to type `Y`". It closely mirrors the structure of the subtype relation, and in fact, is related by the following theorem:

$$\exists \mathtt{Z}.\ \mathtt{Z} \in \mathtt{X} \underset{\mathtt{?T}}{\Rightarrow} \mathtt{Y} \text{ if and only if } \mathtt{X}\ \mathtt{<:}\ \mathtt{Y}$$

which says that the set of types that flow to `T` exists if and only if `X` is a subtype of `Y`.

$\textsc{CapVar}_1$ and $\textsc{CapVar}_2$ defines the behaviour of captured type parameters. $\textsc{CapVar}_1$ is the critical rule and says that if a type, `X`, is passed to captured type parameter `T`, then `X` flows into `T`. If `X` is passed to some other captured type parameter `S`, then $\textsc{CapVar}_2$ says that `Void` flows into `T`.

$\textsc{TypeVar}$ specifies that `Void` flows into `T` if a type variable is passed to itself.

$\textsc{Named}_1$, $\textsc{Named}_2$, and $\textsc{Named}_3$ define the flow behaviour of named types. $\textsc{Named}_1$ says that a named type can be passed to another named type if the names match and all the type arguments of the left-hand type can be respectively passed to the type arguments of the right-hand type. Any type that flows to `T` due to passing the type arguments also flow to `T` when passing the named types. If the named types have *no* type arguments, then $\textsc{Named}_2$ says that `Void` flows to `T`. $\textsc{Named}_3$ says that any type that flows to `T` when the parent of the named type is passed also flows to `T` when the named type is passed. The parent is computed in the same fashion as for the subtype relation.

$\textsc{Tuple}_1$, $\textsc{Tuple}_2$, and $\textsc{Tuple}_3$ define the flow behaviour of tuple types. $\textsc{Tuple}_1$ says that a tuple type can be passed to another tuple type if they have the same arity and all the element types of the left-hand tuple can be respectively passed to the element types of the right-hand tuple. Any type that flows to `T` when passing the element types also flows to `T` when passing the tuple type. If the tuples are zero-arity, then $\textsc{Tuple}_2$ says that `Void` flows to `T`. $\textsc{Tuple}_3$ says that any type that flows to `T` when the collapsed form of the tuple is passed also flows to `T` when the tuple type is passed. The collapsed form is computed in the same fashion as for the subtype relation.

$\texttt{Z} \in \texttt{X} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}$ means that Z exists in the set of types that flow into type parameter T when X is passed to Y.

$$\frac{}{\texttt{X} \in \texttt{X} \underset{\texttt{?T}}{\Rightarrow} \texttt{?T}}\ \textsc{CapVar}_1 \qquad \frac{}{\texttt{Void} \in \texttt{X} \underset{\texttt{?T}}{\Rightarrow} \texttt{?S}}\ \textsc{CapVar}_2 \qquad \frac{}{\texttt{Void} \in \texttt{S} \underset{\texttt{?T}}{\Rightarrow} \texttt{S}}\ \textsc{TypeVar}$$

$$\boxed{\begin{array}{l} \texttt{?} \sim \texttt{T} \\ \texttt{?} \sim \texttt{A<?, ..., ?>} \\ \texttt{?} \sim \texttt{[?, ..., ?]} \\ \texttt{?} \sim \texttt{(?, ..., ?) -> ?} \end{array}}$$

$$\frac{\forall i.\ \texttt{Z}i \in \texttt{X}i \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}i}{\texttt{Z}i \in \texttt{A<X1, ..., Xn>} \underset{\texttt{?T}}{\Rightarrow} \texttt{A<Y1, ..., Yn>}}\ \textsc{Named}_1 \qquad \frac{}{\texttt{Void} \in \texttt{A<>} \underset{\texttt{?T}}{\Rightarrow} \texttt{A<>}}\ \textsc{Named}_2$$

$$\frac{\begin{array}{l} \texttt{deftype A<T1, ..., Tn> <: P} \\ \texttt{Z} \in \texttt{P[T1 := X1, ..., Tn := Xn]} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y} \end{array}}{\texttt{Z} \in \texttt{A<X1, ..., Xn>} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}}\ \textsc{Named}_3 \qquad \frac{\underset{i\geq 1}{\forall i}.\ \texttt{Z}i \in \texttt{Y}i \underset{\texttt{?T}}{\Rightarrow} \texttt{X}i \qquad \texttt{Z0} \in \texttt{X0} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y0}}{\texttt{Z}i \in \texttt{(X1, ..., Xn) -> X0} \underset{\texttt{?T}}{\Rightarrow} \texttt{(Y1, ..., Yn) -> Y0}}\ \textsc{Arrow}$$

$$\frac{\forall i.\ \texttt{Z}i \in \texttt{X}i \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}i}{\texttt{Z}i \in \texttt{[X1, ..., Xn]} \underset{\texttt{?T}}{\Rightarrow} \texttt{[Y1, ..., Yn]}}\ \textsc{Tuple}_1 \qquad \frac{}{\texttt{Void} \in \texttt{[]} \underset{\texttt{?T}}{\Rightarrow} \texttt{[]}}\ \textsc{Tuple}_2 \qquad \frac{\texttt{Z} \in \texttt{Tuple<X1|...|Xn>} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}}{\texttt{Z} \in \texttt{[X1, ..., Xn]} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}}\ \textsc{Tuple}_3$$

$$\frac{\texttt{?} \sim \texttt{Y} \qquad \texttt{Z} \in \texttt{X} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}}{\texttt{Z} \in \texttt{X} \underset{\texttt{?T}}{\Rightarrow} \texttt{?}}\ \textsc{Unknown}_1 \qquad \frac{\texttt{?} \sim \texttt{Y} \qquad \texttt{Z} \in \texttt{Y} \underset{\texttt{?T}}{\Rightarrow} \texttt{X}}{\texttt{Z} \in \texttt{?} \underset{\texttt{?T}}{\Rightarrow} \texttt{X}}\ \textsc{Unknown}_2 \qquad \frac{}{\texttt{Void} \in \texttt{?} \underset{\texttt{?T}}{\Rightarrow} \texttt{?}}\ \textsc{Unknown}_3$$

$$\frac{\texttt{Z} \in \texttt{Y} \underset{\texttt{?T}}{\Rightarrow} \texttt{X1}}{\texttt{Z} \in \texttt{Y} \underset{\texttt{?T}}{\Rightarrow} \texttt{X1|X2}}\ \textsc{Union}_1 \qquad \frac{\texttt{Z} \in \texttt{Y} \underset{\texttt{?T}}{\Rightarrow} \texttt{X2}}{\texttt{Z} \in \texttt{Y} \underset{\texttt{?T}}{\Rightarrow} \texttt{X1|X2}}\ \textsc{Union}_2 \qquad \frac{\texttt{Z1} \in \texttt{X1} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y} \qquad \texttt{Z2} \in \texttt{X2} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}}{\texttt{Z}i \in \texttt{X1|X2} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}}\ \textsc{Union}_3$$

$$\frac{\texttt{Z1} \in \texttt{Y} \underset{\texttt{?T}}{\Rightarrow} \texttt{X1} \qquad \texttt{Z2} \in \texttt{Y} \underset{\texttt{?T}}{\Rightarrow} \texttt{X2}}{\texttt{Z}i \in \texttt{Y} \underset{\texttt{?T}}{\Rightarrow} \texttt{X1\&X2}}\ \textsc{Intersection}_1 \qquad \frac{\texttt{Z} \in \texttt{X1} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y} \qquad \texttt{Y} \neq \texttt{?T}}{\texttt{Z} \in \texttt{X1\&X2} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}}\ \textsc{Intersection}_2$$

$$\frac{\texttt{Z} \in \texttt{X2} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y} \qquad \texttt{Y} \neq \texttt{?T}}{\texttt{Z} \in \texttt{X1\&X2} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y}}\ \textsc{Intersection}_3$$

Figure 4.2: Stanza Flow Relation

Union$_1$, Union$_2$, and Union$_3$ define the flow behaviour of union types. Union$_1$ says that if a type, Y, can be passed to a type, X1, then Y can also be passed to the union type X1|X2. Any type that flows to T when Y is passed to X1 also flows to T when Y is passed to X1|X2. Union$_2$ says the same about passing Y to X2. Union$_3$ says that both X1 and X2 must be passable to Y for X1|X2 to be passable to Y. Any type that flows to T due to passing X1 or X2 also flows to T when passing X1|X2.

Intersection$_1$, Intersection$_2$, and Intersection$_3$ define the flow behaviour of intersection types. Intersection$_1$ says that Y must be passable to both X1 and X2 for Y to be passable to X1&X2. Any type that flows to T when Y is passed to X1 or X2 also flows to T when Y is passed to X1&X2. Intersection$_2$ says that any type that flows to T when X1 is passed to Y also flows to T when X1&X2 is passed to Y. However, this rule applies only if Y is not the captured type parameter ?T. Intersection$_3$ says the same about passing X2 to Y.

Arrow defines the flow behaviour of the arrow type. If the two arrows have the same arity, if all the argument types of the right-hand arrow can be passed to the argument types of the left-hand arrow, and if the return type of the left-hand arrow can be passed to the return type of the right-hand arrow, then the left-hand arrow can be passed to the right-hand arrow. Any type that flows to T when passing the argument or return types also flow to T when passing the arrow.

Unknown$_1$, Unknown$_2$, and Unknown$_3$ define the flow behaviour of the unknown type, and rely upon the unknown expansion relation ($\sim$). Unknown$_1$ says that a type, X, can be passed to the unknown type if X can be passed to some expanded type Y. Any type that flows to T when passing to the expanded type also flows to T when passing to the unknown type. Unknown$_2$ says that the unknown type can be passed to some type, X, if some expanded type, Y, can be passed to X. Any type that flows to T when passing the expanded type also flows to T when passing the unknown type. Finally Unknown$_3$ says that Void flows to T when the unknown type is passed to the unknown type.

Capturing a Single Type Variable

Consider the case of calling a polymorphic function with a value of type X. The function has a single captured type parameter, T, and expects a single argument of type Y. Then the type that is captured by T, $\theta_{\texttt{T}}$, is defined as the solution to the following equation:

$$\theta_{\texttt{T}} = \bigcup \{\texttt{Z}|\texttt{Z} \in \texttt{X} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y[T := }\theta_{\texttt{T}}\texttt{]}\}$$

where we use $\bigcup S$ to denote the union of all types in the set S. For example:

$$\bigcup \{\texttt{X1, X2, X3}\} = \texttt{X1|X2|X3}$$

The equation says that $\theta_{\texttt{T}}$, the type captured by T, is defined to be the union of all types that flow to T when X is passed to the result of substituting all occurrences of T in Y with $\theta_{\texttt{T}}$.

As an example, consider calling the following function:

```
defn set-element<?T> (x:[Array<?T>, Int, T]) -> False :
  body ...
```

with a value of type `[Array<Animal>, Int, Dog]`. Then $\theta_{\texttt{T}}$, the type captured by `T`, must satisfy the equation:

$$\theta_{\texttt{T}} = \bigcup \{\texttt{Z}|\texttt{Z} \in \texttt{[Array<Animal>, Int, Dog]} \underset{\texttt{?T}}{\Rightarrow} \texttt{[Array<?T>, Int, }\theta_{\texttt{T}}\texttt{]}\}$$

which can be verified to have solution $\theta_{\texttt{T}} = \texttt{Animal}$ – assuming that `Dog` is a subtype of `Animal`. Note that `Void|T`, the union of the `Void` type with any type `T`, is equivalent to just `T`.

For the purposes of the capture mechanism, calling functions with more than a single argument is equivalent to calling a function with arguments wrapped in a tuple – as was done in the previous example.

Capturing Multiple Type Variables

In the case of calling a polymorphic function with multiple captured type parameters, the types that are captured are defined as the solution to a system of equations. Suppose the function has two captured type parameters, `T` and `S`, expects an argument of type `Y`, and is called with a value of type `X`. Then the types that are captured by `T` and `S`, $\theta_{\texttt{T}}$ and $\theta_{\texttt{S}}$, are defined as follows:

$$\theta_{\texttt{T}} = \bigcup \{\texttt{Z}|\texttt{Z} \in \texttt{X} \underset{\texttt{?T}}{\Rightarrow} \texttt{Y[T := }\theta_{\texttt{T}}\texttt{, S := }\theta_{\texttt{S}}\texttt{]}\}$$

$$\theta_{\texttt{S}} = \bigcup \{\texttt{Z}|\texttt{Z} \in \texttt{X} \underset{\texttt{?S}}{\Rightarrow} \texttt{Y[T := }\theta_{\texttt{T}}\texttt{, S := }\theta_{\texttt{S}}\texttt{]}\}$$

$\theta_{\texttt{T}}$ is defined to be the union of all types that flow to `T` when `X` is passed to the result of substituting all occurrences of `T` in `Y` with $\theta_{\texttt{T}}$ and all occurrences of `S` with $\theta_{\texttt{S}}$. Similarly, $\theta_{\texttt{S}}$ is defined as the union of all types that flow to `S`.

As an example, consider calling the following function, `mymap`:

```
defn mymap<?T,?S> (f:T -> ?S, xs:List<?T>) -> List<S> :
  body ...
```

with arguments of type `Int -> String` and `List<Int>`. Then $\theta_{\texttt{T}}$ and $\theta_{\texttt{S}}$, the types captured by `T` and `S`, must satisfy the system of equations:

$$\theta_{\texttt{T}} = \bigcup \{\texttt{Z}|\texttt{Z} \in \texttt{[Int -> String, List<Int>]} \underset{\texttt{?T}}{\Rightarrow} \texttt{[}\theta_{\texttt{T}}\texttt{ -> ?S, List<?T>]}$$

$$\theta_{\texttt{S}} = \bigcup \{\texttt{Z}|\texttt{Z} \in \texttt{[Int -> String, List<Int>]} \underset{\texttt{?S}}{\Rightarrow} \texttt{[}\theta_{\texttt{T}}\texttt{ -> ?S, List<?T>]}$$

The equations can be verified to have solutions $\theta_{\texttt{T}} = \texttt{Int}$, and $\theta_{\texttt{S}} = \texttt{String}$.

4.7 Type Inference

Type inference for Stanza is complicated by the operational semantics of its optional type system. In purely statically-typed languages, type annotations do not affect the runtime behaviour of a program – they affect only whether a program is accepted or rejected by the typechecker. In the case of Stanza, a type annotation introduces a consistency check that is enforced at runtime.

The following shows an example of declaring a function annotated to expect an `Int` argument, and calling this function with a value of type `?`:

```
defn myfunction (x:Int) :
  ...

val x:? = ...
myfunction(x)
```

Stanza will enforce, at runtime, that the passed argument to `myfunction` is an `Int` or otherwise abort the program. If `myfunction` were annotated to require a `String` instead, then Stanza will enforce that the passed argument to `myfunction` is a `String`. This is an example where both programs typecheck successfully, but where a different type annotation leads to a different runtime behaviour.

Because type annotations can affect runtime behaviour, predictability is of paramount importance for Stanza's type inference system. In the event that a program fails a runtime consistency check, the user should be able to easily understand which type annotation was violated and also agree that it has been violated. This is especially important if the type annotation was inferred and not given explicitly by the user.

Stanza's inference algorithm infers type annotations for the following four cases:

1. return types of functions,
2. argument types of anonymous functions,
3. types of value declarations, and
4. types of local variable definitions.

Return Type Inference

Stanza uses a conservative algorithm for inferring the return type of functions. If the function contains no (direct or indirect) recursive calls then the return type can always be inferred by inspecting the type of the last expression. For recursive functions, Stanza uses a simple simplification procedure to solve type equations involving union types. The following shows an example of a recursive function, `loop`, that requires union type simplification in order to infer its return type:

```
defn loop (i:Int) :
  if i == 0 : 0
  else : loop(i - 1)
```

If we let `X` denote the return type of `loop`, then, by analyzing the body of `loop`, we find that `X` must satisfy the following equation:

$$\texttt{X} = \texttt{X|Int}$$

The most precise solution is `X` = `Int`.

If the return type of a function cannot be inferred, then Stanza issues an error requesting for the user to explicitly provide it.

Argument Type Inference

Stanza infers the argument types of anonymous functions from the context in which they are used. The reason this is done only for anonymous functions, and not named functions, is because anonymous functions are guaranteed to be used only in a single context. The following shows an example of calling a higher-order function with an anonymous function:

```
defn do-to-n (f:Int -> ?, n:Int) :
  for i in 0 to n do :
    f(i)

do-to-n(
  fn (x) :
    println(x),
  10)
```

The function `do-to-n` expects two arguments: a function that can be called with `Int` and returns `?`, and an integer `n`. We then call `do-to-n` with an anonymous function and the literal `10`. The type of the argument `x` is left unspecified. Because the anonymous function is passed to a location that is expecting the type `Int -> ?`, Stanza can infer that the argument type of `x` is `Int`.

The design of Stanza's core library relies extensively upon argument type inference. As an example, recall that the syntax:

```
for i in 0 to 10 do :
  println(i)
```

is a shorthand for the following call to the `do` function:

```
do(
  fn (i) :
    println(i),
  0 to 10)
```

The type of the loop variable `i` is inferred from the type expected by the `do` function.

Similar to return type inference, if the argument type of an anonymous function cannot be inferred, then Stanza issues an error requesting the user to explicitly provide it.

Note that argument types for named functions are *not* inferred. If the type of an argument to a named function is left unspecified, then it is assumed to be `?` by default.

Value Type Inference

If left unspecified, Stanza will automatically infer the types of value declarations based on their initializing value. The following shows the declaration for a value, `x`, that is initialized to the result of `10 + 32`:

```
val x = 10 + 32
```

`x` is inferred to have type `Int`.

The types of local values can *always* be inferred, and the Stanza convention is to leave the types of values left unspecified. Because top-level definitions may reference each other, it may be impossible on rare occasions to infer the type of a global value. The following shows an example of this:

```
val X = Y
val Y = X
```

In this case, Stanza will issue an error requesting explicit type annotations. Note that Stanza's top-level expressions execute sequentially, so the above code will halt with an error when `Y` is read before it is initialized.

Variable Type Inference

If left unspecified, Stanza will automatically infer the types of local variable declarations based upon the expressions assigned to the variable. Similar to return type inference, Stanza uses a simplification procedure to solve type equations involving union types. The following shows an example of a variable declaration where inference requires union type simplification:

```
var y = "Hello"
y = if pred() : y
    else : 42
```

If we let `X` denote the type of `y`, then, by analyzing the expression assigned to `y`, we find that `X` must satisfy the following equation:

$$\texttt{X} = \texttt{String|X|Int}$$

The most precise solution is $\texttt{X} = \texttt{String|Int}$.

There is one complication that is unique to the inference of variable types. When a variable is assigned an expression with a type containing a type variable that is not in scope at the variable declaration site, then the variable type cannot be inferred. The following shows an example of this complication:

```
var y = 42
defn f<T> (x:T) :
  y = x
```

If we let X denote the type of y, then from the assignments to y we find that X must satisfy the following equation:

$$\texttt{X} = \texttt{Int|T}$$

However, T is not in scope where y is defined, and thus Stanza cannot infer the type of y. If the variable type cannot be inferred, then Stanza issues an error requesting for the user to explicitly provide it.

Note that *global* variables are required to have explicit type annotations. This is because Stanza allows packages to be compiled separately from each other, and we do not want the inferred type of a variable to be affected by assignments to it from other packages.

Chapter 5

The Stanza Multimethod Object System

Similar to how Stanza's type system was designed to bridge the dynamically-typed and statically-typed paradigms, Stanza's object system was designed to bridge the object-oriented programming (OOP) and functional programming (FP) paradigms. We feel that OOP offers an intuitive and convenient paradigm that is well-suited for day-to-day programming tasks, while FP leads ultimately to better software architecture and shorter code. Stanza's object system is built from a small number of constructs that combine to offer the advantages of both paradigms.

5.1 Object Oriented Programming

Object Oriented Programming (OOP) is a programming paradigm based on the concept of "objects" – encapsulated bundles of state that interact with each other through well-defined interfaces. It was first introduced in Simula [16], a language for discrete event simulation, where objects often mirrored structures in the physical world – such as animals, creatures, cars, etc.

This programming practice of having software objects parallel physical objects is a key aspect of OOP, and is reflected in the common convention of using nouns as names for objects. Niklaus Wirth had this to say about OOP:

> "This paradigm closely reflects the structure of systems 'in the real world', and it is therefore well suited to model complex systems with complex behaviours." [64]

The close relationship between OOP and real-world objects makes it an intuitive and friendly paradigm for beginner programmers. Out of the top ten most popular languages on the Tiobe index [54], eight of them are OOP languages: Java [2], C++ [55], C# [27], Python [47], PHP [34], Visual Basic .NET [40], JavaScript [18], and Delphi [7].

Desirable Aspects of OOP

The following shows an example function written in the OOP language Java [2]:

```
void animalCare(Animal[] animals) {
  Shower s = new Shower("Floofer 2000");
  for(int i=0; i<animals.length; i++){
    Animal a = animals[i];
    a.walk();
    a.wash(s);
    a.dry();
    a.bedTime();
  }
}
```

It accepts an array of `Animal` objects, creates a shower, and then for each animal, walks, washes, dries, and tucks it into bed. It demonstrates four critical aspects of object-oriented programming that we desire from Stanza:

1. *Encapsulation and Protection:* The implementation details of an `Animal` are hidden from the implementation of the `animalCare` function. The function depends only upon the set of defined methods, `walk`, `wash`, `dry`, and `bedTime`. The implementation of these methods can be changed at will, and – as long as they satisfy the method contract – `animalCare` is guaranteed to continue working as before.

   The representation of an `Animal` is concealed and can be changed without affecting the correctness of `animalCare`. For instance, additional fields can be added to keep track of an animal's name, and floating-point numbers can be used instead of integers to keep track of an animal's weight, both without affecting `animalCare`.

2. *Type-Specific Namespaces:* In daily programming, a great deal of effort is spent simply on naming things appropriately. The names should be both clear and succinct, so as to make code easier to read, but also avoid clashes according to the language's namespace rules.

   In the example above, `dry` is chosen for the name of the method that dries an animal. If the program were to contain a `Humor` class, Java would conveniently permit it to also contain a method named `dry`. These two different `dry` methods do not clash according to Java's namespace rules, as each class can be thought of as having its own namespace for method names. This allows for short method names and natural-reading code.

   In contrast, the equivalent program in C [33] would likely use the names `animal_dry` and `humor_dry` for the analogous functions. Some libraries, such as OpenGL [51], even go so far as to add a prefix to *every* function in the library to avoid clashes.

3. *Dynamic Dispatch:* There are many different kinds of animals – such as cats, dogs, horses, and humans to name just a few – and each of these animals walk in their own particular way. Java allows the `Cat`, `Dog`, `Horse`, and `Human` classes to each provide

their own implementation of `walk`. In the call to `a.walk()`, Java will automatically call the appropriate version depending the type of animal `a` is referring to. This is a convenient aspect of object-oriented programming that corresponds to how algorithms are often described in natural language.

4. *Extensibility:* Suppose that the programmer responsible for writing `animalCare` is separate from the programmers implementing the behaviours of each different kind of animal. Java allows new types of animals to be implemented separately from the implementation of `animalCare`. If we were to later implement, say, a `Goat` class, `animalCare` would automatically support the new `Goat` objects.

Weaknesses of OOP

By this point in time, OOP has been proven to be an effective organizational philosophy for architecting large software and many large successful programs have been written in object-oriented languages. But over time, we have noticed that adherence to the OOP paradigm has led to the following deficiences that we wish to avoid in Stanza:

1. *Awkward Division of Behaviours into Classes:* It is strongly encouraged, in many object-oriented languages, to keep behaviour *contained* in classes. This can be enforced either by mandating behaviour to be implemented as methods, as is done in Java, or through a set of guidelines for establishing best practices. However, it is often awkward to force a certain behaviour into being contained by one specific class. The call to `a.wash(s)`, for example, presumes that the `wash` method is best contained within the `Animal` class, and the shower object, `s`, is passed as an argument. An alternative design would be to put the `wash` method in the `Shower` class instead, and pass in the `Animal` object as an argument. Yet another design could be to define a new class, `FarmEmployee`, for containing the `wash` method and pass in both the `Shower` and the `Animal` object as arguments. Forcing the eager division of behaviour into classes leads to suboptimal program architectures that is cumbersome to change later.

2. *Vertical Separation of Concerns:* Over time, software architected according to object-oriented principles becomes hard to maintain. Classes become larger, contain an increasing number of fields, and are more tightly coupled. We believe this is due to OOP's over-emphasis on a *vertical* separation of concerns.

   A prominent example of this is in the typical design of graphical user interface (GUI) libraries. Consider a class, `Window`, for representing a window interface element that supports the following methods: `close`, `open`, `resize`, `draw-border`, `draw-bevel`, `repaint-component`, and `get-component-texture`. The `close`, `open`, and `resize` methods manage the state of the window, and are the only methods that should be called by users of the GUI library. `draw-border` and `draw-bevel` are high-level drawing methods for controlling the look of the window borders and are meant to be called only by the GUI framework. `repaint-component` and `get-component-texture` are critical low-level drawing methods used by the GUI painting subsystem, and will typically crash the entire GUI framework if they are implemented incorrectly.

The `Window` class is a prime example of vertically separating concerns: software modules are divided not by their level of abstraction but by which application feature they implement. The same class is responsible for every aspect of a window element, from the high-level user-facing functions to the low-level drawing primitives.

3. *Fragile Base Classes:* Inheritance is an occasionally useful feature for enabling code reuse that is supported by the majority of object-oriented languages. A class, `B`, can inherit all the behaviour and state of another class, `A`, by declaring `B` to be a *subclass* of `A`. For example, a `Dog` and a `Cat` class, due to both representing animals, will likely contain a lot of code in common. To reduce the redundancy, the common code can be factored out into a base class called `Animal`, and `Dog` and `Cat` can be subclassed from `Animal` to inherit the common functionality. Later on, new subclasses of `Dog` can also be declared, such as `Retriever` or `Collie`, and these subclasses will inherit all the functionality common to dogs.

   Gradually, however, the depth of the class hierarchy grows deeper as more specialized classes are introduced. At the same time, the base class also starts to accrue more fields and methods to satisfy the needs of its increasing number of subclasses. Without extreme discipline, it is easy for the base class to grow to an unmaintainable size, become unfocused in its purpose, and become too entrenched to change. As an example, the `MetalScrollButton` class in the Java Swing library is subclassed from *six* parent classes. Its immediate parent is `BasicArrowButton`, which is itself a subclass of `JButton`, which in turn is subclassed from `AbstractButton`, and so on through `JComponent`, `Container`, and `Component`.

4. *Scattered Algorithms:* Due to an over-emphasis on inheritance and dynamic dispatch, otherwise simple algorithms often end up spread across multiple classes and source files when written in an object-oriented language. Consider the following procedure, which is employed by an animal center for bathing all the cats in its care:

   a) Separate the cats from the rest of the animals.

   b) Gather enough materials for the number of cats to be washed.

   c) For each cat, restrain the cat, and then wash it.

   d) Turkish Van cats do not need restraining as they enjoy baths naturally.

   The following shows an implementation of part of the above algorithm in Java:

   ```
   class AnimalCare {
     ...
     void washCats (Animal[] animals) {
       ArrayList cats = new ArrayList();
   ```

```
    for(int i=0; i<animals.length; i++)
      animals[i].addIfCat(cats);
    Materials m = washingMaterials(cats.size());
    for(int i=0; i<cats.length; i++){
      Cat c = (Cat)cats[i];
      c.restrain();
      c.wash(m);
    }
  }
}
```

The `washCats` method takes a collection of animals, creates a list for storing cats, and then asks each of the animals to add themselves to the list if they are a cat. Materials are collected to wash the cats, and each cat is then restrained and washed.

Notice that `washCats` shows an incomplete representation of the procedure. Nowhere is it shown how we decide whether an animal is a cat, or how to exclude Turkish Van cats from being restrained. That information is contained in the other classes shown below:

```
class Animal {
  ...
  void addIfCat (ArrayList xs){}
}

class Cat extends Animal {
  ...
  void addIfCat (ArrayList xs){
    xs.add(this);
  }

  void restrain (){
    can_move = false;
  }
}

class TurkishVan extends Cat {
  ...
  void restrain (){}
}
```

The default method for `addIfCat` in the `Animal` class is empty. This method is overridden in the `Cat` class, which adds itself to the array. The `restrain` method in the `Cat` class sets a flag that forbids the cat from moving. The `TurkishVan` subclass of `Cat` then overrides the `restrain` method to do nothing as they do not require restraining before bath time.

Our simple algorithm for washing cats is spread across four different classes. In order to fully grasp the procedure, readers must be familiar with the default *and overridden* implementations of `addIfCat` and `restrain`.

Our example may seem contrived but such code is often found in practice. The following shows the implementation of a method from the Java Swing library with a similar flavor:

```
class AbstractButton extends JComponent {
  ...
  public void setLayout (LayoutManager mgr){
    setLayout = true;
    super.setLayout(mgr);
  }
}
```

The following quote, sometimes attributed to Adele Goldberg, an early proponent of object-oriented programming, eloquently characterizes the above phenomenon:

> "In Smalltalk, everything happens somewhere else."

5.2 Functional Programming

Functional Programming (FP) is built upon the lambda calculus [13] developed by Alonzo Church, a theoretical framework for modeling computation based on function abstraction and application. One of the earliest functional languages was the programming language Lisp [32] by John McCarthy, which included support for first-class functions and list datastructures – features that singularly distinguished it from other languages of the period.

Since its early beginnings in Lisp, many functional languages have been designed and become successful in their niches, such as Scheme [56], ML [41], Miranda [59], and later, Haskell [31]. Moreover, features that previously were supported only by functional languages, such as first-class and higher-order functions, have gradually crept into the design of modern object-oriented languages.

While functional programming has traditionally been regarded as being academic and impractical, its productivity advantages have come to be recognized and it is seeing more use throughout industry. Java [2], Python [47], Javascript [18], and Ruby [22], four of the most widespread languages currently in use, all have introduced features for supporting functional programming to some extent.

Desirable Aspects of FP

We find the following aspects of functional programming desirable and wish to include it in the design of Stanza:

1. *First-class Functions:* Functional programming languages encourage the pervasive use of first-class functions, a powerful technique that allows for considerable code reuse. The following example shows a succinct and clear Scheme [56] implementation of computing the total length of a list of strings:

```
(define (total-length strings)
  (reduce + 0 (map string-length strings)))
```

While many object-oriented languages do support first-class functions, FP languages are set apart by their ease-of-use, and by the extent of their use in the design of the language and core libraries. For instance, unlike Scheme, the OOP Ruby [22] language opted to provide `map` as a method in their collections classes instead of as a function. This choice fits better within Ruby's overall philosophy but does limit the language's power. Unlike functions, methods in Ruby cannot be treated as values.

2. *Horizontal Separation of Concerns:* In contrast to how programs are typically architected in object-oriented languages, functional programming languages encourage organizing code according to a horizontal, rather than vertical, separation of concerns. Instead of compartmentalizing code into modules that each implement a different application feature, code is grouped by their abstraction level. A large program is divided into a series of layers in which the bottom-most layer directly deals with the abstractions provided by the machine or operating system. Each successive layer then makes use of the underlying layer to introduce its own set of simplifying abstractions to be used by the layers above. The top-most layer interfaces directly with the end user and is coded against abstractions perfectly tuned for the application domain.

   Consider again the example of the `Window` class for representing a window interface element. Instead of holding a single class responsible for every aspect of managing a window element, a functional language would encourage division along the following abstraction boundaries:

   a) The lowest layer would handle memory and resource management.
   b) The next layer, the low-level drawing layer, would provide abstractions for drawing simple shapes and managing colors.
   c) The high-level drawing layer would then make use of the low-level drawing layer to provide abstractions such as drawing borders and beveling.
   d) Finally the top-most layer, the user interface layer, would provide the API for creating and closing windows, creating buttons and scrollbars, and so on.

   We believe that the horizontal separation of concerns ultimately leads to more maintainable and stable software architecture. This is reflected in the design of our general compute infrastructure. The operating system drivers provides the abstractions necessary for controlling the machine hardware; the operating system then makes use of these abstractions to provide abstractions for handling files and allocating memory; interface libraries then allow for the creation of windows, buttons, and scrollbars; and finally the application developer writes programs for some specific domain which the user interacts with directly.

