Dynamix: A Domain-Specific Language for Dynamic Semantics

Master's Thesis

Thijs Molendijk

Dynamix: A Domain-Specific Language for Dynamic Semantics

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Thijs Molendijk
born in Ede, the Netherlands

TUDelft
Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

The source code for this thesis can be found online:
`https://github.com/metaborg/spoofax-pie`
`https://github.com/metaborgcube/metaborg-tiger`
`https://github.com/molenzwiebel/metaborg-chocopy`
`https://github.com/molenzwiebel/metaborg-ministratego`

Dynamix: A Domain-Specific Language for Dynamic Semantics

Author: Thijs Molendijk
Student id: 4730739
Email: `t.molendijk@student.tudelft.nl`

Abstract

The dynamic semantics of a programming language formally describe the runtime behavior of any given program. In this thesis, we present Dynamix, a meta-language for dynamic semantics. By writing a specification for a language in Dynamix, a compiler for the language can be derived automatically.

Dynamix specifications compile source programs to the Tim intermediate representation, a language-agnostic target IR designed to be able to be efficiently interpreted or compiled. Dynamix and Tim make use of the continuation-passing style to abstract over control flow, giving language developers fine-grained access to control flow primitives in their specification. A novel abstraction in Dynamix allows the construction of these CPS terms without the traditional friction involved. Dynamix is fully typed and integrated within the Spoofax language workbench. This allows language developers to interact directly with other parts of the workbench, including automatic type signature generation and the ability to query the results of static analysis.

Through case studies for miniStratego and ChocoPy with exceptions, we show that Dynamix is capable of succinctly representing a wide range of source language features and paradigms. Current performance is acceptable, with the foundations for a future efficient compiler for the Tim IR already in place.

Thesis Committee:

Chair:	Prof. dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft
Committee Member:	Prof. dr. A. Tolmach, Dept. of Computer Science, Portland State University

Thesis advisor:	Prof. dr. E. Visser, Faculty EEMCS, TU Delft

Preface

I did not expect to be here, 9 months later, without Eelco. He approached me back when I was but a simple compiler construction student, proclaiming that he had the perfect thesis idea for me. As a true salesman, he sold me the idea of working on Dynamix. Little did I know what I had gotten myself into.

My time working on Dynamix has been wonderful. Language development has always been a hobby, and Dynamix gave me the opportunity to design something that was truly new. Eelco was heavily invested in the project, and his spirit remains in Dynamix even after his untimely passing.

I am enormously grateful for the others that helped me throughout the project. In particular, Andrew Tolmach, whose feedback was already impeccable before Eelco's passing, and who continued his support without missing a beat even after Eelco's death, Casper Bach Poulsen, who gracefully stepped in as thesis advisor after Eelco's passing and helped guide me through the remainder of my thesis, and Ruben van Baarle, who was always present even though his own thesis has yet to start.

An additional thank you goes out to my parents and my sister, who continued supporting and encouraging me even though they had no clue what I was working on, despite my best efforts at explaining. A final thank you to William and Matěj, for taking the time out of your day to proofread my occasionally incoherent ramblings.

Let this be a warning to all future master's students. If someone approaches you with a "fun" thesis idea, don't be tempted. Before you know it, you'll be spending all your time on the formal definition of not one, but *two* programming languages.

Thijs Molendijk
Delft, the Netherlands
August 24, 2022

Contents

Chapter 1

Introduction

While the advent of computing has brought with it a large swathe of programming languages, wildly varying in levels of abstraction, capabilities, and use cases, we can reduce each of them down to just three components acting in unison to form a ”programming language”: the grammar, the static semantics, and the dynamic semantics.

The grammar governs how the programmer writes code. It specifies what lexical structures the language expects, what keywords, operators, literals, and expressions it supports, and how the language converts this textual representation to a format more suitable to work on, the abstract syntax tree (AST).

The static semantics govern what programs are *statically valid*. It encompasses all rules that can be checked at compile time, such as the validity of name binding within the program and whether it violates any typing rules. Some languages thrive around (strict) static semantics, whereas other languages, particularly dynamic scripting languages, may not have any at all.

Finally, the dynamic semantics of a language describe *what* a program produces. It formalizes that an addition of two numbers will indeed lead to a single number that is the sum of the two. At the same time, it describes what might happen to the result should the addition of the numbers overflow.

It is the collection of these properties that forms a programming language, not a specific implementation. Anyone can write a compiler for the C++98 programming language, as long as they adhere to the formal specification of C++98 as written in ISO/IEC 14882:1998 [26]. Someone wanting to implement a new JavaScript engine needs only to follow ECMA-262 [18]. Any virtual machine compliant with the Java Language Specification [23] is able to run any and all valid Java programs.

That's the theory, at least. In practice, bugs in implementations, ambiguities in the natural language used in the specification, as well as choices left up to the implementation mean that even trivial C++ programs that run flawlessly when compiled by GCC [21] may behave entirely different when compiled by LLVM's Clang [14]. Which one of the observed behaviors is the *correct* one can only be judged by consulting the specification.

Many other programming languages, even extremely popular ones, do not have a specification. This is completely understandable. After all, it is a herculean task to formally describe every construct and behavior in one's programming language. For these languages, the reference implementation of the language *is* the specification. Creating an alternative implementation of these programming languages amounts to carefully observing and replicating

$$\frac{E \vdash e_1 \Downarrow i_1 \qquad E \vdash e_2 \Downarrow i_2 \qquad v = i_1 +_{\text{signed 32-bit}} i_2}{E \vdash e_1 \texttt{ + } e_2 \Downarrow v}$$

```
rules
  compileExpr(Add(left, right)) = {
    lv <- compileExpr(left)
    rv <- compileExpr(right)
    #i32-add(lv, rv)
  }
```

Figure 1.1: A formal dynamic specification for the behavior of integer addition (top), and an equivalent Dynamix specification expressing the same behavior (bottom).

what the reference compiler does, and ensuring that this behavior stays consistent with the reference as both implementations continue to improve and evolve.

In this thesis, we introduce a new domain-specific language (DSL) called Dynamix for use in the Spoofax language workbench [57]. Dynamix aims to unify the specification and implementation of the dynamic semantics of a programming language, by offering a language that is high-level enough to function as a formal specification of the language, while at the same time allowing this specification to be turned into an efficient compiler for the language.

An example of Dynamix in action can be seen in Figure 1.1. It describes that the addition operator should be implemented by evaluating the values of each sub-term in left-to-right order, then yielding the 32-bit signed addition result of the two values. The Dynamix source describes exactly the same behavior as the formal definition above it, while additionally being capable of turning this specification directly into a compiler for the language.

Dynamix is designed to be used within the Spoofax language workbench [57], an environment designed for the design and implementation of (domain-specific) programming languages. Dynamix's integration within Spoofax offers many benefits, including the ability to interact with the results of static analysis, the ability to type check specifications using the algebraic signature of the source language, and the ability to invoke Dynamix as part of a language implementation test-suite. Dynamix requires no specific language design, allowing existing Spoofax workbench projects to seamlessly start using the meta-language.

This thesis makes the following contributions:

- We informally describe Dynamix, a new meta-language used for defining the dynamic semantics of a programming language, through numerous examples (Chapters 6 and 8).
- We introduce and formalize Tim, a language-agnostic intermediate representation for programs and the target language for Dynamix (Chapter 5).
- We formalize a limited subset of Dynamix, called Dynamix Core, defining the formal static and dynamic semantics of the language (Chapter 7).
- We provide a complete implementation of the Dynamix interpreter, as well as a baseline interpreter for the Tim IR. These implementations are fully integrated in the Spoofax language workbench, including signature generation and interoperation with other meta-languages (Chapters 5 and 8).

- We present two case studies that evaluate the performance and capabilities of Dynamix by implementing Dynamix specifications for the ChocoPy programming language [48] extended with exception handling, and a subset of the Stratego [67] programming language (Chapter 9).

The remainder of this document is structured as follows:

- We discuss the background around the subject of programming language specifications, the Spoofax language workbench, and previous iterations of dynamic specification meta-languages within this workbench (Chapter 2).
- We establish the main objectives of the Dynamix project, so that they may guide our design process in the remainder of the document (Chapter 3). Based on the established objectives, we discuss how one might design a DSL for runtime semantics, and what properties such a DSL should have (Chapter 4).
- We introduce and subsequently formalize the Tim intermediate representation, which functions as the compilation target for the Dynamix meta-language (Chapter 5).
- We informally introduce the Dynamix meta-language, by discussing a specification for a simple language and describing the abstractions it uses to simplify the creation of specifications (Chapter 6).
- We formalize a subset of the Dynamix language, discussing the formal semantics of the most interesting parts of the meta-language (Chapter 7).
- We discuss how the Dynamix meta-language is integrated within the Spoofax language workbench and how it allows for tight interoperation with other parts of the workbench (Chapter 8).
- We evaluate the performance and capabilities of Dynamix by discussing two different case studies implementing a specification for ChocoPy with exceptions [48] and a subset of Stratego [67] respectively (Chapter 9).
- We discuss how Dynamix performs compared to alternative implementations and competing tools for the specification of runtime semantics (Chapter 10). We continue by discussing what future steps might be taken to improve the Dynamix meta-language and the Tim intermediate representation (Chapter 11).

Chapter 2

Background

In this chapter, we will first explore some background related to both formal (dynamic) language specifications, as well as the Spoofax language workbench. Both are essential parts of the Dynamix language and some understanding of their design, conventions, and use will greatly help us contextualize the requirements, goals, and results outlined in the remainder of this thesis.

Section 2.1 elaborates on the current state of (formal) language specifications, discussing the various styles of language specifications, the goals of writing such a specification, as well as how these specifications relate back to their concrete language implementation(s).

Section 2.2 will provide a brief introduction to the Spoofax language workbench for readers unfamiliar with it. It discusses the ”programming language design pipeline” of which Dynamix will become a part. As part of this, we will briefly introduce the other meta-languages in the workbench, and specifically the features relevant for the Dynamix language. Readers already familiar with the Spoofax language workbench may want to skip this section.

Finally, Section 2.3 specifically discusses the history of dynamic specifications in the Spoofax workbench. Dynamix is hardly the first foray into this domain, so we ought to learn from our predecessors.

2.1 Programming language specifications

At its core, a language specification is some form of documentation that outlines the behavior of a language, whose contents are agreed upon by both the implementors and the users of the language. A language specification is the ”single source of truth” for these languages: if the real-world behavior of the language differs from the specification, the language implementation is divergent from the specification and hence incorrect[1]. Through this, a specification allows one to unambiguously decide what the *meaning* of any program is.

Specifications for a language exist for various reasons. From a mathematical perspective, having a specification is simply the *right thing to do*. After all, how could one possibly trust a language that does not have a (proven) formal definition? Others may create a specification as a form of direct user documentation, such that a user does not have to consult the language implementation to find out the exact behavior of a certain language feature. Similarly,

[1]In practice, humans are not perfect. Specifications often contain errors or oversights, or the language designers may consider a previously formally defined behavior to be unwanted. It is often more appropriate to say that, especially in languages with only a single implementation, the specification and the implementation evolve hand-in-hand.

a language specification might be created to help unify several different (semi-)incompatible language implementations under a common semantics, or as an attempt to promote alternative implementations.

Specifications come in several forms. Most commonly seen are explicit definitions: documents outlining the grammar, static, and dynamic semantics of the language using either natural language or formal semantics. Not infrequently, these documents are written by a committee and part of a standardization process. Examples include ISO/IEC 9899:1990 [29] for the C programming language, ISO/IEC 14882:1998 [26] for the C++ programming language, and ECMA-262 [18] for the JavaScript programming language. Others may exist as publications, such as the Java Language Specification [23] or ChocoPy language specification [49].

These explicit definitions are generally written in either natural language or formal notation. When formal notation is used, it is often through one of several mathematical frameworks designed for such semantics, such as the big-step notation (also known as natural semantics) popularized by Gilles Kahn [31]. Examples of language specifications that use formal notation are the definition of Standard ML by Milner et al. [44] and the ChocoPy language specification [49] (we discuss the ChocoPy specification in more detail in Chapters 4 and 9). The benefit of using a formal notation is that it prevents the ambiguities that are inherent in natural language. By using a formal notation where every operation has a clearly defined meaning, it should[2] be unambiguous exactly how each language feature is defined. Going beyond just specifying the language, it is also possible to use formal specifications to build a mechanized proof[3] that a certain compiler or runtime performs exactly as stated by the programming language. One such example is the CompCert compiler [11], which implements a formally verified compiler for ISO C99 and ANSI C.

For specifications written in natural language, authors must be careful to be explicit in their intentions to avoid the ambiguities inherent in natural language. An example of a formal definition written in natural language can be seen in Figure 2.1. Observe that one must be careful to list exactly all of the possible behaviors, even if some of those behaviors are invalid or undesirable. However, a specification in natural language is often easier to both read and write, causing it to remain a popular choice for formal specifications. We further discuss both formal notation, natural notation, and the differences between the two in Chapter 4.

Despite explicit definition documents being the most common, they are not the only form of language specification. Two other common forms are that of a reference implementation, in which a single implementation of a language is designated to be the ”correct” implementation from which the behavior should be derived, as well as that of a designated test suite for the language, which defines behavior in terms of examples and their expected outcome. Many languages also combine these: Test262[4] is an official test suite for conformance against the ECMAScript specification [18].

[2]Even with formal notation, it is still possible for ambiguity to slip in. Authors must be careful to ensure that only a single rule applies at a time, that rules do not contradict each other, and that every possible situation has been accounted for.

[3]A proof that can be checked by a machine. Mechanized proofs are often written using ”proof assistants”, such as Coq [62] or Agda [1].