Weaknesses of FP

Despite its power, the following are areas of weakness in existing functional languages that we wish to avoid in Stanza:

1. *Extensibility:* Unlike object-oriented languages, which include systematic mechanisms to help architect extensible software – such as subclassing and inheritance – functional languages place less emphasis on extensibility. Programmers are expected to build in hooks for future extensibility out of first-class functions. Neither OCaml [35] nor Haskell's [31] tagged union types, for example, can be extended with additional constructors. Scheme [56] has a multitude of libraries that allow for programming in an object-oriented style, but they are non-standard and generally at odds with the design of the base language.

2. *Namespace Conflicts:* Object-oriented languages allow for a straightforward method to reduce namespace conflicts. Since classes are guaranteed to contain no duplicate methods, no call to a method will ever conflict. The method is resolved, either statically or dynamically, in the method namespace of the class of the receiver object.

   Functional languages have no such namespace mechanism and consequently suffer more from name conflicts. Conventionally, methods in object-oriented languages are represented in FP languages as functions that take the receiver object as the first argument. However, under this convention, there is a high likelihood of there being multiple functions with the same name.

   OCaml [35] and Haskell [31] programs work around this problem by relying heavily on its destructuring and pattern-matching mechanisms, at the expense of breaking encapsulation. Scheme [56] has taken the approach of prefixing its functions with the type of the receiver object – leading to a plethora of functions with names like `array-get`, `array-set!`, `vector-get`, and `vector-set!`. While workable, these solutions lack the elegance of the object-oriented languages.

3. *Steep Learning Curve:* A subset of functional languages have a reputation for having a steep learning curve. OCaml [35] and Haskell [31], for instance, for known for being difficult to learn, but Scheme [56] is seen as beginner friendly in contrast. We believe the difference arises primarily due to OCaml and Haskell's emphasis on immutability, and the rigidity of their type systems.

   To address immutability, we do not hold any strong beliefs against mutation. Stanza is meant to be a pragmatic programming language, and mutation is a natural concept for expressing many algorithms. We believe in neither eliminating mutation from the language nor in designing the language to discourage its use.

   To address the rigidity of type systems, our experience leads us to believe that the major learning barrier is due to the need to work around OCaml and Haskell's lack of support for subtyping and heterogeneity. Programmers are comfortable with the concepts of hierarchy and heterogeneity, and these concepts are reflected in the structure of their code.

For instance, dogs and cats are different types of animals; retrievers and collies are different types of dogs; and golden retrievers and black retrievers are different types of retrievers. An animal care shelter will, at a given time, be taking care of an assorted collection of animals. The programmer may define a set of functions that are applicable to all animals, such as `feed` and `sleep`, and a set of functions that are application only to dogs, such as `fetch` and `roll-over`. They may use a list to keep track of all the animals in the shelter, and expect this list to be able to hold an *assorted* set of animals, not simply one particular type of animal. Because of OCaml's and Haskell's poor support for modeling hierarchy and heterogeneity, programmers must find an alternate and less natural encoding for these concepts.

5.3 The Multimethod Object System

Stanza's multimethod object system is built upon only three constructs: `defmulti`, `defmethod`, and `new`.

Multis and Methods

The `defmulti` construct declares the existence of a *multi* with the specified name and type signature. The following declares a multi called `feed` that accepts an `Animal` and returns `False`:

```
defmulti feed (a:Animal) -> False
```

The `defmethod` construct provides an implementation for an existing multi. The following declares a method for `feed` that prints a message to the screen:

```
defmethod feed (a:Animal) :
  println("Feed an animal: Generic grain")
```

From the perspective of callers, the above definitions of the `feed` multi and its method is indistinguishable from the following function:

```
defn feed (a:Animal) -> False :
  println("Feed an animal: Generic grain")
```

Multis exist in the same namespace as functions, have the same visibility rules, and can be treated as values, just like functions.

There are two important architectural benefits of declaring `feed` as a multimethod versus as a function:

1. The implementation can now be provided separately from the declaration. The `feed` method can even be declared in a different package residing in a different source.

2. *Multiple* implementations can be provided for a single multi declaration.

The following illustrates the second point by providing two additional methods for the `feed` multi – one for when `feed` is called with a `Dog` and another for when it is called with a `Cat`:

```
defmethod feed (a:Dog) :
  println("Feed a dog: Kibble")

defmethod feed (a:Cat) :
  println("Feed a cat: Tuna")
```

With these methods, the `feed` multi now has a total of three different implementations: one for dogs, one for cats, and a generic one for animals. When a programmer calls the `feed` multi, Stanza will *dynamically dispatch* the call to the most specific method. Calling `feed` on a dog will print:

```
Feed a dog: Kibble
```

Calling `feed` on a cat will print:

```
Feed a cat: Tuna
```

And calling `feed` on some other animal, say a bird, will print:

```
Feed an animal: Generic grain
```

Instances and Instance Methods

The `new` construct creates a new instance of a given type with a set of *instance methods* – methods that are specific to the created instance. Instance methods must be declared within the body of a `new` construct, and must have exactly one argument with the name `this`, which refers to the instance being created. The following demonstrates creating an instance of a dog, `d`, with one instance method:

```
val d = new Dog :
  defmethod feed (this) :
    println("Feed me something yummy")
```

Whenever `feed` is called specifically on `d`, it prints out the message:

```
Feed me something yummy
```

The real expressiveness of instance methods come from their ability to reference binders defined in their lexical scope. The following demonstrates an instance method that refers to two binders declared in its lexical scope:

```
defn Dog (name:String, taste:String) :
  new Dog :
    defmethod feed (this) :
      println("%_ wants something %_." % [name, taste])
```

The `feed` method refers to the binders `name` and `taste` which are defined outside the `new` construct.

`Dog` is just a simple function, and the following demonstrates calling it twice to create two dog instances, `d1` and `d2`:

```
val d1 = Dog("Whiskey", "yummy")
val d2 = Dog("Rummy", "tasty")
```

When `feed` is called on `d1`, the program prints:

```
Whiskey wants something yummy.
```

And when `feed` is called on `d2`, the program prints:

```
Rummy wants something tasty.
```

Notice that the instance methods remember the values of the lexical binders of the context in which the instance was created. In this regard, the `new` construct behaves similarly to a function closure.

Specificity Relation

The specificity relation governs which method is executed when a multi is called. A method, m_1, is considered *more specific* than another method, m_2, if the arguments of m_1 are all subtypes of the arguments of m_2 respectively. When a multi is called with a list of values, a method is considered a *candidate method* if the values are instances of the method argument types, *and* there is no other applicable method that is more specific than it.

If there is a *unique* candidate method, then that method is executed. If there is no candidate method, then Stanza issues an error indicating that the multi cannot be called with values of those types. If there are multiple candidate methods, then Stanza issues an error indicating that the choice of method to be executed is ambiguous.

As an example, the following defines a two-argument `feed` multi that takes an `Animal` and a `Food` value, and returns `False`, along with three methods:

```
defmulti feed (a:Animal, f:Food) -> False

defmethod feed (a:Animal, f:Food) :
  println("Feed an animal some generic food.")

defmethod feed (a:Animal, f:Tuna) :
  println("Feeding an animal tuna.")

defmethod feed (c:Cat, f:Food) :
  println("Cat's won't eat generic food.")
```

Given a `Dog` and a `Tuna`, there are two methods that are applicable: the one defined for `(Animal, Food)` and the one defined for `(Animal, Tuna)`. Out of the two, the `(Animal, Tuna)` method is more specific and is the one that is executed.

Given a `Cat` and a `Tuna`, there are three methods that are applicable: the `(Animal, Food)`, the `(Animal, Tuna)`, and the `(Cat, Food)` methods. Neither is the `(Animal, Tuna)` method more specific than the `(Cat, Food)` method nor vice versa, and thus Stanza issues an ambiguity error.

Function Overloading and Automatic Function Mixing

Stanza provides two features, *function overloading* and *automatic function mixing*, for relaxing the namespace rules and decreasing the chance of name conflicts.

Consider the following Stanza code:

```
defpackage dogs
defn Dog () -> Dog : ...
defmulti bark (d:Dog) -> False

defpackage trees
defn Tree () -> Tree : ...
defmulti bark (t:Tree) -> String

defpackage user :
  import dogs
  import trees

val d = Dog()
val t = Tree()
bark(d)
bark(t)
```

There are two independent packages, `dogs` and `trees`, both of which define a multi called `bark`. The `bark` in the `dogs` package accepts a `Dog` value and plays the dog's bark through the speaker. The `bark` in the `trees` package accepts a `Tree` value and returns a string representing the type of bark it has.

The `user` package imports both the `dogs` and `trees` package, creates a `Dog` and a `Tree` instance and calls `bark` on both of them. The language design question is: how do we solve the name conflict that arises from there being two `bark` multis visible from the `user` package?

To start, one important observation is that the `dogs` and `trees` package are completely independent of each other. They each were written without any knowledge of the other, and *must* be able to be written without knowledge of the other. This eliminates the possibility of subsuming both multis in a single declaration:

```
defmulti bark (d:Dog|Tree) -> False|String
```

Beyond this multi making no semantic sense, this would also require coordination between the author of the `dogs` package and the author of the `trees` package.

Let us study the analogous code in an OOP language, and reflect on how it avoids the namespace problem:

```
class Dog {
  method bark () { ... }
}

class Tree {
  method bark () { ... }
}

function main (x) {
  val d = new Dog()
  val t = new Tree()
  d.bark()
  t.bark()
  x.bark()
}
```

The two classes, `Dog` and `Tree`, both define a method called `bark`. The `main` function creates `Dog` and `Tree` objects and calls `.bark` on each of them, and also calls `.bark` on the passed-in argument `x`.

Let us first consider the calls to `.bark` on `d` and `t`. It is clear from inspection that `d` and `t` have types `Dog` and `Tree` respectively. Therefore the call to `d.bark()` must refer to the `bark` method in the `Dog` class, and the call to `t.bark()` must refer to the `bark` in the `Tree` class. Thus the name conflict can be disambiguated in these cases by statically inferring the type of the receiver object.

Now let us consider the call to `.bark` on `x`. Since `x` is an arbitrary object that is passed into main, we cannot statically determine anything about its type. However, if classes are not allowed to contain methods with duplicate names, then we can delay the dispatch until runtime – when the type of `x` will be known – and jump to the single `.bark` method in whatever class `x` turns out to be from.

Thus in both cases, the name conflict can be disambiguated by inspecting the type of the receiver object – either statically or at runtime. We can use an analogous strategy in Stanza to disambiguate calls to multis and functions. To disambiguate a call to an overloaded function:

1. We first statically infer the types of all the arguments, and remove from consideration all the functions that cannot be validly called with arguments of those types. If there is only one applicable function remaining, then that function is called. This feature is called *function overloading*.

2. If there is more than one applicable function remaining, then we check whether there is any overlap in their domains, i.e. if there exists a set of argument types for which multiple functions are applicable. If there is overlap, then the typechecker issues a function ambiguity error. If not, then the function call is converted to a `match` expression that dynamically dispatches to the appropriate overloaded function. This feature is called *automatic function mixing.*

5.4 Relation to OOP and FP

The combination of the three object constructs provide us the advantages we sought from the OOP and FP paradigms while also avoiding their disadvantages:

1. *Encapsulation and Protection:* Notice that the representation of an object is never directly specified by the programmer. Instead, the programmer only specifies how an object implements the multis defining its interface. Additionally, multis respect the same visibility rules as other declarations, and can simply be declared as private to conceal it from code outside the package.

   The result is that the set of possible interactions with an object is defined solely by its interface, and programmers have fine control over how this interface is exposed to users.

2. *Type-Specific Namespaces:* Function overloading and automatic function mixing allows for multiple functions to have the same name so long as they accept arguments of different types. This allows for short intuitive names in the vast majority of cases.

   In fact, together the two mechanisms offer exactly the same disambiguation power as the class-specific method namespaces of OOP languages. Consider systematically mapping the following OOP method definitions:

   ```
   class A {
     mymethod (x, y) { ... }
   }
   class B {
     mymethod (x, y) { ... }
   }
   ```

   to multi definitions by representing the receiver object with the first argument:

   ```
   defmulti mymethod (receiver:A, x, y) -> ?
   defmulti mymethod (receiver:B, x, y) -> ?
   ```

   Then function overloading and automatic function mixing disambiguates a call to `mymethod(obj, x, y)` in exactly the same way as OOP disambiguates a call to `obj.mymethod(x, y)`: based upon the type of the receiver object.

3. *Dynamic Dispatch:* The multimethod system dynamically dispatches to the appropriate method based on the types of the arguments that the multi is called with. As an additional advantage over the class-based OOP languages, the multimethod system dispatches based upon the types of *all* the arguments instead of solely on the type of the receiver object. Algorithms that depend upon the type of more than one argument – such as a function to test whether two shapes intersect – are *much* more naturally expressed.

4. *Extensibility:* The multimethod system allows for users to easily design an extensible software architecture. A multi declaration creates an extension point with a clear interface that must be satisfied by future types. If a program cannot be extended in a desirable way, then it can re-architected by replacing functions with multi definitions.

5. *Division of Behaviours into Classes:* Stanza has no concept of classes, and consequently also no requirements to keep behaviour contained within classes. In fact, there are no constraints at all upon which package and which source files the `defmulti`, `defmethod`, and `new` constructs appear in. Programmers can and do organize these constructs freely in order to best suit the architecture of their program.

   Here are some examples of constraints that do not exist in Stanza:

   - The methods that operate on values of one type are not required to be declared in the same file.
   - The multis that operate on values of one type do not need to be defined together with the type. Users are free to later define additional multis on existing types in different source files.
   - The creation of objects of one type are not required to be in the same source file as the type definition. Users are free to create additional objects of an existing type in another source file.

6. *Horizontal Separation of Concerns:* Stanza's multimethod system removes the extraneous pressure to keep code grouped into classes. Without this pressure, it is both easy and natural to architect programs according to a horizontal separation of concerns. Stanza programs are similar in architecture to programs written in other functional programming languages.

7. *Scattered Algorithms:* In comparison to OOP languages, algorithms are less scattered in Stanza for two reasons:

   a) The primary purpose of the multimethod system is for extensibility and architecture. Contrary to OOP philosophy, it is *not* encouraged to use multimethods for the purpose of performing dynamic dispatch, for which the `match` expression should be used instead. Here is the `washCats` example written in Stanza using `match` expressions:

```
defn wash-cats (animals:Array<Animal>) :
  val cats = Vector<Cat>()
  for a in animals do :
    match(a) :
      (a:Cat) : add(cats, a)
      (a) : false
  val m = washing-materials(length(cats))
  for cat in cats do :
    match(cat) :
      (cat:TurkishVan) : false
      (cat) : restrain(cat)
    wash(cat, m)
```

In contrast to the OOP version, the Stanza version is self-contained within one function.

b) Because multis and methods are not constrained to any specific source file, we can keep the relevant multis and methods grouped with the algorithm instead of distributed across many classes. Here is the `washCats` example written using multimethods:

```
defn wash-cats (animals:Array<Animal>) :
  val cats = Vector<Cat>()
  for a in animals do :
    add-if-cat(cats, a)
  val m = washing-materials(length(cats))
  for cat in cats do :
    restrain(cat)
    wash(cat, m)

defmulti add-if-cat (cs:Vector<Cat>, a:Animal) -> False
defmethod add-if-cat (cs:Vector<Cat>, a:Animal) : false
defmethod add-if-cat (cs:Vector<Cat>, c:Cat) : add(cs, c)

defmulti restrain (c:Cat) -> False
defmethod restrain (c:Cat) : set-can-move(c, false)
defmethod restrain (c:TurkishVan) : false
```

Though the program structure is identical to the OOP version, the mere placement of the code within one section of one source file significantly improves clarity.

8. *Base "Classes":* In our experience, Stanza programs do not suffer from the uncontrolled growth of base classes. There are two reasons for this:

   a) Inheritance of state is not supported in Stanza. Stanza's multimethod system allows for inheritance of behaviour but not inheritance of state. This removes the possibility – and thus temptation – of reusing code through inheriting the state of the parent class and then overriding some definitions. Stanza provides many other tools for enabling code reuse and inheritance is not necessary.

   b) It is not necessary to perform dynamic dispatch by abusing the object system. An algorithm requiring dynamic dispatch can be expressed directly using the `match` expression. This has the advantages of keeping algorithms clear, and also preventing objects from amassing state that is specific to one algorithm.

9. *First-class Functions:* From the perspective of the caller, multis are indistinguishable from functions. They can be stored in datastructures and passed as arguments, just as

functions can. The multimethod object system suits the functional programming style seamlessly.

10. *Learning Curve:* Unlike OCaml [35] and Haskell [31], the Stanza object and type system is designed to embrace both mutation and heterogeneity. These are natural and intuitive concepts and supported by all of the popular OOP languages, so programmers familiar with Python [47], Ruby [22], or Java [2], feel at ease with this aspect of Stanza.

5.5 Relation to Class-Based OOP Systems

The minimalism of the object system also has the following useful properties:

1. *No Concept of Constructor:* Due to the design of the `new` construct and disallowing inheritance of state, it is not necessary to introduce the concept of a *constructor* in Stanza. This has a number of positive consequences.

   Syntactically, the object being created by `new` cannot be referenced until after the object is created and fully initialized. This prevents the possibility of referencing a partially initialized object as arises in Java [2].

   Simple helper functions are used for creating objects, and thus there is no distinguished syntax necessary for creating a new object. This eliminates much of the need for complicated dependency injection frameworks to facilitate testing and debugging.

   Because there is no separate concept of constructors, there is also no need for the Factory design pattern [61], nor are there esoteric restrictions on when a constructor can call another constructor.

2. *Less Name Conflicts for Multiple Parent Types:* Due to the method namespace in class-based object-oriented languages, there can only be one method of a given name (and type signature) in a class. This can lead to problems if a class needs to implement multiple interfaces in Java [2], as demonstrated in the following example:

```
package officers;
public interface Officer {
  Order bark ();
}

package dogs;
public interface Dog {
  void bark ();
}

package mainprogram;
import officers.*;
import dogs.*;
```

```
public class PuppyOfficer implements Officer, Dog {
  void bark(){
    //Implementation of barking for a puppy.
  }
  Order bark(){
    //Implementation of barking an order.
  }
}
```

The interfaces `Officer` and `Dog` both have a method called `bark` – one is for retrieving the officer's `Order`, and the other is for making a loud sound. However, Java (and other class-based OOP languages) disallows declaring multiple methods with the same name and type signature in a single class. This unfortunate naming conflict makes it impossible for a single class to implement both interfaces, as in the case for `PuppyOfficer`.

In contrast, the following translation to Stanza has no such problem:

```
defpackage officers
public deftype Officer
public defmulti bark (o:Officer) -> Order

defpackage dogs
public deftype Dog
public defmulti bark (d:Dog) -> False

defpackage main-program :
  import officers
  import dogs

public deftype PuppyOfficer <: Officer & Dog
public defn PuppyOfficer () :
  new PuppyOfficer :
    defmethod officers/bark (this) :
      ;Implementation of barking an order.
    defmethod dogs/bark (this) :
      ;Implementation of barking for a puppy.
```

The type `PuppyOfficer` is a subtype of both `Officer` and `Dog`, and provides instance methods for both the `bark` multi in the `officers` package, and the `bark` multi in the `dogs` package. The caller has the option to import either the `dogs` or the `officers` package depending upon the multi to call. This allows a `PuppyOfficer` to be treated equally as a `Dog` when appropriate, or as an `Officer`.

3. *No Concept of Fields:* The multimethod system does not need a distinguished concept of "fields" nor the standard suite of mechanisms for declaring and manipulating them. Fields are modeled by declaring multis for retrieving a value from the object; fields can be made mutable by declaring multis for setting their value; and field visibility can be controlled by using the package visibility modifiers on the multi declarations.

4. *No Identity-Comparison Operator:* Stanza provides no identity-comparison operator for comparing whether two objects are represented by the same pointer. Comparisons of values must be done by comparing their contents. This subtle property means that two objects with the same underlying content are indistinguishable. An important optimization made possible by this property is the ability to *inline objects*. Consider the following example:

   ```
   defn f (p:Point2D) :
     ...
   f(Point2D(1.0, 3.0))
   ```

   The function `f` requires a `Point2D` object, and it is called with a point freshly created from the values `1.0` and `3.0`. With object inlining, the Stanza compiler is free to pass the numbers `1.0` and `3.0` individually to `f` instead of first wrapping them up in a `Point2D` object.

Chapter 6

The Stanza Targetable Coroutine System

The targetable coroutine system plays an integral role in the overall design of the Stanza language by simultaneously serving two separate but related purposes:

1. It serves as a powerful construct for expressing cooperative multitasking.

2. It serves as Stanza's *only* non-local control flow operator.

In regards to purpose 1, Stanza's targetable coroutine system plays a similar role as Lua's [30] and Python's [47] coroutine constructs, and Ruby's [22] and CLU's [37] generator constructs. Cooperative multitasking is a powerful concept and can be applied towards a variety of applications, including:

- *On-Demand Computation:* Coroutines can be used to compute successive items in a sequence on-demand. A common example is to use coroutines to compute the breadth-first or depth-first traversal of a tree datastructure. If an algorithm does not require the entire sequence then time is saved by not computing the rest of the traversal.

- *Representing Infinite Sequences:* Coroutines naturally represent sequences of infinite length. This is often useful from a software engineering perspective as it obviates the need to compute the required length ahead of time.

- *Modeling Concurrency:* Many applications contain conceptually concurrent components. Video games, for instance, often create the illusion of multiple animated characters acting independently and concurrently. A separate coroutine can be used to drive each character's behaviour.

- *Distributing a Computation Across Time:* For interactive applications, it is important that there aren't long periods of inactivity where the program is unresponsive to user input. Long computations should be broken up into shorter segments and the program should poll for user input in-between executing each segment. Coroutines automatically save the state of a computation and can be used to trivially break up a long computation.

In regards to purpose 2, Stanza's targetable coroutine system plays a similar role as Scheme's continuation construct. The Stanza language and its core libraries is designed around the use of coroutines as a general-purpose control flow operator, and relies upon it for:

1. returning early from functions,
2. breaking out of loops,
3. skipping to the next iteration of a loop, and
4. handling error conditions and exceptional circumstances.

From a language design perspective, the decision to have coroutines serve as a control flow operator is an unconventional one. We will explain the motivation underlying this decision by first examining the shortcomings of the standard suite of control flow operators – `return`, `break`, `continue`, etc. – in the context of Stanza's design.

6.1 Shortcomings of Standard Control Flow Operators

Recall that Stanza's `for` construct, despite looking like a looping construct, is syntactic sugar for calling a higher-order function. Thus the following loop in the `main` function:

```
defn main () :
  for x in xs do :
    println(x)
```

can be re-expressed equivalently as the following:

```
defn main () :
  defn loop-body (x) :
    println(x)
  do(loop-body, xs)
```

This example is archetypal of Stanza's philosophy of maximizing expressiveness through the design of a minimal set of orthogonal constructs: iteration is expressed through tail-recursion, and looping constructs are abstracted as higher-order functions.

Now consider changing Stanza by adding support for the standard `return` construct, which immediately returns a value from a function. We will use this construct to have `main` return `false` if it encounters a negative number in `xs`:

```
defn main () :
  for x in xs do :
    if x < 0 : return false
    println(x)
```

While seemingly innocuous, this use of `return` exposes an ambiguity in our design. The following re-expresses the above code without the `for` macro to emphasize the ambiguity:

```
defn main () :
  defn loop-body (x) :
    if x < 0 : return false
    println(x)
  do(loop-body, xs)
```

It is easy to see in this form that it is unclear what `return false` should do.

1. Should it return from the `loop-body` function immediately, in which case it prevents negative numbers in `xs` from being printed?

2. Or should it return from the `main` function immediately, in which case printing stops as soon as the loop reaches the first negative number in `xs`?

Depending on the circumstance, it is reasonable that a programmer might want to do either.

The root cause of the ambiguity is the `return` construct's lack of an explicit *target*, and is one of the key problems we were required to solve in the design of Stanza's targetable coroutine system. The other standard operators, such as `break` and `continue`, suffer from similar ambiguities.

6.2 Core Functions

The core of the targetable coroutine mechanism is defined by three functions:

1. the `Coroutine` function, for creating a new coroutine,

2. the `resume` function, for resuming an existing coroutine, and

3. the `suspend` function, for suspending an existing coroutine.

Creating a Coroutine

The coroutine creation function has the following type signature:

```
defn Coroutine<I,O> (body: (Coroutine<I,O>, I) -> O) -> Coroutine<I,O>
```

It requires two type arguments and a two-arity function, and returns a new coroutine. A coroutine of type `Coroutine<I,O>` allows values of type `I` to enter, and values of type `O` to exit.

Here is an example of creating a coroutine that allows `Int` values to enter, and `String` values to exit:

```
defn body (co:Coroutine<Int,String>, x0:Int) -> String :
  ...
val co = Coroutine<Int,String>(body)
```

The first argument of the `body` function represents the coroutine to be created; the second argument represents the first value to enter the coroutine; and the return value of the function is implicitly the last value to exit the coroutine.

Note that the `body` function is not executed until the coroutine is first resumed. The creation of a coroutine does not have any side-effects.

Resuming a Coroutine

The coroutine resume function has the following type signature:

```
defn resume<?I,?O> (co:Coroutine<?I,?O>, v-in:I) -> O
```

It requires two arguments – the coroutine to resume, and the value to send into the coroutine – and returns the next value to exit the coroutine. The type arguments for `resume` are captured from the coroutine input/output types.

Here is an example of resuming our previously created coroutine of type `Coroutine<Int,String>`:

```
val v-out:String = resume(co, 42)
```

The coroutine requires entering values to be of type `Int` – we send in `42` – and promises that exiting values are of type `String` – the next one of which we bind to `v-out`.

Suspending a Coroutine

The coroutine suspend function has the following type signature:

```
defn suspend<?I,?O> (co:Coroutine<?I,?O>, v-out:O) -> I
```

It requires two arguments – the coroutine to suspend, and the value to send out of the coroutine – and returns the next value to enter the coroutine. Like `resume`, the type arguments for `suspend` are captured from the coroutine input/output types.

Here is an example of suspending our previously created coroutine of type `Coroutine<Int,String>`:

```
val v-in:Int = suspend(co, "Hello World")
```

The coroutine requires exiting values to be of type `String` – we send out `"Hello World"` – and promises that entering values are of type `Int` – the next one of which we bind to `v-in`.

6.3 Semantics

The semantics of the core coroutine functions will be demonstrated through the following series of progressively more complex examples:

1. The creation of a coroutine.

2. The first call to `resume` on a coroutine with no calls to `suspend`.

3. The first call to `resume` on a coroutine with calls to `suspend`.

4. Subsequent calls to `resume` on a coroutine with calls to `suspend`.

Creation of a Coroutine

The following creates a coroutine of type `Coroutine<Int,String>`, in which integers enter the coroutine, and strings exit the coroutine:

```
defn body (co:Coroutine<Int,String>, x0:Int) -> String :
  println("Started coroutine (x0 = %_)" % [x0])
  println("Leaving coroutine")
  "Hello world"
val co = Coroutine<Int,String>(body)
```

A coroutine, `co`, is created from a function that prints out `x0`, the first value to enter it, and then prints a message to indicate it is finished before returning `"Hello World"`. Note that `body` does not contain any calls to `suspend`.

As previously mentioned, creating a coroutine does not cause its body function to be executed, thus nothing is printed to the screen when the above code is evaluated.

First Call to Resume with No Calls to Suspend

We now call `resume` on the `co` coroutine created previously, with `42` as the initial value to be sent into the coroutine:

```
val result = resume(co, 42)
```

This begins the execution of the coroutine body function with the entering value `x0` bound to `42`. Because there are no calls to `suspend`, the body function executes to completion and prints the following to the screen:

```
Started coroutine (x0 = 42)
Leaving coroutine
```

When the body function finishes executing, the control flow returns to just after where `resume` is called. The `resume` function returns the result of evaluating the body function, which is `"Hello world"` in this case.

Thus, in the absence of `suspend`, calling `resume` is identical to calling the coroutine body function with some initial entering value.

First Call to Resume with Calls to Suspend

We now revise the body function of the co coroutine and insert a call to suspend with the exiting value "Live":

```
defn body (co:Coroutine<Int,String>, x0:Int) -> String :
  println("Started coroutine (x0 = %_)" % [x0])
  val x1 = suspend(co, "Live")
  println("Resumed coroutine (x1 = %_)" % [x1])
  println("Leaving coroutine")
  "Hello world"
val co = Coroutine<Int,String>(body)
```

We call resume on the co coroutine as we did previously, with 42 as the initial value to be sent in:

```
val result = resume(co, 42)
```

which again begins the execution of the coroutine body function with the entering value x0 bound to 42. The following is printed to the screen:

```
Started coroutine (x0 = 42)
```

At this point however, instead of executing to completion, the call to suspend causes the control flow to immediately return to just after where resume is called. The resume function now returns the argument that was passed to suspend, which is "Live" in this case.

Thus, the suspend function acts as a way to immediately leave a coroutine with some exiting value.

Subsequent Calls to Resume

Now we consider the case of calling resume repeatedly on the same coroutine. As demonstrated previously, the first call to resume:

```
val result = resume(co, 42)
```

results in the following being printed to the screen:

```
Started coroutine (x0 = 42)
```

with result being bound to "Live".

The next call to resume:

```
val result2 = resume(co, 13)
```

resumes execution of the body function starting from the previous call to suspend. The argument passed to resume is returned as the result of calling suspend, thus x1 is bound to 13. From there, the body function executes until completion, printing out:

```
Resumed coroutine (x1 = 13)
Leaving coroutine
```

before returning `"Hello World"`. Upon completion, the control flow returns to just after where `resume` is called for the second time, and `result2` is bound to `"Hello World"`.

Thus, the `suspend` function automatically saves the state of executing the body function before leaving the coroutine. Subsequent calls to `resume` resume the body function from its saved state.

6.4 Applications of Coroutines

We now show three example applications of Stanza's coroutine system:

1. Using a coroutine to compute, on-demand, the breadth-first traversal of a binary tree.
2. Using a coroutine as a control flow operator for returning early from a function.
3. Using a coroutine as a control flow operator for skipping to the next iteration of a loop.

On-Demand Computation

In this example, we will write a utility function, `traverse`, that computes the breadth-first traversal of a binary tree, and use it to find the first negative integer in a tree. To avoid unnecessarily continuing to traverse the tree once the integer has been found, we will use a coroutine to compute the traversal *on-demand*.

Here is the definition of a node in the binary tree:

```
defstruct Node :
  value: Int
  left: Node|False
  right: Node|False
```

Every node contains an integer value, and may recursively contain a left and right node.

Here is the definition of the `traverse` function:

```
defn traverse (root:Node) -> Coroutine<False,Int|False> :
  Coroutine<False,Int|False> $ fn (co, x0) :
    let loop (node:Node|False = root) :
      match(node) :
        (node:Node) :
          suspend(co, value(node))
          loop(left(node))
          loop(right(node))
        (node:False) :
          false
```

It accepts a root node to traverse, and returns a coroutine of type `Coroutine<False,Int|False>`. The coroutine accepts `false` as entering values, and sends out all the integers in the tree in depth-first order, followed by `false` at the end. The syntax `f $ x` is a shorthand for `f(x)`.

To find the first negative integer in a tree, we then call traverse on its root to obtain the traversal coroutine, and resume it repeatedly to retrieve each subsequent integer in the traversal:

```
defn first-negative (root:Node) -> Int :
  val co = traverse(root)
  let loop () :
    match(resume(co, false)) :
      (x:Int) : x when x < 0 else loop()
      (x:False) : fatal("No negative number in tree.")
```

We halt the program with an error message if we reach the end of the traversal without encountering a negative number.

Returning Early from a Function

In this example, we will use a coroutine to emulate the common return construct that is found in other programming languages for immediately returning from a function.

To demonstrate, we will translate the earlier example in this chapter:

```
defn main () :
  for x in xs do :
    if x < 0 : return false
    println(x)
```

to proper Stanza code.

Our strategy stems from the observation that the coroutine suspend function behaves very similarly to the common return construct. In addition to immediately exiting a block of code, it *also* saves the state of the computation so that the block of code may be later resumed. Thus, all that is needed to emulate the return construct is to refrain from resuming the coroutine after the first call to suspend:

```
defn main () :
  defn body (co:Coroutine<False,False>, x0:False) :
    for x in xs do :
      if x < 0 : suspend(co, false)
      println(x)
  resume(Coroutine<False,False>(body), false)
```

The main function wraps up its entire previous body in a new coroutine, and then resumes the coroutine a single time. Within the coroutine body, suspend is used to emulate return.

For the sake of clarity, the above pattern can be abstracted into a convenient higher-order function:

```
defn with-exit<T> (block: (T -> Void) -> T) :
  defn body (co:Coroutine<False,T>, x0:False) :
    defn exit (x:T) :
      suspend(co, x)
      fatal("Unreachable")
    block(exit)
  resume(Coroutine<False,T>(body), false)
```

The `with-exit` function accepts as input a function representing a block of code, and passes it an *exit function* that can be used for immediately exiting the block.

Using `with-exit`, we can re-express the `main` function as the following:

```
defn main () :
  with-exit<False> $ fn (return) :
    for x in xs do :
      if x < 0 : return(false)
      println(x)
```

Skipping to the Next Loop Iteration

In this example, we will use a coroutine to emulate the common `continue` construct that is found in other programming languages for skipping to the next iteration of a loop.

To demonstrate, we will translate the following:

```
defn main () :
  for x in xs do :
    if x < 0 : continue
    println(x)
```

to proper Stanza code. Instead of returning from `main` upon encountering a negative number, we wish now to skip to the next iteration of the loop.

Our strategy for this example stems from noticing the similaries between the `return` construct and the `continue` construct. The former immediately exits a function, while the latter can be thought of as a construct for immediately exiting a loop body. We can therefore reuse the `with-exit` function to execute the `for` body, and use the provided exit function to emulate `continue`:

```
defn main () :
  for x in xs do :
    with-exit<False> $ fn (continue) :
      if x < 0 : continue(false)
      println(x)
```

6.5 The Label Construct

The `with-exit` function defined previously is satisfactory for the purposes of emulating the `return` and `continue` constructs, but its usage is syntactically unpleasant. Because it is used so commonly, the Stanza core library provides the `label` macro as a syntactic shorthand for calling the `with-exit` function.

The expression:

```
label<T> exit :
  body
```

is equivalent to the following explicit call to `with-exit`:

```
with-exit<T> $ fn (exit) :
  body
```

There is also a version of the `label` construct that does not require an explicit return type – in which case we assume it returns `false`, and the exit function correspondingly requires no arguments.

Using the `label` construct, the example emulating the `return` construct can be re-expressed as:

```
defn main () :
  label return :
    for x in xs do :
      if x < 0 : return()
      println(x)
```

And the example emulating the `continue` construct can be re-expressed as:

```
defn main () :
  for x in xs do :
    label continue :
      if x < 0 : continue()
      println(x)
```

6.6 Nested Coroutines

One key aspect of Stanza's coroutine mechanism is the requirement for programmers to explicitly specify which coroutine to suspend. This is unlike the coroutine mechanisms of other languages – such as Lua [30], Python [47], or Ruby [22] – which allows suspension only from the currently-running coroutine. Stanza's coroutines are called *targetable* coroutines for this reason: the target coroutine must be specified explicitly. This difference allows for coroutines – and constructs built upon coroutines – to be well-behaved when nested within each other, and is key to enabling the use of Stanza's coroutines as a general-purpose control flow operator.

As an example, the following function, `product-of-non-negative`, computes the product of all the non-negative integers in an array:

```
defn product-of-non-negative (xs:Array<Int>) :
  label<Int> return :
    var product = 1
    for x in xs do :
      label continue :
        if x < 0 : continue()
        if x == 0 : return(0)
        product = product * x
    product
```

It is implemented with two `label` constructs, one nested within the other. When iterating through the array, the outer `label` allows the function to return `0` immediately if any of the numbers is `0`. Within the `for` loop, the inner `label` allows us to skip to the next iteration if the number is negative.

Because the respective scopes corresponding to the two exit functions (`return` and `continue`) is made explicit, there is no ambiguity in the above code.

6.7 Implementation

Stanza uses an implementation technique known as *segmented stacks* for both its coroutine system and its automatic stack extension system. A common language implementation strategy – such as for C [33] and C++ [55] – is to separate the machine memory into two sections: stack memory and heap memory. The stack memory holds the activation records containing the local variables and return addresses for each function call, while the heap memory holds all dynamically-allocated structures. In this scheme, the recursion depth is limited by the amount of stack memory, as each recursive call requires the allocation of an activation record.

Stanza does not distinguish between stack memory and heap memory, and instead uses all available memory as heap memory. The stack for holding the function activation records are allocated as small 4KB datastructures from the heap memory, and the machine stack register holds a pointer to the currently active stack. Function calls allocate their activation records from the top of the currently active stack, and when the current stack is full, Stanza's automatic stack extension mechanism allocates another 4KB stack from the heap. In this scheme, the recursion depth is limited only by the total available memory.

Each coroutine saves the state of its computation in its own dedicated stack. Resuming and suspending a coroutine involves simply assigning its stack to the machine stack register. To optimize coroutine creation and stack extension, we use a free-list to maintain a set of pre-allocated stack datastructures. Coroutine creation, resumption, and suspension, can all be done in a handful of machine instructions.

Coroutine States

To implement the targetable coroutine system, where the user explicitly requests for the resumption or suspension of a *specific* coroutine, every coroutine holds:

1. a pointer to the coroutine that resumed it, and
2. a flag for indicating the state of the coroutine.

Here is the terminology we will use. If coroutine A resumed coroutine B, then we define A as the *direct parent* of B, and B as the *direct child* of A. A coroutine can only have a single direct parent and a single direct child. The parent of A (and its parent and so forth) is defined as an *indirect parent* of B. Vice versa, the child of B (and its child and so forth) is defined as an *indirect child* of A. The coroutine that is currently executing is defined as the *current coroutine.* The initial coroutine in which a Stanza program starts is defined as the *main coroutine.*

At any given time, a coroutine can be in one of four states:

1. *Active:* The coroutine is currently running and is free to be suspended.

2. *Open:* The coroutine is not running and is waiting to be resumed.

3. *Suspended:* The coroutine is not running but cannot be resumed.

4. *Closed:* The coroutine has finished running and can no longer be resumed.

A Stanza program starts in the main coroutine which is always *active.*

When a coroutine is first created, it is *open* to indicate that it is free to be resumed.

The `resume` function can only be called on an *open* coroutine. When a coroutine A resumes another coroutine B, the parent coroutine A stays *active*, B changes from *open* to *active*, and all of the children of B change from *suspended* to *active.* If B has no children, then B is set as the current coroutine. Otherwise, the most distant child of B – the one with no child of its own – is set as the current coroutine.

The `suspend` function can only be called on an *active* coroutine – which necessarily implies that it is either the current coroutine or a parent of the current coroutine. The coroutine upon which `suspend` is called changes from *active* to *open*, all of its children changes from *active* to *suspended*, and its direct parent will be set as the current coroutine.

A coroutine is *closed* once its body function has finished evaluating, and can no longer be resumed.

Figure 6.1 illustrates the effect of the core functions on the coroutine states:

1. In the initial configuration, we assume that `suspend` has been called twice already, resulting in two suspended chains of coroutines: the cde chain and the fg chain. Coroutines c and f are *open* and free to be resumed, and the current coroutine is b.

2. We then decide to call `resume` on c, which sets b as the parent of c. Coroutines c, d, and e become *active*, and the current coroutine is now set to e.

3. We then call `suspend` on b, which sets the current coroutine to a. As in the initial configuration, we are back to two suspended chains of coroutines, but now b is *open* and c is *suspended.*

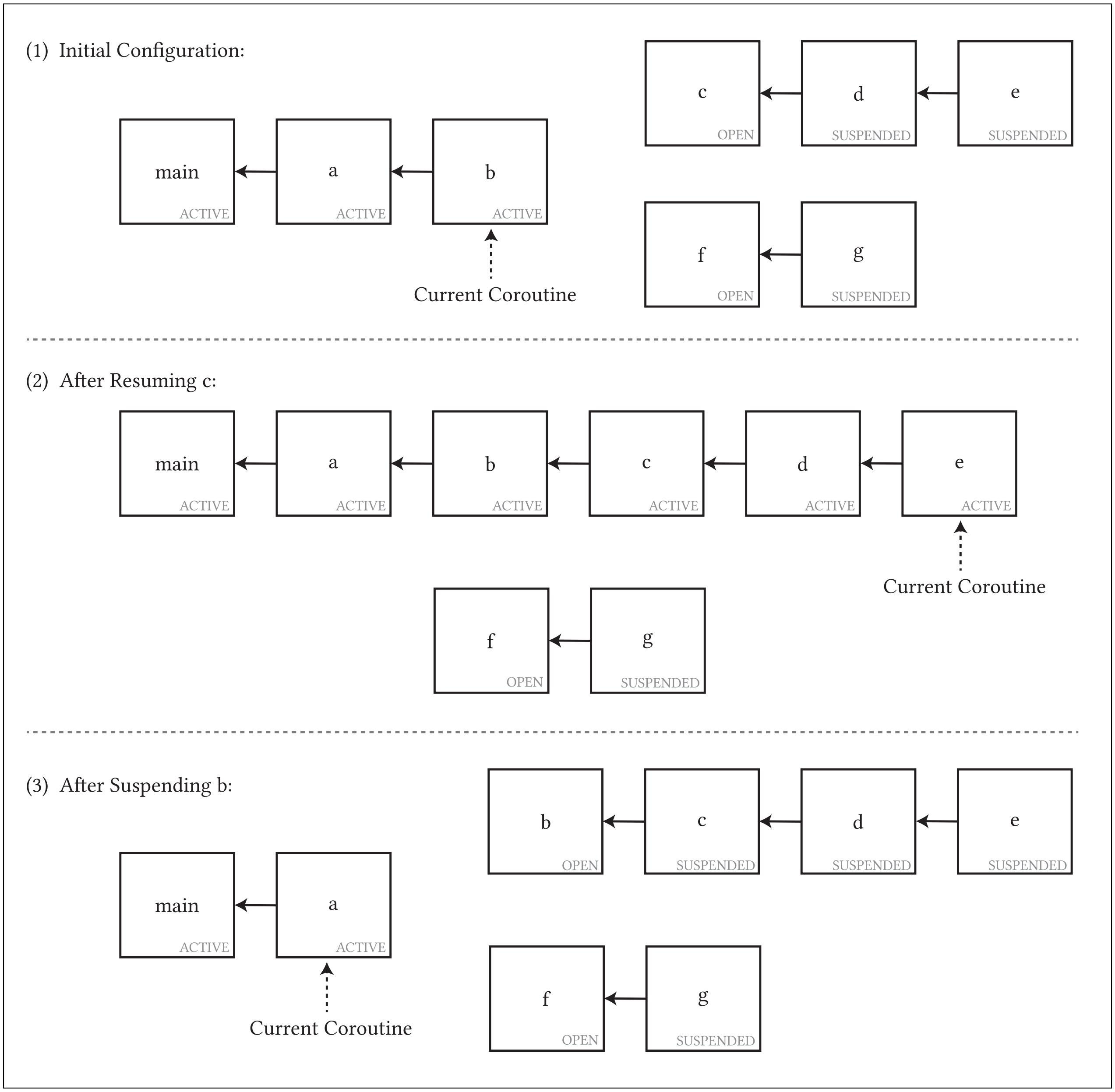

Figure 6.1: Semantics of Nested Coroutines

Optimization Opportunities

The requirement to explicitly specify which coroutine to suspend provides the compiler more information for optimizing the code. Consider the following example implementation of `sum`:

```
defn sum (f: () -> Int|False) -> Int :
  label<Int> return :
    var accum = 0
    while true :
      match(f()) :
        (x:Int) : accum = accum + x
        (x:False) : return(accum)
    fatal("Unreachable")
```

`sum` accepts a function `f` that returns either an integer or `false`, and calls it repeatedly until `false` is returned. The result of calling `sum` is the sum of all the integers returned by `f`.

The design of Stanza's targetable coroutine system provides two key pieces of information to the compiler:

1. The `return(accum)` call is the only invocation of the `return` exit function, which is the only way of suspending the coroutine.

2. The `return` exit function does not *escape* outside of `sum`. It is not possible for `return` to be called from any other location. In particular, `f` is guaranteed not to call `return`.

From that information, the compiler can deduce *all* points at which the coroutine can be suspended, and implement the function using simple goto instructions:

```
defn sum (f: () -> Int|False) -> Int :
  var accum = 0

LOOP_START:
  match(f()) :
    (x:Int) : accum = accum + x
    (x:False) : goto END_OF_BLOCK
  goto LOOP_START

LOOP_END:
  fatal("Unreachable")

END_OF_BLOCK:
  accum
```

Hence, the overhead of allocating a new coroutine, resuming, and suspending it, is completely eliminated.

Languages like Lua [30], Python [47], or Ruby [22], which allow users to implicitly suspend the current coroutine, prohibit the above optimization. The compiler is forced to allocate and launch a new coroutine because `f` may choose to suspend it.

6.8 Other Functions

Here are the rest of the functions available for operating on coroutines:

1. The `open?` function has the following type signature:

   ```
   defn open? (c:Coroutine) -> True|False
   ```

   and returns `true` if the coroutine is *open* and returns `false` otherwise. A coroutine can only be resumed if `open?` returns `true`.

2. The `active?` function has the following type signature:

   ```
   defn active? (c:Coroutine) -> True|False
   ```

   and returns `true` if the coroutine is *active* and returns `false` otherwise. A coroutine can only be suspended if `active?` returns `true`.

3. The `close` function has the following type signature:

   ```
   defn close (c:Coroutine) -> False
   ```

   and is used to close an *open* coroutine. It indicates that the program will not resume the coroutine again, and frees its associated resources.

4. The `break` function has the following type signature:

   ```
   defn break<?O> (c:Coroutine<?,?O>, v-out:O) -> Void
   ```

   and is used to suspend and close an *active* coroutine. It behaves identically to calling `suspend` followed by `close` on a coroutine.

6.9 Design Benefits of a General-Purpose Control Flow Operator

Stanza's philosophy of using higher-order functions for representing common iteration constructs was heavily inspired by the Smalltalk [24] and Ruby [22] programming languages. It is an elegant philosophy that simultaneously decreases the language complexity while increasing its expressiveness. But, though we agree with this philosophy, we feel that its full benefits can only be realized when combined with a general-purpose control flow operator, which is lacking in both Smalltalk and Ruby.

As an example, the following is a snippet of Ruby code for iterating through each element of a collection:

```
def main ()
  xs.each do |x|
    puts(x)
  end
end
```

The `.each` method plays a very similar role as Stanza's `do` function in the overall design of Ruby. It accepts a *block* – a form of Ruby closure – and calls the block once for each element in the collection. The block in the above example accepts an argument, `x`, and prints it to the screen.

Now we use Ruby's `return` construct to immediately exit the function upon encountering a negative number:

```
def main ()
  xs.each do |x|
    return nil if x < 0
    puts(x)
  end
```

At this time, the suspicious reader should wonder how Ruby avoids the ambiguity discussed in Section 6.1. Why does `return nil` return from the `main` function instead of returning from the block passed to the `.each` method? And furthermore, what control flow operator *would* return from the block instead of from `main`?

Because of its lack of a general-purpose control flow operator, Ruby instead introduces a variety of function constructs that differ in their behaviour with respect to control flow operators. In the example above, `return nil` returns from `main` instead of from the block because the `return` construct is defined to return from the lexically-enclosing method and ignore blocks.

Ruby provides five different ways of wrapping up code – *blocks*, *procs*, *lambdas*, *methods*, and *Method objects* – along with a suite of control flow operators – `return`, `break`, and `next`, among others. Blocks and methods are not first-class values, while procs, lambdas, and Method objects are. The `return` construct is used to return from a lambda, method, or Method object. The `next` construct is used to return from a lambda, block, or proc. Calling `return` within a block or proc returns from its lexically-enclosing method or lambda. Calling `break` within a block or proc returns from the method that called the block or proc.

In contrast, Stanza provides one way of wrapping up code – functions – and one control flow operator – targetable coroutines (with `label` as a convenient shorthand).

Though we used Ruby to illustrate the issue, the interaction between control flow operators and function constructs is a fundamental design challenge. The Smalltalk language also provides two ways to wrap up code – *methods* and *blocks*. The return operator returns from the lexically-enclosing method, but there is no operator for returning early from a block.

6.10 Comparison to Continuations

The decision to use coroutines as the sole non-local control flow operator in Stanza was inspired by Scheme's [56] `call/cc` function for reifying the current continuation. Like Stanza, the `call/cc` function is Scheme's only non-local control flow operator, and other common constructs – such as throwing exceptions and exception handlers – are built upon `call/cc` as a set of macros.

The semantics of Stanza's targetable coroutines can be perfectly emulated using `call/cc`, which makes the `call/cc` function strictly *more expressive* than coroutines. However, this expressivity comes at the cost of being notoriously difficult to learn, being inefficient to execute, and interacting poorly with mutation.

Learning Difficulty

Coroutines can be intuitively understood as representing a separate computation that is interleaved with the main program. The programmer's intuitive notion of time is undisturbed: *time flows forward*. Each coroutine takes turns to advance for some number of time steps.

In contrast, `call/cc` reifies a continuation that represents a single snapshot in time to which the program can jump arbitrarily. Thus, time is no longer restricted to only flow forward, and programmers can no longer rely upon their intuitive notion of time to understand code that makes heavy use of continuations.

The following shows a notoriously perplexing usage of continuations known as the Yin-Yang puzzle by David Madore [38]:

```
(let* ((yin
         ((lambda (cc) (newline) cc)
          (call/cc (lambda (cc) cc))))
       (yang
         ((lambda (cc) (display #\*) cc)
          (call/cc (lambda (cc) cc)))))
  (yin yang))
```

Though only seven lines long, predicting the printed output of the Yin-Yang puzzle has proven to be far from trivial.

For the curious reader, it turns out that it prints out an increasing number of asterisks on each line:

```
*
**
***
****
*****
...
```

There are many explanations for how the code works, none of them simple.

Execution Efficiency

As described in Section 6.7, the state of a computation is fully described by the stack of activation records corresponding to each function call. Thus, reifying the current continuation can be done by creating a copy of the current stack of activation records. To invoke a continuation, a copy of this saved stack is set as the active stack.

Because the same continuation can be invoked repeatedly, the saved stack must be copied before being set as the active stack. If the stack was quite deep at the time that the continuation was reified, then copying it is an expensive procedure. This is substantially less efficient than Stanza's coroutine implementation, which requires only a few instructions for creation, resumption, and suspension.

One-shot continuations – continuations which can only be invoked once – do not suffer from the same copying overhead as full continuations. They have the same efficiency as Stanza's coroutines, but consequently are also less expressive than full continuations.

Interactions with Mutation

Both Stanza and Scheme are impure languages that support an assignment construct for mutating local variables. In the presence of mutation, the intuitive understanding of continuations as a snapshot in time is now incomplete, as state that has been mutated is *not* reverted when a continuation is invoked.

Consider the following Scheme function, `do-n-times`, which calls the argument function, `f`, repeatedly for `n` times:

```
(define (do-n-times n f)
  (let ((i 0))
    (while (< i n)
      (f)
      (set! i (+ i 1)))))
```

Internally, it defines a loop variable, `i`, and loops while `i` is less than `n`. On each iteration we call `f` and increment `i`.

The following uses `do-n-times` to print ten asterisks on a line:

```
(do-n-times 10 (lambda (i) (display #\*)))
```

Though the function works, the implementation of `do-n-times` is not idiomatic of Scheme code, which prefers expressing loops through tail-recursion instead of mutating a loop variable. Here is a version of `do-n-times` implemented in terms of tail-recursion:

```
(define (do-n-times n f)
  (when (> n 0)
    (f)
    (do-n-times (- n 1) f)))
```

Ideally, the choice of how to implement `do-n-times` *should* be an internal implementation detail. That is, from the perspective of the caller, no code should be able to discern whether `do-n-times` is implemented using tail-recursion or using a mutable loop variable. However, the following example demonstrates that this is not true in the presence of `call/cc`:

```
(let ((saved-cc #f)
      (resumed #f))
  (do-n-times 10
    (lambda ()
      (set! resumed
        (call/cc (lambda (cc)
          (when (not saved-cc) (set! saved-cc cc))
          (or resumed #f))))
      (display #\*)))
  (newline)
  (when (not resumed)
    (saved-cc #t)))
```

A variable `saved-cc` is used to save the continuation of the first iteration, and then this continuation is invoked once. The `resumed` variable prevents the infinite loop resulting from invoking the continuation repeatedly.

When `do-n-times` is implemented using tail-recursion, the following is printed:

```
**********
**********
```

However, when `do-n-times` is implemented through mutating a loop variable, the following is printed:

```
**********
*
```

The consequence of this demonstration is that it is extraordinarily difficult to know whether two implementations of a function are actually equivalent in the presence of full continuations. Simply refactoring a loop to use tail-recursion instead of mutation may result in unpredictable changes in program behaviour.

Chapter 7

The LoStanza Sublanguage

In the design of any programming language there is a fundamental tension between the desire for a well-defined high-level machine-independent semantics and the desire for fine-grained transparent control over the underlying hardware. Stanza has been presented so far as a high-level language, and the details of how programs are actually mapped onto the primitive operations of the machine have been intentionally concealed. How objects are represented, how memory is addressed, how memory is allocated and deallocated, how to track the state of coroutines, and how to track whether an object can be safely deallocated, are just a few examples of details that are automatically managed by the Stanza system.

Ultimately, however, a practical programming language must be able to interface with the rest of the software ecosystem: the operating system and existing code written in other programming languages. This requires the ability to execute code not originally written in Stanza and manipulate datastructures not originally created within Stanza. Since their low-level details are not managed by Stanza, programmers require fine-grained control to manipulate memory and execute code according to the conventions set by the other systems. The task of the language designer is to provide an interface to the software ecosystem without sacrificing the benefits of having a high-level machine-independent semantics.

7.1 LoStanza

There are three broad methods for providing an interface between a high-level programming language and the surrounding ecosystem:

1. Provide the necessary low-level constructs for controlling the hardware.

2. Control the hardware through primitive operations written in a different programming language.

3. Separate the language into a high-level and a low-level sublanguage. This is the method adopted by Stanza.

Stanza is divided into two sublanguages: a high-level and low-level language respectively called HiStanza and LoStanza. HiStanza has a well-defined machine-independent semantics, and is what the majority of Stanza code is written in – including all the examples presented thus far. HiStanza is strongly-typed and guarantees that all programs either execute to completion or fail gracefully with an error message.

LoStanza is a low-level language that provides constructs for direct control over memory and for executing code written in other languages. Unlike HiStanza, details of the machine architecture - such as word width - *are* exposed to programmers. LoStanza also offers no protection against inadvertently writing to an inappropriate memory location nor provides any guarantees about the system behaviour if it occurs.

LoStanza provides constructs for control over the following details:

1. *Memory Addresses:* Machine addresses are directly exposed to programmers as *pointers*. Pointers can be created by requesting the address of an in-memory object or directly by casting from an integer, and can be manipulated using arithmetic instructions.

2. *Memory Load/Store:* Values may be loaded from and stored to arbitrary memory locations through pointers.

3. *Object Layout:* Objects can be defined with an explicit layout in memory.

4. *Calls to Foreign Code:* Functions written in other languages can be called from Stanza.

5. *Calls from Foreign Code:* Stanza functions can be declared such that they can be called from other languages.

Example

The following shows the declaration of a LoStanza function:

```
lostanza defn sum-to (n:int) -> int :
  var accum:int = 0
  for (var i:int = 0, i < n, i = i + 1) :
    accum = accum + i
  return accum
```

The `sum-to` function accepts a single *primitive* integer argument, and returns a primitive integer. All LoStanza functions are preceded by the `lostanza` keyword, and must have explicit argument and return types. Unlike HiStanza, which is an expression-oriented language, LoStanza separates the concept of statements from expressions. Thus the function ends with the `return` statement to return the final value of `accum`, which, unlike in HiStanza, *is* a built-in construct.

7.2 LoStanza Core Forms

The following lists the complete set of LoStanza core forms:

```
Top Level Forms:
  ($ls-def name type value)                                (Define Value)
  ($ls-defvar name type value)                             (Define Variable)
  ($ls-extern name type)                                   (External Variable)
  ($ls-deftype name parent (fields ...) rfield             (Define Type)
                           (types ...) rtype)
  ($ls-defn name (args ...) (a1 ...)  a2 body ...)         (Define Function)
  ($ls-extern-fn name (args ...) (a1 ...) a2 body ...)     (External Function)
  ($ls-defmethod name (args ...) (a1 ...) a2 body ...)     (Define Method)

Statement Forms:
  ($ls-set name exp)                                 (Assignment)
  ($ls-labels blocks ...)                            (Labeled Blocks)
  ($ls-block name (args ...) (ts ...) body ...)      (Block)
  ($ls-goto name args ...)                           (Goto Block)
  ($ls-return exp)                                   (Return)
  ($ls-let stmts ...)                                (New Scope)
  ($ls-if pred conseq alt)                           (If Statement)
  ($ls-match (args ...) branches ...)                (Match Statement)
  ($ls-branch (args ...) (ts ...) body ...)          (Match Branch)
  ($ls-func f)                                       (Closure Reference)

Expression Forms:
  ($ls-new type args ...)      (New Object)
  ($ls-struct type args ...)   (Struct)
  ($ls-addr exp)               (Address-of)
  ($ls-addr! exp)              (Forced Address-of)
  ($ls-deref exp)              (Dereference)
  ($ls-slot exp index)         (Indexed Slot)
  ($ls-field exp name)         (Named Field)
  ($ls-do f args ...)          (Function Call)
  ($ls-call-c f args ...)      (C Function Call)
  ($ls-prim f args ...)        (Primitive Call)
  ($ls-sizeof type)            (Size of Type)
  ($ls-tagof name)             (Tag of Type)
  ($ls-as exp type)            (Cast)
  ($ls-and a b)                (Short-circuiting And)
  ($ls-or a b)                 (Short-circuiting Or)

Type Forms:
  ($ls-byte)                (Byte)
  ($ls-int)                 (Int)
  ($ls-long)                (Long)
  ($ls-float)               (Float)
  ($ls-double)              (Double)
  ($ls-?)                   (Unknown Type)
  ($ls-of name args ...)    (Struct)
```

```
($ls-ptr t)                (Pointer)
($ls-ref t)                (Reference)
($ls-fn (a1 ...) ar a2)    (Function)
```

7.3 Low-Level Types

LoStanza defines its own notions of types separately from HiStanza. Unlike HiStanza types – which are names given to categories of values with common interfaces, and are independent from the machine – LoStanza types denote specific bit patterns as stored in machine memory.

The following *primitive* types denote numbers:

- `byte` denotes an 8-bit unsigned integer.
- `int` denotes a 32-bit signed integer in two's complement format.
- `long` denotes a 64-bit signed integer in two's complement format.
- `float` denotes a 32-bit real number in IEEE-754 single precision floating-point format.
- `double` denotes a 64-bit real number in IEEE-754 double precision floating-point format.

The *pointer* type, `ptr<t>`, denotes the address of some location in memory where a bit pattern of type `t` is stored. The memory address can be either 32 or 64 bits depending on the machine architecture. The special type `ptr<?>` denotes a generic memory address and does not specify the type of the bit pattern stored at the address.

The *function* type, `(t1, t2, ..., tn) -> tr`, denotes a sequence of machine instructions that implements a function taking *n* arguments of type `t1`, `t2`, ..., `tn` respectively, and returns a value of type `tr`. The list of argument types may optionally end with an ellipsis to denote that the function takes a variable number of arguments. For example, the type `(int, float ...) -> float` denotes a function that takes a single `int` argument followed by a variable number of `float` arguments. While variable-arity functions cannot be defined in LoStanza, the ellipsis is useful for modeling the types of foreign functions such as `printf` from C.

The *reference* type, `ref<T>`, denotes the address of some location in the automatically-managed heap memory, where a HiStanza value of type `T` is stored. The reference type allows LoStanza code to easily interact with values created within HiStanza.

7.4 LoStanza Objects and Arrays

LoStanza allows programmers to declare and manipulate objects with an explicit memory layout. The following declares a struct called `Coord` that contains two 32-bit floating point numbers for representing its `x` and `y` coordinates:

```
lostanza deftype Coord :
  x: float
  y: float
```

Within a LoStanza function, the type Coord will then refer to the struct comprised of two 32-bit floating point numbers. In total, the struct is then 64 bits. Within a LoStanza function, a Coord struct can be created using the following syntax:

```
var c:Coord = Coord{1.0f, 3.0f}
```

The dot (.) operator can be used to retrieve from and store to a field:

```
  var x:float = c.x  ;Retrieve the x field in c
  c.y = 4.0f         ;Store 4.0f into the y field in c
```

The code above allocates the object locally – either in registers, or on the stack if necessary. To create a heap-allocated reference to a struct we can use the new operator:

```
var c:ref<Coord> = new Coord{1.0f, 3.0f}
```

Array types are specified in LoStanza by following the last field declaration in a struct with an ellipsis. This specifies that the struct ends with a variable number of fields of that type. The following example declares a Company type that holds some number of employees:

```
lostanza deftype Company :
  num-employees: long
  tax-id: long
  annual-revenue: long
  employees: ref<String> ...
```

The Company type has a num-employees field for specifying the number of employees, a tax-id field for specifying the tax identification number of the company, an annual-revenue field for specifying the annual revenue of the company, and an employees field that holds num-employees number of employees. The number of values held by the last field in an array type is always dictated by the first field in the struct which must have type long.

The new operator is also used to instantiate an array object. All fields except the last one must be provided during instantiation:

```
var c:ref<Company> = new Company{3L, 401431282L, 2000000L}
```

The index operator ([]) is used to store to and load from a specific location in the variable field of an array type:

```
c.employees[0] = String("Patrick")      ;Store into 1st c.employees slot.
c.employees[1] = String("Luca")         ;Store into 2nd c.employees slot.
c.employees[2] = String("Emmy")         ;Store into 3rd c.employees slot.
var luca:ref<String> = c.employees[1]   ;Load from 2nd c.employees slot.
```

Note that in the examples above we explicitly provide the types of all LoStanza variable (var) declarations. For purposes of clarity, LoStanza offers extremely limited type inference. The types of all arguments and variables must be explicitly provided. Type annotations are inferred only for value (val) declarations.

7.5 Interaction between HiStanza and LoStanza

LoStanza is designed to be a simple language and does not offer any novel constructs that cannot be found in other low-level languages such as C [33]. What sets LoStanza apart is the ease with which it can interact with code written in HiStanza. This is accomplished by two attributes of LoStanza:

1. HiStanza and LoStanza definitions share the same namespaces. LoStanza can directly refer to types, values, variables, functions, and multis declared in HiStanza and vice versa.
2. LoStanza can create and interact with objects in the garbage-collected heap memory.

Shared Namespaces

To demonstrate the shared HiStanza/LoStanza namespaces, consider again our example of the LoStanza type `Coord`:

```
lostanza deftype Coord :
  x: float
  y: float
```

We now write a constructor function, and two getter functions, to allow the `Coord` type to be freely used from HiStanza:

```
lostanza defn Coord (x:ref<Float>, y:ref<Float>) -> ref<Coord> :
  return new Coord{x.value, y.value}

lostanza defn x (c:ref<Coord>) -> ref<Float> :
  return new Float{c.x}

lostanza defn y (c:ref<Coord>) -> ref<Float> :
  return new Float{c.y}
```

From the perspective of HiStanza, the `Coord` type represents an immutable object with two `Float` fields that can be retrieved using the `x` and `y` functions.

As an example, the following `parabola` function takes a starting and ending x coordinate, and a step size, and returns a `Vector` of `Coord` values representing the function $y = x^2$:

```
defn parabola (x0:Float, x1:Float, inc:Float) :
  val coords = Vector<Coord>()
  let loop (x:Float = x0) :
    if x <= x1 :
      add(coords, Coord(x, x * x))
      loop(x + inc)
  coords
```

Note that the `Coord` type can be transparently referred to from HiStanza. The constructor function `Coord` is used to create `Coord` values which are stored in a `Vector` created from within HiStanza.

From the caller's perspective, whether a function is implemented in HiStanza or LoStanza is indistinguishable. LoStanza functions can be called, passed as arguments, or stored in datastructures, identically to HiStanza functions. The following example tests whether a `Vector` of `Coord` values contains a point lying below the x-axis:

```
defn below-x-axis? (ps:Vector<Coord>) :
  any?({_ < 0}, seq(y, ps))
```

There is only one restriction that applies to interactions between HiStanza and LoStanza functions: only LoStanza functions that accept and return `ref<T>` types can be referenced from HiStanza. This is why the `Coord` constructor function accepts its arguments as `ref<Float>` values, and why the `x` and `y` getter functions return `ref<Float>` values.