[4]`https://github.com/tc39/test262`

Bitwise shift operators

Syntax

shift-expression
  additive-expression
  shift-expression `<<` *additive-expression*
  shift-expression `>>` *additive-expression*

Constraints

Each of the operands shall have integral type.

Semantics

The integral promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width in bits of the promoted left operand, the behavior is undefined.

The result of **E1** « **R2** is **E1** left-shifted **E2** bit positions; vacated bits are filled with zeros. If **E1** has an unsigned type, the value of the result is **E1** multiplied by the quantity 2 raised to the power **E2**, reduced modulo `ULONG_MAX+1` if **E1** has type `unsigned long`, `UINT_MAX+1` otherwise. (The constants `ULONG_MAX` and `UINT_MAX` are defined in the header `<limits.h>`.)

[...]

Figure 2.1: An example of a language specification written in natural language. This describes the syntax and behavior of the bitwise-shift operator in the C programming language. Parts are omitted for the sake of brevity. Adapted from the ANSI C definition, ISO/IEC 9899:1990 [29].

Official language specifications are common in programming languages, but not ubiquitous. This can be seen even in some of the most popular programming languages: an informal survey of some of the most popular programming languages currently in use yields the following list[5]:

- **JavaScript**: Formalized in ECMA-262 [18]. Defines dynamic semantics using natural language.
- **HTML/CSS**: Formalized by the W3C across several different specifications, including HTML 5.3 [45] and CSS2 [40]. Dynamic semantics not applicable.
- **Python**: No official specification.
- **SQL**: First formalized as ISO/IEC 9075:1992 [28]. Several (incompatible) dialects exist. Dynamic semantics not applicable.
- **Java**: Formalized per language version, most recently as "The Java Language Specification, Java SE 18 Edition" [22] and "The Java Virtual Machine Specification, Java SE 18

[5]These languages have been chosen according to their popularity as indicated on the StackOverflow developer survey 2021 [58], a survey on programming languages and technologies taken by more than 80,000 developers.

Edition" [41] for the Java language and virtual machine, respectively. Defines dynamic semantics using natural language.

- **TypeScript**: No official specification.
- **C#**: First formalized as ISO/IEC 23270:2003 [27]. Defines dynamic semantics using natural language.
- **Bash/Shell**: Base feature set formalized as part of the POSIX standards family in IEEE 1003.1-2008 [25]. Defines dynamic semantics using natural language.
- **C++**: First formalized as ISO/IEC 14882:1998 [26]. Defines dynamic semantics using natural language.
- **PHP**: Efforts ongoing to write a specification[6]. Current efforts define dynamic semantics using natural language.

2.2 The Spoofax language workbench

The Spoofax language workbench [57] is a software suite that combines various meta-languages[7] and tools to allow users to easily design and implement (domain-specific) programming languages. Beyond parsing, static analysis, and execution, the language workbench is capable of automatically generating various IDE features including syntax highlighting, inline error markers, go-to-definition, hover annotations, and more. A standalone plugin for the Eclipse IDE providing these features can be automatically generated from a Spoofax language project.

The most recent stable version of Spoofax, as of the time of writing, is Spoofax 2 [34]. However, work has been ongoing on Spoofax 3, which retains the same meta-languages and pipeline setup, but integrates this pipeline using PIE [37], a framework for incrementalizing build tasks. Using PIE, various aspects of Spoofax 3 gain support for incremental compilation "for free", leading to considerable performance gains in both the language development process, as well as the performance of the artifacts produced by the workbench. All projects discussed in this paper, particularly Tim and Dynamix, have been integrated directly into Spoofax 3.

The Spoofax language workbench consists of a family of meta-languages that each describe an aspect of a programming language. These meta-languages are often declarative and inspired by academic work, such that it is easy for users to get started if they are familiar with the programming languages academic field. Creating a new language using the Spoofax language workbench amounts to defining each section of the "compiler pipeline" using the appropriate meta-language. Figure 2.2 shows the current compiler pipeline for use in Spoofax. We briefly discuss each step in more detail.

[6]`https://github.com/php/php-langspec`

[7]A programming language that describes some aspect of another programming language.

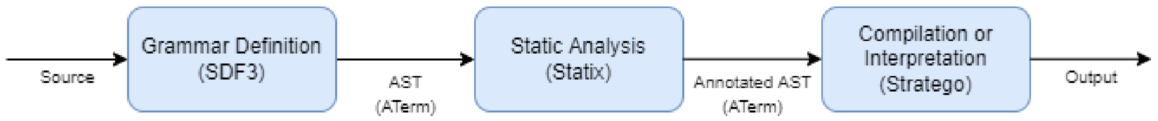


Figure 2.2: The Spoofax language workbench pipeline. Source code is parsed using SDF3, analyzed using Statix, then compiled or executed using Stratego.

```
1   lexical sorts INT ID
2   lexical syntax
3     INT = [1-9] [0-9]*
4     ID = [A-Za-z] [A-Za-z0-9]*
5     LAYOUT = [\ \n\r\t]
6
7   context-free sorts Exp
8   context-free syntax
9     Exp = <(<Exp>)> {bracket}
10    Exp.Add = <<Exp> + <Exp>> {left}
11    Exp.Sub = <<Exp> - <Exp>> {left}
12    Exp.Mul = <<Exp> * <Exp>> {left}
13    Exp.Div = <<Exp> / <Exp>> {left}
14
15    Exp.Var = <<ID>>
16    Exp.Int = <<INT>>
17
18    Exp.Let = <let <ID> = <Exp> in <Exp>>
19
20  context-free priorities
21    {left: Exp.Mul Exp.Div} > {left: Exp.Add Exp.Sub} > {Exp.Let}
```

Figure 2.3: A simple SDF3 grammar for an arithmetic language that supports variable bindings.

2.2.1 Syntax definition and parsing

The natural first part of a language workbench is the ability to declare a grammar for the language. For this purpose, Spoofax offers the Syntax Definition Formalism 3 (SDF3) meta-language [55]. Using SDF3, a user can declare both lexical and context-free syntax productions for their language. From this declaration, SDF3 generates a scannerless parser with support for error recovery, a pretty-printer, and a syntax highlighter [56].

Figure 2.3 shows an example of an SDF3 definition for a simple arithmetic language. Two lexical sorts are defined, `INT` and `ID`, which represent an integer literal and variable identifier respectively. A single context-free sort is defined, `Exp`, which represents an arbitrary expression. The distinction between lexical and context-free sorts determines is similar to that between grammar productions and lexical tokens in other parsers, and mainly determines the way in which the resulting value is represented in the AST (a lexical production yields a string of its contents, whereas a context-free production produces a tuple containing its subterms).

SDF3 has various features that simplify the creation of a grammar. The `LAYOUT` syntax production on line 5, for example, indicates which layout characters (e.g. whitespace) are allowed between nonterminals in context-free productions. Similarly, the `context-free priorities` section allows for disambiguation between productions with different precedence levels without resorting to unique sorts per precedence level. By marking a production with `{bracket}` (line 9), we can indicate to the generated pretty-printer that brackets should only be included if they are necessary to preserve the meaning of the AST.

The parser generated by SDF3 outputs an AST in the annotated term format (ATerm). This

```
let x = 10 in
  let y = 20 in
    let z = x + y in
      (10 + x) * y / z
```

```
Let(
  "x",
  Int("10"),
  Let(
    "y",
    Int("20"),
    Let(
      "z",
      Add(Var("x"), Var("y")),
      Div(
        Mul(Add(Int("10"), Var("x")), Var("y")),
        Var("z")
      )
    )
  )
)
```

Figure 2.4: An example expression in the arithmetic language declared in Figure 2.3[8]. The parsed AST in ATerm format as produced by the generated parser can be seen on the right.

term	::=	*int literal*	*Integer literal*
	|	*string literal*	*String literal*
	|	*identifier*`(`*terms*`)`	*Constructor*
	|	`[`*terms*`]`	*List literal*
	|	`(`*terms*`)`	*Tuple literal*
	|	*term*`{`*terms*`}`	*Annotated term*
terms	::=	ϵ	
	|	*term*	
	|	*term*`,` *terms*	
int literal	::=	`'-'? [1-9] [0-9]*`	
string literal	::=	`"` *string char** `"`	
identifier	::=	`[A-Za-z] [A-Za-z0-9._-]*`	

Grammar 2.1: The grammar for the ATerm data representation format.

format originated in the ASF+SDF formalism [17], a precursor to the Spoofax language workbench. It is a simple data transfer format that supports strings, integers, lists, tuples, tagged tuples (constructors), and annotations. The grammar for the ATerm format can be seen in Grammar 2.1. ATerms are used pervasively across the Spoofax language workbench and are supported as data format in all Spoofax meta-languages. Figure 2.4 shows an example of the parsing output for a simple program using the grammar defined in Figure 2.3.

Spoofax will automatically derive an *algebraic signature* from all SDF3 declarations. Such a signature dictates the exact structure that an ATerm expression can take such that it is a valid AST[9]. These signatures are used by various other meta-languages in the Spoofax language workbench to provide better static analysis features. Figure 2.5 shows the signature for the

[8]The syntax highlighting in this snippet is exactly the syntax highlighting that the Spoofax language workbench automatically generates from the given grammar definition.

[9]Here, valid AST means that it is an AST that can be produced by the parser (i.e. it has a lexical equivalent), and not necessarily that this AST is valid according to the static and dynamic semantics of the language.

```
signature
  sorts
    INT = string
    ID = string
    Exp

  constructors
    Add : Exp * Exp -> Exp
    Sub : Exp * Exp -> Exp
    Mul : Exp * Exp -> Exp
    Div : Exp * Exp -> Exp
    Var : ID -> Exp
    Int : INT -> Exp
    Let : ID * Exp * Exp -> Exp
```

$$S = \{\texttt{INT}, \texttt{ID}, \texttt{Exp}\}$$

$$\begin{aligned}\Sigma = \{\ & \texttt{Add} : \texttt{Exp} \times \texttt{Exp} \to \texttt{Exp}, \\ & \texttt{Sub} : \texttt{Exp} \times \texttt{Exp} \to \texttt{Exp}, \\ & \texttt{Mul} : \texttt{Exp} \times \texttt{Exp} \to \texttt{Exp}, \\ & \texttt{Div} : \texttt{Exp} \times \texttt{Exp} \to \texttt{Exp}, \\ & \texttt{Var} : \texttt{ID} \to \texttt{Exp}, \\ & \texttt{Int} : \texttt{INT} \to \texttt{Exp}, \\ & \texttt{Let} : \texttt{ID} \times \texttt{Exp} \times \texttt{Exp} \to \texttt{Exp}\ \}\end{aligned}$$

Figure 2.5: The algebraic signature for the grammar in Figure 2.3. Left shows the signature of the grammar defined in the Statix meta-language, right shows the grammar in equivalent formal term algebra notation. The left source is automatically derived from the grammar definition by SDF3 as part of the compilation process.

example grammar in both formal and Spoofax syntax formats.
Other features that SDF3 offers include the ability to specify layout constraints for grammar productions (often crucial for languages that have significant whitespace such as Python and Haskell) and automatic error recovery and placeholder insertion (needed for robust autocomplete). For more information on SDF3 and parsing within the Spoofax language workbench, the reader is invited to consult the relevant academic work, particularly the work by Visser et al. across various papers [33, 57, 55, 56, 32, 69].

2.2.2 Static analysis with constraint solvers

The current meta-language used for static analysis in the Spoofax workbench is Statix [5]. Statix is a declarative language that performs static analysis through the use of constraints. A Statix specification consists of a set of declarative rules that specify constraints that should hold for a program to be valid. If the Statix logic solver can find a solution for which all constraints hold for a given input AST, a program is considered well-typed. Name binding correctness is asserted through the use of *scope graphs* [4], although their exact semantics are beyond the scope of this paper. The reader is referred to the work by van Antwerpen et al. [4, 5, 3] should they be interested in their exact workings.

Figure 2.6 shows an example typing rule in both formal notation and Statix syntax. The `typeOfExpression` rule and $\Gamma \vdash v : \texttt{T}$ notation are equivalent, in that they both describe the relationship between an expression and its type. Within the *head* of the Statix rule, the `Div(a, b)` node is pattern-matching on exactly the ATerm that is output by the parser. Indeed, all values in Statix (including the `INT()` term) are ATerms. Using the signatures generated by SDF3 (see Section 2.2.1, Figure 2.5), the Statix meta-language can statically ensure that these pattern-matching operations are valid.

Beyond simply indicating whether some input AST is valid according to the Statix specification, the Statix solver also allows users to attach *properties* to arbitrary AST nodes. These properties are arbitrary ATerm values that can later be read from outside Statix. Common properties used in Statix specifications are the `ref` and `type` properties, which represent the declaring node and the type of a node respectively. The Spoofax language workbench has

```
rules
  typeOfExpression(s, Div(a, b)) = INT() :-
    typeOfExpression(s, a) == INT(),
    typeOfExpression(s, b) == INT().
```

$$\frac{\Gamma \vdash a : \texttt{INT} \qquad \Gamma \vdash b : \texttt{INT}}{\Gamma \vdash a \,/\, b : \texttt{INT}} \text{ T-Division}$$

Figure 2.6: An example typing rule for integer division in both Statix notation and formal notation. Here s and Γ are analogous and correspond to the context in which the expression appears. Other rules, such as the type of an integer literal, are omitted.

```
rules
  typeOfExpression(s, n@Div(a, b)) = INT() :-
    typeOfExpression(s, a) == INT(),
    typeOfExpression(s, b) == INT(),
    @n.type := INT().
```

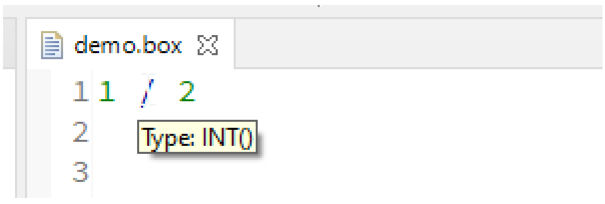

```
demo.box
1 1 / 2
2      Type: INT()
3
```

Figure 2.7: An example showing the ability to attach properties to nodes in Statix. The highlighted line (left) sets the type property on the AST node representing the division expression. This value is later read by Spoofax to provide editor services such as hover information (right).

built-in support for reading the values of these two properties to provide editor services such as hover hints and go-to-definition. An example of the type annotation, as well as the tooltip generated as a result, can be seen in Figure 2.7. It is worth noting that both the type and ref properties are simply conventions. A user can assign arbitrary values to a property, and can read these values from other meta-languages within the Spoofax language workbench.