Automatic Garbage Collection

In general, two key pieces of information must be provided by the user for a language to support automatic garbage collection:

1. The user must explicitly request the allocation of a block of automatically-managed memory.

2. The user must state when a block of memory is still in use, so that the system knows when it can be safely deallocated.

These two details are provided by the LoStanza `ref` type.

First, a user explicitly requests the allocation of a block of automatically-managed memory through the `new` construct, which returns a value of type `ref<T>`. The `new` construct is also the *only* construct that can create a `ref<T>` value.

Second, Stanza uses the `ref<T>` type to track whether a block of memory is still in use. Starting from the global variables, Stanza assumes that any memory reachable through a `ref<T>` reference is under use, and that any memory that is not reachable is safe to deallocate.

Alternatively, programming languages such as Python and Ruby, that require the use of a separate low-level language (such as C) to interface with the software ecosystem, provide hooks to register and unregister pointers with the garbage collector. These hooks, typically named `register-gc-root` and `unregister-gc-root`, are used to explicitly tell the system whether a piece of memory is still under use, and their misuse is the source of many errors.

When the heap is full and no more objects can be allocated, the system will automatically call the LoStanza function `core/extend-heap` to run the garbage collector and free any memory that is no longer being used.

7.6 LoStanza and HiStanza Primitives

Because HiStanza code cannot operate directly on LoStanza primitive types, there is an associated HiStanza type that wraps up each primitive type. Each of these HiStanza types contains a single value field of the requisite LoStanza type.

For instance, the Float HiStanza type is defined as follows:

```
lostanza deftype Float :
  value: float
```

which wraps up a primitive floating point number in a struct called Float. The primitive types byte, int, long, and double have the corresponding HiStanza types Byte, Int, Long, and Double defined similarly. The Char HiStanza type contains a byte value.

The LoStanza constructs new and dot (.) operator can be used to convert between the LoStanza and HiStanza representations of primitives. For instance, a HiStanza ref<Float> value can be constructed from a LoStanza float value, x, using the new operator:

```
new Float{x}
```

and the LoStanza float value wrapped within a HiStanza ref<Float> value, y, can be retrieved by accessing its value field:

```
y.value
```

7.7 Interacting with the Software Ecosystem

Interactions between Stanza and the surrounding software ecosystem is also handled through LoStanza, which supports the constructs necessary to both:

1. execute foreign code from within Stanza, and
2. call Stanza functions from foreign code.

Executing Foreign Code from Stanza

LoStanza currently supports declaring and calling foreign code using the *C calling convention* through the special extern and call-c constructs. Suppose we wish to call the following C function for computing the sum up to n:

```
int c_sum (int n) {
  int accum = 0;
  for(int i=0; i<n; i++)
    accum += i;
  return accum;
}
```

First, the extern construct is used to declare the existence and type of a foreign function. The following definition declares the existence of a function called c_sum that accepts a primitive int value and returns a primitive int value:

```
extern c_sum: int -> int
```

Second, the call-c operator is used to call a function using the C calling convention:

```
lostanza defn sum (x:ref<Int>) -> ref<Int> :
  val result = call-c c_sum(x.value)
  return new Int{result}
```

The LoStanza `sum` function takes a `ref<Int>` argument, retrieves the wrapped primitive `int`, and then calls `c_sum`. A `ref<Int>` value is then constructed from the result and returned.

Here is another example of the `call-c` operator, in which we call the C `printf` function to display the message "There are 3 ducks sitting in a row.":

```
extern printf: (ptr<byte>, ? ...) -> int

lostanza defn print-ducks (n:ref<Int>) -> ref<False> :
  call-c printf("There are %d ducks sitting in a row.", n.value)
  return false

print-ducks(3)
```

The `extern` declaration declares `printf` to accept the format string as a byte pointer, followed by a list of values of unspecified types. The `print-ducks` function retrieves the `int` value within its `ref<Int>` argument and calls `printf` using the C calling convention.

Calling Stanza Functions from Foreign Code

It is often desirable to have Stanza functions be callable from foreign code. This need arises often when using libraries whose design relies heavily upon callbacks, such as those for constructing graphical user interfaces. LoStanza supports this feature by allowing LoStanza functions to be annotated with an `extern` declaration that tells the compiler that the function will be called with the C calling convention.

As an example, consider the following C function for computing fibonacci numbers:

```
void report_number (int);

void c_fibonacci (int n) {
  int a = 1;
  int b = 1;
  for(int i=0; i<n; i++){
    report_number(a);
    int a2 = b;
    int b2 = a+b;
    a = a2;
    b = b2;
  }
}
```

The `c_fibonacci` function accepts an integer, `n`, that specifies how many numbers in the fibonacci sequence to compute. We assume the existence of a callback function named `report_number`, and have `c_fibonacci` call it once for each computed fibonacci number.

Here is the `report_number` callback function written in Stanza:

```
val NUMBERS = Vector<Int>()

extern defn report_number (n:int) -> int :
  add(NUMBERS, new Int{n})
  return 0
```

The `extern` keyword preceding `defn` declares that the function will be called using the C calling convention, and that the name `report_number` should be made visible to foreign code. The callback simply constructs a `ref<Int>` value from its argument and adds it to the global `NUMBERS` vector.

We declare and call `c_fibonacci` using the `call-c` operator as described before:

```
extern c_fibonacci: int -> int

lostanza defn fibonacci (n:ref<Int>) -> ref<False> :
  clear(NUMBERS)
  call-c c_fibonacci(n.value)
  return false
```

The LoStanza `fibonacci` function first clears the contents of `NUMBERS`, and then calls `c_fibonacci` using the C calling convention.

Executing the following code:

```
fibonacci(20)
println(NUMBERS)
```

prints out the twenty numbers stored in the `NUMBERS` vector:

```
[1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765]
```

Note that LoStanza does not have a type analogous to the C `void` type. The `void` type in C indicates that the function does return, but that it has no meaningful result. We simply use `int` in place of `void` and ignore the returned value.

Function Pointers

The previous example was unrealistic in the sense that the name of the callback function, `report_number`, was hard coded. This is rarely done by C libraries – instead, the callback function is typically passed in as a function pointer.

Here is the definition of `c_fibonacci` changed to accept `report_number` as a function pointer argument:

```
void c_fibonacci (void ( *report_number )(int), int n) {
  int a = 1;
  int b = 1;
  for(int i=0; i<n; i++){
    report_number(a);
    int a2 = b;
    int b2 = a+b;
    a = a2;
    b = b2;
  }
}
```

In order to call the new definition of `c_fibonacci` we have to change its `extern` declaration to reflect its new type signature:

```
extern c_fibonacci: (ptr<(int -> int)>, int) -> int
```

`c_fibonacci` now requires two arguments. The first argument is a pointer to a callback function that takes an `int` and returns an `int`. The second argument is an `int`.

To call `c_fibonacci`, we now retrieve a pointer to the `report_number` function using the `addr` operator, and pass that in as the first argument. The implementation of `report_number` is left unchanged:

```
lostanza defn fibonacci (n:ref<Int>) -> ref<False> :
  clear(NUMBERS)
  call-c c_fibonacci(addr(report_number), n.value)
  return false
```

7.8 Comparisons to Other Solutions

We made the choice to separate Stanza into high-level and low-level sublanguages to interface with the surrounding software ecosystem. There are broadly two other solutions we could adopted:

1. Directly provide the necessary low-level constructs for controlling the hardware.
2. Control the hardware with primitive operations written in a different programming language.

Directly Provide Low-Level Constructs

The most direct solution is to have the language support the low-level constructs necessary for controlling hardware. This is the solution taken by C [33], C++ [55], Go [25], and Rust [17] for example – which we recognize as "systems programming languages" to reflect the amount of control they offer. The goal of a systems programming language is to expose as much of the raw abilities of the machine as possible to the programmer – and less or no emphasis is placed on maintaining a high-level machine-independent semantics.

All else being equal, control over the machine is always a desirable aspect. The tension between a systems programming language and a high-level language arises from the fact that many of the features offered by a high-level language results from *limiting* the ways in which a programmer can interact with the machine. As a trivial example, a language that provides no construct for observing the word-width of the machine guarantees that programs written in it are portable between machines of different word-widths.

The HiStanza/LoStanza sublanguage solution has the following advantages over a systems programming language:

1. *Portability:* HiStanza's semantics are machine-independent. The vast majority of Stanza programs do not require the use of LoStanza and can execute as-is on different machines. For the small subset of programs that do require LoStanza, only a small portion of the complete program needs to be written in LoStanza. Programmers can thus port a program by focusing their efforts upon these isolated LoStanza sections.

2. *Easy-of-Use:* Language features that offer significant control over the machine also come with the added burden of controlling the machine *correctly*. Systems programming languages are known for being difficult to learn because of the abundance of details that must be managed explicitly by the programmer. In contrast, high-level languages such as Python can be used effectively without understanding machine architecture, and is often taught in introductory programming courses.

   While Stanza does offer low-level constructs that require a competent understanding of machine architecture, these constructs are cleanly isolated to the LoStanza sublanguage. HiStanza has a learning curve comparable to Python.

3. *Abstraction:* Low-level constructs over-constrain a program by specifying details that are unnecessary for describing its operation. Consider the following C function that computes the total sum of an array:

   ```
   float arraysum (float* xs, int n) {
     float sum = 0.0f;
     for(int i=0; i<n; i++)
       sum += xs[i]
     return sum;
   }
   ```

   with the equivalent Stanza code:

   ```
   defn arraysum (xs:Array<Float>) -> Float :
     var sum = 0.0f
     for i in 0 to length(xs) do :
       sum = sum + xs[i]
     sum
   ```

   Compared to the Stanza code, the C code has more constraints upon its operation: the `xs` array *must* be passed in as a pointer to a sequence of `float` values laid out contiguously in memory. The Stanza code assumes only that `xs` supports two operations: one for retrieving its length, and another for retrieving the number at a specified index.

   The following shows the Stanza definition of an implicitly defined *interval array* – which contains the number `1.0f` between the specified `start` and `end` index, and is `0.0f` everywhere else:

```
defn IntervalArray (start:Int, end:Int, length:Int) :
  new Array<Float> :
    defmethod length (this) :
      length
    defmethod get (this, i:Int) :
      if i >= start and i <= end : 1.0f
      else : 0.0f
```

The Stanza implementation of `arraysum` can be directly called with the array created by `IntervalArray`. The C version of `arraysum` would require an explicitly allocated block of contiguous memory, populated with `0.0f` and `1.0f`.

4. *Opportunities for Optimization:* Because a high-level language imposes less constraints upon the implementation of an algorithm, the compiler has greater flexibility when mapping the algorithm to the machine.

Consider the following code, which defines a `Pair` type containing an `x` and a `y` coordinate:

```
deftype Pair
defmulti x (p:Pair) -> Float
defmulti y (p:Pair) -> Float

defn Pair (x:Float, y:Float) :
  new Pair :
    defmethod x (this) : x
    defmethod y (this) : y

defn main () :
  val p = Pair(1.0f, 2.0f)
  println(x(p))
  println(y(p))

main()
```

The `main` function creates a `Pair` object and prints out its coordinates.

The details of how and where the `p` value is stored is not constrained by the code. The compiler is free to store `p` in any of the following locations:

a) It can be stored on the heap and be represented as a pointer to heap memory.

b) It can be stored on the activation frame of the `main` function, which would relieve the garbage collector of managing its allocation.

c) Since it's a small object – comprised of only two floating point numbers – it can be stored directly in two machine registers. This would be an extremely efficient mapping: the garbage collector does not have to manage its allocation, and no memory operations are required to access the pair's coordinates.

A trivial analysis would also reveal that `Pair` objects are *immutable*: a pair's coordinates cannot change once the pair is created. Thus the calls to `x(p)` and `y(p)` are guaranteed to return `1.0f` and `2.0f` as those are the coordinates used to create the pair. Consequently, the `main` function can be equivalently mapped to the following:

```
deftype Pair
defmulti x (p:Pair) -> Float
defmulti y (p:Pair) -> Float

defn Pair (x:Float, y:Float) :
  new Pair :
    defmethod x (this) : x
    defmethod y (this) : y

defn main () :
  val p = Pair(1.0f, 2.0f)
  println(1.0f)
  println(2.0f)

main()
```

Finally, since `p` is no longer used in the `main` function, and that the `Pair` constructor function contains no side-effects, the call to `Pair` can be removed entirely. The definition of the `Pair` type can be removed as it is no longer referenced by any executable code. And the `main` function can be inlined, as it is called only once. The final program can be thus mapped to the following:

```
println(1.0f)
println(2.0f)
```

As a more extreme optimization example, consider the following, which computes the sum of the first hundred even numbers:

```
defn sum-of-even-numbers () :
  defn even? (x:Int) : x % 2 == 0
  reduce(plus, 0, take-n(100, filter(even?, 0 to false)))
```

The expression `0 to false` creates an infinite sequence containing the natural numbers. We then use `filter` to keep only the even numbers, `take-n` to take the first hundred remaining, and `reduce` with the `plus` function to compute the result.

After a suite of aggressive inlining and partial evaluation optimizations, the `sum-of-even-numbers` function can be transformed by the compiler into the following simple loop:

```
defn sum-of-even-numbers () :
  let loop (accum:Int = 0, x:Int = 0, n:Int = 0) :
    if x % 2 == 0 :
      if n < 100 : loop(accum + x, x + 1, n + 1)
      else : accum
    else :
      loop(accum, x + 1, n)
```

Write Primitive Operations Using a Different Language

The other method for interfacing with the software ecosystem is to implement a set of primitive operations in a separate programming language that will then be called from the high-level language. The separate programming language should have the low-level constructs necessary for interfacing with the hardware, and is typically a systems programming language such as C [33] or C++ [55]. This is the most common solution and is used by Scheme [56], Python [47], Ruby [22], Java [2], C# [27], Perl [63], and PHP [34], among many other languages.

In the simplest case, the set of primitive operations is fixed and implemented only by the language designer. The specific programming language chosen to implement these primitive operations are then an implementation detail and does not concern the language user. This design decision is acceptable for languages targeted at a niche domain for which the required set of primitive operations is well-defined and stable.

General-purpose languages, on the other hand, must allow users to extend the set of primitive operations themselves. This is done by directly writing code in the systems language. In this case, the choice of systems language is no longer an implementation detail, and detailed instructions must be provided for correctly manipulating the datastructures created within the high-level language, and for satisfying any required invariants.

Compared to this method, the HiStanza/LoStanza sublanguage solution is significantly easier to use. Because the systems language was designed independently from the high-level language, it provides no facilities for easily working with constructs from the high-level language. Communication between the high-level language and the systems language is often complicated and a frequent source of errors.

To illustrate the issues, consider the following LoStanza function, `store-x-coordinates`, which, given a vector of pairs, stores their x coordinates contiguously to a special memory location called `GRAPHICS-X-COORDS`:

```
lostanza var GRAPHICS-X-COORDS: ptr<float>

lostanza defn store-x-coordinates (ps:ref<Vector<Pair>>) -> ref<False> :
  val n = length(ps).value
  var mem:ptr<float> = GRAPHICS-X-COORDS
  for (var i:int = 0, i < n, i = i + 1) :
    val p = get(ps, new Int{i})
    [mem] = x(p).value
    mem = mem + sizeof(float)
  return false
```

Here is a listing of all the details that are automatically handled by LoStanza that would otherwise need to be explicitly managed by the user.

1. We retrieve the length of the `ps` vector by calling the `length` function. LoStanza can transparently call functions defined in HiStanza as the two sublanguages share the same namespace. To accomplish the same task in a separate systems language, the user must somehow retrieve the address of the high-level function, and then call it *with the appropriate calling convention*.

2. The `length(ps)` call returns a value of type `ref<Int>` which represents a HiStanza integer. To convert that to a primitive machine integer, we retrieve its `.value` field. To accomplish the same task in a separate systems language, the user must understand the mapping between the high-level primitives and their equivalent primitive values. This conversion is sometimes referred to as *tagging* and *untagging*.

3. To retrieve the x coordinate from a pair, we call the `x` function. To accomplish the same task in a separate systems language, the user must understand the details of how fields of the `Pair` type are laid out in memory. Note also that the Stanza compiler is able to inline calls to `x` to avoid any function call overhead.

4. The `Vector` and `Pair` implementations are abstract and the `length`, `get`, and `x` functions can execute arbitrary Stanza code – including code that may *trigger a garbage collection cycle*. Because Stanza employs a relocating garbage collector, the address of the `ps` array before the call to `get` may be different from its address after the call. In the event of a garbage collection cycle, LoStanza automatically updates `ps` to the new address. In a separate systems language, the user would have to explicitly request the potentially changed address. This burden is so high and so error-prone that many languages, such as Python [47] and Ruby [22], implement less efficient but non-relocating garbage collectors.

5. The `length`, `get`, and `x` functions are free to throw an exception or yield to a different coroutine. LoStanza will automatically handle jumping out of the `store-x-coordinates` function (in the case of a thrown exception) or saving and reloading its state (in the case of yielding a coroutine). In a separate systems language, the user would have to explicitly check whether an exception is thrown and appropriately jump out of the `store-x-coordinates` function. To our knowledge, no existing language allows for a function call from the systems language to yield to a different coroutine.

7.9 Summary

LoStanza is a simple low-level language where the details of memory-management, interacting with foreign code, and controlling hardware can be kept isolated away from the high-level logic of the program. It is designed to offer all the low-level constructs of C, while also being able to interoperate easily with HiStanza code.

Stanza, comprised of the HiStanza and LoStanza sublanguages, forms a complete language system that is expressive enough to implement itself. The compiler is written in

HiStanza, the library routines for interoperation with the operating system are written in LoStanza, and the garbage collector is written in LoStanza.

Chapter 8

The Stanza System

The Stanza system consists of the Stanza compiler, and the Stanza core libraries. The compiler accepts as input a combination of `.stanza` source files and precompiled `.pkg` package files, and produces a single text file containing x86-64 assembly instructions. The GNU Compiler Collection (GCC) [46] is used to link the resulting assembly text file against the standard C libraries to create the final program executable.

The latest version of the compiler is implemented in the Stanza language itself, and we take full advantage of Stanza's power and expressivity to keep the code short and manageable. We make prevalent use of higher-order functions, macros, coroutines, and custom domain specific languages to ensure that code is well-factored and redundancy is minimized. As a result, the entire implementation fits in 28782 lines of code.

The compiler is divided into the following components:

1. *Reader:* Reads the input source files into memory and converts their characters to s-expressions.

2. *Macroexpander:* Expands all macros in the input s-expressions into their resulting core forms.

3. *Syntax Checker:* Verifies the core forms to be syntactically well-formed.

4. *Symbol Resolver:* Assigns all binders a unique identifier, and resolves all references to binders. Shadowed binders are handled during this pass.

5. *Type Inferencer and Typechecker:* Infers the unspecified types for all binders, and checks the program for ill-typed expressions. This is the last pass in the compiler frontend. Once a program passes this stage with no reported errors, the compiler guarantees that the resulting program will execute to completion or fail with a precise error message.

6. *Midend Optimizer:* Optimizes the program by re-expressing slow expressions as more efficient but equivalent forms.

7. *Abstract Machine Compiler:* Implements the program as instructions for an abstract machine. The abstract machine has an infinite number of registers and abstract instructions for simplifying code generation – such as calling functions, performing dispatch based on type, allocating memory from the garbage collector, and extending the stack.

8. *Register Allocator:* Converts the abstract machine instructions into instructions for a machine with a finite number of registers and simple instructions.

9. *x86-64 Emitter:* Emits the final instructions using AT&T syntax to be linked using GCC.

10. *Separate Compilation and the Pkg System:* For increasing the compilation speed of large programs, the compiler allows packages to be compiled separately into `.pkg` files. When working on a large project, only the modified source files need to be recompiled. The separate compilation system ensures, by inspecting the function type signatures, that `.pkg` files can be linked to form a correct program and issues an error otherwise.

The core library consists of 11037 lines of code, and includes:

1. the garbage collector,

2. file input and output routines,

3. standard datastructures,

4. commonly used operations on sequences,

5. commonly used numerical types,

6. useful utilities for writing macros,

7. support for converting strings to s-expressions, and

8. support for parsing s-expression datastructures.

8.1 Organization of the Compiler

The Stanza compiler is organized as a series of *passes* that transform the program from one representation to another. Each subsequent representation contains less information than the previous one, and accumulates more concrete details about how the program is mapped onto the physical machine. From a high-level perspective, the compiler adopts a pure functional architecture, where each pass is implemented as a pure function that takes an immutable datastructure as input and outputs another immutable datastructure. Mutation is used internally within a pass when convenient.

The Reader

The reader is the first pass of the compiler and is responsible for outputting the s-expression that is represented by the input character sequence. A minimal amount of syntax checking ensures that the characters represent a valid s-expression. The following lists some properties that are checked:

- Parentheses, braces, and brackets, must be balanced.
- Strings must have a starting and ending double-quote character.
- Characters must have a starting and ending single-quote character.
- Numbers must be formatted properly and be within the proper bounds.

The indentation structuring mechanism and reader shorthands are also handled during this stage, such that the output datastructure is a simple s-expression.

The Macroexpander

The macroexpander accepts as input an s-expression and outputs the s-expression resulting from expanding all macros in the input s-expression. During macroexpansion, each macro is responsible for checking whether its syntax is well-formed, and issues an error otherwise. For instance, the `for` macro, upon encountering the following code:

```
for i in xs :
  println(i)
```

will issue the error:

```
Missing operating function in for expression.
Did you forget to put a do after the bindings?
```

The macroexpander pass succeeds if all macros expand without errors.

Syntax Checker

The syntax checker accepts as input an s-expression, and verifies that it is comprised solely of syntactically correct Stanza core forms. The following lists some properties that are verified:

- All core forms must begin with the appropriate tag, e.g. `$defn`, and have the proper arity.
- Some core forms should only appear at the top-level.
- LoStanza core forms should not appear in the bodies of HiStanza functions, and vice versa.
- Instance methods should have a single argument named `this`.

- The last expression in a function body should not be a declaration.
- LoStanza functions should contain a `return` statement.
- The number of arguments in a `match` branch should equal the number of values that were matched against.

Once the syntax checker has verified that the core forms are syntactically correct, it outputs an intermediate datastructure called *Input IR* – short for *Input Intermediate Representation* – that represents the program. The Input IR uses a separate type to represent each core form, as is typical of datastructures for representing abstract syntax trees (ASTs).

Symbol Resolver

The symbol resolver accepts as input a program represented in Input IR, and outputs the same program in Input IR but with binder definitions and binder references replaced with unique numeric identifiers.

The resolver ensures the program contains no binders with duplicate names when inappropriate. Function overloading allows functions to share the same name, but no other kind of binder can share the same name within the same scope. A feature called *shadowing* allows a binder defined within an inner scope to have the same name as a binder defined in an outer scope, as shown in the following example:

```
val x = 3
let :
  val x = 4
  println(x)
```

The resolver will assign unique identifiers to all binders and resolve references to these shadowed binders appropriately. So the above code, after symbol resolution, will be treated equivalently to the following:

```
val x0 = 3
let :
  val x1 = 4
  println(x1)
```

The resolver will issue an error for any referenced binders that have not been declared. By the end of this pass, the program is guaranteed to be *syntactically* correct (though not yet well-typed).

Type Inferencer and Checker

The type inferencer and checker accepts a program represented in Input IR; converts the program to *Typed IR* – short for *Typed Intermediate Representation*; infers the types of all binders; and outputs the program in Typed IR if all expressions are well-typed.

The Typed IR is similar to Input IR except all expressions now have an extra `type` field for storing their inferred type. The inference algorithm scans through the program and builds a set of type constraints that are solved using a custom dataflow solver. The solver infers function return types, value types, variable types, captured type arguments, and the argument types of anonymous functions. Errors are issued for any unspecified type that cannot be inferred by the solver.

After the types of all expressions have been inferred, references to overloaded functions can be resolved based upon either the types of the arguments that they are called with, or upon the type expected by the context in which they are used.

After inference, the typechecker then sweeps through the program and ensures that all expressions are well-typed. For example:

- A function must be called with arguments of the appropriate types and the appropriate number of type arguments.
- A variable must be initialized with a value of the appropriate type.
- A variable must be assigned a value of the appropriate type.
- The arguments of a `match` expression must be able to match against all the types in a match branch.

Once all expressions are ensured to be well-typed, the program is guaranteed to run until completion or fail with a precise error message.

Midend Optimizer

The midend optimizer accepts a program represented in Typed IR; converts the program to *K IR* – short for *K-Form Intermediate Representation*; performs a number of optimizing transformations; and outputs the optimized program in K IR.

K IR is an intermediate transformation with a convenient normalized form that simplifies the implementation of many optimizing transformations. In K-normalized form, all compound expressions are simplified by binding nested expressions to temporary values. For example, the following code:

```
val x = f(1, g(0))
val y = let :
  val z = h(1, g(1))
  z + 3
w(y)
```

is K-normalized to:

```
val t0 = g(0)
val x = f(1, t0)
val t1 = g(1)
val z = h(1, t1)
val y = z + 3
w(y)
```

Some examples of optimizing transformations done in the midend include:

- *Tail-Recursion Elimination:* Self tail-recursive functions are transformed into explicit loops.
- *Inlining:* When it is profitable to do so, calls to functions are inlined to eliminate the function call overhead.
- *Coroutine Elimination:* Some specific but common uses of coroutines can be transformed to simple jumps.
- *Check Elimination:* A runtime typecheck for ensuring that a value is of a specific type can be eliminated if it can be proven that the check will never fail.
- *Lambda Lifting:* Nested functions can be lifted to the top-level by passing in any closed over binders as arguments.
- *Method Resolving:* Calls to a multi can be replaced by a direct call to the method if it can be proven that the call will always dispatch to that method.
- *Loop Hoisting:* Some operations that are recomputed repeatedly inside a loop can instead be computed once outside of the loop if it does not depend on any loop binders.
- *Constant Folding:* Some expressions – such as `1 + 1` – can be replaced by their statically computed results.

Abstract Machine Compiler

The abstract machine compiler accepts a program in K IR and outputs the program's implementation in *TG IR* – short for *Target Intermediate Representation.*

The TG IR is comprised of a set of simple instructions for an abstract machine. The abstract machine has an infinite number of registers, is provided the type of data stored in each of its registers, and also supports abstract instructions such as for performing function calls, dispatching based on type, switching the active coroutine stack, and creating a new coroutine stack.

The following example shows the printed representation of the TG IR:

```
goto 1043125 when arity-ne(2)
def 1043126 : Ref
def 1043127 : Ref
def 1043129 : Long
new-stack
(V1043126, V1043127) = args
V1043129 = [M1000049 + 0]
```

```
goto 1043128 when ge(V1043129, 3)
() = call/0 M1000001()
label 1043128
match(V1043127): (($of 1000015)) => 1043130 ($top) => 1043131
label 1043131
return (0)
label 1043130
return (1)
label 1043125
def 1043132 : Ref
def 1043134 : Long
new-stack
(V1043132) = args
V1043134 = [M1000049 + 0]
goto 1043133 when ge(V1043134, 3)
() = call/0 M1000001()
label 1043133
defdata 1043135 : Int
def 1043136 : Ref
(V1043136) = call/1 M1000012(M1043135)
def 1043137 : Ref
def 1043139 : Long
def 1043140 : Long
def 1043141 : Long
V1043139 = [M1000044 + 0]
V1043141 = [M1000045 + 0]
V1043140 = add(V1043139, 24)
goto 1043138 when ule(V1043140, V1043141)
() = call/1 M1000000(24)
V1043139 = [M1000044 + 0]
V1043140 = add(V1043139, 24)
label 1043138
[M1000044 + 0] = V1043140
V1043137 = add(V1043139, 1)
[V1043137 + -1] = T1000026
[V1043137 + 7] = 1
[V1043137 + 15] = V1043136
return (V1043137)
```

Notice that registers are declared before they are used. The following declares the registers `1043127` and `1043129` with types `Ref` and `Long` respectively:

```
def 1043127 : Ref
def 1043129 : Long
```

The `Ref` type is of particular importance and indicates that register `1043127` holds a pointer to the automatically-managed heap memory. The compiler will track when this register is no longer used to determine when to deallocate the memory.

The `new-stack` abstract instruction:

```
new-stack
```

indicates to create and switch to a new active stack.

The `call` abstract instruction:

```
(V1043136) = call/1 M1000012(M1043135)
```

indicates to call the function at memory location `M1000012` with memory address `M1043135` and store the result in register `1043136`.

The `match` abstract instruction tests the type of its given arguments and jumps to different labels depending on the result:

```
match(V1043127): (($of 1000015)) => 1043130 ($top) => 1043131
```

The above code says to jump to label `1043130` if the value in register `1043127` is of type `1000015`, and jump to label `1043131` otherwise.

Register Allocator

The register allocator accepts as input instructions in TG IR; and outputs concrete instructions in ASM IR – short for *Assembly Intermediate Representation.*

Instructions in ASM IR are expressed in terms of a finite number of registers, and map directly to instructions on the underlying hardware. All abstract instructions are expressed now as concrete instructions. For instance:

- Abstract instructions for creating a new coroutine stack are replaced by concrete instructions for allocating a new coroutine and assigning to the stack register.
- Abstract instructions for calling a function are replaced by instructions for storing arguments to the locations dictated by the Stanza calling convention.
- Abstract instructions for performing dynamic dispatch are replaced by instructions for loading the type tag of a value from its header word and testing it against the tags of each type.

The allocator also computes the stack map that indicates which stack locations contain a heap reference and need to be scanned by the garbage collector.

To perform the register allocation, we use a variant of the linear scan algorithm that performs allocation on the basic blocks individually and then stitches the results together. Our algorithm performs automatic live range splitting, which, in addition to improving the quality of the allocation, also obviates the need to reserve scratch registers for spilling.

x86-64 Emitter

The emitter is a simple pass that takes as input the concrete instructions in ASM IR and outputs them to a text file using AT&T syntax.

Specific to the x86 architecture, there are two details that need to be handled carefully:

1. ASM IR is a representation of 3-address instructions whereas x86 uses 2-address instructions. Thus some ASM IR instructions expand into multiple x86 instructions. For instance, the ASM IR instruction:

   ```
   R0 = R1 + R2
   ```

   is expanded into the following x86 instructions:

   ```
   movq $rbx, $rax
   addq $rcx, $rax
   ```

2. x86 imposes certain restrictions upon some instructions that must be worked around when converting from ASM IR. For instance, a literal cannot be the first operand in a comparison operation. Thus the following sequence of instructions, which jumps to the label `L0` if `42` is less than the register `$rax`, is not valid:

   ```
   cmp $rax, $42
   jl L0
   ```

   To work around the restriction, the instruction can be re-expressed as:

   ```
   cmp $42, $rax
   jg L0
   ```

   which jumps to the label `L0` if the register `$rax` is greater than `42`.

   Note that the ordering of the operands in the above examples are not a mistake. AT&T syntax requires the operands in swapped order.

Separate Compilation and the Pkg System

The separate compiler allows packages to be compiled separately and output as `.pkg` files. Now when working on a large program, only the modified source files need to be recompiled, thus greatly reducing the compilation time. The Stanza distribution includes the standard library as pre-compiled `.pkg` files to remove the cost of compiling them for every project.

The compiler passes are extended to support `.pkg` files through the following modifications:

1. *Symbol Resolver:* Every `.pkg` file contains a listing of all externally visible bindings declared within the package. When resolving a reference to a binder, the resolver must search within all imported packages – in `.stanza` files and in `.pkg` files – for the binder's declaration.

2. *Typechecker:* Every `.pkg` file also contains the type signatures of all externally visible bindings declared within the package. The typechecker uses these type signatures when checking whether imported bindings are used in a type consistent manner.

   Additionally, a package can be recompiled *without* recompiling its dependent packages if the type signatures of its externally visible bindings are unchanged.

3. *Register Allocator:* The result of register allocation is saved within the `.pkg` file with one caveat. All objects stored on the Stanza heap begin with eight bytes for indicating the type of the object. This header is called the *type tag* and the compiler issues a tag for each type *once all types in the program are known*. Thus the type tags are not yet known when compiling a single package.

   The allocator supports separate compilation by keeping abstract all the instructions that depend upon type tags. Once all the packages comprising a program are known, and tags have been issued for all types, these abstract instructions are then expanded into concrete instructions.

8.2 Automatic Garbage Collection

One of the most difficult aspects in the implementation of a managed programming language is the coordination between the code generator and the garbage collector.

From the perspective of Stanza's code generator, automatic management of memory consists of the following:

1. Keeping track of the size of the heap. Whenever a new object is allocated, Stanza assigns it the pointer to the current top of the heap, and then increments the size of the heap by the size of the object.

2. Signaling for the garbage collector to run. If allocating a new object will increase the size of the heap to beyond its capacity, then the code generator inserts a call for running the garbage collector. The garbage collector will either return, indicating that enough space has been freed to allocate the object, or it will halt the program, indicating that there is no more memory available.

The Stanza garbage collector is implemented as a LoStanza function which vastly simplifies a few details:

1. Running the garbage collector becomes a simple function call. There is no need to save and restore special-purpose registers for switching from "executing normal code" to "executing the garbage collector".

2. Because LoStanza is able to easily refer to HiStanza datastructures, object layouts are easily accessible to the garbage collector, and do not need to be hard coded. For instance, the following LoStanza type defines the layout for a coroutine stack:

   ```
   lostanza deftype Stack :
     position: int
     mark: int
     parent: ref<False|Stack>
     sp: ptr<?>
     frames: StackFrame ...
   ```

The sp, parent, mark, and position fields can be rearranged at will without affecting the implementation of the garbage collector or the code generator.

3. Some special operations need to occur after running the garbage collector, such as running the finalizers for objects that are no longer live. And these operations can, themselves, allocate objects and again trigger the garbage collector. In LoStanza, these operations can be performed directly. If instead the garbage collector executes in a special enviroment, then we need to first transition back to "executing normal code" with a hook for performing the special operations before resuming the program.

The garbage collector requires the following information from the compiler:

1. It needs the list of binders, called the *roots*, that are always live in the program. For Stanza, the roots consist of the global variables and the current coroutine stack.

2. For each object in the heap, it needs to know the locations of any internal heap references so that it may scan those references.

3. For each stack, it needs to know the locations of the heap references in each stack frame so that it may scan those references.

This information is computed by the compiler and stored in three tables that are accessible from LoStanza: the global map, the object map, and the stack map.

Currently Stanza uses a simple single-threaded non-generational stop-and-copy garbage collector.

8.3 Domain Specific Languages for Compilers

To minimize redundancy and keep code short, the Stanza compiler makes extensive use of custom domain specific languages (DSLs) in its implementation. The DSLs are for internal use only, and are implemented using Stanza's macro facilities. As an example of the philosophy, the following describes one of the DSLs in the compiler: the *renamer* language.

The renamer language is used to write the implementation of Stanza's renaming algorithm, which scans through the program to assign unique identifiers to all binders, and also detects binders with duplicate names. The following shows part of the algorithm implementation:

```
defrenamer rename-exp (e:IExp, eng:Engine) :
  e :
    ;Stanza
    IDefType: (class:c+, {args:t+}, children:e)
    IDefChild: ({args:t+})
    IDef: (name:v+, value:e)
    IDefVar: (name:mv+, value:e)
    IDefn: (name:f+, {targs:t+, args:v+, body:e})
    IDefmulti: (name:m+, {targs:t+})
    IDefmethod: ({targs:t+, args:v+, body:e})
    IFn: ({args:v+, body:e})
    IBranch: ({args:v+, body:e})
    IDo: (func:f, args:e)
    INew: (methods:e)
    ISet: (value:e)
    ILet: ({exp:e})
    LetRec: (group{defns:e}, body:e)
    IPublic: (public{exp:e})

    ;Fallthrough
    IExp: (_:e)
  f :
    IOf: (class:e)
    IExp: e
  v+ :
    IVar: register var
    ITuple: (_:v+)
  mv+ :
    IVar: register mutable-var
  t+ :
    IVar: register tvar
    ICap: (name:cv+)
  cv+ :
    IVar: register capvar
  c+ :
    IVar: register class
  m+ :
    IVar: register multi
  f+ :
    IVar: register fn
```

Focusing in on the `IDefn` node:

```
IDefn: (name:f+, {targs:t+, args:v+, body:e})
```

The `name:f+` indicates to register the function in the function (`fn`) namespace, as can be seen in the definition of the `f+` rule:

```
f+ :
  IVar: register fn
```

Similarly, the `targs:t+` and `args:v+` indicates to register the function type arguments and standard arguments in the type variable (`tvar` and `capvar`) and variable (`var`) namespaces.

```
t+ :
  IVar: register tvar
  ICap: (name:cv+)
cv+ :
  IVar: register capvar
v+ :
  IVar: register var
  ITuple: (_:v+)
```

The `body:e` indicates to recursively rename the nodes in the function body using the `e` rule.

The curly brackets (`{}`) surrounding the fields indicate the beginning of a new *scope*. Binders within the same scope are not allowed to share the same name (except for functions). The curly bracket notation is also used in the `IPublic`, `LetRec`, `ILet`, `IBranch`, `IFn`, `IDefmethod`, `IDefmulti`, `IDefChild`, and `IDefType` nodes.

The renamer language allows us to condense the entire renaming algorithm to one page, which serves as both a specification and implementation. And though the implementation of the macro itself is sophisticated, it is unlikely to be subtlely wrong. If it is wrong, it is catastrophically wrong, which is a counter-intuitive but useful technique for avoiding bugs.

Chapter 9

Experience

Stanza's implementation is now mature and stable enough for developing sizable applications. This chapter will report on our own experiences with using Stanza to develop our lab software, and with teaching Stanza to students.

9.1 Our Experiences Using Stanza

The thesis author and members of our laboratory have used Stanza to develop four major projects thus far – the Stanza compiler, the FIRRTL hardware design language, the Feeny teaching language, and a printed circuit board (PCB) design system – along with a handful of smaller ones.

The Stanza Compiler

The largest project is the Stanza compiler itself, which consists of nearly 30KLoc, and was written entirely by a single graduate student, the author of this thesis. It has been rewritten many times now as the language design evolved, with the last full rewrite taking about four months. The early Stanza compilers were originally written in Gambit Scheme, and then ported to Stanza once the language reached reasonable maturity.

Our experience with developing in Scheme was mixed: on the one hand, Scheme was flexible and expressive, but on the other hand, it lacked a static typechecker. Scheme's flexibility enabled us to experiment quickly with Stanza's design, and we made pervasive use of Scheme's support for higher-order functions and macros to keep productivity high. But we did waste a lot of time on finding and fixing bugs that would have been trivially caught by a typechecker.

To compensate for the lack of a typechecker, we adopted a disciplined and incremental development methodology. The compiler was broken into subcomponents, and each component was tested extensively before we started developing and attaching the subsequent component. By being reasonably disciplined, we could ensure that the code base was always at an acceptable level of correctness. If the system failed after attaching a new component, then the error was likely in the implementation of the new component, and not in the existing code base.

But adjustments in the design required us to make changes deep in some component in the middle of the compiler pass chain. A change to an interface would require us to consistently update all the passes downstream. Now the cause of an error could lie anywhere. It could be that the new algorithm itself was wrong, or it could be that one of the later passes was not properly updated. A persistent fear of change loomed over us during development in Scheme. Even a change as simple as reordering the arguments to a function could not be made with full confidence in its correctness. These fears went away entirely after we bootstrapped the compiler in Stanza.

The process of porting the compiler from Scheme to Stanza greatly benefitted from the optional type system. Because Stanza supports all of Scheme's core datastructures and language constructs, the code base could be translated line-by-line. No additional type annotations were added during the initial porting.

Once ported to Stanza, type annotations and typed datastructures were gradually added to transition the code base to the statically-typed paradigm. Type annotations were added to all top-level functions, and new types were introduced for representing AST nodes that were previously stored as Scheme s-expressions. The code base for the most recent implementation of the compiler is almost entirely typed.

A major advantage gained from the port to Stanza was significantly reduced time for developing new intermediate passes. In Scheme, a large portion of development time was spent finding and fixing errors resulting from mismatched interfaces between passes. Stanza's type-checker automatically detects errors resulting from ill-typed interfaces and has completely eliminated the burden of this stage.

The introduction of typed datastructures has also significantly improved the architecture of the compiler. In Scheme, many core datastructures were encoded as simple s-expressions and were re-used often by different passes in the compiler out of convenience. The sharing of datastructures led to a proliferation of false dependencies between passes that made it difficult to update any isolated pass. When we moved to a typed paradigm, these poor architectural decisions came to the surface and were easily fixed.

The FIRRTL Hardware Design Language

The second major project developed in Stanza was the FIRRTL – short for *Flexible Intermediate Representation for Register Transfer Level* – hardware design language. FIRRTL serves as a target language for representing RTL circuitry that is output by *Chisel*, a frontend hardware construction language also developed in our laboratory. The idea was to keep frontend conveniences isolated to Chisel, and perform all optimization and lowering transformations as passes in FIRRTL.

The project was co-developed by the thesis author and Adam Izraelevitz, whom we trained in Stanza to work on this project. Adam had an electrical engineering background and minor programming experience, but within three weeks was fluent in Stanza. In fact, he was comfortable with the language mechanics after only a week and a half, and spent the remaining time familiarizing himself with the core library. Stanza, at the time, was mature but documentation was still lacking, and the student's main complaint was having to read through the source code to learn the core library. Nonetheless, within three months, we had completed the datastructures for the intermediate representation, the lowering and verification passes, and the bitwidth constraint solver.

The Feeny Teaching Language

Feeny was a minimal programming system consisting of an interpreter, bytecode compiler, virtual machine, and just-in-time compiler, written in about 7KLoc. It was co-developed by the thesis author and Mario Wolczko at OracleLabs, and was used to teach a graduate course at U.C. Berkeley on virtual machines and managed runtimes.

The frontend for Feeny was written using Stanza's decorated s-expression reader, and the Stanza parsing system. Students were provided the frontend and were responsible for developing, from scratch, a complete just-in-time compiler and virtual machine for executing Feeny.

The course was organized as a series of assignments that each required students to implement some subcomponent of the final system, and that incrementally built up to the full implementation. A final end-of-class competition ranked the performance of each student's system on a Feeny solver for the Sudoku puzzle.

The PCB Design System

The PCB design system [4], by Jonathan Bachrach, David Biancolin, Austin Buchan, Duncan Haldane, and Richard Lin, automatically generates manufacturable designs from declarative specifications of circuit boards. As input, it is given a listing of desired peripherals, and it computes and generates the circuit board layout, the wire routing, and any required startup and networking code. It is written in about 10KLoc of Stanza.

The PCB design system was the first project to make extensive use of the LoStanza sublanguage. For performance reasons, some of the routing algorithms were written in C and then called from LoStanza; and to display the resulting circuits, LoStanza was used to call into an OpenGL-based [51] graphics library.

Summary

The following summarizes our impressions of day-to-day programming with Stanza.

The types are intuitive and expressive enough to easily type the typical coding styles of programmers with Java [2], C++ [55], Scheme [56], or Python [47] experience. For the rare instances where it is not obvious how to type a segment of code, it is trivial to leave it untyped. Type errors are well-localized, easily understood, and fixed. In our experience writing the

Stanza compiler, FIRRTL, Feeny, and the PCB design system, we have consistently felt that Stanza's type system guided us towards writing well-documented and well-architected software, and we have never gone out of our way to satisfy the type system.

Using the standard library macros in Stanza feels natural. The fact that common language constructs are implemented as macros instead of as compiler-recognized constructs is indiscernible to users. The macro system is powerful enough to express sophisticated DSLs that are useful both internally (as demonstrated by `defrenamer`) and for user-facing libraries (as demonstrated by `defsyntax`).

Stanza's basic language constructs feel equivalent in expressivity to Scheme, as evidenced by our ability to do a line-by-line port of the compiler. The multimethod object system works seamlessly together with the functional programming style, and feels like a natural extension to the base language. Stanza code that operates on objects is comparable to Scheme code that operates on list and array datastructures.

On the rare occasion that it is necessary, LoStanza allows programmers to conveniently work with external datastructures and foreign code – either by directly manipulating them through LoStanza constructs, or by writing a HiStanza interface for them.

9.2 Our Experiences Teaching Stanza

Together with Jonathan Bachrach, the thesis author has held two bootcamps at U.C. Berkeley specifically dedicated to teaching Stanza, each lasting for six sessions of one and a half hours. Each bootcamp consisted of a series of presentations intermixed with hands-on exercises, which students were expected to do on their own laptops. We had two main goals in mind when organizing the teaching material:

1. Quickly teach the students enough of the core language and libraries for them to be as productive with Stanza as they are with existing languages.

2. Introduce them to features that are not common to mainstream languages such that students understand their purpose and can read up on their usage as necessary.

Our experiences showed that students can comfortably learn the core language and libraries after roughly ten hours of instruction. We credit this to the design of Stanza's concrete syntax – which largely resembles Python [47] – and to the expressivity of the type system – which can type typical Java [2] and Python idioms. After ten hours, students were able to code easily in Stanza, but still in the same style as the language they are most familiar with. The majority of students were most fluent in an imperative programming language – such as Java or Python – and this was reflected in their coding style. Students with experience using Scheme [56], OCaml [35], or Haskell [31] naturally adopted a more functional programming style. Though we ourselves prefer a more functional style, Stanza supports both equally, and students did not find it awkward to code in an imperative style.

We were quite happy that we had no difficulty teaching the type system, even to students who have never before programmed in a statically-typed language. This includes the concepts of parametric types, and both calling and defining their own parametric polymorphic functions using the captured type system. Students with extensive Java [2] or C++ [55] experience reported a vague sense of there being less busy-work required to satisfy Stanza's typechecker. We believe this is because, for the few places that are legitimately difficult to type, it is trivial to simply leave them untyped.

We encouraged students without any static-typing experience to take an exploratory approach towards learning the type system. By adding a few type annotations and studying the resulting error messages, these students gradually learned the relation between the static and dynamic semantics of Stanza. They have commented that the type system feels straightforward and have reported few to no "false positives".

One interesting observation is that once students have reasonable experience with the type system, they began to add type annotations even during the initial nprototyping stage. When asked about this change in their workflow, they commented that the types were helpful for documentation purposes and that the time spent adding type annotations was negligible compared to the time later spent debugging.

For students without experience in functional programming, three concepts were harder to grasp: first-class functions, higher-order functions, and immutable datastructures. In exercises, students were able to understand, call, and define first-class and higher-order functions but had trouble recognizing when to use them in practice. Even after having written their own versions of the common `map` function, students still preferred to write explicit loops. Similarly, Stanza's `List` type is immutable and students easily understood its basic operations but still preferred the mutable `Vector` type in their own programming. Note that students familiar with at least one functional programming language did not have difficulty with these concepts. We have also observed that, over time, the students' coding styles gradually became more functional as they discovered common idioms for shortening their code.

The remaining concept that proved to take some time to learn for students with both imperative and functional programming experience was Stanza's multimethod object system. Students have no difficulty understanding the basic mechanism and usage of Stanza's `defmulti` and `defmethod` constructs, but when writing their own code, they continue abiding by the restrictions and structure imposed by a traditional class-based object system – restrictions that do not exist in Stanza. In its worst manifestation, students would create a single file to hold all the definitions relating to one type, then write a single constructor function that holds all of the "instance variables" and "class methods" for that type – essentially mimicking a Java or Python class definition.

We have noticed that there is a learning "hump" that occurs after roughly one month of Stanza programming, when students are able to set aside what they've learned about software architecture from Java, and take full advantage of the flexibility of the multimethod object system. Past this hump, students have consistently reported that the object system is one of Stanza's greatest strengths, and that they are significantly more productive programming in Stanza than they were in existing languages.

In addition to the Stanza bootcamps, Jonathan Bachrach has also used Stanza to teach a computational design course. In this case, the focus was not on Stanza specifically, and students were expected to pick up the language on their own time. The Stanza website comes with extensive reference documentation on the core library functions and constructs, as well as an introductory textbook called *Stanza By Example* [36] that gradually teaches readers about each of the Stanza subsystems. No student reported any trouble with learning Stanza, and the majority of students commented that they simply regarded Stanza as Python with some minor syntactic differences. They programmed predominantly without type annotations but were appreciative of the errors given by the compiler.

Chapter 10

Experiments

One of our goals in the design of the type system was to enable users to gradually add type annotations to an untyped code base in order to have the typechecker catch more errors. The expectation is that the probability of statically catching an error would gradually increase with the number of type annotations added. This chapter describes an experiment for quantifying the effectiveness of the typechecker as a function of the number of annotations in a program.

10.1 Experimental Setup

If we do not consider the effects of function overloading or multimethod dispatch, then any binder in Stanza can have its type annotation replaced with the `?` type without affecting the semantics of the program or its ability to typecheck. The experiment will take advantage of this property to artificially obtain a set of programs with a smoothly increasing number of annotations.

We start with an existing program, e, that is correct and typechecks, and randomly remove a fraction, p, of its type annotations to obtain the program, e'. Our type system guarantees that e' will also typecheck. We then make a number of small random perturbations to e' to obtain an "incorrect" program e'', and test whether our typechecker is capable of detecting the errors in e''. We would expect a low probability of the typechecker catching errors in an completely unannotated program, and a higher probability of catching errors in a completely annotated program.

We selected the potential perturbations to be representative of common coding errors. They include:

1. reordering the arguments in a call to a function,
2. removing an argument in a call to a function,
3. inserting an extra argument in a call to a function,

4. reordering a sequence of expressions,
5. and swapping a reference to a binder with a reference to a different (in-scope) binder.

Our dataset contains five different programs to use as e: Calculus, Lexer, Feeny, FIRRTL, and the Stanza compiler. Calculus is a small example that implements a basic symbolic differentiator and algebraic simplifier. Lexer is a small program for parsing a given file into an s-expression. Feeny, FIRRTL, and the Stanza compiler are the code bases for the projects described in chapter 9. The following table lists the program sizes. Note that the Stanza core libraries are included in the listed sizes.

Program Name	Lines of Code
Calculus	7856
Lexer	8444
F	13071
RTL	19606
X Compiler	40698

A single trial consists of removing a fraction of the type annotations in e to obtain e', perturbing e' to obtain e'' (the "fuzzed" program), and testing whether the errors in e'' are caught by the typechecker. We use a perturbation rate of 0.0002 – meaning that an expression has a 0.02% chance of being perturbed in one of the listed ways, and results in roughly 2-6 perturbations per program. We sweep the annotation percentage from 0% annotated to 100% annotated in 5% increments and perform 300 trials at each percentage.

10.2 Experimental Data

The experimental results for each of the five datasets are shown in the following table:

	% Fuzzed Programs Caught for				
Annotation (%)	Calculus	Lexer	Feeny	FIRRTL	Stanza Compiler
0	50%	38%	52%	37%	65%
5	49%	41%	58%	33%	64%
10	53%	38%	60%	41%	72%
15	57%	40%	69%	40%	71%
20	54%	46%	66%	47%	76%
25	59%	47%	73%	45%	74%
30	64%	54%	72%	47%	84%
35	60%	50%	80%	46%	82%
40	62%	59%	81%	47%	88%
45	68%	65%	85%	50%	87%
50	67%	59%	86%	58%	87%
55	70%	59%	90%	58%	90%
60	78%	63%	91%	58%	89%
65	74%	67%	92%	63%	94%
70	77%	69%	94%	64%	94%
75	77%	71%	95%	65%	93%
80	79%	79%	96%	70%	92%
85	81%	79%	96%	72%	94%
90	85%	73%	99%	75%	92%
95	87%	80%	98%	79%	91%
100	88%	79%	100%	75%	90%

Each row denotes, for each dataset, the percentage of fuzzed programs that were able to be caught by the typechecker given a specific annotation percentage.

To highlight the trends, Figure 10.1, 10.2, 10.3, 10.4, and 10.5 shows the same data as a scatter plot.

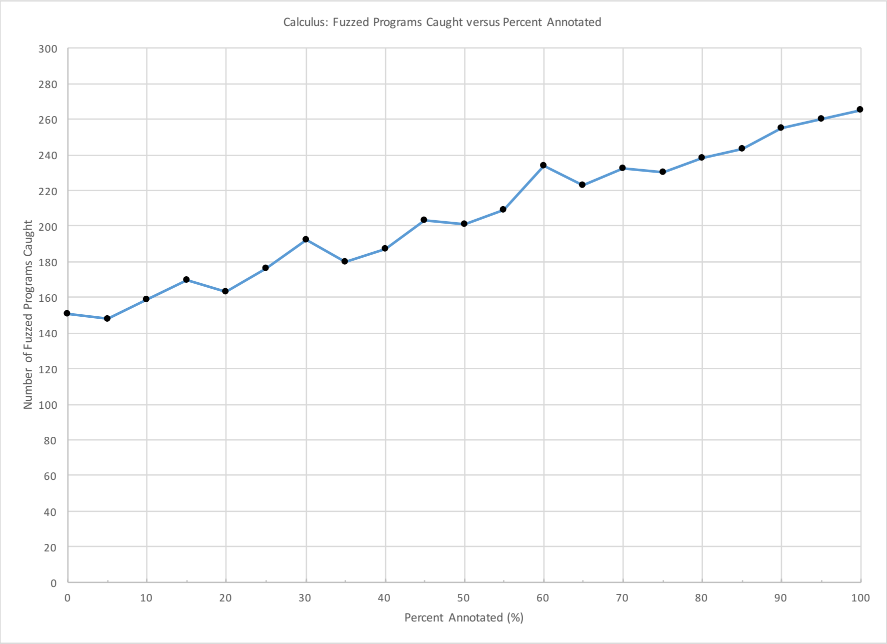

Figure 10.1: Effect of Annotations for Calculus

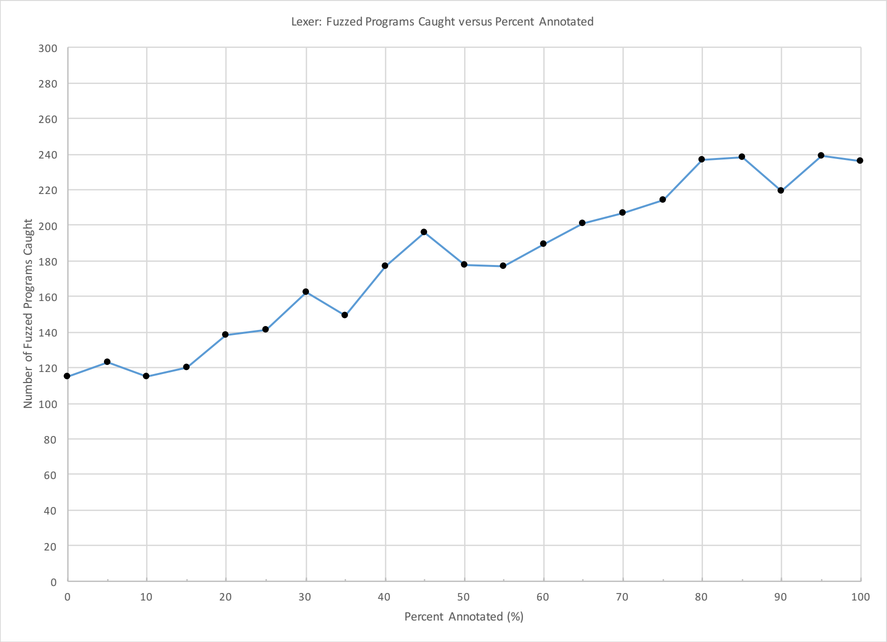

Figure 10.2: Effect of Annotations for Lexer

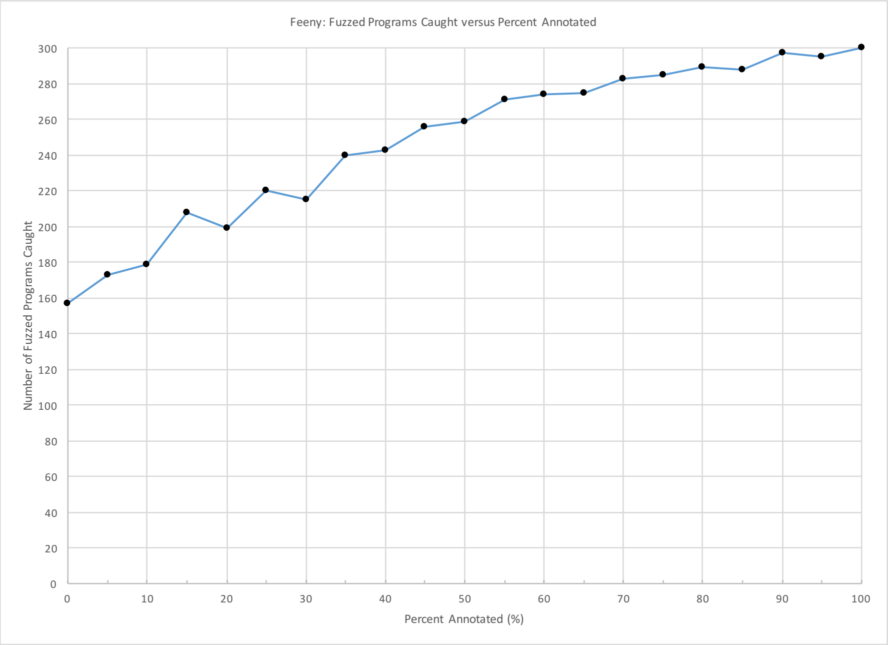

Figure 10.3: Effect of Annotations for Feeny

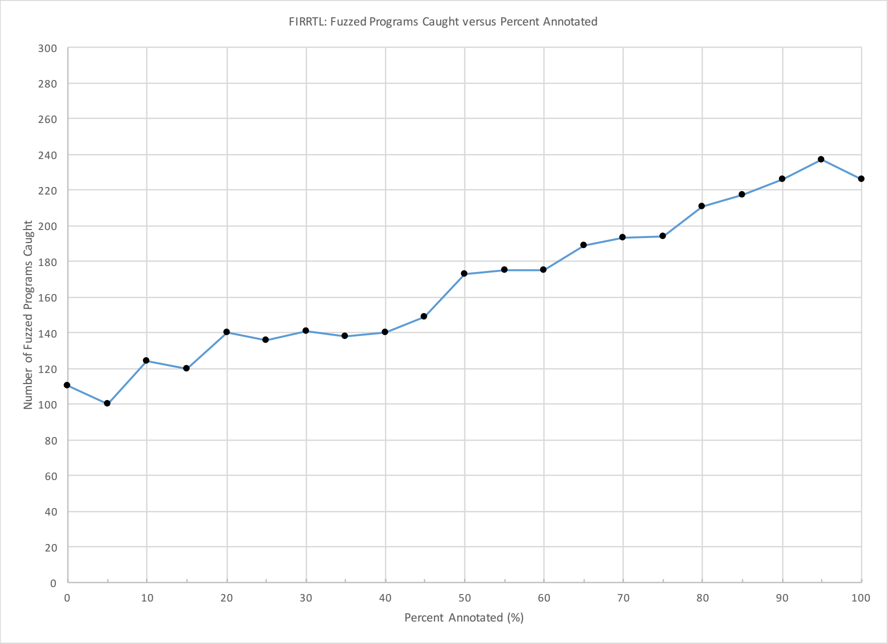

Figure 10.4: Effect of Annotations for FIRRTL

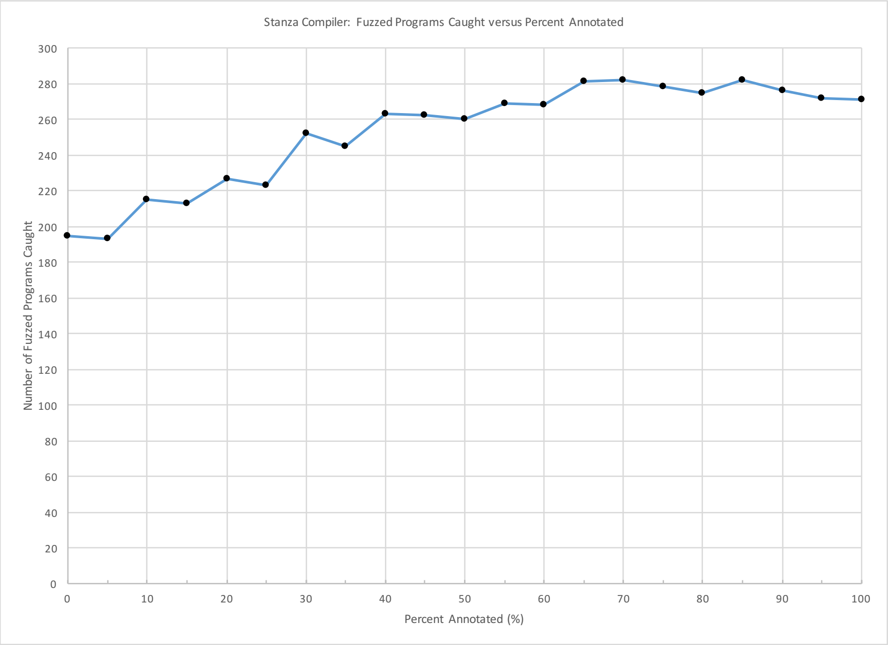

Figure 10.5: Effect of Annotations for Stanza Compiler

10.3 Discussion

As can be seen, there is a clear and smooth relationship between the number of fuzzed programs caught by the typechecker and the degree to which the program is typed. Two extremes are worth pointing out. Even with 0% of annotations, the typechecker manages to catch a substantial number of incorrect programs. This behaviour results from types being inferred from subexpressions. For example, it is invalid to add an anonymous expression to an integer, because an anonymous function is inferred to have an arrow type. This is an exciting property, as it means that before adding *any* annotations, programmers already reap some of the benefits of the typechecker.

At the other extreme, even with 100% of annotations, not all perturbations are caught by the typechecker. This is to be expected as not all perturbations manifest as a type inconsistency. We do, however, see that there is variation in the percent of programs caught depending on the choice of program e. The programs are written in different styles, some by different people, and have different characteristics. Beyond the number of annotations, the table also shows that programming style has an effect on the effectiveness of the typechecker.

Due to the randomized selection of type annotations, the number of programs caught does not increase monotically with the annotation percentage. If we were to ensure that a variable remains annotated once it is first annotated, the number of programs caught would increase monotically with the annotation percentage.

Chapter 11

The Core Type System

This chapter presents a formal definition of the core of our optional type system. It supports first-class functions, parametric polymorphic functions, two example ground types, integers and strings, and builds optional typing upon the subtyping relation. The Stanza type system features that are left out of the formal calculus include: union and intersection types, multi-arity functions, function overloading, and multimethods.

11.1 Syntax of Types and Expressions

Figure 11.1 presents the syntax of types and expressions in the core language. The system supports arrow types, tuple types, two example ground types – *int* and *str* – and type variables α. Optional typing is supported through the addition of the ? type, similar in fashion to [53].

Type $\tau =$	$\tau_1 \rightarrow \tau_2$	Arrows	Expression $e =$	$e_1 \;\; e_2$	Function Application
$\mid$	$\tau_1 \times \tau_2$	Tuples	$\mid$	$e_1[\tau] \;\; e_2$	Polymorphic Call
$\mid$	int	Integers	$\mid$	$\lambda x : \tau.e$	Functions
$\mid$	str	Strings	$\mid$	$\forall\alpha.\lambda x : \tau.e$	Polymorphic Functions
$\mid$	α	Type Variables	$\mid$	(e_1, e_2)	Tuple Construction
$\mid$	?	Unknown Type	$\mid$	$e.\text{fst}$	Tuple First Projection
			$\mid$	$e.\text{snd}$	Tuple Second Projection
			$\mid$	$e_1 + e_2$	Addition
			$\mid$	$e.\text{length}$	Length
			$\mid$	$\text{let } x = e_1 \text{ in } e_2$	Let
			$\mid$	x	Reference
			$\mid$	n	Literal Integer
			$\mid$	s	Literal String

Figure 11.1: Syntax of Types and Expressions

$\tau_1 <: \tau_2$ means τ_1 is a subtype of τ_2.

$$\frac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'} \qquad \frac{}{\mathit{int} <: \mathit{int}} \qquad \frac{}{\tau <: ?}$$

$$\frac{}{\mathit{str} <: \mathit{str}}$$

$$\frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{\tau_1 \times \tau_2 <: \tau_1' \times \tau_2'} \qquad \frac{}{\alpha <: \alpha} \qquad \frac{}{? <: \tau}$$

Figure 11.2: Subtyping Relationship

As is standard, language expressions are provided for function application and creation, tuple construction and projection, operations on ground types, and literal integers and strings. For the purposes of formalization, all function arguments are explicitly typed, and the ? type is used to model unannotated binders. As examples of operations on ground types, we include the addition operation on integers and the length operation on strings. Parametric polymorphism is supported through the polymorphic function expression, a special polymorphic call expression, and an explicit let expression.