2.2.3 Transformations with Stratego

Finally, we will briefly discuss the Stratego transformation language [67]. Unlike SDF3 or Statix, Stratego is more of a general imperative programming language centered around the concept of transformations, instead of a (hyper-specialized) meta-language. As a result, it generally functions as a general purpose language that, while traditionally used for interpretation or compilation at the end of the Spoofax pipeline, may also appear between other stages to perform intermediate transformations.

An example of a simple Stratego program can be seen in Figure 2.8. Pattern-matching is performed on input terms, with matches transformed accordingly. The data format in use for Stratego is again the ATerm format, with additional static analysis provided using the algebraic signature automatically derived from the SDF3 grammar.

Due to its general nature, many Spoofax features and meta-languages offer a Stratego API. This allows Stratego strategies to interact with other parts of the Spoofax workbench, such as the ability to query the results of static analysis. Consider the example in Figure 2.9, which uses Statix APIs to distinguish between integer addition and string concatenation, both of which are syntactically represented as Add(a, b).

We will further discuss Stratego and its semantics when we introduce the mini-Stratego case study in Chapter 9. Readers interested in learning more about Stratego and its term rewriting paradigm are encouraged to read the appropriate work by Visser et al. [32, 67].

```
rules
  fold = innermost(fold-term)

  fold-term: Add(Int(a), Int(b)) -> Int(<addS> (a, b))
  fold-term: Sub(Int(a), Int(b)) -> Int(<subtS> (a, b))
  fold-term: Mul(Int(a), Int(b)) -> Int(<mulS> (a, b))
  fold-term: Div(Int(a), Int(b)) -> Int(<divS> (a, b))

  fold-term: Eq(Int(a), Int(a)) -> Int("1")
  fold-term: Eq(Int(a), Int(b)) -> Int("0") where not(<eq> (a, b))
```

Figure 2.8: A simple Stratego program that performs constant folding in an arithmetic language. The `fold` strategy will apply the `fold-term` strategy until a fixpoint is reached.

```
imports
  statix/api

rules
  compile-expr: n@Add(a, b) -> <compile-int-addition> (a, b)
    where a := <stx-get-ast-analysis> n; INT() := <stx-get-ast-type(|a)> n

  compile-expr: n@Add(a, b) -> <compile-string-concatenation> (a, b)
    where a := <stx-get-ast-analysis> n; STRING() := <stx-get-ast-type(|a)> n
```

Figure 2.9: Many Spoofax meta-languages expose Stratego APIs. Here, the result of a Statix analysis is used to emit integer addition or string concatenation based on the type of the expression.

2.3 Dynamic semantics in Spoofax

As already briefly alluded to, Dynamix is not the first attempt at adding dynamic semantics support to the Spoofax language workbench. In fact, several projects, including some under the Dynamix name, predate the work done in this thesis.

The first effort at implementing dynamic semantics in the Spoofax language workbench was the DynSem meta language [64]. DynSem was conceived with goals similar to Dynamix, but was intentionally designed to mimic the notation often used by big-step operational semantics. The DynSem compiler converts such a specification into an interpreter for the language written in Java. An example of a DynSem specification can be seen in Figure 2.10. We further discuss DynSem as part of the related work section in Chapter 10.

One of the main problems with DynSem was that its use of big-step rules with explicit control flow meant that certain language features, particularly non-linear control flow such as exceptions and generators, require non-trivial boilerplate and state threading in virtually every rule, including those not concerned with the feature. This meant that as these features were added, the complexity of the rules increased to a degree where rules were no longer ergonomic to write. Additionally, the interpreter generated by DynSem was not particularly performant. While this performance was greatly increased in follow-up work by Vergu et al. [66, 65], they note that the generated interpreter still performed a factor of 4 slower than a hand-written interpreter for the same language. As a result of these two main drawbacks,

```
rules
  Lit(s) --> NumV(parseI(s)).

  Plus(e1, e2) --> NumV(addI(i1, i2))
  where
    e1 --> NumV(i1);
    e2 --> NumV(i2).

  Ifz(NumV(ci), e1, e2) --> v
  where
    case ci of {
      0 => e1 --> v
      otherwise => e2 --> v
    }.
```

Figure 2.10: An example of a specification written using the DynSem meta-language [64]. The `-->` ”arrow” is intended to be analogous to a Kahn-style big-step evaluation relation.

the DynSem language saw little use within the Spoofax workbench and was eventually deprecated.

To remedy these issues, the Dynamix meta-language[10] was conceived as part of Chiel Bruin’s master’s thesis [13]. Inspired by DynSem, the Dynamix language was designed as a dynamic specification meta-language compatible with the semantics of the ”Roger” target language, introduced in the same thesis. Figure 2.11 shows an example snippet in Bruin’s Dynamix meta-language.

Bruin’s Dynamix targeted the FrameVM, a virtual machine designed around the concepts of scopes-as-frames [52] and control frames. Control frames were designed as an abstraction over a direct continuation-passing style (CPS)-based language: Bruin argues that pure CPS needlessly increases bookkeeping and complexity by enforcing explicit control flow even in situations where implicit control flow would suffice. Bruin’s Dynamix compiled to Roger, an instruction set that runs natively on the FrameVM. We discuss the FrameVM in more detail as part of the section on related work in Chapter 10.

While Bruin’s work showed that an approach for dynamic specifications using continuations (or abstractions around them) was capable of producing specifications for even complex constructs like Scheme’s `call/cc`, it was not always practical to use. The introduction of control frames meant that the language was hard to pick up, and the Dynamix language was perhaps too low-level. Performance, while not a consideration, was also a large blocker: programs were regularly several orders of magnitude slower than a hand-written implementation. Ultimately, these issues meant that Bruin’s Dynamix and the FrameVM were never integrated in the Spoofax language workbench.

Despite this, using continuations as a tool to model dynamic semantics seemed promising. In late 2020, Eelco Visser and Andrew Tolmach started exploratory work on ”Dynamix V2”. The intent was to retain most of the ideas behind the Dynamix language, but to compile to a new continuation-based target language. If the resulting target language was able to be effi-

[10]Not to be confused with the Dynamix language presented in this paper. Except for this section, the term Dynamix will always refer to the language presented in this paper.

```
rules
  eval-exp(Int(v)) = return(int(v))

  eval-exp(Plus(left, right)) =
    v1 <- eval-exp(left);
    v2 <- eval-exp(right);
    return(iadd(v1, v2))

  eval-exp(Var(name)) =
    path = resolve(name, "Var");
    return(get(cur(), path))

  eval-exp(Ifz(cond, then, else)); k(v1) =
    !v1;
    c <- eval-exp(cond);
    jumpz(c, else_b, then_b);

    then_b = <
      ~v1 <- eval-exp(~then);
      jump(~k)
    >;
    else_b = <
      ~v1 <- eval-exp(~else);
      jump(~k)
    >
```

Figure 2.11: An example program in the Dynamix meta-language designed by Chiel Bruin. Adapted from Figures 6.5 and 6.7 from Bruin's thesis [13].

ciently compiled or interpreted, this approach could resolve the runtime performance issues that both DynSem and the FrameVM suffered from. DynSem's other main problem, that of control flow, could also be resolved by using continuations, as evident from Bruin's work. At the same time, this was a good opportunity to explore potential approaches for DSLs to simplify the process of building CPS terms.

The Dynamix language discussed in this thesis[11] is a spiritual successor to these efforts by Visser and Tolmach. Many components were revamped in order to generalize their experimental prototype to a large array of languages and programming paradigms, and to fully integrate the resulting product in the Spoofax language workbench. The remainder of this thesis discusses these efforts and shows that the resulting meta-language is indeed capable of resolving many of DynSem's original issues. We should stress that no previous knowledge of the Dynamix meta-language as it appeared in Bruin's thesis is required: we will introduce the language from scratch.

[11]Dynamix, as presented in this paper, should really be called Dynamix v2.5. However, since the version of Dynamix designed by Bruin never made it into the Spoofax language workbench, the Dynamix presented in this paper is the first publicly usable iteration of the meta-language. To avoid potential confusion for Spoofax users, we decided that it would be more appropriate to call the language Dynamix, without a version number.

Chapter 3

Objectives

Now that we have discussed all the necessary background, let us elaborate on the objectives of the Dynamix project as introduced in Chapter 1. The remainder of this thesis will use these objectives as pillars to guide the design and implementation of the Dynamix project.

3.1 Overarching goal

The primary goal of the Dynamix project is to **provide a widely applicable and performant meta-language for the specification of dynamic semantics within the Spoofax language workbench**. Dynamix must offer all the infrastructure required to implement dynamic specifications within Spoofax, with a level of support, stability, and integration that is to be expected from a first-class Spoofax meta-language.

As a guiding principle, **Dynamix should strive to be as declarative as possible, so as to retain its correspondence with the domain of formal programming language specifications**. Where possible, Dynamix should operate at an abstraction level that allows it to be treated as a formal specification, with the additional benefit that a language implementation can be directly derived from this specification. As a consequence of this requirement, the Dynamix meta-language should be largely declaratively oriented and offer an abstraction level comparable to those used in formal specifications. Only in cases where this abstraction level may hinder the usability or applicability of the Dynamix meta-language should other, possibly imperative, language features be considered.

3.2 Concrete requirements

Based on the aforementioned overarching goals, we now discuss several concrete requirements for both the Dynamix meta-language, as well as its implementation within the Spoofax language workbench.

Simplicity

A large focus should be on the simplicity and adoptability of the Dynamix DSL, especially for existing users of the Spoofax language workbench. One vehicle for achieving this simplicity is a resemblance to existing formalisms and conventions for (formal) dynamic specifications. Such resemblances will help users familiarize themselves by giving them the ability to associate constructs within the DSL with concepts and approaches that they already understand. A similarity with existing formalisms additionally helps establish a close tie with formal specifications, supporting the guiding principle that a Dynamix specification should be similar to a formal language specification.

Applicable to a wide variety of language paradigms

The Dynamix DSL should offer abstractions capable of handling a large number of programming paradigms. Users should be able to implement their language of choice without being limited by a lack of DSL features. This does not imply that *every* language should be supported, but rather that features within the DSL should be chosen and designed in a way where they do not unnecessarily limit the general applicability of the DSL.

Must be able to yield a performant runtime

Desiring a fast runtime for a language should be a valid motivator for writing a language specification using the Dynamix DSL. The Dynamix project must be designed in a way where it is possible to produce language runtimes that perform within an order of magnitude compared to a hand-written implementation. The initial implementation, as discussed in this thesis, does not have to perform at this level, but there should be no technical limitations preventing a performant runtime from being created. The abstraction level of both the DSL and the target language should consider the runtime performance.

High-quality integration within the Spoofax workbench

The Dynamix project should have a tight and idiomatic integration with other meta-languages in the Spoofax language workbench. Users should be able to easily write dynamic specifications for their existing Spoofax projects, without overhauling parts of their projects. The design and behavior of the language should, where applicable, be consistent with other Spoofax meta-languages to allow for easier adoption.

Chapter 4

Designing a DSL for runtime semantics

In this chapter, we will establish some of the core design decisions behind the Dynamix meta-language by discussing approaches for designing and implementing a DSL for runtime semantics. This design approach is guided by the objectives outlined in Chapter 3, and will discuss how we might structure the language, what features would help widen the range of source languages the DSL might be applicable for, and what kind of interaction with other parts of the compilation process is appropriate.

Within this chapter, we use the term **source language** to refer to the language under compilation (i.e. the language for which a runtime semantics specification is written). Unless otherwise specified, the term **DSL** refers to the DSL for dynamic specification whose design we are considering (i.e. the Dynamix meta-language). The term **rule** refers to the runtime semantics for a single source language construct (e.g. a single rule may define the behavior of integer multiplication). The term **specification** refers to the set of all runtime semantics **rule**s (for the source language) written in the DSL.

4.1 The anatomy of a formal runtime semantics specification

We will start our design process by taking a look at existing formal specifications of several programming languages. Doing so allows us to derive a minimal set of features needed to replicate these specifications in our new DSL, while at the same time giving us a general idea of what the syntax of the DSL may look like. In particular, we will consider the formal specifications of the plus (+) operator, responsible for both numerical addition and string concatenation. This operation was chosen for several reasons: it is a common and well-understood operation, it involves different behavior based on the types of its sub-expressions, and one has to consider the underlying data storage (such as the bit-width of the numeric types and the wrapping behavior when addition may overflow) when defining the operation.