Note that our type system is first-order and does not support universally quantified types. A well-typed program requires that polymorphic functions appear only in the let expression.

11.2 Subtyping Relation

We choose to model optional typing by introducing the ? type on top of a subtyping framework. The inference rules are listed in Figure 11.2. Informally, our system issues an error whenever the programmer attempts to use a value of type τ_1 in a location expecting type τ_2 and τ_1 is *not* a subtype of τ_2.

The inference rules are mostly standard. Arrows are contravariant in their argument types and covariant in their return types. Both tuple element types are covariant. Subtyping for ground types and type variables are reflexive.

The special type ? is governed by two rules:

1. Every type is a subtype of the ? type. This rule allows any value to be passed to a location expecting a ? type.

2. The ? type is itself a subtype of any type. This allows an expression of type ? to be passed to any location.

These two rules allow us to model the semantics of a fully dynamically-typed programming language by treating every binder as if annotated with the ? type. Figure 11.3 shows some examples of subtyping relationships. Notice that ? acts somewhat like a wildcard.

The decision to model optional typing directly within a subtyping framework comes with two key advantages:

$$
\boxed{
\begin{array}{c}
\mathit{int} <: ? \\
? <: \mathit{int} \\
\mathit{int} \times \mathit{str} <: ? \\
\mathit{int} \times \mathit{str} <: \mathit{int} \times ? \\
? \to ? <: \mathit{int} \to \mathit{int} \\
? <: \mathit{int} \to \mathit{str} \\
\mathit{int} \to (\mathit{int} \times \mathit{str}) <: \mathit{int} \to (? \times \mathit{str})
\end{array}
}
$$

Figure 11.3: Subtyping Examples

1. Subtyping is a mature theoretical framework, and is also a familiar paradigm for programmers accustomed to Java and C++, two of the most widely used statically-typed languages.
2. The core system can be easily extended with other features also originally developed in the subtyping framework – such as nominal subtyping and union types. The full Stanza type system builds extensively upon the subtyping framework.

A Note on Transitivity

As warned about in [53], modeling optional typing within a subtyping framework can be potentially problematic. Trouble arises if we include transitivity explicitly using the following inference rule:

$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

If such an inference rule were included, then we can derive the following nonsensical theorem:

$$\frac{\tau_1 <: ? \qquad ? <: \tau_2}{\tau_1 <: \tau_2}$$

which says that every type is a subtype of every other type, and thus the type lattice collapses to a single point.

We carefully avoid this degeneracy by simply not including transitivity as an explicit inference rule. In our system, in the absence of the ? type, transitivity can be derived as a property from the given inference rules. However, as desired, transitivity does not hold in the presence of the ? type. As an example, *int* <: ?, and ? <: *str*, but *int* <: *str* does *not* hold.

11.3 Example

The following is a small example demonstrating how a hypothetical user might interact with our type system. The following code defines a function *inc-fst* that increments the first element of a tuple, and calls it with three different arguments. The initial version of *inc-fst* is left untyped, which is represented in our framework with the ? annotation.

$$
\begin{array}{l}
\text{let inc-fst} = \lambda x : ?.(x.\text{fst} + 1, x.\text{snd}) \text{ in} \\
\quad (\text{inc-fst } \; 0, \\
\quad (\text{inc-fst } \; (\text{“hi, “world”}), \\
\quad\;\; \text{inc-fst } \; (0, \text{“world”})),
\end{array}
$$

The above code contains no static type errors according to our subtyping relation, but will fail at runtime at the first call to x.fst. The value 0 does not support the .fst operation. From the failure, the user can deduce that *inc-fst* must have been called with a non-tuple value, and then insert an annotation to make explicit the requirement that x be a tuple.

$$
\begin{array}{l}
\text{let inc-fst} = \lambda x : ? \times ?.(x.\text{fst} + 1, x.\text{snd}) \text{ in} \\
\quad (\text{inc-fst } \; 0, \qquad \textit{[TypeError]} \\
\quad (\text{inc-fst } \; (\text{“hi, “world”}), \\
\quad\;\; \text{inc-fst } \; (0, \text{“world”})),
\end{array}
$$

With the added annotation, our type system will flag the first call to *inc-fst* as an error. The value 0, with type *int*, cannot be passed to a function that expects an argument of type $? \times ?$. The user can then inspect the erroneous call and conclude that, indeed, it shouldn't be there. After deleting this erroneous line, the next failure will occur at the call to $x.\text{fst} + 1$, as the value "hi" cannot be an operand to the plus operation. Similarly, from this failure, the user can deduce that *inc-fst* was called with a tuple whose first element was *not* an integer, and can further refine the annotation to make this requirement explicit.

$$
\begin{array}{l}
\text{let inc-fst} = \lambda x : \textit{int} \times ?.(x.\text{fst} + 1, x.\text{snd}) \text{ in} \\
\quad (\text{inc-fst } \; 0, \qquad \textit{[TypeError]} \\
\quad (\text{inc-fst } \; (\text{“hi, “world”}), \qquad \textit{[TypeError]} \\
\quad\;\; \text{inc-fst } \; (0, \text{“world”})),
\end{array}
$$

With the added annotation, the system will flag the second call to *inc-fst* as an error. The value ("hi", "world"), with type $\textit{str} \times \textit{str}$, cannot be passed to a function that expects an argument of type $\textit{int} \times ?$. Finally, after deleting all the erroneous lines, the program will run correctly without failures.

The example shows the incremental workflow that we envision will be adopted by users. The untyped version of *inc-fst* is akin to what may be written in, for example, Python. As programs mature, users can incrementally add and refine type annotations to help locate the source of runtime failures.

$\Delta \vdash \tau$ means τ is well-formed under type environment Δ.

$$\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \rightarrow \tau_2} \qquad \frac{}{\Delta \vdash \mathit{int}} \qquad \frac{}{\Delta \vdash ?}$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \times \tau_2} \qquad \frac{}{\Delta \vdash \mathit{str}} \qquad \frac{\alpha \in \Delta}{\Delta \vdash \alpha}$$

Figure 11.4: Well-Formed Types

11.4 Typing Judgement

The typing judgements and syntax of the type environment for our system is given in Figure 11.5. The type well-formedness relation, which is needed by the type judgements, is defined in Figure 11.4.

The type environment is defined as a list of environment entries. The entry $x : \tau$ indicates that the binder x is in scope and has type τ. The entry α indicates that the type variable α is in scope. The special entry $x : \forall\alpha.\tau_1 \rightarrow \tau_2$ indicates that the polymorphic function x is in scope, and has type $\tau_1 \rightarrow \tau_2$ quantified over α. ϵ is used to represent the empty type environment.

A type is well-formed if it does not refer to any type variables that are out of scope.

The Int and Str rules are standard and type literal expressions. The Add and Len rules type the addition and length operations on integers and strings. Note the use of the subtype constraint on these two rules. This allows expressions of type ? to be valid operands.

The Tuple, Fst$_1$, and Snd$_1$ rules are standard, typing tuple construction and projection expressions. The Fst$_2$ and Snd$_2$ rules specifically type tuple projections for expressions of type ?. In these cases, the results of the projections have type ?.

The Call$_1$ and Call$_2$ rules type function application expressions. The Call$_1$ rule is standard. Similar to the Fst$_2$ and Snd$_2$ rules, the Call$_2$ rule specifically types function application expressions where the function has type ?. The result of such a call has type ?.

The Let and Ref rules are standard and type the let expression and references to binders. Note that the Ref rule requires specifically an entry for $x : \tau$ in the environment. This means that polymorphic functions cannot be directly referred to by name.

Function expressions are typed using the Fn rule. We require that the argument type be explicitly given, and we automatically infer the return type.

The PolyLet rule types a polymorphic let expression, and is the only expression in which a polymorphic function expression can appear. We ensure that the argument type is well-formed assuming that α is in scope, and infer the return type τ_2. The body e is then typed in an environment with a polymorphic function entry for f.

The PolyCall rule types a polymorphic call expression, and is the only expression that may refer to a bound polymorphic function. The type argument τ must be well-formed and

$\Delta \vdash e : \tau$ means expression e has type τ under type environment Δ.

$$\frac{\Delta \vdash e_1 : \tau_1 \qquad \Delta \vdash e_2 : \tau_2}{\Delta \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ \textsc{Tuple} \qquad \frac{}{\Delta \vdash n : \mathit{int}}\ \textsc{Int} \qquad \frac{}{\Delta \vdash s : \mathit{str}}\ \textsc{Str}$$

$$\frac{\Delta \vdash e : \tau_1 \times \tau_2}{\Delta \vdash e.\mathrm{fst} : \tau_1}\ \textsc{Fst}_1 \qquad \frac{\Delta \vdash e : ?}{\Delta \vdash e.\mathrm{fst} : ?}\ \textsc{Fst}_2 \qquad \frac{\Delta \vdash e : \tau_1 \times \tau_2}{\Delta \vdash e.\mathrm{snd} : \tau_2}\ \textsc{Snd}_1 \qquad \frac{\Delta \vdash e : ?}{\Delta \vdash e.\mathrm{snd} : ?}\ \textsc{Snd}_2$$

$$\frac{\Delta \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta \vdash e_2 : \tau_1' \qquad \tau_1' <: \tau_1}{\Delta \vdash e_1\ \ e_2 : \tau_2}\ \textsc{Call}_1 \qquad \frac{\Delta \vdash e_1 : ? \qquad \Delta \vdash e_2 : \tau}{\Delta \vdash e_1\ \ e_2 : ?}\ \textsc{Call}_2$$

$$\frac{\Delta \vdash e_1 : \tau_1 \qquad \Delta \vdash e_2 : \tau_2 \qquad \tau_1 <: \mathit{int} \qquad \tau_2 <: \mathit{int}}{\Delta \vdash e_1 + e_2 : \mathit{int}}\ \textsc{Add} \qquad \frac{\Delta \vdash e : \tau \qquad \tau <: \mathit{str}}{\Delta \vdash e.\mathrm{length} : \mathit{int}}\ \textsc{Len}$$

$$\frac{\Delta \vdash e_1 : \tau_1 \qquad x : \tau_1, \Delta \vdash e_2 : \tau}{\Delta \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : \tau}\ \textsc{Let} \qquad \frac{x : \tau \in \Delta}{\Delta \vdash x : \tau}\ \textsc{Ref} \qquad \frac{\Delta \vdash \tau_1 \qquad x : \tau_1, \Delta \vdash e : \tau_2}{\Delta \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}\ \textsc{Fn}$$

$$\frac{f : \forall \alpha . \tau_1 \to \tau_2 \in \Delta \qquad \Delta \vdash \tau \qquad \Delta \vdash e : \tau_1' \qquad \tau_1' <: \{\tau/\alpha\}\tau_1}{\Delta \vdash f[\tau]\ \ e : \{\tau/\alpha\}\tau_2}\ \textsc{PolyCall}$$

$$\frac{\alpha, \Delta \vdash \tau_1 \qquad x : \tau_1, \alpha, \Delta \vdash e_f : \tau_2 \qquad f : \forall \alpha . \tau_1 \to \tau_2, \Delta \vdash e_2 : \tau}{\Delta \vdash \mathrm{let}\ f = \forall \alpha . \lambda x : \tau_1 . e_f\ \mathrm{in}\ e_2 : \tau}\ \textsc{PolyLet}$$

$$\begin{array}{rcll}
\text{Type Environment } \Delta & = & \epsilon & \text{Empty Environment} \\
 & | & x : \tau, \Delta & \text{Binder Entry} \\
 & | & \alpha, \Delta & \text{Type Variable Entry} \\
 & | & x : \forall \alpha . \tau_1 \to \tau_2, \Delta & \text{Polymorphic Function Entry}
\end{array}$$

Figure 11.5: Typing Judgements

$$\begin{array}{rlll} \text{Value } v = & \langle n \rangle & \text{Integer} \\ \mid & \langle s \rangle & \text{String} \\ \mid & \langle (v_1, v_2) \rangle & \text{Tuple} \\ \mid & \langle \lambda x : \tau . e \rangle & \text{Function} \end{array}$$

Figure 11.6: Syntax of Values

$v \sim \tau$ means value v is top-level consistent with type τ.

$$\langle n \rangle \sim \mathit{int} \qquad \langle s \rangle \sim \mathit{str} \qquad v \sim ?$$

$$\langle (v_1, v_2) \rangle \sim \tau_1 \times \tau_2 \qquad \langle \lambda x : \tau . e \rangle \sim \tau_1 \rightarrow \tau_2$$

Figure 11.7: Top-Level Consistency

the function expression must refer to a polymorphic function. The restrictions upon the argument e and the resulting type mirror the $\textsc{Call}_1$ rule after substituting the type variable α for τ.

11.5 Operational Semantics

Before introducing the operational semantics, we first introduce the concept of *values* and *top-level consistency*.

The syntax for values is shown in Figure 11.6. In our operational semantics, all reducible expressions reduce to either a value or an error. Values are irreducible and are differentiated from expressions by being surrounded in angle brackets. Supported values are integers, strings, tuples, and functions. Additionally, the function value is well-formed if and only if $x : \tau \vdash e : \tau'$.

Top-level consistency is the concept that a given value matches a given type up to its "top-most" level. Top-level consistency can be cheaply checked at runtime in constant time [14]. The relation is defined formally in Figure 11.7. Note that a tuple value is consistent with *any* tuple type, and a function value is consistent with *any* arrow type. All values are consistent with the ? type.

To simplify the presentation of the operational semantics, we introduce two new expressions: the check and annotated value expressions, whose syntax is shown in Figure 11.8. The Chk and AVal inference rules are added for typing the new expressions. The $\textsc{PolyCall}_2$ rule is a combination of the previous PolyLet and PolyCall rules and is introduced to handle polymorphic functions after let substitution.

The operational semantics will be expressed using small-step semantics with expressions factored into evaluation contexts substituted with reducible expressions. The syntax for contexts and reducible expressions is shown in Figure 11.9.

$$\begin{array}{lcll}
\text{Expression } e = & & \ldots & \\
& \mid & e \text{ as } \tau & \text{Check} \\
& \mid & v : \tau & \text{Annotated Value}
\end{array}$$

$$\frac{\Delta \vdash e : \tau'}{\Delta \vdash e \text{ as } \tau : \tau}\ \textsc{Chk} \qquad \frac{v \sim \tau}{\Delta \vdash (v : \tau) : \tau}\ \text{A}\textsc{Val}$$

$$\frac{\begin{array}{c}\alpha, \Delta \vdash \tau_1 \qquad x : \tau_1, \alpha, \Delta \vdash e_f : \tau_2 \\ \Delta \vdash \tau \qquad \Delta \vdash e_2 : \tau_1' \qquad \tau_1' <: \{\tau/\alpha\}\tau_1\end{array}}{\Delta \vdash (\forall \alpha.\lambda x : \tau_1.e_f)[\tau]\ \ e_2 : \{\tau/\alpha\}\tau_2}\ \textsc{PolyCall}_2$$

Figure 11.8: New Expressions

$$\begin{array}{ll}
\text{Context } H = & H\ \ e \\
& \mid\ v : \tau\ \ H \\
& \mid\ (H, e) \\
& \mid\ (v : \tau, H) \\
& \mid\ H.\text{fst} \\
& \mid\ H.\text{snd} \\
& \mid\ H + e \\
& \mid\ v : \tau + H \\
& \mid\ H.\text{length} \\
& \mid\ \text{let } x = H \text{ in } e \\
& \mid\ e[\tau]\ \ H \\
& \mid\ H \text{ as } \tau \\
& \mid\ \bullet
\end{array}
\qquad
\begin{array}{ll}
\text{Reducible Expression } r = & (v_1 : \tau_1)\ \ (v_2 : \tau_2) \\
& \mid\ (\forall \alpha.\lambda x : \tau_x : e)[\tau]\ \ (v_2 : \tau_2) \\
& \mid\ \lambda x : \tau.e \\
& \mid\ (v_1 : \tau_1, v_2 : \tau_2) \\
& \mid\ (v : \tau).\text{fst} \\
& \mid\ (v : \tau).\text{snd} \\
& \mid\ (v_1 : \tau_1) + (v_2 : \tau_2) \\
& \mid\ (v : \tau).\text{length} \\
& \mid\ \text{let } x = v : \tau \text{ in } e \\
& \mid\ \text{let } f = \forall \alpha.\lambda x : \tau_x.e_f \text{ in } e \\
& \mid\ n \\
& \mid\ s
\end{array}$$

Figure 11.9: Syntax for Contexts and Reducible Expressions

Every context has one and only one occurrence of the hole $\bullet$. We use the notation $H[e]$ to mean the expression resulting from substituting the hole in H with the expression e. Reducible expressions, r, are a subset of expressions that can be reduced in a single step, either to an annotated value or to an error.

The inference rules defining the small-step semantics are listed in Figure 11.10. The E.Ctxt and E.CtxtErr rules are standard and handle evaluations within a context. The E.Fn and E.Tuple rules are standard and handle creation of functions and tuples.

The E.Call$_1$ rule handles the case of calling a value known to have an arrow type. Note that the argument value v must be consistent with both the statically-inferred argument type τ_1 and the actual argument type τ_x, otherwise the call fails as specified in E.CallErr$_1$. The argument value is annotated with the *actual* argument type τ_x before being substituted into the function body e. Additionally, a check expression enforces that the result is consistent with the statically-inferred return type τ_2.

The E.CALL$_2$ rule handles the case of calling a value with type ?. It mirrors the E.CALL$_1$ rule except the argument value is only checked against the actual argument type, and the return type is ?. The call fails if the argument check fails, as in E.CALLERR$_2$, or if the called value is not a function, as in E.CALLERR$_3$.

The polymorphic E.POLYCALL and E.POLYCALLERR rules mirror the regular calling rules E.CALL$_1$ and E.CALLERR$_1$, except with occurrences of the type variable α substituted for the type argument τ.

The tuple projection rules E.FST$_1$ and E.SND$_1$ are standard and extract the appropriate elements from the tuple value. The result is enforced to be consistent with the expected statically-inferred type, or else the projection fails, as shown in E.FSTERR$_1$ and E.SNDERR$_1$. Rules E.FST$_2$ and E.SND$_2$ handle the case when the value to project has type ?. In this case, the result is annotated with type ?. E.FSTERR$_2$ and E.SNDERR$_2$ handle the cases when the value to project is not a tuple.

Literal integer and string expressions reduce to annotated integer and string values, E.INT and E.STR. The add expression requires two integer values, E.ADD, or else fails, E.ADDERR. Similarly, the length expression requires a string value, E.LEN, or else fails, E.LENERR.

The E.LET rule substitutes every occurrence of the binder x in the body e with the annotated value $v : \tau$. The polymorphic E.POLYLET rule substitutes every occurrence of the binder f with the polymorphic function.

The E.CHK rule performs a dynamic check to enforce that the value is consistent with the check type. If not, then the operation fails, as shown in E.CHKERR.

$e \Rightarrow e'$ means that the expression e evaluates to the new expression e' in a single step.
$e \Rightarrow$ error means that the expression e evaluates to an error.

$$\frac{r \Rightarrow v : \tau}{H[r] \Rightarrow H[v : \tau]}\ \textsc{E.Ctxt} \qquad \frac{r \Rightarrow \text{error}}{H[r] \Rightarrow \text{error}}\ \textsc{E.CtxtErr}$$

$$\frac{x : \tau_1 \vdash e : \tau_2}{\lambda x : \tau_1.e \Rightarrow \langle \lambda x : \tau_1.e \rangle : \tau_1 \rightarrow \tau_2}\ \textsc{E.Fn} \qquad \frac{v \nsim \tau_x}{(\langle \lambda x : \tau_x.e \rangle : ?)\ \ (v : \tau_v) \Rightarrow \text{error}}\ \textsc{E.CallErr}_2 \qquad \frac{v_1 \neq \langle \lambda x : \tau_x.e \rangle}{(v_1 : ?)\ \ (v_2 : \tau_v) \Rightarrow \text{error}}\ \textsc{E.CallErr}_3$$

$$\frac{v \sim \tau_1 \qquad v \sim \tau_x}{(\langle \lambda x : \tau_x.e \rangle : \tau_1 \rightarrow \tau_2)\ \ (v : \tau_v) \Rightarrow (\{v : \tau_x/x\}e) \text{ as } \tau_2}\ \textsc{E.Call}_1 \qquad \frac{v \nsim \tau_1 \text{ or } v \nsim \tau_x}{(\langle \lambda x : \tau_x.e \rangle : \tau_1 \rightarrow \tau_2)\ \ (v : \tau_v) \Rightarrow \text{error}}\ \textsc{E.CallErr}_1$$

$$\frac{v \sim \tau_x}{(\langle \lambda x : \tau_x.e \rangle : ?)\ \ (v : \tau_v) \Rightarrow (\{v : \tau_x/x\}e) \text{ as } ?}\ \textsc{E.Call}_2 \qquad \frac{v \sim \{\tau/\alpha\}\tau_x}{(\forall\alpha.\lambda x : \tau_x.e_f)[\tau]\ \ (v : \tau_v) \Rightarrow \{\tau/\alpha\}\{v : \tau_x/x\}e_f}\ \textsc{E.PolyCall}$$

$$\frac{}{\text{let } f = \forall\alpha.\lambda x : \tau.e_f \text{ in } e \Rightarrow \{(\forall\alpha.\lambda x : \tau.e_f)/f\}e}\ \textsc{E.PolyLet} \qquad \frac{v \nsim \{\tau/\alpha\}\tau_x}{(\forall\alpha.\lambda x : \tau_x.e_f)[\tau]\ \ (v : \tau_v) \Rightarrow \text{error}}\ \textsc{E.PolyCallErr}$$

$$\frac{v_1 \sim \tau_1}{(\langle(v_1, v_2)\rangle : \tau_1 \times \tau_2).\text{fst} \Rightarrow v_1 : \tau_1}\ \textsc{E.Fst}_1 \qquad \frac{}{(\langle(v_1, v_2)\rangle : ?).\text{fst} \Rightarrow v_1 : ?}\ \textsc{E.Fst}_2 \qquad \frac{v_1 \nsim \tau_1}{(\langle(v_1, v_2)\rangle : \tau_1 \times \tau_2).\text{fst} \Rightarrow \text{error}}\ \textsc{E.FstErr}_1$$

$$\frac{v_2 \sim \tau_2}{(\langle(v_1, v_2)\rangle : \tau_1 \times \tau_2).\text{snd} \Rightarrow v_2 : \tau_2}\ \textsc{E.Snd}_1 \qquad \frac{}{(\langle(v_1, v_2)\rangle : ?).\text{snd} \Rightarrow v_2 : ?}\ \textsc{E.Snd}_2 \qquad \frac{v_2 \nsim \tau_2}{(\langle(v_1, v_2)\rangle : \tau_1 \times \tau_2).\text{snd} \Rightarrow \text{error}}\ \textsc{E.SndErr}_1$$

$$\frac{v \neq \langle(v_1, v_2)\rangle}{(v : ?).\text{fst} \Rightarrow \text{error}}\ \textsc{E.FstErr}_2 \qquad \frac{v \neq \langle(v_1, v_2)\rangle}{(v : ?).\text{snd} \Rightarrow \text{error}}\ \textsc{E.SndErr}_2 \qquad \frac{}{(v_1 : \tau_1, v_2 : \tau_2) \Rightarrow \langle(v_1, v_2)\rangle : \tau_1 \times \tau_2}\ \textsc{E.Tuple}$$

$$\frac{}{\langle n_1 \rangle : \tau_1 + \langle n_2 \rangle : \tau_2 \Rightarrow \langle n_1 + n_2 \rangle : \mathit{int}}\ \textsc{E.Add} \qquad \frac{v_1 \neq \langle n_1 \rangle \text{ or } v_2 \neq \langle n_2 \rangle}{v_1 : \tau_1 + v_2 : \tau_2 \Rightarrow \text{error}}\ \textsc{E.AddErr} \qquad \frac{v \sim \tau}{(v : \tau_v) \text{ as } \tau \Rightarrow v : \tau}\ \textsc{E.Chk}$$

$$\frac{}{(\langle s \rangle : \tau).\text{length} \Rightarrow \langle s.\text{length} \rangle : \mathit{int}}\ \textsc{E.Len} \qquad \frac{v \neq \langle s \rangle}{(v : \tau).\text{length} \Rightarrow \text{error}}\ \textsc{E.LenErr} \qquad \frac{v \nsim \tau}{(v : \tau_v) \text{ as } \tau \Rightarrow \text{error}}\ \textsc{E.ChkErr}$$

$$\frac{}{\text{let } x = v : \tau \text{ in } e \Rightarrow \{v : \tau/x\}e}\ \textsc{E.Let} \qquad \frac{}{n \Rightarrow \langle n \rangle : \mathit{int}}\ \textsc{E.Int} \qquad \frac{}{s \Rightarrow \langle s \rangle : \mathit{str}}\ \textsc{E.Str}$$

Figure 11.10: Small-Step Semantics

Chapter 12

Properties of the Core Type System

Our system is type-safe, satisfies the standard progress and preservation theorems, and also satisfies two incrementality theorems. We formally state and prove the theorems in this chapter.

12.1 Equivalence Modulo Types

The incrementality theorems require the concept of *equivalence modulo types*. Informally, an expression e_1 is equivalent to another expression e_2 if e_2 is syntactically equivalent to e_1 except with some subset of type annotations replaced with the ? type. Figures 12.1 and 12.2 shows the formal definitions of the $\overset{?}{\approx}$ and $<_?$ relations.

$e_1 \overset{?}{\approx} e_2$ means that expression e_1 is equivalent to expression e_2 modulo type annotations.
$v_1 \overset{?}{\approx} v_2$ means that value v_1 is equivalent to value v_2 modulo type annotations.

$$\frac{}{x \overset{?}{\approx} x} \qquad \frac{}{n \overset{?}{\approx} n} \qquad \frac{}{s \overset{?}{\approx} s} \qquad \frac{}{\langle n\rangle \overset{?}{\approx} \langle n\rangle} \qquad \frac{}{\langle s\rangle \overset{?}{\approx} \langle s\rangle} \qquad \frac{v_1 \overset{?}{\approx} v_1' \qquad v_2 \overset{?}{\approx} v_2'}{\langle(v_1, v_2)\rangle \overset{?}{\approx} \langle(v_1', v_2')\rangle} \qquad \frac{e \overset{?}{\approx} e' \qquad \tau' = \tau \text{ or } \tau' = ?}{\langle \lambda x : \tau.e\rangle \overset{?}{\approx} \langle \lambda x : \tau'.e'\rangle}$$

$$\frac{e \overset{?}{\approx} e' \qquad \tau' = \tau \text{ or } \tau' = ?}{\lambda x : \tau.e \overset{?}{\approx} \lambda x : \tau'.e'} \qquad \frac{e_1 \overset{?}{\approx} e_1' \qquad e_2 \overset{?}{\approx} e_2'}{\text{let } x = e_1 \text{ in } e_2 \overset{?}{\approx} \text{let } x = e_1' \text{ in } e_2'} \qquad \frac{e \overset{?}{\approx} e'}{e.\text{fst} \overset{?}{\approx} e'.\text{fst}} \qquad \frac{v \overset{?}{\approx} v' \qquad \tau_1 <_? \tau_1'}{v : \tau \overset{?}{\approx} v' : \tau'}$$

$$\frac{e \overset{?}{\approx} e'}{e.\text{length} \overset{?}{\approx} e'.\text{length}} \qquad \frac{e_1 \overset{?}{\approx} e_1' \qquad e_2 \overset{?}{\approx} e_2'}{e_1 + e_2 \overset{?}{\approx} e_1' + e_2'} \qquad \frac{e \overset{?}{\approx} e' \qquad \tau <_? \tau'}{e \text{ as } \tau \overset{?}{\approx} e' \text{ as } \tau'} \qquad \frac{e \overset{?}{\approx} e'}{e.\text{snd} \overset{?}{\approx} e'.\text{snd}}$$

$$\frac{e_1 \overset{?}{\approx} e_1' \qquad e_2 \overset{?}{\approx} e_2'}{(e_1, e_2) \overset{?}{\approx} (e_1', e_2')} \qquad \frac{e_1 \overset{?}{\approx} e_1' \qquad e_2 \overset{?}{\approx} e_2'}{e_1 \;\; e_2 \overset{?}{\approx} e_1' \;\; e_2'} \qquad \frac{e_1 \overset{?}{\approx} e_1' \qquad e_2 \overset{?}{\approx} e_2'}{e_1[\tau] \;\; e_2 \overset{?}{\approx} e_1'[\tau] \;\; e_2'} \qquad \frac{e \overset{?}{\approx} e' \qquad \tau' = \tau \text{ or } \tau' = ?}{\forall\alpha.\lambda x : \tau.e \overset{?}{\approx} \forall\alpha.\lambda x : \tau'.e'}$$

Figure 12.1: Equivalence Modulo Types

$\tau_1 <_? \tau_2$ means τ_1 contains less "information" than τ_2.

$$\frac{\tau_1 <_? \tau_1' \qquad \tau_2 <_? \tau_2'}{\tau_1 \to \tau_2 <_? \tau_1' \to \tau_2'} \qquad \frac{}{\mathit{int} <_? \mathit{int}} \qquad \frac{}{\alpha <_? \alpha}$$

$$\frac{\tau_1 <_? \tau_1' \qquad \tau_2 <_? \tau_2'}{\tau_1 \times \tau_2 <_? \tau_1' \times \tau_2'} \qquad \frac{}{\mathit{str} <_? \mathit{str}} \qquad \frac{}{\tau <_? ?}$$

Figure 12.2: Subinfo Relation

12.2 Safety

Our type system is safe by satisfying the standard progress and preservation theorems.

Theorem 1 (Preservation)**.** *The inferred type of an expression is preserved across the small step operational semantics. If $\epsilon \vdash e : \tau$ and $e \Rightarrow e'$, then $\epsilon \vdash e' : \tau$.*

Theorem 2 (Progress)**.** *A well-typed expression never gets "stuck". It always evaluates to an annotated value expression or else an error is detected during execution. If $\epsilon \vdash e : \tau$ then either:*

1. *$e \Rightarrow e'$ for some e',*
2. *$e \Rightarrow$ error,*
3. *or $e = v : \tau$.*

12.3 Incrementality

Our type system satisfies two incrementality theorems, one stating that programs remain well-typed, and the other stating that semantics are preserved.

Theorem 3 (Typing Incrementality)**.** *A well-typed program can have any arbitrary subset of its type annotations replaced with the ? type, and remain well-typed. If $\epsilon \vdash e_1 : \tau_1$ and $e_1 \overset{?}{\approx} e_2$, then $\epsilon \vdash e_2 : \tau_2$.*

Theorem 4 (Evaluation Incrementality)**.** *A well-typed program can have any arbitrary subset of its type annotations replaced with the ? type and still evaluate to the same value (modulo types). That is, assume that e_2 is equivalent to e_1 but with some type annotations replaced with the ? type, and that e_1 is well-typed and reduces to e_1'. Then e_2 will reduce to e_2' and e_2' is guaranteed to be equivalent to e_1' modulo types. If $e_1 \overset{?}{\approx} e_2$ and $\epsilon \vdash e_1 : \tau_1$ and $e_1 \Rightarrow e_1'$, then $e_2 \Rightarrow e_2'$ and $e_1' \overset{?}{\approx} e_2'$.*

$\Delta \overset{?}{\approx} \Delta'$ means that type environment Δ is equivalent to type environment Δ' modulo type annotations.

$$\frac{\tau <_? \tau' \qquad \Delta \overset{?}{\approx} \Delta'}{x : \tau, \Delta \overset{?}{\approx} x : \tau', \Delta'} \qquad \frac{}{\epsilon \overset{?}{\approx} \epsilon} \qquad \frac{\Delta \overset{?}{\approx} \Delta'}{\alpha, \Delta \overset{?}{\approx} \alpha, \Delta'}$$

$$\frac{\tau_1 \rightarrow \tau_2 <_? \tau_1' \rightarrow \tau_2' \qquad \Delta \overset{?}{\approx} \Delta'}{x : \forall\alpha.\tau_1 \rightarrow \tau_2, \Delta \overset{?}{\approx} x : \forall\alpha.\tau_1' \rightarrow \tau_2', \Delta'}$$

Figure 12.3: Equivalence Modulo Types of Type Environments

$H \overset{?}{\approx} H'$ means that context H is equivalent to context H' modulo type annotations.

$$\frac{H \overset{?}{\approx} H' \qquad e \overset{?}{\approx} e'}{H + e \overset{?}{\approx} H' + e'} \qquad \frac{H \overset{?}{\approx} H' \qquad v : \tau \overset{?}{\approx} v' : \tau'}{v : \tau + H \overset{?}{\approx} v' : \tau' + H'} \qquad \frac{H \overset{?}{\approx} H' \qquad e \overset{?}{\approx} e'}{\text{let } x = H \text{ in } e \overset{?}{\approx} \text{let } x = H' \text{ in } e'}$$

$$\frac{H \overset{?}{\approx} H'}{H.\text{fst} \overset{?}{\approx} H'.\text{fst}} \qquad \frac{H \overset{?}{\approx} H'}{H.\text{snd} \overset{?}{\approx} H'.\text{snd}} \qquad \frac{H \overset{?}{\approx} H' \qquad v : \tau \overset{?}{\approx} v' : \tau'}{(v : \tau, H) \overset{?}{\approx} (v' : \tau', H')} \qquad \frac{H \overset{?}{\approx} H' \qquad e \overset{?}{\approx} e'}{(H, e) \overset{?}{\approx} (H', e')}$$

$$\frac{H \overset{?}{\approx} H' \qquad e \overset{?}{\approx} e'}{H \;\; e \overset{?}{\approx} H' \;\; e'} \qquad \frac{H \overset{?}{\approx} H' \qquad e \overset{?}{\approx} e'}{e[\tau] \;\; H \overset{?}{\approx} e'[\tau] \;\; H'} \qquad \frac{H \overset{?}{\approx} H'}{H.\text{length} \overset{?}{\approx} H'.\text{length}}$$

$$\frac{H \overset{?}{\approx} H' \qquad v : \tau \overset{?}{\approx} v' : \tau'}{v : \tau \;\; H \overset{?}{\approx} v' : \tau' \;\; H'} \qquad \frac{H \overset{?}{\approx} H' \qquad \tau <_? \tau'}{H \text{ as } \tau \overset{?}{\approx} H' \text{ as } \tau'} \qquad \frac{}{\bullet \overset{?}{\approx} \bullet}$$

Figure 12.4: Equivalence Modulo Types of Evaluation Contexts

12.4 Required Lemmas and Definitions

The proofs of safety and incrementality require the additional definitions of the equivalence modulo types relation for type environments and evaluation contexts. These are shown in Figures 12.3 and 12.4.

Additionally, we state and prove the following lemmas which will be referred to later.

Lemma 1. *All well-typed expressions are either annotated value expressions, or contexts substituted with a reducible expression.*

If $\epsilon \vdash e : \tau$ *then either:*

1. $e = H[r]$

2. $e = v : \tau$

Proof. By induction on derivation of $\epsilon \vdash e : \tau$. □

Lemma 2. *If a context, H, substituted with a reducible expression, r, is well-typed, then the reducible expression r is also well-typed.*
If $\epsilon \vdash H[r] : \tau$ then $\epsilon \vdash r : \tau'$.

Proof. By induction on the syntax of H. □

Lemma 3. *If e has type τ in a type environment containing entry $x : \tau'$, and expression e' has type τ', then the expression $\{e'/x\}e$ will have type τ in the same type environment without the entry $x : \tau'$.*
If $\Delta_1, x : \tau', \Delta_2 \vdash e : \tau$ and $\epsilon \vdash e' : \tau'$, then $\Delta_1, \Delta_2 \vdash \{e'/x\}e : \tau$

Proof. By induction on the derivation of $\Delta_1, x : \tau', \Delta_2 \vdash e : \tau$. □

Lemma 4. *If the polymorphic function $f = \forall\alpha.\lambda x : \tau_1.e_f$ is well-typed (τ_1 is well-formed and e_f is well-typed), and e has type τ in an environment containing an entry for f, then substituting $\forall\alpha.\lambda x : \tau_1.e_f$ for f in e will still have type τ.*
If $\alpha \vdash \tau_1$ and $\alpha, x : \tau_1 \vdash e_f : \tau_2$ and $f : \forall\alpha.\tau_1 \to \tau_2 \vdash e : \tau$, then $\epsilon \vdash \{\forall\alpha.\lambda x : \tau_1.e_f/f\}e : \tau$.

Proof. By induction on the derivation of $f : \forall\alpha.\tau_1 \to \tau_2 \vdash e : \tau$. □

Lemma 5. *If expression e has type τ in a type environment containing type variable α, then the expression resulting from substituting α for τ' in e will have type $\{\tau'/\alpha\}\tau$.*
If $\Delta_1, \alpha, \Delta_2 \vdash e : \tau$, then $\Delta_1, \Delta_2 \vdash \{\tau'/\alpha\}e : \{\tau'/\alpha\}\tau$.

Proof. By induction on derivation of $\Delta_1, \alpha, \Delta_2 \vdash e : \tau$. □

Lemma 6. *If the expression $H[e_1]$ has type τ, and the expressions e_1 and e_2 have the same type, then the expression $H[e_2]$ will also have type τ.*
If $\Delta \vdash e_1 : \tau'$, and $\Delta \vdash e_2 : \tau'$, and $\Delta \vdash H[e_1] : \tau$, then $\Delta \vdash H[e_2] : \tau$.

Proof. By induction on the syntax of H. □

Lemma 7. *If value v is top-level consistent with type* int, *then v is an integer.*
If $v \sim \text{int}$ then $v = \langle n \rangle$ for some integer n.

Proof. By induction on syntax of v, and then applying inversion on the typing judgement. □

Lemma 8. *If value v is top-level consistent with type* str, *then v is a string.*
If $v \sim \text{str}$ then $v = \langle s \rangle$ for some string s.

Proof. By induction on syntax of v, and then applying inversion on the typing judgement. □

Lemma 9. *If value v is top-level consistent with a tuple type, then v is a tuple.*
If $v \sim \tau_1 \times \tau_2$ then $v = \langle(v_1, v_2)\rangle$ for some value v_1 and v_2.

Proof. By induction on syntax of v, and then applying inversion on the typing judgement. □

Lemma 10. *If value v is top-level consistent with an arrow type, then v is a function.*
If $v \sim \tau_1 \rightarrow \tau_2$ then $v = \langle \lambda x : \tau_x . e\rangle$ for some binder x, type τ_x, and expression e.

Proof. By induction on syntax of v, and then applying inversion on the typing judgement. □

Lemma 11. *If $\tau_1 <: \tau_2$, and τ_1, τ_2 contain less information than τ_1', τ_2' respectively, then $\tau_1' <: \tau_2'$.*
If $\tau_1 <: \tau_2$ and $\tau_1 <_? \tau_1'$ and $\tau_2 <_? \tau_2'$, then $\tau_1' <: \tau_2'$.

Proof. By induction on derivation of $\tau_1 <: \tau_2$. □

Lemma 12. *If the ? type has less information than type τ, then τ must be the ? type itself.*
If $? <_? \tau$, then $\tau = ?$.

Proof. By induction on the derivation of $? <_? \tau$. □

Lemma 13. *If an arrow type, $\tau_1 \rightarrow \tau_2 <_? \tau$, has less information than type τ, then τ must be either the ? type or an arrow type.*
If $\tau_1 \rightarrow \tau_2 <_? \tau$, then either $\tau = ?$, or $\tau = \tau_1' \rightarrow \tau_2'$ where $\tau_1 <_? \tau_1'$ and $\tau_2 <_? \tau_2'$.

Proof. By induction on the derivation of $\tau_1 \rightarrow \tau_2 <_? \tau$. □

Lemma 14. *If a tuple type, $\tau_1 \times \tau_2 <_? \tau$, has less information than type τ, then τ must be either the ? type or a tuple type.*
If $\tau_1 \times \tau_2 <_? \tau$, then either $\tau = ?$, or $\tau = \tau_1' \times \tau_2'$ and $\tau_1 <_? \tau_1'$ and $\tau_2 <_? \tau_2'$.

Proof. By induction on the derivation of $\tau_1 \times \tau_2 <_? \tau$. □

Lemma 15. *If the polymorphic entry $f : \forall\alpha.\tau_1 \rightarrow \tau_2$ exists in type environment Δ_1, and Δ_1 is equivalent to Δ_2 modulo types, then a polymorphic entry for f will also exist in Δ_2.*

If $f : \forall\alpha.\tau_1 \rightarrow \tau_2 \in \Delta_1$ and $\Delta_1 \overset{?}{\approx} \Delta_2$, then $f : \forall\alpha.\tau_1' \rightarrow \tau_2' \in \Delta_2$ where $\tau_1 <_? \tau_1'$ and $\tau_2 <_? \tau_2'$.

Proof. By induction on the derivation of $\Delta_1 \overset{?}{\approx} \Delta_2$. □

Lemma 16. *If the binder entry $x : \tau$ exists in type environment Δ_1, and Δ_1 is equivalent to Δ_2 modulo types, then a binder entry will also exist for x in Δ_2.*
If $x : \tau \in \Delta_1$ and $\Delta_1 <_? \Delta_2$, then $x : \tau' \in \Delta_2$ where $\tau <_? \tau'$.

Proof. By induction on the derivation of $\Delta_1 \overset{?}{\approx} \Delta_2$. □

Lemma 17. *If type τ is well-formed under Δ_1, and Δ_1 is equivalent to Δ_2 modulo types, then τ is also well-formed under Δ_2.*

If $\Delta_1 \vdash \tau$ and $\Delta_1 <_? \Delta_2$, then $\Delta_2 \vdash \tau$.

Proof. By induction on the derivation of $\Delta_1 \vdash \tau$. □

Lemma 18. *If $\tau_1 <: \{\tau/\alpha\}\tau_2$, and types τ_1 and τ_2 have less information respectively than τ_1' and τ_2', then $\tau_1' <: \{\tau/\alpha\}\tau_2'$.*

If $\tau_1 <: \{\tau/\alpha\}\tau_2$ and $\tau_1 <_? \tau_1'$ and $\tau_2 <_? \tau_2'$, then $\tau_1' <: \{\tau/\alpha\}\tau_2'$.

Proof. By induction on derivation of $\tau_1 <: \{\tau/\alpha\}\tau_2$. □

Lemma 19. *Substituting a type variable for a given type does not affect the $<_?$ relation.*

If $\tau_1 <_? \tau_2$, then $\{\tau/\alpha\}\tau_1 <_? \{\tau/\alpha\}\tau_2$.

Proof. By induction on the derivation of $\tau_1 <_? \tau_2$. □

Lemma 20. *If expressions e_1 and e_2 are respectively equivalent to e_1' and e_2' modulo types, then the expressions $\{e_2/x\}e_1$ and $\{e_2'/x\}e_1'$ are also equivalent modulo types.*

If $e_1 \overset{?}{\approx} e_1'$ and $e_2 \overset{?}{\approx} e_2'$, then $\{e_2/x\}e_1 \overset{?}{\approx} \{e_2'/x\}e_1'$.

Proof. By induction on derivation of $e_1 \overset{?}{\approx} e_1'$. □

Lemma 21. *If value v_1 is consistent with type τ_1, v_1 is equivalent to v_2 modulo types, and τ_1 has less information than τ_2, then value v_2 is also consistent with type τ_2.*

If $v_1 \sim \tau_1$ and $v_1 \overset{?}{\approx} v_2$ and $\tau_1 <_? \tau_2$, then $v_2 \sim \tau_2$.

Proof. By induction on derivation of $v_1 \sim \tau_1$. □

Lemma 22. *If expression e_1 can be expressed as $H_1[r_1]$, and e_1 is equivalent to e_2 modulo types, then e_2 can also be expressed as $H_2[r_2]$.*

If $e_1 = H_1[r_1]$ and $e_1 \overset{?}{\approx} e_2$, then $e_2 = H_2[r_2]$ where $H_1 \overset{?}{\approx} H_2$ and $r_1 \overset{?}{\approx} r_2$.

Proof. By induction on syntax of H. □

Lemma 23. *If contexts H_1 and H_2 are equivalent modulo types, and expressions e_1 and e_2 are equivalent modulo types, then $H_1[e_1]$ and $H_2[e_2]$ are also equivalent modulo types.*

If $H_1 \overset{?}{\approx} H_2$ and $e_1 \overset{?}{\approx} e_2$, then $H_1[e_1] \overset{?}{\approx} H_2[e_2]$.

Proof. By induction on derivation of $H_1 \overset{?}{\approx} H_2$. □

12.5 Proof of Preservation

If $\epsilon \vdash e : \tau$ and $e \Rightarrow e'$, then $\epsilon \vdash e' : \tau$.

Proof. By induction on the derivation of $e \Rightarrow e'$.

Case:

$$\frac{r \Rightarrow v : \tau'}{H[r] \Rightarrow H[v : \tau']}$$

By Lemma 2, we know that $\epsilon \vdash r : \tau_1$ given that $H[r]$ is well-typed. From the typing judgement, we know that $\epsilon \vdash (v : \tau') : \tau'$. From the induction hypothesis, then $\tau_1 = \tau'$. Finally from Lemma 6, we therefore know that $\epsilon \vdash H[v : \tau'] : \tau$.

Case:

$$\frac{v \sim \tau_1 \qquad v \sim \tau_x}{(\langle \lambda x : \tau_x . e \rangle : \tau_1 \to \tau_2)\ \ (v : \tau_v) \Rightarrow (\{v : \tau_x / x\}e) \text{ as } \tau_2}$$

By inversion of the typing judgement, we know that $\epsilon \vdash (\langle \lambda x : \tau_x . e \rangle : \tau_1 \to \tau_2) : \tau_1 \to \tau_2$ and that $\tau = \tau_2$. Well-formedness of the function value requires that $x : \tau_x \vdash e : \tau'$, which leads to $\epsilon \vdash \{v : \tau_x / x\}e : \tau'$ from Lemma 3. Finally, we can derive $\epsilon \vdash (\{v : \tau_x / x\}e)$ as $\tau_2 : \tau_2$ from the typing judgement.

Case:

$$\frac{v \sim \tau_x}{(\langle \lambda x : \tau_x . e \rangle : ?)\ \ (v : \tau_v) \Rightarrow (\{v : \tau_x / x\}e) \text{ as } ?}$$

By inversion of the typing judgement, we know that $\tau = ?$. Well-formedness of the function value requires that $x : \tau_x \vdash e : \tau'$, which leads to $\epsilon \vdash \{v : \tau_x / x\}e : \tau'$ by Lemma 3. Finally, we can derive $\epsilon \vdash (\{v : \tau_x / x\}e)$ as $? : ?$ from the typing judgement.

Case:

$$\frac{}{(v_1 : \tau_1, v_2 : \tau_2) \Rightarrow \langle (v_1, v_2) \rangle : \tau_1 \times \tau_2}$$

By inversion of the typing judgement, we know that $\tau = \tau_1 \times \tau_2$. Then $\epsilon \vdash (\langle (v_1, v_2) \rangle : \tau_1 \times \tau_2) : \tau_1 \times \tau_2$ follows trivially from the typing judgement.

Case:

$$\frac{v_1 \sim \tau_1}{(\langle (v_1, v_2) \rangle : \tau_1 \times \tau_2).\text{fst} \Rightarrow v_1 : \tau_1}$$

By inversion of the typing judgement, $\epsilon \vdash (\langle(v_1, v_2)\rangle : \tau_1 \times \tau_2) : \tau_1 \times \tau_2$, and therefore $\tau = \tau_1$. Then $\epsilon \vdash (v_1 : \tau_1) : \tau_1$ follows trivially from the typing judgement.

Case:

$$\frac{}{(\langle(v_1, v_2)\rangle : ?).\text{fst} \Rightarrow v_1 : ?}$$

By inversion of the typing judgement, $\epsilon \vdash (\langle(v_1, v_2)\rangle : ?) : ?$, and therefore $\tau = ?$. Then $\epsilon \vdash (v_1 : ?) : ?$ follows trivially from the typing judgement.

Case:

$$\frac{v_1 \sim \tau_1}{(\langle(v_1, v_2)\rangle : \tau_1 \times \tau_2).\text{snd} \Rightarrow v_2 : \tau_2}$$

Symmetric to case for e.fst.

Case:

$$\frac{}{(\langle(v_1, v_2)\rangle : ?).\text{snd} \Rightarrow v_2 : ?}$$

Symmetric to case for e.fst.

Case:

$$\frac{}{n \Rightarrow \langle n\rangle : \mathit{int}}$$

Follows trivially from the typing judgement.

Case:

$$\frac{}{s \Rightarrow \langle s\rangle : \mathit{str}}$$

Follows trivially from the typing judgement.

Case:

$$\frac{}{\langle n_1\rangle : \tau_1 + \langle n_2\rangle : \tau_2 \Rightarrow \langle n_1 + n_2\rangle : \mathit{int}}$$

Follows trivially from the typing judgement.

Case:

$$\frac{}{(\langle s\rangle : \tau).\text{length} \Rightarrow \langle s.\text{length}\rangle : \mathit{int}}$$

Follows trivially from the typing judgement.

Case:

$$\frac{}{\text{let } x = v : \tau_v \text{ in } e \Rightarrow \{v : \tau_v / x\}e}$$

By inversion of the typing judgement we know that $\epsilon \vdash (v : \tau_v) : \tau_v$ and that $x : \tau_v \vdash e : \tau$. From this we can derive $\epsilon \vdash \{v : \tau_v / x\}e : \tau$ from Lemma 3.

Case:

$$\frac{}{\text{let } f = \forall\alpha.\lambda x : \tau_1.e_f \text{ in } e \Rightarrow \{(\forall\alpha.\lambda x : \tau_1.e_f)/f\}e}$$

By inversion of the typing judgement we know that $f : \forall\alpha.\tau_1 \rightarrow \tau_2 \vdash e : \tau$ and $\alpha \vdash \tau_1$ and $\alpha, x : \tau_1 \vdash e_f : \tau_2$. From Lemma 4, we can derive $\epsilon \vdash \{(\forall\alpha.\lambda x : \tau_1.e_f)/f\}e : \tau$.

Case:

$$\frac{x : \tau_1 \vdash e : \tau_2}{\lambda x : \tau_1.e \Rightarrow \langle\lambda x : \tau_1.e\rangle : \tau_1 \rightarrow \tau_2}$$

By inversion of the typing judgement, we can deduce that $x : \tau_1 \vdash e : \tau_2$ and $\tau = \tau_1 \rightarrow \tau_2$. $\epsilon \vdash (\langle\lambda x : \tau_1.e\rangle : \tau_1 \rightarrow \tau_2) : \tau_1 \rightarrow \tau_2$ follow trivially from the typing judgement.

Case:

$$\frac{v \sim \{\tau_1/\alpha\}\tau_x}{(\forall\alpha.\lambda x : \tau_x.e_f)[\tau_1]\ \ (v : \tau_v) \Rightarrow \{\tau_1/\alpha\}\{v : \tau_x/x\}e_f}$$

By inversion of the typing judgement we can deduce that $\alpha, x : \tau_x \vdash e_f : \tau_2$ and that $\tau = \{\tau_1/\alpha\}\tau_2$. Lemma 3 allows us to derive $\alpha \vdash \{v : \tau_x/x\}e_f : \tau_2$. Lemma 5 allows us to derive $\epsilon \vdash \{\tau_1/\alpha\}\{v : \tau_x/x\}e_f : \{\tau_1/\alpha\}\tau_2$.

□

12.6 Proof of Progress

Proof of Progress of Reducible Expressions

Lemma 24. *If $\epsilon \vdash r : \tau$ then either:*

1. $r \Rightarrow v : \tau$

2. $r \Rightarrow$ *error*

Proof. By induction on derivation of $\epsilon \vdash r : \tau$.

Case:

$$\frac{\epsilon \vdash (v_1 : \tau_{v1}) : \tau_1 \to \tau_2 \qquad \epsilon \vdash (v_2 : \tau_{v2}) : \tau_1' \qquad \tau_1' <: \tau_1}{\epsilon \vdash (v_1 : \tau_{v1})\ (v_2 : \tau_{v2}) : \tau_2}$$

By inversion of the typing judgement we know that $\tau_{v1} = \tau_1 \to \tau_2$ and $\tau_{v2} = \tau_1'$. By Lemma 10, we can thus deduce $v_1 = \langle \lambda x : \tau_x.e \rangle$. Now we consider whether $v_2 \sim \tau_1$ and $v_2 \sim \tau_x$. If it holds, then $(\langle \lambda x : \tau_x.e \rangle : \tau_1 \to \tau_2)\ (v_2 : \tau_1') \Rightarrow (\{v_2 : \tau_x/x\}e)$ as τ_2. Otherwise the expression reduces to an error.

Case:

$$\frac{\epsilon \vdash (v_1 : \tau_{v1}) : ? \qquad \epsilon \vdash (v_2 : \tau_{v2}) : \tau'}{\epsilon \vdash (v_1 : \tau_{v1})\ (v_2 : \tau_{v2}) : ?}$$

By inversion of the typing judgement we can deduce $\tau_{v1} = ?$ and $\tau_{v2} = \tau'$. We now assume that $v_1 = \langle \lambda x : \tau_x.e \rangle$. If this is not true, then the expression reduces to an error. Following the assumption, we can consider whether $v_2 \sim \tau_x$. If it holds then $(\langle \lambda x : \tau_x.e \rangle : ?)\ (v_2 : \tau') \Rightarrow (\{v_2 : \tau_x/x\}e)$ as ?. Otherwise, the expression reduces to an error.

Case:

$$\frac{\epsilon \vdash (v_1 : \tau_{v1}) : \tau_1 \qquad \epsilon \vdash (v_2 : \tau_{v2}) : \tau_2}{\epsilon \vdash (v_1 : \tau_{v1}, v_2 : \tau_{v2}) : \tau_1 \times \tau_2}$$

Follows trivially from the $\Rightarrow$ relation.

Case:

$$\frac{\epsilon \vdash (v : \tau_v) : \tau_1 \times \tau_2}{\epsilon \vdash (v : \tau_v).\text{fst} : \tau_1}$$

By inversion of the typing judgement, $\tau_v = \tau_1 \times \tau_2$. From Lemma 9, we can deduce $v = \langle (v_1, v_2) \rangle$. We can now consider whether $v_1 \sim \tau_1$. If it holds then $(\langle (v_1, v_2) \rangle : \tau_1 \times \tau_2).\text{fst} \Rightarrow v_1 : \tau_1$. Otherwise, the expression reduces to an error.

Case:

$$\frac{\epsilon \vdash (v : \tau_v) : ?}{\epsilon \vdash (v : \tau_v).\text{fst} : ?}$$

By inversion of the typing judgement $\tau_v = ?$. We then assume that $v = \langle (v_1, v_2) \rangle$. If this assumption is not true, then the expression reduces to an error. If it is true, then $(\langle (v_1, v_2) \rangle : ?).\text{fst} \Rightarrow v_1 : ?$.

Case:

$$\frac{\epsilon \vdash (v : \tau_v) : \tau_1 \times \tau_2}{\epsilon \vdash (v : \tau_v).\text{snd} : \tau_2}$$

Symmetric to case for $(v : \tau_v).\text{fst}$.

Case:

$$\frac{\epsilon \vdash (v : \tau_v) : ?}{\epsilon \vdash (v : \tau_v).\text{snd} : ?}$$

Symmetric to case for $(v : \tau_v).\text{fst}$.

Case:

$$\frac{}{\epsilon \vdash n : \mathit{int}}$$

Follows trivially from $\Rightarrow$ relation.

Case:

$$\frac{}{\epsilon \vdash s : \mathit{str}}$$

Follows trivially from $\Rightarrow$ relation.

Case:

$$\frac{\epsilon \vdash (v_1 : \tau_{v1}) : \tau_1 \qquad \epsilon \vdash (v_2 : \tau_{v2}) : \tau_2 \qquad \tau_1 <: \mathit{int} \qquad \tau_2 <: \mathit{int}}{\epsilon \vdash (v_1 : \tau_{v1}) + (v_2 : \tau_{v2}) : \mathit{int}}$$

We assume that $v_1 = \langle n_1 \rangle$ and $v_2 = \langle n_2 \rangle$. If this is not true, then the expression reduces to an error. If true, then $\langle n_1 \rangle : \tau_{v1} + \langle n_2 \rangle : \tau_{v2} \Rightarrow \langle n_1 + n_2 \rangle : \mathit{int}$.

Case:

$$\frac{\epsilon \vdash (v : \tau_v) : \tau' \qquad \tau' <: \mathit{str}}{\epsilon \vdash (v : \tau_v).\text{length} : \mathit{int}}$$

We assume that $v = \langle s \rangle$. If this is not true, then the expression reduces to an error. If true, then $(\langle s \rangle : \tau_v).\text{length} \Rightarrow \langle s.\text{length} \rangle : \mathit{int}$.

Case:

$$\frac{\epsilon \vdash \{v : \tau_x / x\}e : \tau}{\epsilon \vdash \text{let } x = v : \tau_x \text{ in } e : \tau}$$

Follows trivially from ⇒ relation.

Case:

$$\frac{\alpha, \Delta \vdash \tau_1 \qquad x : \tau_1, \alpha, \Delta \vdash e_f : \tau_2 \qquad f : \forall \alpha . \tau_1 \rightarrow \tau_2, \Delta \vdash e_2 : \tau}{\Delta \vdash \text{let } f = \forall \alpha . \lambda x : \tau_1 . e_f \text{ in } e_2 : \tau}$$

Follows trivially from ⇒ relation.

Case:

$$\frac{\epsilon \vdash \tau_1 \qquad x : \tau_1 \vdash e : \tau_2}{\epsilon \vdash \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2}$$

Follows trivially from ⇒ relation.

Case:

$$\frac{\alpha \vdash \tau_x \qquad x : \tau_x, \alpha \vdash e_f : \tau_2 \qquad \epsilon \vdash \tau_a \qquad \epsilon \vdash (v : \tau_v) : \tau_1 \qquad \tau_1 <: \{\tau_a/\alpha\}\tau_x}{\epsilon \vdash (\forall \alpha . \lambda x : \tau_x . e_f)[\tau_a] \ (v : \tau_v) : \{\tau_a/\alpha\}\tau_2}$$

We consider whether $v \sim \{\tau_a/\alpha\}\tau_x$. If it is not true then the expression reduces to an error. If it is true, then $(\forall \alpha . \lambda x : \tau_x . e_f)[\tau_a] \ (v : \tau_v) \Rightarrow \{\tau_a/\alpha\}\{v : \tau_x/x\}e_f$.

Case:

$$\frac{\epsilon \vdash (v : \tau_v) : \tau'}{\epsilon \vdash (v : \tau_v) \text{ as } \tau : \tau}$$

We consider whether $v \sim \tau$. If it is not true then the expression reduces to an error. If it is true then $(v : \tau_v)$ as $\tau \Rightarrow v : \tau$.

□

Proof of Progress of Expressions

If $\epsilon \vdash e : \tau$ then either:

1. $e \Rightarrow e'$ for some e'
2. $e \Rightarrow$ error
3. $e = v : \tau$

Proof. According to Lemma 1, either $e = v : \tau$ or $e = H[r]$. The first case satisfies the theorem trivially. In the second case, Lemma 2 implies that r is well-typed, $\epsilon \vdash r : \tau'$. Therefore from Lemma 24, either $r \Rightarrow v' : \tau'$ or $r \Rightarrow$ error. Then from the ⇒ relation, either $H[r] \Rightarrow H[v' : \tau']$ or $H[r] \Rightarrow$ error respectively. □

12.7 Proof of Typing Incrementality

Lemma 25. *If $\Delta_1 \vdash e_1 : \tau_1$ and $e_1 \overset{?}{\approx} e_2$ and $\Delta_1 <_? \Delta_2$, then $\Delta_2 \vdash e_2 : \tau_2$ and $\tau_1 <_? \tau_2$.*

Proof. By induction on derivation of $e_1 \overset{?}{\approx} e_2$.

Case:

$$\frac{e_a \overset{?}{\approx} e'_b \qquad e_a \overset{?}{\approx} e'_b}{e_a \;\; e_b \overset{?}{\approx} e'_a \;\; e'_b}$$

By inversion of the typing judgement, either $\Delta_1 \vdash e_a : ?$ or $\Delta_1 \vdash e_a : \tau_a \rightarrow \tau_1$. In the first case, $\Delta_2 \vdash e'_a : ?$ from the induction hypothesis and Lemma 12. e'_b is well-typed from the induction hypothesis, and thus $\tau_2 = ?$.

In the second case, we know additionally $\Delta_1 \vdash e_b : \tau_b$ and $\tau_b <: \tau_a$. From the induction hypothesis and Lemma 13, we know that either $\Delta_2 \vdash e'_a : ?$ or $\Delta_2 \vdash e'_a : \tau'_a \rightarrow \tau_2$ where $\tau_a <_? \tau'_a$ and $\tau_1 <_? \tau_2$. If the former, then $\tau_2 = ?$. If the latter, then from the induction hypothesis, $\Delta_2 \vdash e'_b : \tau'_b$ and $\tau_b <_? \tau'_b$. From Lemma 11, we know $\tau'_b <: \tau'_a$.

Case:

$$\frac{e_a \overset{?}{\approx} e'_a \qquad e_b \overset{?}{\approx} e'_b}{e_a[\tau] \;\; e_b \overset{?}{\approx} e'_a[\tau] \;\; e'_b}$$

By inversion of the typing judgement, $e_a : \forall\alpha.\tau_{a1} \rightarrow \tau_{a2} \in \Delta_1$, $\Delta_1 \vdash \tau$, $\Delta_1 \vdash e_b : \tau_b$, and $\tau_b <: \{\tau/\alpha\}\tau_{a1}$. We need to show that:

1. $e'_a : \forall\alpha.\tau'_{a1} \rightarrow \tau'_{a2} \in \Delta_2$,
2. $\Delta_2 \vdash \tau$,
3. $\Delta_2 \vdash e'_b : \tau'_b$,
4. $\tau'_b <: \{\tau/\alpha\}\tau'_{a1}$, and
5. $\{\tau/\alpha\}\tau_{a2} <_? \{\tau/\alpha\}\tau'_{a2}$.

(1) follows from Lemma 15. (2) follows from Lemma 17. (3) follows from the induction hypothesis. (4) follows from the induction hypothesis and Lemma 18. (5) follows from the induction hypothesis, Lemma 11, and Lemma 19.

Case:

$$\frac{e \overset{?}{\approx} e' \qquad \tau' = \tau \text{ or } \tau' = ?}{\lambda x : \tau.e \overset{?}{\approx} \lambda x : \tau'.e'}$$

By inversion of the typing judgement, $\Delta_1 \vdash \tau$ and $x : \tau, \Delta_1 \vdash e : \tau_2$. Consider the cases of $\tau' = \tau$ and $\tau' = ?$ separately. In the first case, we need to show that:

1. $\Delta_2 \vdash \tau$,
2. $x : \tau, \Delta_2 \vdash e' : \tau_2'$, and
3. $\tau \rightarrow \tau_2 <_? \tau \rightarrow \tau_2'$.

(1) follows from Lemma 17. (2) follows from the induction hypothesis. (3) follows from the $<_?$ relation.

In the second case, we need to show that:

1. $\Delta_2 \vdash ?$,
2. $x : ?, \Delta_2 \vdash e' : \tau_2'$, and
3. $\tau \rightarrow \tau_2 <_? ? \rightarrow \tau_2'$.

(1) is trivial. (2) follows from induction hypothesis. (3) follows from the $<_?$ relation.

Case:

$$\frac{e_a \overset{?}{\approx} e_a' \qquad e_b \overset{?}{\approx} e_b'}{\text{let } x = e_a \text{ in } e_b \overset{?}{\approx} \text{let } x = e_a' \text{ in } e_b'}$$

We consider separately the case when e_a is a typable expression and when it is a polymorphic function. In the first case, by inversion of the typing judgement, $\Delta_1 \vdash e_a : \tau_a$ and $x : \tau_a, \Delta_1 \vdash e_b : \tau_1$. We need to show that:

1. $\Delta_2 \vdash e_a' : \tau_a'$,
2. $x : \tau_a', \Delta_2 \vdash e_b' : \tau_2$, and
3. $\tau_1 <_? \tau_2$,

which all follow from the induction hypothesis.

The second case is similar, and the proof is a combination of the first case and the proof for function creation.