The languages specifications that we will consider are those of the ChocoPy programming language [48], as well as the ECMAScript specification for the JavaScript programming language [18]. These were chosen for several reasons: they vary in notational style (the ChocoPy specification uses big-step denotational semantics[1], the ECMAScript specification uses natural language), they represent different language designs (ChocoPy is statically typed, JavaScript is dynamically typed), and they target different platforms (ChocoPy is intended to be com-

[1]Big-step denotational semantics, also known as natural semantics or Kahn-style semantics, are a notational style for dynamic semantic specifications due to Gilles Kahn [31]. In big-step semantics, the notation $\rho \vdash E \Rightarrow \alpha$ indicates that the expression E, when evaluated in the environment ρ, yields the value α.

$$\frac{\begin{array}{c} G, E, S \vdash e_1 : str(n_1, s_1), S_1, \_ \\ G, E, S_1 \vdash e_2 : str(n_2, s_2), S_2, \_ \\ v = str(n_1 + n_2, s_1 || s_2) \end{array}}{G, E, S \vdash e_1 + e_2 : v, S_2, \_} \quad [\textsc{Str-Concat}]$$

$$\frac{\begin{array}{c} G, E, S \vdash e_1 : int(i_1), S_1, \_ \\ G, E, S_1 \vdash e_2 : int(i_2), S_2, \_ \\ v = int(i_1 + i_2) \end{array}}{G, E, S \vdash e_1 + e_2 : v, S_2, \_} \quad [\textsc{Int-Addition}]$$

Figure 4.1: Examples of the runtime semantics for string concatenation and integer addition in formal notation for the ChocoPy language. Adapted from the ChocoPy reference manual by Padhye et al. [49].

EvaluateStringOrNumericBinaryExpression(***leftOperand*, +, *rightOperand***)

- Let *lref* be the result of evaluating *leftOperand*.
- Let *lval* be ? `GetValue`(*lref*).
- Let *rref* be the result of evaluating *rightOperand*.
- Let *rval* be ? `GetValue`(*rref*).
- Let *lprim* be ? `ToPrimitive`(*lval*).
- Let *rprim* be ? `ToPrimitive`(*rval*).
- If `Type`(*lprim*) is String or `Type`(*lprim*) is String, then
  - Let *lstr* be ? `ToString`(*lprim*).
  - Let *rstr* be ? `ToString`(*rprim*).
  - Return the string-concatenation of *lstr* and *rstr*.
- Note: at this point it must be a numeric operation.
- Let *lnum* be ? `ToNumeric`(*lval*).
- Let *rnum* be ? `ToNumeric`(*rval*).
- [*NaN and infinite cases omitted*]
- Assert: *lnum* and *rnum* are both finite.
- If *lnum* is $-0_{\mathbb{F}}$ and *rnum* is $-0_{\mathbb{F}}$, return $-0_{\mathbb{F}}$.
- Return $\mathbb{F}(\mathbb{R}($*lnum*$) + \mathbb{R}($*rnum*$))$.

Figure 4.2: Examples of the runtime semantics for string concatenation and number addition in the JavaScript programming language. Certain steps are omitted for the sake of brevity. Adapted from the ECMAScript 2023 language specification [18].

piled to RISC-V assembly, JavaScript is generally interpreted). Figures 4.1 and 4.2 show formal specifications of the plus operator for ChocoPy and JavaScript, respectively.

The ChocoPy specification of the plus operator (Figure 4.1) is described using formal notation. The *store* parameter S, which represents the values of all locations currently in use by the program, is explicitly specified as an input and output parameter to each operation. Since the evaluation of e_2 uses store S_1, it must logically be evaluated after e_1. As a result, if evaluating e_1 had side effects, those effects must be visible when e_2 is evaluated. The specification also considers the underlying data representation of the values. String values (`str`) are defined to carry both the length and the contents of the string as separate fields, and the concatenation explicitly notes that the resulting string length is the sum of the individual lengths. How exactly integers are stored, and how overflow should be handled, is mentioned elsewhere in the ChocoPy manual [49]: integers are signed 32-bit values, and the behavior when this addition overflows is undefined[2].

The ECMAScript specification of the plus operator (Figure 4.2) uses natural language instead of formal notation to describe the operation. Here, unlike the ChocoPy specification, there is no explicit "state-threading". Instead, it is implicitly assumed that operations are performed in the order in which they are listed, and that any underlying changes to the program environment as a result of these operations are reflected in the next step. Indeed, the current environment in which the expressions are evaluated (G, E, S in the ChocoPy specification) is implicitly inherited from the environment that contained the addition expression. The specific data representations of both numbers and strings, as well as the exact behavior of string-concatenation and the $\mathbb{F}$ and $\mathbb{R}$ operators, are defined elsewhere in the specification.

Both specifications perform *conditional* behavior to determine whether to apply numeric or string addition. For the ECMAScript specification, this branching is explicitly done at runtime: the type of the operands is derived only after they are both evaluated. This is a natural choice for JavaScript: it is impossible to always statically predict the runtime types of expressions due to its dynamically typed nature[3]. The ChocoPy specification is more ambiguous in how a specific rule is selected. It is clear that the premises of the [Str-Concat] and [Int-Addition] rules are disjoint (after all, a value cannot be both an integer and a string at the same time), but there is no explicit process described for exactly *when* to select a rule. Since ChocoPy is statically typed, a compiler that unconditionally emits either string concatenation or integer addition code based on the types involved would adhere to the specification. However, an implementation that performed this check at runtime would be equally valid (since the ChocoPy specification imposes no specific restrictions on both the data representation of values and the time complexity of the addition operation). This shows an interesting aspect of formal specifications for programming languages: their primary aim is to describe *how* programs are executed, but this description does not necessarily translate to the most optimal[4] implementation.

[2]"Undefined behavior" is a common term in formal language specifications and indicates a complete lack of obligations on a particular implementation of the language when such behavior occurs. Returning an arbitrary value, crashing the program, or even shutting down the computer would all be behaviors consistent with the specification.

[3]It should be noted that even though JavaScript is dynamically typed, advanced JavaScript engines are able to elide the vast majority of type checks through a combination of static and runtime analysis. This elision does not conflict with the specification, since it is only performed in situations where the engine is able to prove that it is safe to do so.

[4]Generally, a language implementation is considered more optimal if it runs faster. However, different requirements may call for a different definition of optimal (e.g. consider a lightweight implementation meant for low-power electronics, which may favor simplicity over performance).

Based on the two discussed specification examples, we can derive the following categories of features that our DSL should support. These give a baseline set of requirements that the designed DSL should conform to in order to replicate the ChocoPy and ECMAScript specifications. We justify these requirements by highlighting the appropriate sections in the discussed examples.

- **Static rule selection**. The DSL should be able to offer some deterministic method for selecting which rule(s) are applicable for any specific language construct. It should be possible to indicate that one rule applies to the multiplication operator, whereas another applies to the subtraction operator.
  - In the formal notation used by the ChocoPy specification, the applicable rules for some expression e are the ones for which $G, E, S \vdash e : v, S', \_$ appears in the conclusion. The ECMAScript specification uses natural language to indicate that a binary expression should defer to the appropriate `EvaluateStringOrNumericBinaryExpression` implementation.
- **Runtime conditional behavior**. A rule in the DSL should be able to perform conditional behavior based on values derived at runtime, for situations where an applicable rule cannot be selected statically. It should be possible to only conditionally evaluate (sub-)expressions.
  - The ECMAScript specification performs conditional behavior at runtime to differentiate between numerical addition and string concatenation. The discussed ChocoPy examples do not perform any conditional behavior. Conditional evaluation is essential to in order to be able to specify branching behavior (e.g. if-then-else statements, loops).
- **Clear ordering of operations**. It should be unambiguous in what order operations are evaluated in a DSL rule, and how these operations affect the surrounding state of the program.
  - The ChocoPy example explicitly threads stores by using S_n. The ECMAScript specification implicitly specifies that operations are executed in exactly the listed order. Both specifications explicitly require the left-hand side of the addition operator to be evaluated before the right-hand side.
- **Access to primitive operations**. It should be possible to express "primitive" operations in the DSL, such as arithmetic operations, I/O operations, and interacting with the operating system.
  - Both the ChocoPy and ECMAScript specifications use the addition operator in its mathematical sense. This is a primitive operation, as it cannot be expressed using other operations within the language. Such operations must therefore be provided by the runtime platform (e.g. ChocoPy addition might be performed using the `add` RISC-V instruction), and exposed by the DSL.
- **Appropriate control over data representation**. A rule in the DSL should be able to control the runtime data representation of values within the language, to a certain degree. A user should be able to specify representation-dependent options, such as the bit-width of integer values and their wrapping behavior. A user should be able to form composite values, such as tuples, records, or structs.
  - Neither the ChocoPy nor the ECMAScript specification fully specifies the runtime data representation of its values, opting to allow the implementation to choose an

appropriate representation. However, both specifications put requirements on the data representation of certain values: integers in ChocoPy **must** be 32-bit signed, and numbers in ECMAScript must be 64-bit IEEE 754-2019 [24] double-precision floating point numbers.

4.2 From specification to evaluation

So far, we have only discussed the definition aspects of a runtime semantics DSL. An equally important part is the ability to transform a specification into some method of running the source programs. Without this capability, the DSL would simply be yet another notation for runtime semantics. In this section, we discuss several different approaches for using a specification written in our DSL to automatically run source language programs.

Broadly speaking, we have established that a formal specification essentially acts as a list of instructions to perform when a certain language construct should be executed. These instructions interact with the runtime environment of the program (such as its memory or variables) and dictate which constructs should be evaluated next, in what order, and how their results are used. Logically, this means that "running" a specification with some source program as input effectively involves repeatedly selecting the appropriate rule to apply, then evaluating all steps for that rule until the program reaches termination. There are multiple ways of doing this, each with their own set of upsides and downsides. The most common solutions include:

- **A DSL interpreter.** A single program takes both a specification and a source program AST as parameters. The interpreter selects the appropriate "entry rule" for the input AST, then (recursively) executes each instruction in the rule. This style of interpretation is also known as *meta-interpretation*, since it is the meta-language that is the subject of interpretation.
- **Automatic interpreter/compiler generation.** The specification is used to generate a specialized interpreter or compiler for the target language. The resulting program can be used to directly run or compile source programs.
- **A DSL meta-compiler.** The dynamic specification is interpreted using a source program AST as input. Instead of directly performing runtime operations as they are interpreted (as is done with meta-interpretation), operations are instead collected into some target intermediate representation (IR). The resulting output is a language-agnostic set of instructions corresponding to exactly the operations taken by the input program, which can subsequently be compiled or interpreted directly. The meta-compiler name derives from the fact that this approach compiles a source program to a language-agnostic target language, with this compilation being driven by the meta-language rules.

Let us briefly discuss each of these approaches and consider their strengths and weaknesses in more detail.

Direct interpretation

Arguably the simplest approach is to write a direct interpreter for the DSL. This involves using an input (desugared, type-checked) AST to guide which rule(s) should be executed, with each specification instruction executed in order. Since a specification is effectively a formal description of an interpreter for the language, this approach typically is not very performant due to the "nested" interpretation (the interpreter interprets the DSL, which in turn interprets the source language). However, this approach is very portable (a single interpreter is capable of running every specification on every source snippet) and generally easy to implement and debug. This is the approach taken by PLT Redex [42] (further discussed in Chapter 10).

Specialized interpreter/compiler generation

The abstract specification is transpiled[5] to an interpreter or compiler in a concrete programming language. The resulting program can be used to directly run source ASTs. This is often a more performant approach to running the specification, as the overhead of the meta-language largely disappears. An example of a project that uses this approach is the DynSem [64] dynamic specification language, which automatically translates a specification into an AST-based interpreter running on the Java virtual machine (we further discuss DynSem in Chapter 10). The final performance of the generated interpreter or compiler depends largely on how it is implemented. For example, subsequent work by Vergu et al. [66] on the same DynSem framework demonstrated performance increases of up to 15x by improving the generated interpreter.

Meta-compilation

The final approach we will discuss is that of *meta-compilation*. This approach lends itself to the observation that, due to the nature of a dynamic specification, a program where every source construct is replaced with the body of the appropriate dynamic specification rule behaves exactly the same (after all, the behavior of the source program is defined through the specification). As we will discuss later, this is the approach used by Dynamix for executing source programs. An example of this approach can be seen in Figure 4.3.

The meta-compilation technique is quite similar to symbolic execution[6], but unlike symbolic execution it only considers rule selection and recursive rule invocation based on the input source language AST. Any operations that require runtime state (such as access to memory, variables, or conditional jumps) are deferred. The resulting output is effectively the (partial) *application* of the specification on the specific input. For the example in Figure 4.3, observe that the numeric literal `10` has been inlined into the declaration for `LetDeclaration`, but that all memory operations have been retained.

The benefit of the meta-compilation technique is that it is able to generically transform every source language for which a specification exists into a single language-agnostic format. The resulting format can be directly interpreted or compiled into a target program. Since this language-agnostic format is common between all possible specifications and source languages, it is comparatively simpler to efficiently compile or run the program (compared to the interpreter generation approach, which cannot make any assumptions about the language which it is interpreting). As a downside, such a language-agnostic IR must be generic enough to support a wide range of languages, and therefore may not be capable of representing every language construct as efficiently as would be possible with a more specialized approach.

[5] Source-to-source compilation between two languages of a similar abstraction level.