Case:

$$\frac{e \overset{?}{\approx} e' \qquad \tau' = \tau \text{ or } \tau' = ?}{\forall\alpha.\lambda x : \tau.e \overset{?}{\approx} \forall\alpha.\lambda x : \tau'.e'}$$

Polymorphic functions cannot be typed in isolation, and thus we do not have to consider this case.

Case:

$$\frac{e_1 \overset{?}{\approx} e_1' \qquad e_2 \overset{?}{\approx} e_2'}{(e_1, e_2) \overset{?}{\approx} (e_1', e_2')}$$

Follows trivially from typing judgement and induction hypothesis.

Case:

$$\frac{e \overset{?}{\approx} e'}{e.\text{fst} \overset{?}{\approx} e'.\text{fst}}$$

By inversion of the typing judgement, either $\Delta_1 \vdash e : ?$ or $\Delta_1 \vdash e : \tau_1 \times \tau$. In the first case, $\Delta_2 \vdash e' : ?$ from the induction hypothesis and Lemma 12. And hence, $\Delta_2 \vdash e'.\text{fst} : ?$.

In the second case, by Lemma 14, either $\Delta_2 \vdash e' : ?$ or $\Delta_2 \vdash e' : \tau_2 \times \tau'$ and $\tau_1 <_? \tau_2$. If the former, then $\tau_2 = ?$. If the latter, then $\Delta_2 \vdash e'.\text{fst} : \tau_2$.

Case:

$$\frac{e \overset{?}{\approx} e'}{e.\text{snd} \overset{?}{\approx} e'.\text{snd}}$$

Symmetrical to the case for e.fst.

Case:

$$\frac{e_1 \overset{?}{\approx} e_1' \qquad e_2 \overset{?}{\approx} e_2'}{e_1 + e_2 \overset{?}{\approx} e_1' + e_2'}$$

Follows trivially from inversion of typing judgement, induction hypothesis, and Lemma 11.

Case:

$$\frac{e \overset{?}{\approx} e'}{e.\text{length} \overset{?}{\approx} e'.\text{length}}$$

Follows trivially from inversion of typing judgement, induction hypothesis, and Lemma 11.

Case:

$$\frac{}{x \overset{?}{\approx} x}$$

Follows trivially from inversion of typing judgement, and Lemma 16.

Case:

$$\frac{}{n \overset{?}{\approx} n}$$

Follows trivially from typing judgement.

Case:

$$\frac{}{s \overset{?}{\approx} s}$$

Follows trivially from typing judgement.

Case:

$$\frac{e \overset{?}{\approx} e' \qquad \tau <_? \tau'}{e \text{ as } \tau \overset{?}{\approx} e' \text{ as } \tau'}$$

Follows trivially from inversion of typing judgement and induction hypothesis.

Case:

$$\frac{v \overset{?}{\approx} v' \qquad \tau_1 <_? \tau_1'}{v : \tau \overset{?}{\approx} v' : \tau'}$$

Follows trivially from inversion of typing judgement and Lemma 21.

$\square$

12.8 Proof of Evaluation Incrementality

Evaluation Incrementality of Reducible Expressions

Lemma 26. *If $r_1 \overset{?}{\approx} r_2$ and $\epsilon \vdash r_1 : \tau_1$ and $r_1 \Rightarrow r_1'$, then $r_2 \Rightarrow r_2'$ and $r_1' \overset{?}{\approx} r_2'$.*

Proof. By induction on derivation of $r_1 \Rightarrow r_1'$.

Case:

$$\frac{v_1 \sim \tau_1' \qquad v_1 \sim \tau_{x1}}{(\langle \lambda x : \tau_{x1}.e_1 \rangle : \tau_1' \to \tau_1)\ (v_1 : \tau_{v1}) \Rightarrow (\{v_1 : \tau_{x1}/x\}e_1) \text{ as } \tau_1} \textsc{E.Call}_1$$

From inversion of $\overset{?}{\approx}$ relation, we can deduce $r_2 = (\langle \lambda x : \tau_{x2}.e_2 \rangle : \tau_2')\ (v_2 : \tau_{v2})$, $\tau_{x1} <_? \tau_{x2}$, $e_1 \overset{?}{\approx} e_2$, $\tau_1' \to \tau_1 <_? \tau_2'$, $v_1 \overset{?}{\approx} v_2$, and $\tau_{v1} <_? \tau_{v2}$. From Lemma 13, we know either $\tau_2' = \tau_2'' \to \tau_2'''$ or $\tau_2 = ?$. If $\tau_2 = ?$, then $r_2' = (\{v_2 : \tau_{x2}/x\}e_2)$ as ? and $r_1' = r_2'$ from Lemma 20.

If $\tau_2' = \tau_2'' \to \tau_2'''$, then we know additionally $\tau_2'' <_? \tau_1'$ and $\tau_2''' <_? \tau_1$. $r_2' = (\{v_2 : \tau_{x2}/x\}e_2)$ as τ_2''', and $r_1' = r_2'$ from Lemma 20.

Case:

$$\frac{v_1 \sim \tau_{x1}}{(\langle \lambda x : \tau_{x1}.e_1 \rangle : ?)\ (v_1 : \tau_{v1}) \Rightarrow (\{v_1 : \tau_{x1}/x\}e_1) \text{ as } ?}\ \textsc{E.Call}_2$$

From inversion of $\overset{?}{\approx}$ relation, and Lemma 12, we can deduce $r_2 = (\langle \lambda x : \tau_{x2}.e_2 \rangle : ?)\ (v_2 : \tau_{v2})$, $\tau_{x1} <_? \tau_{x2}$, $e_1 \overset{?}{\approx} e_2$, $v_1 \overset{?}{\approx} v_2$, and $\tau_{v1} <_? \tau_{v2}$. Then $r_2' = (\{v_2 : \tau_{x2}/x\}e_2)$ as ? and $r_1' = r_2'$ from Lemma 20.

Case:

$$\frac{}{(v_1 : \tau_1, v_2 : \tau_2) \Rightarrow \langle (v_1, v_2) \rangle : \tau_1 \times \tau_2}\ \textsc{E.Tuple}$$

Follows trivially from inversion of $\overset{?}{\approx}$ relation.

Case:

$$\frac{v_1 \sim \tau_1}{(\langle (v_1, v_2) \rangle : \tau_1 \times \tau_2).\text{fst} \Rightarrow v_1 : \tau_1}\ \textsc{E.Fst}_1$$

From inversion of $\overset{?}{\approx}$ relation, $r_2 = (\langle (v_1', v_2') \rangle : \tau_3).\text{fst}$. From Lemma 14, either $\tau_3 = ?$ or $\tau_3 = \tau_1' \times \tau_2'$. If $\tau_3 = ?$, then $r_2' = v_1' : ?$.

Otherwise, if $\tau_3 = \tau_1' \times \tau_2'$, then we additionally know $\tau_1 <_? \tau_1'$ and $\tau_2 <_? \tau_2'$. In this case, $r_2' = v_1' : \tau_1'$.

Case:

$$\frac{}{(\langle (v_1, v_2) \rangle : ?).\text{fst} \Rightarrow v_1 : ?}\ \textsc{E.Fst}_2$$

From inversion of $\overset{?}{\approx}$ relation and Lemma 12, we can deduce that $r_2 = (\langle (v_1', v_2') \rangle : ?).\text{fst}$. Then $r_2' = v_1' : ?$ from $\Rightarrow$ relation.

Case:

$$\frac{v_2 \sim \tau_2}{(\langle (v_1, v_2) \rangle : \tau_1 \times \tau_2).\text{snd} \Rightarrow v_2 : \tau_2}\ \textsc{E.Snd}_1$$

Symmetrical to case for e.fst.

Case:

$$\frac{}{(\langle(v_1, v_2)\rangle : ?).\text{snd} \Rightarrow v_2 : ?}\ \text{E.S}{\small\text{ND}}_2$$

Symmetrical to case for e.fst.

Case:

$$\frac{}{n \Rightarrow \langle n\rangle : \mathit{int}}\ \text{E.I}{\small\text{NT}}$$

Follows trivially from inversion of $\overset{?}{\approx}$ relation.

Case:

$$\frac{}{s \Rightarrow \langle s\rangle : \mathit{str}}\ \text{E.S}{\small\text{TR}}$$

Follows trivially from inversion of $\overset{?}{\approx}$ relation.

Case:

$$\frac{}{\langle n_1\rangle : \tau_1 + \langle n_2\rangle : \tau_2 \Rightarrow \langle n_1 + n_2\rangle : \mathit{int}}\ \text{E.A}{\small\text{DD}}$$

Follows trivially from inversion of $\overset{?}{\approx}$ relation.

Case:

$$\frac{}{(\langle s\rangle : \tau).\text{length} \Rightarrow \langle s.\text{length}\rangle : \mathit{int}}\ \text{E.L}{\small\text{EN}}$$

Follows trivially from inversion of $\overset{?}{\approx}$ relation.

Case:

$$\frac{}{\text{let } x = v : \tau \text{ in } e \Rightarrow \{v : \tau/x\}e}\ \text{E.L}{\small\text{ET}}$$

From inversion of $\overset{?}{\approx}$ relation, $r_2 = \text{let } x = v' : \tau' \text{ in } e'$, $v \overset{?}{\approx} v'$, $\tau <_? \tau'$, and $e \overset{?}{\approx} e'$. Thus $r_2' = \{v' : \tau'/x\}e'$ and $r_1' \overset{?}{\approx} r_2'$ from Lemma 20.

Case:

$$\frac{x : \tau_1 \vdash e : \tau_2}{\lambda x : \tau_1.e \Rightarrow \langle \lambda x : \tau_1.e\rangle : \tau_1 \to \tau_2}\ \text{E.F}{\small\text{N}}$$

From inversion of $\overset{?}{\approx}$ relation, $r_2 = \lambda x : \tau_1'.e'$ where $\tau_1 <_? \tau_1'$ and $e \overset{?}{\approx} e'$. From Typing Incrementality, we know that $x : \tau_1' \vdash e' : \tau_2'$ and $\tau_2 <_? \tau_2'$. Therefore $r_2' = \langle \lambda x : \tau_1'.e' \rangle : \tau_1' \rightarrow \tau_2'$, and $r_1' \overset{?}{\approx} r_2'$.

Case:

$$\frac{v \sim \{\tau/\alpha\}\tau_x}{(\forall \alpha.\lambda x : \tau_x.e_f)[\tau]\ \ (v : \tau_v) \Rightarrow \{\tau/\alpha\}\{v : \tau_x/x\}e_f}\ \textsc{E.PolyCall}$$

Mirrors the case for rule E.$\textsc{Call}_1$.

Case:

$$\frac{}{\text{let } f = \forall \alpha.\lambda x : \tau.e_f \text{ in } e \Rightarrow \{(\forall \alpha.\lambda x : \tau.e_f)/f\}e}\ \textsc{E.PolyLet}$$

Mirrors the case for rule E.Let.

Case:

$$\frac{v \sim \tau}{(v : \tau_v) \text{ as } \tau \Rightarrow v : \tau}\ \textsc{E.Chk}$$

By inversion of $\overset{?}{\approx}$ relation, we can deduce $r_2 = (v' : \tau_v')$ as τ' where $v \overset{?}{\approx} v'$, $\tau_v <_? \tau_v'$, and $\tau <_? \tau'$. By Lemma 21 we know that $v' \sim \tau'$. Therefore $r_2' = v' : \tau'$ and $r_1' \overset{?}{\approx} r_2'$.

□

Evaluation Incrementality of Expressions

If $e_1 \overset{?}{\approx} e_2$ and $\epsilon \vdash e_1 : \tau_1$ and $e_1 \Rightarrow e_1'$, then $e_2 \Rightarrow e_2'$ and $e_1' \overset{?}{\approx} e_2'$.

Proof. Either $e_1 = r_1$ or $e_1 = H_1[r_1]$ from Lemma 1. If the former, then theorem follows from Lemma 26. Otherwise, $e_2 = H_2[r_2]$ from Lemma 22, and $r_2 \Rightarrow v_2 : \tau_2$ from Lemma 26. From $\Rightarrow$ relation, therefore $H_2[r_2] \Rightarrow H_2[v_2 : \tau_2]$, and $H_1[v_1 : \tau_1] \overset{?}{\approx} H_2[v_2 : \tau_2]$ from Lemma 23. □

Chapter 13

Related Work

The design of the Stanza language and its underlying theory draws its inspiration from the numerous influential programming languages that have come before it as well as the vast existing literature in the field.

13.1 The Stanza Language

The surface syntax of Stanza was inspired heavily by Python [47], a language that emphasized readability and familiarity over conciseness. The most visibly-obvious trait in common is the use of indentation to denote code structure, and the use of a colon (`:`) to denote the start of a block. Despite being a controversial feature, we greatly appreciate the clean aesthetics that it lends to our code, and it is one of our cherished features of Stanza.

Just underneath the surface syntax, Stanza uses s-expressions to represent code, an essential trait that we adopted from the Lisp [32] family of languages. We have always considered Stanza to be a modern dialect of Lisp, even though the surface syntax of Stanza is distinctly unlike Lisp. The choice of using s-expressions has sweeping consequences and directly affect the design of Stanza's macro system, the core language constructs, and even the design philosophy. The decision to conceal the s-expressions beneath an appealing surface syntax was inspired by the Dylan [19, 50] programming language.

The programmatic s-expression-based macro system was almost entirely taken from Lisp [32] and Scheme [56]. Macros are expressed using arbitrary Stanza code to transform one s-expression into another. To help write the macros, Stanza provides a Lisp-like quasiquote operator, and an s-expression template engine inspired by Scheme's `syntax-rules` construct. For the issues of macro hygiene, we take the Common Lisp approach and provide `gensym` and package-qualified identifiers. The design philosophy of using macros to implement common language constructs so as to minimize the number of core forms is also borrowed from Lisp. Some syntax is borrowed from the Clojure [28] language, such as treating commas as whitespace, and using the `~` and `~@` symbols to represent `unquote` and `unquote-splice`. We were inspired by the emphasis that the REBOL [48] language placed on domain specific languages, but pursued it within the context of s-expression macros and intentionally did not allow any flexibility in the lexical structure of programs.

One important difference between Stanza's and Lisp's macro systems is the property that multiple Stanza s-expressions may map to a single core form, whereas Lisp macros transform only a single s-expression to a single core form. This key difference enables Stanza's support for natural syntax, but is also why Stanza requires an additional parsing stage during macroexpansion. The parser system is built upon Parsing Expression Grammars [23] (PEGs), and the syntax of macros are expressed in a variant of Backus-Naur Form [5] (BNF). The binding behaviour of nested ellipsis patterns (...) are inspired by Scheme's `syntax-rules` construct.

The overall organization of a Stanza project is influenced heavily by both Java [2] and Lisp. A project is divided into packages, as they are in Java, and packages are allowed to be cyclically dependent. The visibility modifiers that control whether declarations can be referenced by code outside of their package are inspired by the class member visibility modifiers of Java. Unlike Java [2], C [33], or C++ [55], Stanza top-level expressions are executable as they are in Lisp, and there is no distinguished `main` function.

Stanza's object system was inspired heavily by Common Lisp [32], Dylan [19,50], Smalltalk [24], Java [2], and C++ [55]. Both Common Lisp and Dylan championed building an object system on top of multimethods because of their flexibility and compatibility with a functional programming style. We chose multimethods for similar reasons, though Stanza's design differs in that its object system is built *solely* upon multimethods. The emphasis on encapsulation and interfaces is inherited from Smalltalk. Stanza's *"everything is a function call"* philosophy is analogous to Smalltalk's *"all you can do is send a message"* philosophy. In Stanza, all values have a type that is fixed upon construction and that can be tested dynamically. For example, the user can write code to test whether or not a value is of type `Duck` and respond appropriately. Languages have differing philosophies on whether this is a desirable feature. Java shares this property, but Ruby and Javascript, in contrast, strongly discourages this practice. Additionally, though Stanza's multimethod object system is most heavily inspired by the Common Lisp Object System (CLOS), it is the function overloading feature borrowed from Java and C++ that makes it a convenient system to use in practice.

Though Stanza's type system is fairly dissimilar to the Hindley-Milner [29] style type systems of OCaml [35] and Haskell [31], those languages had great influence on our design. During the design of our type system, we constantly compared Stanza code against OCaml and Haskell to evaluate the design tradeoffs – such as whether a feature was sufficiently expressive, or overly complex, or whether it can be generalized, etc. In the end, the type system is most similar to Java's [2] nominal subtyping system. The decision to treat parametric types as covariant was inspired by Java's treatment of arrays, which, contrary to popular opinion, we do *not* feel was a bad design choice. The captured type system was motivated by our negative experiences with Scala's [42] and Java's variance annotations. We decided that, for our target audience, the extra expressivity and safety afforded by variance annotations did not justify their added complexity, which led to our design of the simpler captured type system. The idea of capturing locations was inspired by the unification annotation in the StrongTalk [8] language. The key equations for Stanza's union and intersection types are taken directly from the textbook *Types and Programming Languages* [45].

Stanza's optional typing feature was inspired by Dylan [19, 50], which showed that a programming language can gain the productivity advantages of a static typechecker without losing the flexibility of dynamic-typing. The distinct feeling of starting from a flexible dynamically-typed code base and incrementally adding types to solidify the structure makes it another one of our cherished features. A number of recent languages – such as Typescript [26], Dart [6], and Typed Racquet [58], among others – also incorporate optional typing.

Stanza's targetable coroutines are inspired by Lua's [30] coroutines, Ruby's [22] and CLU's [37] generators, and Scheme's [56] continuations. We were inspired particularly by the elegance of Ruby's incorporation of generators into the overall language design. However, the decision to use Stanza's coroutines as a general-purpose control flow construct are a response to Ruby's overly-complex closure constructs and control flow operators. Scheme's continuations assured us that a general-purpose control flow construct exists, and we settled on Stanza's coroutine design as a balanced tradeoff between generality and simplicity.

For code local to a function, Stanza borrows language constructs from a variety of sources. The `$` operator is borrowed from Haskell [31]. Tuple destructuring was inspired by Clojure [28], OCaml [35], and Haskell [31]. From Scheme [56], we borrow the use of tail-recursion as the fundamental looping construct, the named let construct, and the emphasis on lists as a foundational datastructure. From Java [2], we borrow the exception handling construct, and we generalize the labeled loops construct to arbitrary code blocks. The philosophy of representing common control structures as higher-order functions was inspired by Ruby [22] and Smalltalk [24].

The design of LoStanza closely mirrors the design of the C [33] programming language. Beyond syntactic differences, the two languages are largely identical except for the ability to reference types and functions defined in HiStanza, the addition of the `ref` type, and the ability to provide arguments to the `goto` operator.

13.2 The Optional Type System

Our ? type was directly inspired by the type of the same name first introduced in [53]. Unlike that work, we chose to build our semantics on top of the subtyping relation, which offers us two advantages. First, the system automatically supports subtyping, which is a useful and familiar paradigm. And second, our system can be easily extended to incorporate other typing features that were also originally developed in a subtyping framework, including nominal subtyping and union types. [52] extends the base gradual typing system with support for subtyping by adding the subsumption rule, but it is unclear how to support union types. We also state and prove incrementality, and the operational semantics of our system is presented without a separate cast-insertion pass.

Thatte's Quasi-Static typing [57] was also developed upon a subtyping framework like ours, but *does* include a subsumption and implicit downcast rule, which together raises similar difficulties as the transitivity rule discussed in section 11.2. Our system avoids these difficulties, and hence does not require a separate plausibility checking phase, and is simpler to implement. In addition, unlike our system, Quasi-Static typing does not guarantee that a completely annotated program catches all type errors.

The early work on the Dynamic type by Abadi et al. [1] is related to this work but was developed with different goals. The main purpose of the Dynamic type was to model dynamic data within a statically-typed context, whereas our goal was to facilitate the transition from an untyped codebase to a typed codebase. Thus the Dynamic type and associated typecase construct does not have and was never intended to have any incrementality properties.

Our top-level consistency relation was inspired by Findler and Felleisen's work on Higher Order Contracts [20] which also checks values only up to the top-level. Findler precisely defines the semantics of calling functions with specified contracts from untrusted contexts. In these cases, the arguments are wrapped in a proxy object that dynamically checks contracts and fails if the contracts are violated. Wadler's Blame calculus [62] uses a similar strategy for enforcing type contracts. When a value of type Dynamic is interpreted as a statically-typed function, it is wrapped in a proxy object that dynamically checks the argument and return types up to the top-level. Wadler's Blame calculus has the additional advantage that when a contract has been violated, a precise location and blame is given to help identify the root cause of the error. However, the need for the creation of proxy objects results in difficult implementation and performance issues as discussed in [3]. Our semantics instead treats contracts as having lexical instead of dynamic extent, and consequentially, can be implemented without wrapper objects. It is true that we cannot assign a precise blame in the event that a contract is violated, but this limitation has not been an issue for us in practice. Instead the user receives an accurate line indicator indicating the first time when a value is not top-level consistent with its statically-inferred type. A recent paper by Vitousek et al. [60] shows that blame can be tracked to a limited extent even without wrapper objects, and their algorithm can potentially be adopted by Stanza to improve the quality of error messages.

The work on Typed Racket by Tobin-Hochstadt et al. [58] is similar and differs chiefly in the granularity to which typed and untyped code can be mixed. Typed Racket allows users to choose between dynamic or static-typing on a per-module basis, whereas our system is on a per-binder basis.

The optimization potential offered by our system is also offered by the Soft Typing system introduced in [11]. Static analysis is used to infer the most precise types possible, and the inferred types are then given as input to an optimizer. However, they do not allow for optional type annotations, and the analysis must necessarily be conservative. In contrast, the goal of our system is, first-and-foremost, to improve productivity through notifying the user of static type errors.

Flanagan's Hybrid Type Checking [21] can be used for similar purposes as our system. In that system, static types are combined with refinements expressed using arbitrary predicates.

These predicates are attempted to be satisfied using automated theorem proving, but runtime checks are inserted for when no definitive solution can be found. Our system, in contrast, provides a *predictable* static semantics that users can rely upon for their annotated binders.

There have been several dynamically-typed programming languages that allow for explicit type annotations. These include Dylan [19, 50], Common LISP [32], Cecil [12], Boo [43], BigLoo [49], Dart [6], and Strongtalk [8]. Many of the above systems use the type annotations as performance hints for the compiler and do not offer additional static-checking. Among the systems that provide additional static-checking, there are no guarantees of incrementality or safety of a fully-annotated program. Our framework is fully formalized and we prove incrementality and safety.

To our knowledge, we are the first to quantify the effect of type annotations on the effectiveness of the typechecker for a dataset comprised of reasonably-sized programs.

Chapter 14

Conclusion and Future Research

This dissertation presents the design and implementation of L.B. Stanza, a full-featured general-purpose programming language, and our experiences with developing in and teaching Stanza.

The language has five orthogonal subsystems that are each responsible for a separate facet of software development, but that together form a cohesive design: an optional type system, a targetable coroutine system, a multimethod object system, a programmatic macro system, and a systems sublanguage. This dissertation describes the core mechanisms underneath each subsystem, and most importantly, the interactions between each subsystem and the role they play within the overall philosophy of the language.

The optional type system bridges the divide between the dynamically-typed and statically-typed paradigms. By leaving out all type annotations, Stanza behaves as a dynamically-typed language where type errors are not caught during execution. But by incrementally adding additional type annotations, more and more type errors can be statically detected by the typechecker. Once fully annotated, Stanza behaves as a statically-typed language.

We formalized a subset of the type system and proved that it satisfies *incrementality*, which allows programmers to gradually add type annotations to increase the number of errors detected statically by the typechecker. The relationship between the number of type annotations and the probability of detecting an incorrect program is quantified experimentally on a dataset containing non-trivial programs.

The targetable coroutine system serves both as a powerful construct for expressing cooperative multitasking, and also as a foundational control flow operator. The semantics of the coroutines are well-behaved when coroutines are nested within each other, and thus allows them to be used as primitives in the creation of sophisticated frameworks that require custom control flow, such as animation, data-flow, and exception-handling frameworks.

The multimethod object system is a class-less object system that bridges the object-oriented programming (OOP) and functional programming (FP) paradigms. The system's flexibility comes from removing the restriction of having to contain behaviours within classes, which makes it possible to fluidly change the architecture of the program throughout development. By combining the multimethod system with function overloading, the flexibility of the system comes without having to sacrifice the convenience of a class-based object system. Multimethods are both more expressive and simpler than classes: constructors and fields are no longer necessary; method dispatch depends on the type of all arguments instead of only the receiver object, and there is no longer any distinction between function calls and method calls.

The programmatic macro system allows for programmers to write arbitrary code transformers using Stanza to extend the syntax of the base language. Similar to the Lisp macro system, the code transformers operate on and return s-expressions. We use a lightly decorated s-expression syntax as the surface syntax for Stanza and combine this with a parsing expression grammar (PEG) framework to allow for parsing of infix expressions, and expressions spanning more than a single s-expression. These extensions provide Stanza the full expressivity of a Lisp-like programmatic macro system but with a familiar syntax that is natural to C and Python programmers.

For typical users, Stanza is a high-level memory-safe language that prohibits low-level operations with the ability to crash the system. The LoStanza sublanguage is a small language within Stanza that exposes many unsafe operations that can crash the system if used incorrectly, but that allows programmers to directly manipulate memory, access raw hardware resources, interface with other language runtimes, and tune performance-sensitive code. Semantically, it is similar to the C language, but allows for easy communication and interoperation with high-level code.

At the University of California, Berkeley, Stanza has been used to write a number of successful practical projects, including:

1. FIRRTL: A digital hardware design language and compiler that lowers a high-level representation of register-transfer-level (RTL) circuitry to low-level Verilog.

2. Feeny: A minimal teaching language with just-in-time compiler.

3. A printed-circuit-board (PCB) design system that automatically generates manufacturable designs from declarative specifications of circuit boards.

4. The optimizing compiler for Stanza, which compiles Stanza source code to x86-64 assembly.

Stanza has been used to teach two graduate courses at U.C. Berkeley: a course on virtual machines and dynamic language runtimes, and a course on computational design.

From here, there are several research directions that may be investigated:

1. Type-Based Optimizations: The additional static information available in Stanza in the form of type annotations makes it possible to perform many type-based optimizations. However, these optimizations cannot be borrowed as is from existing literature as only a subset of binders may be annotated, and research remains on how to adapt the existing optimization literature on statically-typed languages to optionally-typed languages.

2. Coroutine Optimizations: Stanza's requirement for the target coroutine to be explicitly provided in the call to `suspend` makes it possible to analyze and inline many common usages of coroutines. As examples, iteration through generators can be re-expressed as simple loops, and calls to `break` on non-escaping coroutines can be re-expressed as a single `jump` instruction. Research remains on the analysis necessary for enabling these coroutine transformations.

3. Optimized Separate Compilation: Stanza's multimethod mechanism enables programmers to develop easily extensible software architectures, which comes at the price of complicating the implementation of a separate compiler. Stanza's current implementation is conservative and performs optimization transforms on a whole-program basis, after all packages in the program have been collected. Research remains on how to perform as much optimization as possible on a per-package basis. We expect that many possible optimizations can be speculated upon and eagerly performed and later invalidated if assumptions are broken.

Stanza is a young, but, we believe, a promising language for general software development. There is ongoing and active work on continuing to improve the language towards this purpose, such as on code quality optimizations, garbage collector optimizations, compiler optimizations, support for multithreading, additional language and type system features, and bindings to existing libraries. Ultimately, in the area of application programming, in the absence of hard real-time constraints, we don't foresee any fundamental technical limitations that would prevent Stanza from becoming the dominant language in this space.

Bibliography

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[3] Asumu, Daniel, Ben, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *43rd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 2016.

[4] Jonathan Bachrach, David Biancolin, Austin Buchan, Duncan Haldane, and Richard Lin. Jitpcb. In *Proceedings of the Intelligent Robots and Systems (IROS)*. IEEE, 2016.

[5] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language algol 60. *Commun. ACM*, 1963.

[6] Lars Bak and Kasper Lund. The Dart programming language. http://www.dartlang.org, 2011.

[7] Borland. The Delphi programming language. https://www.embarcadero.com/products/delphi, 1995.

[8] Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1993.

[9] Andrey Breslav. The Kotlin programming language. http://kotlinlang.org, 2011.

[10] Arthur W Burks, Don W Warren, and Jesse B Wright. An analysis of a logical machine using parenthesis-free notation. *Mathematical tables and other aids to computation*, 1954.

[11] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1991.

[12] Craig Chambers and the Cecil Group. The Cecil language: Specification and rationale. Technical report, University of Washington, Seattle, Department of Computer Science and Engineering, 2004.

[13] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, 1932.

[14] Cliff Click and John Rose. Fast subtype checking in the hotspot jvm. In *Proceedings of the Joint ACM-ISCOPE Conference on Java Grande (JGI)*. ACM, 2002.

[15] Robert Corbett. Gnu Bison. http://www.gnu.org/software/bison, 1988.

[16] Ole-Johan Dahl and Kristen Nygaard. Simula: An algol-based simulation language. *Commun. ACM*, 1966.

[17] The Rust Project Developers. The Rust programming language. http://www.rust-lang.org, 2010.

[18] Brendan Eich. The JavaScript programming language. https://developer.mozilla.org/bm/docs/Web/JavaScript, 1995.

[19] Neal Feinberg, Sonya E. Keene, Robert O. Matthews, and P. Tucker Withinton. *Dylan Programming: An Object-Oriented and Dynamic Language*. Addison Wesley Longman Publishing Co., 1997.

[20] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN Notices*. ACM, 2002.

[21] Cormac Flanagan. Hybrid type checking. In *33rd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 2006.

[22] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, 2008.

[23] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*. ACM, 2004.

[24] Adele Goldberg and Alan Kay. *Smalltalk-72: Instruction Manual*. 1976.

[25] Robert Griesemer, Rob Pike, and Ken Thompson. The Go programming language. http://golang.org, 2009.

[26] Anders Hejlsberg. Introducing TypeScript. *Microsoft Channel*, 2012.

[27] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *C# Programming Language*. Addison-Wesley Professional, 2010.

[28] Rich Hickey. The Clojure programming language. http://clojure.org, 2007.

[29] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 1969.

[30] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua&mdash;an extensible extension language. *Softw. Pract. Exper.*, 1996.

[31] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[32] Guy L. Steele Jr. An overview of COMMON LISP. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*. ACM, 1982.

[33] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 1988.

[34] Rasmus Lerdorf. The PHP programming language. http://php.net, 1995.

[35] X. Leroy and P. Weis. *Manuel de référence du langage Caml*. InterEditions, 1994.

[36] Patrick S. Li. The L.B. Stanza programming language. http://lbstanza.org, 2014.

[37] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. ACM, 1974.

[38] David Madore. The Yin-Yang puzzle. http://www.madore.org/ david/, 1999.

[39] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.

[40] Microsoft. The Visual Basic programming language. https://docs.microsoft.com/en-us/dotnet/visual-basic/, 1991.

[41] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[42] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, 2008.

[43] Rodrigo B. De Oliveira. The Boo programming language. http://boo.codehaus.org, 2005.

[44] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 1995.

[45] Benjamin C Pierce. *Types and Programming Languages*. MIT Press, 2002.

[46] GNU Project. The GNU compiler collection. http://gcc.gnu.org, 1987.

[47] Guido Rossum. Python reference manual. Technical report, 1995.

[48] Carl Sassenrath. *REBOL/Core Users Guide*. REBOL Technologies, 2005.

[49] Manuel Serrano. *Bigloo: a Practical Scheme Compiler*. Inria-Rocquencourt, 2002.

[50] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison Wesley Longman Publishing Co., 1996.

[51] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 2013.

[52] Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2007.

[53] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

[54] TIOBE Software. The tiobe index. https://www.tiobe.com/tiobe-index/, 2017.

[55] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[56] Gerald Jay Sussman and Guy L. Steele, Jr. Scheme: A interpreter for extended lambda calculus. *Higher Order Symbol. Comput.*, 1998.

[57] Satish Thatte. Quasi-static typing. In *17th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 1989.

[58] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*. ACM, 2011.

[59] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag New York, Inc., 1985.

[60] Michael M Vitousek, Cameron Swords, and Jeremy G Siek. Big types in little runtime: open-world soundness and collaborative blame for gradual type systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 2017.

[61] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 1995.

[62] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*. Springer, 2009.

[63] Larry Wall. The Perl programming language. http://www.perl.org, 1987.

[64] Niklaus Wirth. Good ideas, through the looking glass [computing history]. *Computer*, 2006.