[6] A method of executing a program in an abstract manner, where every possible behavior of the source program is considered. Originally introduced by James C. King [35], it is a common analysis technique used for testing, verifying, and debugging programs. Unlike symbol execution, meta-compilation does not consider symbolic values and only evaluates all (potential) program control flow.

```
on Program(stmts):
  - execute each stmt in `stmts` sequentially

on LetDeclaration(name, value):
  - let `v` be the result of executing `value`
  - assert: there is no global variable `name`
  - assign `v` to the global variable `name`

on IntLiteral(value):
  - yield the 32-bit signed representation of `value`

on Print(value):
  - let `v` be the result of executing `value`
  - let `str` be the result of converting `v` to a string representation
  - output `str` to standard output, followed by a newline character (0x0A, '\n')
```

```
let x = 10;
print x;
```

```
let `v` be the 32-bit signed integer 10
assign `v` to the global variable `x`
let `v1` be the value of the global variable `x`
let `str` be the string representation of `v1`
output `str` to standard output, followed by a newline character (0x0A, '\n')
```

Figure 4.3: A dynamic specification (top) and input program (middle) for a fictional language. The bottom program represents the result of applying meta-compilation on the source program, yielding a new program where every source language construct has been replaced with the appropriate instructions in the specification language.

4.3 Considering control flow

One aspect of a language specification we have so far largely glossed over is that of control flow. While Section 4.1 considers control flow from the aspect of evaluation order, we have yet to discuss constructs like conditionals, loops, and exceptions. With our intent to provide a DSL capable of describing a large number of source languages, we must make an effort to support most, if not all, of these constructs in an easy manner.

4.3.1 Control flow in existing specifications

The natural approach to implementing control flow in a dynamic specification is by simply abstracting it through conditional rule selection. Consider the example in Figure 4.4. The method for selecting the appropriate rule that applies to the while statement is used as a method to perform control flow: the conditional check implicitly introduced by the rule selection is used to perform the conditional behavior that a while loop must perform at the start of each iteration. A similar definition can be used to conditionally evaluate statements, such as an if-then-else expression.

This abstraction becomes much less simple in the presence of statements that can abruptly jump to a different part of the program, such as the `return` statement. In fact, the ChocoPy

$$\frac{G, E, S \vdash e_1 : bool(true), S_1, \_}{G, E, S \vdash \texttt{while } e_1 : b_1 : \_, S_1, \_} \quad [\textsc{While-False}]$$

$$\frac{\begin{array}{c} G, E, S \vdash e_1 : bool(false), S_1, \_ \\ G, E, S_1 \vdash b_1 : \_, S_2, \_ \\ G, E, S_2 \vdash \texttt{while } e_1 : b_1 : \_, S_3, \_ \end{array}}{G, E, S \vdash \texttt{while } e_1 : b_1 : \_, S_3, \_} \quad [\textsc{While-True}]$$

Figure 4.4: Control flow as seen in the ChocoPy reference manual [49]. The appropriate rule is chosen based on the evaluation of the condition expression e_1. The [While-True] case uses recursive invocations to the same rule as a means of performing multiple loop iterations.

$$\frac{\begin{array}{c} G, E, S \vdash e_1 : bool(true), S_1, \_ \\ G, E, S_1 \vdash b_1 : \_, S_2, R \\ R \text{ is not } \_ \end{array}}{G, E, S \vdash \texttt{while } e_1 : b_1 : \_, S_2, R} \quad [\textsc{While-True-Return}]$$

Figure 4.5: A specialized case of the while construct as seen in the ChocoPy reference manual [49]. This case aborts control flow if some statement in the body of the loop performed an early return.

reference manual defines another case as part of the while loop semantics, specifically for handling early returns. This case can be seen in Figure 4.5. The last value in the output triplet, *R*, indicates the value returned from the function. Every other rule that affects control flow must consider this value, and possibly immediately return it without evaluating the rest of the construct (e.g. a function body must not execute the remaining instructions if an early return is encountered).

This approach is manageable if one only needs to support early returns, but it doesn't scale very well[7]. The ECMAScript specification [18] handles this by wrapping the result of an evaluation inside a *continuation*. This continuation indicates whether execution should continue normally, or whether it should move elsewhere (e.g. because an exception was thrown). The `?` character prefixed to certain operations in the ECMAScript example discussed earlier (Figure 4.2) is part of this abstraction: it indicates that if the result of the following function call was an *abrupt completion* (i.e. an exception was thrown), this result should immediately be propagated upwards without continuing the remainder of the rule.

This approach is powerful enough to handle a wide range of control flow features, including early returns, loop breaking, generators, asynchronous functions, and exceptions. However, it comes at the cost of additional complexity in the specification (e.g. the possibility of an exception must be handled in every rule that can indirectly cause one) and has substantial overhead if a language implementation directly implements control flow in this manner. Clearly, both the usability and performance of our DSL would benefit from an alternative implementation for control flow that does not suffer from these issues.

4.3.2 Control flow through continuations

An alternative approach to control flow within a dynamic specification borrows from a technique first described by Strachey and Wadsworth [60]. They propose the use of *continu-*

[7]Perhaps this is the reason why ChocoPy does not support the `continue` and `break` constructs.

ations[8], an abstract term representing "the meaning of the rest of the program" (Reynolds, [53]), as a method of modeling the mathematical semantics of a programming language with jumps. The core insight is to no longer consider the semantics of each language construct in isolation. As Strachey and Wadsworth put it:

> The solution to this problem is to abandon the idea of giving the state transformation for each command in isolation. We must define, instead, a semantic function that yields, for every command γ, in a program, the state transformation which would be produced from there to the end of the program.

Particularly, we will add an additional parameter to our evaluation relation, indicating the continuation of the expression. This continuation is some abstract representation of a point in the program (e.g. a label or a function) where evaluation should continue after the evaluation relation has completed.

Consider the example in Figure 4.6. We introduce a new `continuation` parameter, representing a *label* to which execution[9] should jump after the completion of the computation described in the body of the function. For the evaluation of the subexpressions `leftOperand` and `rightOperand`, we simply indicate that they should continue directly on the next line at the AfterLeft and AfterRight labels. Conceptually, this transformation just makes the implicit returning behavior of functions (i.e. that they jump back to the callee directly after the call expression) explicit.

The real benefit of a continuation-based approach becomes clear when we perform control flow. Figure 4.7 contains a fictional implementation of the evaluation of statements in ECMAScript. In the case that the statement is a sole expression, it is directly evaluated, with execution continuing at AfterExpressionStatement. This simply discards the result of the expression, then jumps to *`continuation`* (which presumably represents the next statement). However, if the statement represents an early return, we instead execute the expression with the continuation label $ReturnFromFunction (assumed to have been defined when we entered the current function). This simple change has big consequences: because we never jump to *`continuation`*, control flow never continues "past" the return statement. The callee never has to consider an early return, as it won't even continue execution in the case where such an early return has occurred.

Other control flows can be implemented in a similar way. For example, exceptions can be handled by designating a $HandleException continuation. Throwing an exception can then be implemented by virtue of simply continuing execution at this label, instead of the supplied continuation. Analogously, the `break` statement might be implemented by defining a continuation label positioned after the body of the loop.

As far as I am aware, no current language specification makes use of continuations as an abstraction over control flow, despite their promising abilities. Perhaps the increase in cognitive complexity when reading the specification makes them an unattractive abstraction. However, a continuation-based approach to control flow lends itself excellently to our DSL.

[8]Continuations were independently conceived by several different computer scientists, but the version used by Strachey et al. originates from the work by Mazurkiewicz [43]. Reynolds's "The Discoveries of Continuations" [53] discusses the full origins of the continuation concept.

[9]We use the term execution here, but it is important to consider that this use of continuations is *only within the dynamic specification*. A specific implementation of the language is not required to use the specific continuation implementation, or even use continuations at all. We use the term execution only because it is convenient to view a dynamic specification as a set of instructions to be followed at runtime.

```
EvaluateStringOrNumericBinaryExpression(leftOperand, +, rightOperand,
continuation)
    • Evaluate leftOperand with continuation label AfterLeft.
AfterLeft(lref):
    • Let lval be ? GetValue(lref).
    • Evaluate rightOperand with continuation label AfterRight.
AfterRight(rref):
    • Let rval be ? GetValue(rref).
    • [several steps omitted]
    • Let result be $\mathbb{F}(\mathbb{R}(lnum) + \mathbb{R}(rnum))$.
    • Jump to continuation with value result.
```

Figure 4.6: A rewritten version of the ECMAScript addition example from Figure 4.2 that uses continuation labels for control flow. The highlighted sections indicate changes needed to support continuations. Text in Small Caps indicates a definition or reference to a continuation label.

```
EvaluateStatement(statementType, subExpression, continuation)
    • If statementType is ExpressionStatement:
        – Evaluate subExpression with continuation label AfterExpressionStatement.
    • If statementType is ReturnStatement:
        – Evaluate subExpression with continuation label $ReturnFromFunction.
    • [other cases omitted]
AfterExpressionStatement(discarded):
    • Jump to continuation.
```

Figure 4.7: Using continuations to implement the `return` statement. We assume that the continuation label $ReturnFromFunction has been defined earlier and that it implements the process of returning a value from a function.

Efficient compilation of programs that use continuations is a well-understood problem[10], and a structural approach to continuations aided by effective static analysis will relieve some of the cognitive complexity of working with continuations.

[10]The book "Compiling with Continuations" by Andrew W. Appel [6] is an excellent introduction to techniques for compilation using continuation-passing style (CPS).

Chapter 5

The Tim intermediate representation

Before we discuss the Dynamix language, we must first take a detour and discuss the language that it targets. As briefly alluded to in Chapter 4, Dynamix converts a language specification into a runnable program through the technique of *meta-compilation* (see Section 4.2). However, instead of directly converting a source program into a sequence of Dynamix runtime instructions, we instead transform the source program into a more specialized intermediate representation (IR) containing only the subset of Dynamix instructions that would actually be executed at runtime. By doing so, we can independently work on the Dynamix meta-language and the interpreter or compiler for the output IR.

The specific IR that Dynamix compiles[1] to is a new Spoofax meta-language called Tim[2]. Tim is a human-readable language with a relatively low level of abstraction that is designed to use continuation-passing style (CPS) (this choice coincides with the choice to use continuations as abstractions for control flow in Dynamix). Its grammar and semantics are largely inspired by the CPS IR used by the Standard ML compiler and described in Andrew W. Appel's book, Compiling with Continuations [6].

In this chapter, we first introduce Tim by discussing how a simple source program translates to Tim (Section 5.1). After this, we move on to the formal definition of the language by discussing Tim's syntax (Section 5.2), dynamic semantics (Section 5.3) and static semantics (Section 5.4). Finally, we discuss the Tim interpreter (Section 5.5). We discuss future work for the Tim language later, in Chapter 11.

5.1 A first impression

Let us introduce Tim by considering how a snippet of an imperative source language would look if it was written directly in Tim. Doing so will give us a good intuition of what the language looks like, how "traditional" operations correspond to operations in Tim, and how the control flow of "conventional" languages translates to CPS. Despite us calling Tim an IR in the introduction, it has a fully defined grammar and semantics[3]. Naturally, we can write programs directly in Tim, and run them using Tim's interpreter.

[1]For the remainder of this thesis, whenever we use the term compilation within the context of Dynamix, it will refer to the process of meta-compiling a Dynamix specification on a given input source. The name "compilation" is not inappropriate for this process. After all, Dynamix translates the input source to a lower-level specialized instruction set.

[2]The name Tim originates from Target InterMediate language.

[3]Calling Tim an IR is mainly motivated by the fact that it is not intended to be used as a language directly, but rather only a compilation artifact of running Dynamix. In this aspect, it is quite similar to the LLVM IR in use by the LLVM project [38].

As a source program, we will use the following Tiger[4] snippet. Tiger is a simple imperative language used in Andrew Appel's book *Modern Compiler implementation in Java* [8][5]. Our snippet involves some literals, arithmetic, and function calls:

```
let
  function getRandomNumber(): int = 4 // chosen by fair dice roll
in
  let
    var rnd: int := getRandomNumber() + 1
  in
    print("The random number, incremented by 1, is: ");
    print(rnd)
  end
end
```

If we directly translate this snippet to Tim, attempting to preserve roughly the same layout as the source program, we obtain the following straightforward program. Identical background colors between the two snippets indicate correspondence between the line(s) in the Tiger snippet and the line(s) in the Tim snippet.

```
fix {
  fun getRandomNumber(c) = c(4)
} in
  fix {
    fun c0(a0) =
      #int-add(a0, 1) => rnd;
      fix {
        fun c1() =
          print(rnd, c2)
        fun c2() =
          #exit()
      } in
        print("The random number, incremented by 1, is: ", c1)
  } in
    getRandomNumber(c0)
```

First, let us discuss the translation to continuation-passing style. We add an explicit continuation argument `c` to `getRandomNumber`, transforming the implicit return into an explicit call to the continuation. On the caller's side, we extract the remainder of the program into an anonymous function (`c0`), and pass it as continuation to the function call.[6] We perform similar translations for the calls to `print` (we assume that print is defined elsewhere), and insert an explicit call to `#exit` after the final statement. Continuations `c1` and `c2` capture the `rnd` variable.

[4]We will use Tiger as an example language throughout the remainder of this thesis.

[5]Prof. Stephen A. Edwards has written an excellent reference manual for Tiger: `http://www.cs.columbia.edu/~sedwards/classes/2002/w4115/tiger.pdf`.

[6]When compiling source programs to Tim, later lines often tend to be "sandwiched" by earlier ones. This is simply an artifact of Tim's `fix` syntax, requiring functions (which contain the "continuation") to be defined before they can be used. When reading Tim snippets, be prepared to often shift between the top and bottom of the snippet.

The only real computation in our source snippet, the integer addition, is implemented as a call to `#int-add`. "Functions" like these, prefixed with a `#`, are *primitives*. Primitives are Tim's approach to compiler intrinsics: functions whose body is implemented directly by the Tim compiler or runtime. Through primitives, Tim can expose operations like integer addition and access to composite data structures without the need to introduce explicit new grammar for these actions. Tim also diverges from the IR used in "Compiling with Continuations" in this aspect: while both IRs have the concept of primitive operations, the IR used in Appel's book limits the available primitive operations to a fixed set, whereas primitives in Tim are allowed to be any valid identifier. Similarly, some operations that have their own constructs in Appel's IR, such as `RECORD`, `OFFSET` and `SELECTED`, are represented using primitive calls in Tim instead.

Primitives come in three forms: calls, tail calls, and conditionals. Primitive tail calls usually perform some form of control flow and as a result must appear in the tail call position. Non-tail calls to primitives do not break the CPS expectation that calls must be in a tail call position: while primitives may look like function calls, their bodies are directly inlined and non-tail call primitives are not allowed to perform control flow. Conditional primitives[7] are only valid in if-then-else expressions and allow for conditional branching based on a compiler-implemented test (e.g. integer equality). Figure 5.1 shows an example Tim program that makes use of primitives for records and conditionals.

[7]You may wonder why Tim has a separate if-then-else conditional primitive construct when such a construct can also be implemented by simply passing two separate continuations to a primitive tail call. The reason is simply because it allows Dynamix to have an if-then-else construct that directly correlates with a Tim construct. It has the added benefit of making Tim programs slightly more readable.

```
fix {
  // Print the name of the user. If the first name is
  // empty, only print the last name.
  fun print_name(user, c) =
    #record-read(user, "first") => first;
    #record-read(user, "last") => last;
    if #str-eq(first, "") then
      #print(last) => tmp0;
      c()
    else
      #str-add(first, " ") => tmp0;
      #str-add(tmp0, last) => tmp1;
      #print(tmp1) => tmp0;
      c()
} in
  fix {
    fun c0() =
      #exit()
  } in
    #record-new("first", "John", "last", "Doe") => user;
    print_name(user, c0)
```

Figure 5.1: An example program that shows the use of primitives in a Tim program. Primitives allow for the introduction of new operations and new data structures, without the need for specialized syntax. Here, primitives are used to provide access to record data structures and to perform operations on string values.

5.2 Syntax

Let us now formally define the syntax for Tim. Unlike the CPS IR from ”Compiling with Continuations” [6] that Tim is based on, Tim has a formally defined syntax (Appel only considers the semantics of the CPS IR, and does not propose a syntax). Just like Appel's IR, our grammar syntactically ensures that the program is in a valid CPS form. Grammar 5.1 shows the full grammar for the Tim IR.

Grammar Notation Within this section, the superscript suffix ⋆ represents that the preceding term may be repeated 0 or more times. Similarly, the superscript suffix + represents that the preceding term may be represented 1 or more times, and the superscript suffix ? indicates that some term is optional. If a repetition qualifier is applied to a sequence of terms within angled brackets, it indicates that the given terms may be repeated using the trailing terminal symbols of the group as separator (e.g. the notation $\langle$ *cexp* **,** $\rangle^*$ represents zero or more comma-separated expressions). If the same non-terminal is defined multiple times, the syntax for the non-terminal should be interpreted as the disjunction of all definitions for it. Any number of whitespace is supported between productions in any non-terminal rule, unless (the lack of) such whitespace would cause the resulting code to be parsed differently. On digital versions of this thesis, clicking a non-terminal reference will navigate you to the definition of that non-terminal.

program	::=	*cexp*	
cexp	::=	*cval*`(`⟨ *cval*`,` ⟩*`)`	*tail call*
	|	`#`*primitive*`(`⟨ *cval*`,` ⟩*`) =>` *id*`;` *cexp*	*primitive call*
	|	`#`*primitive*`(`⟨ *cval*`,` ⟩*`)`	*primitive tail call*
	|	`if #`*primitive*`(`⟨ *cval*`,` ⟩*`) then` *cexp* `else` *cexp*	*conditional primitive*
	|	`fix {` *cfun** `} in` *cexp*	*function definition*
	|	`let` ⟨ *id* `=` *cval*`,` ⟩$^+$ `in` *cexp*	*let definition*
cval	::=	*int*	*integer literal*
	|	*string*	*string literal*
	|	*id*	*variable reference*
cfun	::=	`fun` *id*`(`⟨ *id*`,` ⟩*`) =` *cexp*	
id	::=	`[a-zA-Z_$] [a-zA-Z0-9_$-]`*	
int	::=	`-`$^?$ `[1-9] [0-9]`*	
string	::=	`"` *string-char** `"`	
primitive	::=	`[A-Za-z0-9_+*/-]`*	

Grammar 5.1: The grammar of the Tim intermediate representation.

One particular property of the Tim grammar, as well as its corresponding algebraic signature, is that it is *CPS-by-construction*. That is, the design of the grammar automatically enforces that certain properties of the CPS hold for any well-formed Tim term. In particular, the grammar enforces that calls are always in the tail position, and that arguments to calls are atomic values (i.e. literals or variable references, but not complex sub-expressions).

Enforcing proper tail call behavior is done by carefully modeling how multiple expressions combine. Instead of a sequencing operator (e.g. `cexp ',' cexp`) or a flat block statement (e.g. `'{' cexp* '}'`), each expression in Tim directly embeds the next expression. The only expressions that do not specify their subsequent expression are function calls and primitive tail calls. This trivially makes them the final expression in a sequence, and hence enforces that every call is a tail call. Note that this property also means that every expression *must* end in a tail call: it is simply not possible to terminate a sequence of expressions with anything other than a (primitive) tail call.

Enforcing that expressions (and function calls in particular) do not contain complex sub-expressions is done simply by making these a different sort. *cval*s are defined to be solely the expressions that directly yield a value: integer literals, string literals, and variable references. Only *cval*s may be passed to function and primitive calls.

5.3 Dynamic semantics

We first focus on the runtime semantics of Tim. While the Tim language resembles that of Appel's IR in Compiling with Continuations [6], the use of a dynamic set of value, expression, and conditional primitives means that it departs enough from Appel's IR that it is worth fully specifying the language. We defer the discussion of Tim's static semantics until Section 5.4, as they naturally follow from the runtime semantics of the language.

5.3.1 Notation

We formally define the runtime semantics of Tim through Kahn-style big-step operational semantics [31]. To do so, let us first define some notation.

The store, S, is a mapping from variable names to their concrete values. Since Tim variables are immutable once created, we do not require a separate environment and store. In order to still support mutable values, we provide the heap, H, which functions as a mapping from address to value. The heap is not exposed to any Tim language constructs, but it may be freely accessed by the implementations of certain primitives. We use the syntax $S' = S[x \mapsto y]$ (respectively $H' = H[x \mapsto y]$) to extend S with a new mapping from x to y, yielding the store S' (respectively heap H'). If S already contained a mapping for x, it is *shadowed* by the new value y in S', but remains accessible in S. The syntax $S[x]$ retrieves the value bound to x in store S.

Values are denoted as $v = X(a_1, a_2, ..., a_n)$. Here X is the *discriminant*, indicating the type of the value. The content of the data is included in the arguments a_1 through a_n. The core Tim language has three different value sorts: *int*(i), *str*(s) and *func*$(S, \overline{args}, body)$. For the function sort, the associated S indicates the environment that the function closes over. Primitives can return additional data types not in this list. These data values are prefixed with `#` and are opaque to core Tim constructs. They can only be handled by primitives that support these data types.

We will use two different relations to define the behavior of Tim. For `cval`s, we will use the notation $S \vdash e \longrightarrow v$ to indicate that the `cval` e evaluates to the value v within the store S. Since `cval` terms cannot have side effects, we do not have to consider any changes to S. Similarly, `cval` terms cannot access the heap H. For expressions, we use the notation $S, H \vdash e \Downarrow H'$ to indicate that evaluating the expression e eventually yields the heap H'. Note that, due to the nature of CPS, this relation represents the result of evaluating the program to the end from a given expression e. Because of this, we do not consider any values produced by this relation.[8]

In some cases, we will use an overhead bar to indicate that something represents a vector or a vector operation. For instance, the notation $S \vdash \overline{e \longrightarrow v}$ indicates that the list of `cval` terms $\overline{e}$ evaluates to the list of values $\overline{v}$. One special case is that of extending a store with multiple elements at once: $S[\overline{x \mapsto y}]$. This operation is only valid if all names in $\overline{x}$ are distinct[9].

The behavior of primitives is implemented through three functions. We use $\textsc{TailPrimitive}(H, x, \overline{v}) = H'$ to indicate that calling primitive x in a tail call position with arguments $\overline{v}$ and heap H yields the new heap H'. Similarly, we use $\textsc{Primitive}(H, x, \overline{v}) = v, H'$ to indicate that the value primitive x yields value v and changed heap H' when invoked with arguments $\overline{v}$ and heap H. Finally, we use the notation $\textsc{ConditionalPrimitive}(H, x, \overline{v}) = b, H'$ for conditional primitives. Here, b is either $true$ or $false$, depending on which branch of the conditional statement should be taken. We list some example implementations of primitives in Section 5.3.3.

[8] After all, a CPS expression must end with either a tail call or a call to the `#exit` primitive. Neither can produce a value that we could reasonably consider the "result" of the expression.

[9] This is generally asserted as part of the static semantics of the language.

5.3.2 Semantics of the core language

With the notation defined, we can discuss the semantics of the Tim core[10] language. Let us start simple by defining the behavior of `cvals`. String and integer literals directly convert to their runtime equivalents. For variables, we perform a lookup in the store.

$$\frac{}{S \vdash i \rightarrow \mathit{int}(v)}\ \textsc{V-Int Literal}$$

$$\frac{}{S \vdash s \rightarrow \mathit{str}(s)}\ \textsc{V-String Literal}$$

$$\frac{v = S[x]}{S \vdash x \rightarrow v}\ \textsc{V-Variable Reference}$$

With `cvals` out of the way, let us tackle the expressions. A `let`-binding simply extends the store with a new name for a value. Newly defined variables are only visible in the body, and not in subsequent bindings of the same let.

$$\frac{\begin{array}{c} S \vdash \overline{e \rightarrow v} \\ S_l = S[\overline{x \mapsto v}] \\ S_l, H \vdash b \Downarrow H' \end{array}}{S, H \vdash \texttt{let}\ \overline{x = e}\ \texttt{in}\ b \Downarrow H'}\ \textsc{E-Let}$$

`fix` expressions are used to declare functions. Functions declared within the same `fix` block are allowed to refer to each other, as well as any variables declared in a higher scope (they *capture* their scope).

$$\frac{\begin{array}{c} f_1 = \mathit{func}(S', \overline{args_1}, \mathit{body}_1) \\ f_2 = \mathit{func}(S', \overline{args_2}, \mathit{body}_2) \\ \vdots \\ f_n = \mathit{func}(S', \overline{args_n}, \mathit{body}_n) \\ S' = S[x_1 \mapsto f_1, x_2 \mapsto f_2, ..., x_n \mapsto f_n] \\ S', H \vdash b \Downarrow H' \end{array}}{S, H \vdash \texttt{fix \{ fun}\ x_1\texttt{(}\overline{args_1}\texttt{) =}\ \mathit{body}_1 \cdots \texttt{fun}\ x_n\texttt{(}\overline{args_n}\texttt{) =}\ \mathit{body}_n\ \texttt{\} in}\ b \Downarrow H'}\ \textsc{E-Fix}$$

The tail call expression can be used to call functions. For a call to be valid, it must resolve to a value of type $func$ with a matching number of arguments. We substitute the argument names into a new environment based on the captured environment from the function, then evaluate the body with the newly formed environment to compute the result of the function call.

$$\frac{\begin{array}{c} S \vdash e \rightarrow \mathit{func}(S_f, \overline{argn}, \mathit{body}) \\ S \vdash \overline{a \rightarrow v} \\ S'_f = S_f[\overline{argn \mapsto v}] \\ S'_f, H \vdash \mathit{body} \Downarrow H' \end{array}}{S, H \vdash e\texttt{(}\overline{a}\texttt{)} \Downarrow H'}\ \textsc{E-Tail Call}$$

[10]We define "Tim core" to be the parts of the Tim language that are statically defined, i.e. anything but the behavior of primitives.

The primitive tail call is very similar, but we defer the next continuation as well as the resulting heap to the implementation of the primitive. It is an error to call a primitive that is not defined. We discuss the semantics of several primitives in Section 5.3.3.

$$\frac{\begin{array}{c} S \vdash \overline{a \rightarrow v} \\ H' = \textsc{TailPrimitive}(H, \texttt{\#id}, \overline{v}) \end{array}}{S, H \vdash \texttt{\#}id\texttt{(}\overline{a}\texttt{)} \Downarrow H'} \text{ E-Primitive Tail Call}$$

Non-tail calls to primitives are similar, but instead of returning a continuation return a value directly. Note that even non-tail call continuations are allowed to change the heap.

$$\frac{\begin{array}{c} S \vdash \overline{a \rightarrow v} \\ v_p, H_1 = \textsc{Primitive}(H, \texttt{\#id}, \overline{v}) \\ S_1 = S[x \mapsto v_p] \\ S_1, H_1 \vdash e \Downarrow H_2 \end{array}}{S, H \vdash \texttt{\#}id\texttt{(}\overline{a}\texttt{)} \texttt{ => } x\texttt{;}\ e \Downarrow H_2} \text{ E-Primitive Call}$$

Finally, conditional primitives simply invoke the primitive function and branch based on the result of this invocation. It should be noted that the $true$ and $false$ in the semantics here are simply indications on which branch should be taken, with no particular requirements on exactly how these values are represented.

$$\frac{\begin{array}{c} S \vdash \overline{a \rightarrow v} \\ b, H_1 = \textsc{ConditionalPrimitive}(H, \texttt{\#id}, \overline{v}) \\ e = \begin{cases} e_1 & \text{if } b \text{ is } true \\ e_2 & \text{if } b \text{ is } false \end{cases} \\ S, H_1 \vdash e \Downarrow H_2 \end{array}}{S, H \vdash \textbf{if } \texttt{\#}id\texttt{(}\overline{a}\texttt{)} \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow H_2} \text{ E-Primitive Conditional}$$

5.3.3 Semantics of various primitives

Now let us discuss the semantics of some of the primitives available to Tim programs. Since Tim offers an API for language designers to implement their own primitives, it is impossible to provide an exhaustive list of all primitives and their semantics. Rather, this section is intended as a way to show how primitives may be implemented and how they are capable of significantly extending the features available to Tim programs.

The `#exit` primitive is the only way to gracefully terminate execution. Perhaps surprisingly, this is achieved by doing nothing. Due to the nature of CPS, not calling any continuations is effectively the same as terminating execution: the only way in which `#exit` can be called is through a primitive tail-call, which by definition is the last expression. This means that if we do nothing to explicitly continue execution, the program will exit.

$$\frac{}{\textsc{TailPrimitive}(H, \texttt{\#exit}) = H} \text{ Prim-Exit}$$

Simple operations, such as integer addition, `#int-add`, are expressed through primitives. But primitives are not limited to only performing a single operation: the `#str-add` primitive performs string concatenation, a considerably more complex operation.

$$\frac{}{\textsc{Primitive}(H, \texttt{\#int-add}, \mathit{int}(a), \mathit{int}(b)) = \mathit{int}(a+b), H}\ \textsc{Prim-Int Add}$$

$$\frac{}{\textsc{Primitive}(H, \texttt{\#str-add}, \mathit{str}(a), \mathit{str}(b)) = \mathit{str}(a||b), H}\ \textsc{Prim-String Add}$$

Conditional primitives such as `#int-eq` are implemented by evaluating the conditional and returning either $true$ or $false$.

$$\frac{b = \begin{cases} true & \text{if } a = b \\ false & \text{otherwise} \end{cases}}{\textsc{ConditionalPrimitive}(H, \texttt{\#int-eq}, \mathit{int}(a), \mathit{int}(b)) = b, H}\ \textsc{Prim-Int Equality}$$

Tim variables are immutable. The `#ref-new`, `#ref-fetch`, and `#ref-store` primitives allow for (shared) mutable values by allocating a value on the heap, and returning an opaque value that contains a reference to the allocated value, noted as $\mathit{\#ref}(x)$.

$$\frac{\begin{array}{c} x = \text{new address not occupied in } H \\ H' = H[x \mapsto v] \\ v_h = \mathit{\#ref}(x) \end{array}}{\textsc{Primitive}(H, \texttt{\#ref-new}, v) = v_h, H'}\ \textsc{Prim-Ref New}$$

$$\frac{v = H[x]}{\textsc{Primitive}(H, \texttt{\#ref-fetch}, \mathit{\#ref}(x)) = v, H}\ \textsc{Prim-Ref Fetch}$$

$$\frac{H' = H[x/v]}{\textsc{Primitive}(H, \texttt{\#ref-store}, \mathit{\#ref}(x), v) = v, H'}\ \textsc{Prim-Ref Store}$$

Primitives may also be used to introduce new data structures. The `#record-new`, `#record-read`, and `#record-write` primitives define and use a *#record* data structure, which functions as a heap-allocated list of key-value pairs. We use the notation $r = rec(k_1 = v_1, ..., k_n = v_n)$ to construct such a list, $r[k_1]$ to retrieve the value bound to k_1 in record r, and $r[k_x \mapsto v_x]$ to extend r with a new mapping from k_x to v_x, possibly shadowing the previous value. It is a runtime error to retrieve a value for an undefined key.

$$\frac{\begin{array}{c} x = \text{new address not occupied in } H \\ r = rec(k_1 = v_1, k_2 = v_2, ..., k_n = v_n) \\ H' = H[x \mapsto r] \\ v_{out} = \mathit{\#record}(x) \end{array}}{\textsc{Primitive}(H, \texttt{\#record-new}, \mathit{str}(k_1), v_1, \mathit{str}(k_2), v_2, ..., \mathit{str}(k_n), v_n) = r, H'}\ \textsc{Prim-Record New}$$

$$\frac{\begin{array}{c} r = H[x] \\ v = r[k] \end{array}}{\textsc{Primitive}(H, \texttt{\#record-read}, \mathit{\#record}(x), \mathit{str}(k)) = v, H}\ \textsc{Prim-Record Read}$$

$$\frac{\begin{array}{c} r = H[x] \\ r' = r[k \mapsto v] \\ H' = H[x \mapsto r'] \end{array}}{\textsc{Primitive}(H, \texttt{\#ref-store}, \mathit{\#record}(x), \mathit{str}(k), v) = v, H'}\ \textsc{Prim-Record Write}$$

5.4 Static semantics

Let us now discuss the static semantics for Tim. The reason we only discuss them now is because there simply *isn't much to do here*. Just like the CPS IR from "Compiling with Continuations", Tim is dynamically typed. Variables and function arguments have no statically declared type and be assigned any value, regardless of its type. Tim also has arbitrary primitives, with the ability for a language designer to define their own. As a consequence of this, we cannot perform much in terms of static analysis. In fact, the static semantics specification for Tim only defines the behavior of name binding, which we will discuss in Section 5.4.1.

However, the lack of static typing does not prevent us from making an attempt at performing static analysis. While the formal definition of Tim only includes the name binding rules, I have written an Statix specification for Tim for use within the Spoofax workbench. This specification uses the constraint engine from Statix to infer types of variables in order to statically reject *some* invalid programs. We will discuss this specification in Section 5.4.2.

5.4.1 Name binding semantics

Let us define the environment O as the set of variables in scope. We will then define the judgement $O \vdash e$ ($O \vdash v$) to indicate that `cexp` e (respectively `cval` v) has valid name bindings with respect to environment O. To extend an environment O with a new binding x, we use the syntax $O' = O \cup \{x\}$. If O already contained a binding x, it will be *shadowed*, such that any references to x within O' refer to the new value (note that the binding within O remains the original x). We use the shorthand $O \vdash v_1, v_2, ..., v_n$ to indicate that values v_1 through v_n must have valid name binding within the environment O. As with the dynamic semantics notation, we will use the overline bar, $\overline{a}$, to indicate that something represents a vector of values.

Figure 5.2 lists the full name binding rules for Tim. Generally, language features that introduce new bindings do so by extending their scope, shadowing any previous bindings with the same name. There is only a single namespace for both variables and functions (since functions are first-class values in Tim), and no access visibility modifiers. Within `fix` and `let` constructs, names that reside within the same scope (e.g. function names within a `fix` block, argument names in a single function) must be distinct.

5.4.2 Approximating static analysis with Statix

While Tim's design does not lend it well to static analysis beyond name binding, some attempts can be made to infer types of variables and functions based on their declaration sites and uses. To aid language designers in debugging their programs, the Tim implementation in the Spoofax language workbench includes a static specification using Statix [5] (see also Section 2.2.2 for more information on Statix). This specification checks name binding according to the rules discussed in Section 5.4.1, but it also attempts to type-check variables, function calls, and primitive calls by inferring types of values. This way, the specification acts as a "smoke test"[11] that gives users an opportunity to see whether their Tim program has trivial errors.

Within the Statix specification for Tim, types of variables are inferred from their assigned values. Since variables are immutable once created, and it is impossible to declare a name without immediately assigning a value to it (with the exception of function arguments), we can directly infer the type of a variable from the type of the value assigned to it. For function

[11] A test that checks whether some system at least appears to function properly. Smoke tests are often very coarse-grained and assert things like "does the program start without immediately crashing".

$$\frac{}{O \vdash i}\ \textsc{B-Int Literal}$$

$$\frac{}{O \vdash s}\ \textsc{B-String Literal}$$

$$\frac{x \in O}{O \vdash x}\ \textsc{B-Variable Reference}$$

$$\frac{O \vdash v_0, v_1, ..., v_n}{O \vdash v_0\texttt{(}v_1, v_2, ..., v_n\texttt{)}}\ \textsc{B-Call}$$

$$\frac{\begin{array}{c} O \vdash v_1, v_2, ..., v_n \\ O' = O \cup \{x\} \\ O' \vdash e \end{array}}{O \vdash \texttt{\#}primitive\texttt{(}v_1, v_2, ..., v_n\texttt{)} \texttt{ => } x\texttt{;}\ e}\ \textsc{B-Primitive Call}$$

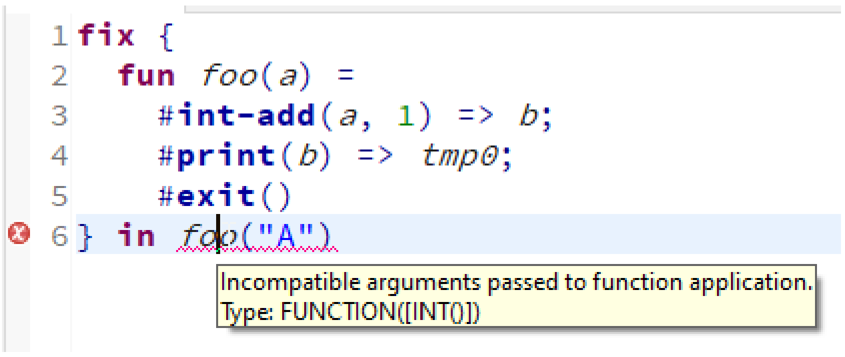
$$\frac{O \vdash v_1, v_2, ..., v_n}{O \vdash \texttt{\#}primitive\texttt{(}v_1, v_2, ..., v_n\texttt{)}}\ \textsc{B-Primitive Tail Call}$$

$$\frac{\begin{array}{c} O \vdash v_1, v_2, ..., v_n \\ S \vdash e_1 \\ S \vdash e_2 \end{array}}{O \vdash \texttt{if \#}primitive\texttt{(}v_1, v_2, ..., v_n\texttt{)}\ \texttt{then}\ e_1\ \texttt{else}\ e_2}\ \textsc{B-Primitive Conditional}$$

$$\frac{\begin{array}{c} f_1, f_2, ..., f_n \text{ distinct} \\ \text{all elements in } \overline{a_i} \text{ distinct for all } i \in [1, n] \\ O_f = O \cup \{f_1, f_2, ..., f_n\} \\ O_1 = O_f \cup \overline{a_1} \\ O_1 \vdash b_1 \\ \vdots \\ O_n = O \cup \overline{a_n} \\ O_n \vdash b_n \\ O_f \vdash e \end{array}}{O \vdash \texttt{fix \{ fun}\ f_1\texttt{(}\overline{a_1}\texttt{) = }b_1 \cdots \texttt{fun}\ f_n\texttt{(}\overline{a_n}\texttt{) = }b_n\ \texttt{\} in}\ e}\ \textsc{B-Fix}$$

$$\frac{\begin{array}{c} x_1, x_2, ..., x_n \text{ distinct} \\ O \vdash e_1, e_2, ..., e_n \\ O' = O \cup \{x_1, x_2, ..., x_n\} \\ O' \vdash b \end{array}}{O \vdash \texttt{let}\ x_1 = e_1, x_2 = e_2, ..., x_n = e_n\ \texttt{in}\ b}\ \textsc{B-Let}$$

Figure 5.2: The name binding rules for Tim.



```
fix {
  fun foo(a) =
    #int-add(a, 1) => b;
    #print(b) => tmp0;
    #exit()
} in foo("A")
```

```
Incompatible arguments passed to function application.
Type: FUNCTION([INT()])
```

Figure 5.3: Type inference in the Statix specification for Tim. Statix is able to determine that the type of `a` is an integer based on its use, and rejects the function call with the wrong argument type.

arguments, we use the constraint-solving engine to infer the type of an argument from its uses, by assigning it an unconstrained wildcard variable. Figure 5.3 shows an example of this inference in action.

There are several drawbacks to the Statix specification that prevent it from becoming a complete static specification for Tim. These drawbacks are also the reason why the language definition for Tim, despite the existence of a complete Statix specification, only formally specifies the name binding rules. In particular, these constraints are the following:

Type signatures of primitives must be defined inside the Statix specification. In order to properly infer types, the Statix specification must know the type signature of each primitive. Since there is no syntax to declare these signatures, they must be directly hardcoded in the specification. As a result, only a select handful of default primitives are supported by the Statix specification.

The type system used is (too) simplistic. In order to avoid specifying a complete type system for Tim, the types used in the Statix specification are very simplistic and have no concept of subtyping. New types introduced by primitives (e.g. records, arrays) are opaque and do not consider their contents. As a result, operations like `#record-read` cannot check whether the field exists within the record. An example of this can be seen in Figure 5.4 (top).

Type inference cannot handle polymorphic functions. The constraint solver used by Statix will unify constraints on a first-come, first-served basis. Once a variable has been unified, any other constraints it participates in will be evaluated as equality constraints. As a result of this, a polymorphic function will unify its arguments with the first function invocation, and assume this to be the function signature. The bottom snippet in Figure 5.4 shows this restriction. The type of `a` cannot be inferred from the body of `id`, so it is inferred from the first call (with an argument of type `INT`). The second call is then rejected, even though it would work properly at runtime.

Despite these flaws, Spoofax ships with the Statix specification enabled by default. Personal experience during the development of various test specifications, as well as the two case studies discussed later in this document (Chapter 9), has shown that the Statix specification delivers tangible benefits and allows for easier tracing of miscompilation artifacts caused by erroneous Dynamix specifications. In cases where the static analysis rejects a Tim program that is valid at runtime, the user is not blocked from actually executing the code; the static

```
#record-new("foo", 1) => a;
#record-read(a, "Foo") => foo;
#print(foo) => tmp0;
#exit()
```

```
fix {
  fun id(x, c) = c(x) // polymorphic fn(T, fn(T)) for all T

  fun c0() = id(1, c1) // OK, types `x` as INT()
  fun c1(_) = id("A", c2) // errors, STRING() != INT()
  fun c2(_) = #exit()
} in c0()
```

Figure 5.4: Two examples of limitations within the Statix specification for Tim. In the above example, no errors are raised even though the `"Foo"` field does not exist in `a`. In the bottom example, an error is raised because the type system used in the Statix specification is unable to represent polymorphic functions.

analysis is completely optional[12].

5.5 The Tim runtime

In order to run Tim programs, the Spoofax implementation of Tim ships with an interpreter for the language written in Stratego. This interpreter was written as a base implementation of Tim and focuses on a correct language implementation over performance. As a result, its main purpose is the ability to run Tim programs from *within* the Spoofax language workbench, such as during language development and as part of language test suites.

Stratego was chosen as an implementation language because it has direct integrations with most other meta-languages within the Spoofax language workbench. This allows it to be invoked from other parts of the language workbench and allows users to easily extend the language with new primitives by invoking the appropriate Stratego strategies. Since Tim reference implementation itself is also a Spoofax project, the Tim interpreter is also able to take advantage of the static type checking available in Stratego 2[13]. The Tim interpreter directly implements the dynamic semantics described in Section 5.3. As such, we will refrain from discussing the entire implementation in this thesis[14].

One aspect we will briefly discuss is how the Tim interpreter performs tail calls. Neither Stratego, nor Java (to which Stratego compiles), has native support for tail calls. Since Tim is a CPS-based language, it is imperative that it supports proper tail calls. A lack of tail call support would mean that any non-trivial Tim program will eventually run out of stack space. Tim resolves this issue by modeling an expression as returning a *continuation* value. This value indicates whether execution should continue (and if yes, where), or whether it should

[12]There have been several discussions during my thesis on whether the static analysis should emit errors or warnings, since it may reject valid programs. In the end, we settled on errors as they are correct the overwhelming majority of the time and almost always indicate a serious issue in the generated code.

[13]Stratego 2 is a partial rewrite of the Stratego language and introduces incremental compilation and gradual typing, among other things. At the time of writing, Stratego 2 is still in development and there have yet to be any publications.

[14]If the reader is interested, the code for the Tim interpreter is available online at `https://github.com/metaborg/spoofax-pie/blob/develop/lwb/metalang/tim_runtime/tim_runtime.spoofax2/trans`

```
signature
  sorts Continuation
  constructors
    ContinueAt : scope * Exp -> Continuation
    Exit : Continuation

strategies
  // `while` implemented iteratively in Java
  external native-while(c, s|)

rules
  eval-exp(|scope): TailCall(target, args) -> ContinueAt(fnBodyScope, fnBody)
    with
      FunctionValue(capturedScope, argNames, fnBody) := <eval-exp(|scope)> target
      // [remaining code omitted]
  eval-exp(|scope): PrimitiveTailCall("exit", []) -> Exit()
  // [other cases omitted]

  eval-continuation: Exit() -> Exit()
  eval-continuation: ContinueAt(scope, exp) -> <eval-exp(|scope)> exp

  eval-program =
    eval-exp(|<scope-new>)
    ; native-while(not(?Exit()), eval-continuation)
```

Figure 5.5: A sketch of how the Tim interpreter in Stratego implements tail-calling by modeling expressions as returning a continuation value. The `native-while` strategy is used to avoid issues with the recursively-implemented `while` strategy from Stratego's standard library.

abort. Evaluating a program is done by iteratively performing execution on an expression using the `while` strategy until a termination continuation term is received. A sketch of this approach can be seen in Figure 5.5. One particularity is that the `while` strategy from the Stratego standard library uses recursion to perform iteration[15]. As a result, early versions of the Tim interpreter were not actually properly tail calling, despite appearing to be. A custom native strategy[16] that works iteratively instead of recursively is needed to achieve proper tail calls.

The Tim language has no major design decisions that prevent a more performant implementation, such as an ahead-of-time compiler, in the future. Such an implementation must support first-class functions that capture scope, first-class tail call support, as well as efficient implementations of common primitives (e.g. the record primitive, which is effectively a hash table). Other points that must be considered include the memory model of the language (the Tim interpreter relies on the Java GC), the runtime representation of types, and potential interoperation with other languages. We briefly discuss such a potential compiler in Chapters 10 and 11.

[15]This is likely an oversight since a similar strategy, `repeat`, does not suffer from this issue. A bug was filed to correct the issue: `https://github.com/metaborg/stratego/issues/34`.

[16]A Stratego strategy implemented directly in Java.

Chapter 6

An introduction to the Dynamix meta-language

In this chapter, we will introduce the Dynamix meta-language in the form of a partial Dynamix specification for the Tiger programming language. Through writing a specification for Tiger, we will gradually introduce the language syntax and features of the Dynamix meta-language, so that we may develop an intuition on how one writes specifications with Dynamix. This introduction will also help us understand some of the peculiarities of the case studies we will later discuss in Chapter 9.

This chapter serves as an informal introduction to the Dynamix meta-language and its conventions. We will first discuss the specification for Tiger in Section 6.1. Afterwards, we discuss how several of Dynamix's abstractions help us keep Dynamix specifications concise, while at the same time generating valid CPS terms.

6.1 Implementing Tiger in Dynamix

One of the best ways to get acquainted with the Dynamix meta-language is to write a simple Dynamix specification for a source language. In this section, we will gradually introduce Dynamix and its features by working on a specification for Tiger. Tiger, a simple programming language used in Andrew Appel's *Modern Compiler Implementation* family of books [8, 7, 9], is a typed imperative language that supports function definitions, records, arrays, control flow, and common arithmetic operations. This makes it an excellent choice for an introduction to Dynamix.

The version of Tiger we will be using is the one described in Prof. Stephen A. Edwards's reference manual for the language[1]. A partial grammar for this language, containing only the constructs which we will discuss in this chapter, can be seen in Grammar 6.1. For the sake of brevity, this chapter only discusses the Dynamix specification for Tiger, omitting the SDF3 grammar and Statix specification aspects of the language definition. It should be noted that our main intention here is to explore the Dynamix meta-language, and not to write a complete specification for Tiger. We will therefore only discuss the specifications for certain language features if their implementation introduces an important part of Dynamix (e.g. we will discuss the implementation of while-loops, but omit the largely similar implementation for for-loops). Readers interested in the full language project, including the grammar and full Dynamix specification, are advised to consult the source code available online[2].

[1]`http://www.cs.columbia.edu/~sedwards/classes/2002/w4115/tiger.pdf`

[2]`https://github.com/metaborgcube/metaborg-tiger`

```
expr     ::= string                                              string literal
           | int                                                 int literal
           | lvalue                                              lvalue
           | expr + expr                                         addition
           | lvalue := expr                                      assignment
           | id(⟨ expr, ⟩*)                                      function call
           | (⟨ expr, ⟩*)                                        expression sequence
           | if exp then exp else exp                            if-then-else
           | type-id { ⟨ id = expr ⟩* }                          record literal
           | while expr do expr                                  while loop
           | break                                               break
           | let decl+ in expr* end                              let binding

lvalue   ::= id                                                  variable
           | lvalue.id                                           record member access

decl     ::= type type-id = type                                 type declaration
           | var id := expr                                      var declaration
           | var id: type-id := expr                             var declaration
           | fundecl+                                            mutually recursive fun declarations

fundecl  ::= function id(⟨ id: type-id, ⟩*) = expr               function declaration
           | function id(⟨ id: type-id, ⟩*): type-id = expr      function declaration

type     ::= type-id                                             type reference
           | { ⟨ id: type-id, ⟩* }                               record type
```

Grammar 6.1: The grammar for the Tiger language constructs discussed in this chapter. Based on the grammar listing from Stephen A. Edwards' reference manual for Tiger[1].

Within this section, we will omit the Dynamix type signatures for any rules we discuss. This is primarily because Dynamix's type system requires an understanding of both the general meta-language, as well as the abstractions it uses to fluently compile to a CPS language, in order to be properly understood. When we formally define Dynamix's type-system in Section 6.2.4, we will return to the signatures of some of the rules discussed.

$$\underbrace{\texttt{evalExp}}_{\text{name}}(\overbrace{\texttt{String(i)}}^{\text{argument pattern}}) = \underbrace{\texttt{str(s)}}_{\text{body}}$$

Figure 6.1: The structure of a rule definition in Dynamix.

6.1.1 Literals

We start our actual specification with the simplest language constructs that Tiger has to offer: literals. Tiger has integer and string literals, represented in the AST as `Int("...")` and `String("...")` respectively. Since Tim natively supports these literals, our specification for these constructs suffices by constructing the equivalent Tim literals:

```
rules
  // The first rule invoked by Dynamix is compileFile. We will delegate
  // it to evalExp, since a file consists of a single expression.
  // compileFile :: [type signature omitted]
  compileFile(Mod(exp)) = evalExp(exp)

  // evalExp :: [type signature omitted]
  evalExp(Int(i)) = int(i)
  evalExp(String(s)) = str(s)
```

A single Dynamix **rule**, such as `evalExp`, consists of a set of implementations. Each implementation follows the structure outlined in Figure 6.1, declaring a set of patterns that the arguments must match in order for the body of the rule to apply. Our `evalExp` rule accepts a single argument representing an `Exp` AST node. Just like other Spoofax languages, Dynamix uses the ATerm format for representing the source AST and allows pattern matching of constructors, literals, lists, and variables. We define two implementations for the rule: one pattern matches the input on an `Int` pattern, and one pattern matches the input on a `String` pattern. Should we try to invoke the `evalExp` rule with an argument that does not match any of the implementation patterns, it will raise an error during the source compilation process. Rule implementations are tried in order of their pattern specificity (see also Section 7.4).

We have named our expression rule `evalExp`, but you should not be deceived by this name. Dynamix remains a declarative language used for compilation of source programs, and not interpretation. However, it turns out that writing Dynamix specifications is quite similar to writing a declarative interpreter. If we view a dynamic specification as a set of steps to perform to evaluate some language construct, then it makes sense for the relation representing the evaluation of some expression to be called `evalExp`.

The bodies of our literal `evalExp` cases invoke the `int` and `str` language constructs. These accept a literal string value (i.e. a value known at compile-time) and return the equivalent Tim literal.

6.1.2 Simple arithmetic

Let us continue with some simple arithmetic operators. We will assume that the source snippet has been type-checked, so we do not need to do any runtime type checks. This means that our arithmetic is as simple as evaluating the sub-expressions, and performing the appropriate native operation. To perform the operations, we will use the `#int-X` family of Tim *primitives* (see Section 5.1) that ships with Dynamix by default.

```
rules
  evalExp(Plus(a, b)) = {
    av <- evalExp(a)
    bv <- evalExp(b)
    #int-add(av, bv)
  }
  // other arithmetic operators omitted
```

Note that we perform pattern matching for the appropriate arithmetic rules here, but we do not redefine the rule. Instead, these rules simply add to the rules for literals that we already defined earlier. Due to the abstractions that Dynamix offers, these implementations are straightforward enough that they can be understood at a glance even by someone not familiar with the language. However, there are some peculiarities involved.

One such peculiarity is that the `<-` operator is *required* in this snippet: directly doing `#int-add(evalExp(a), evalExp(b))` will be rejected by the Dynamix type-checker. We discuss the exact reason behind this in Section 6.2.4, but for now it suffices to say that Dynamix distinguishes between *computations* and *values*. The call to `evalExp` will return a computation, which needs to be performed in order to obtain a value. The `<-` operator will perform this operation, binding the resulting value to the left-hand side identifier[3].

Note that the call to `#int-add` uses orange syntax-highlighting. In general, any Dynamix language constructs that correspond with an output Tim expression (e.g. arithmetic, control flow) use this color to indicate that they will result in computations performed by the compiled program[4]. Constructs with a different color are part of the meta-language and will be interpreted as part of the compilation process. This syntax highlighting is integrated directly in the Eclipse IDE used for Spoofax language development, and all source snippets in this document are highlighted using the same highlighting engine.

6.1.3 Short-circuiting logic operators

Let us now consider a simple expression that requires control flow. Tiger's logic operators, `&` (`And`) and `|` (`Or`), are *short-circuiting*. If evaluating the left-hand side of the operator is enough to determine its value, the right-hand side should never be evaluated. In order to do this, we will need to perform conditional control flow in our Dynamix specification. Consider the implementation of `And`:

[3]At least, that is how it *looks*. We should not forget that the Dynamix meta-language is declarative and performs compilation, not interpretation. The actual behavior of the arrow operator is more akin to giving us an address at which the computed value will reside, instead of the actual value.

[4]We use the orange color for any construct that appears in some form in the output program, regardless of whether that construct performs actual computation. For example, `int(x)` is highlighted as orange, but this does not mean that the integer is parsed from a string at runtime. Rather, the Tim literal derived from the input will appear in the compiled program. If we consider the Dynamix compilation process to be a form of partial evaluation, orange terms would be conceptually similar to the residual expressions emitted by this partial evaluation.

```
rules
  evalExp(And(a, b)) = {
    av <- evalExp(a) // evaluate the left-hand first

    // if the left-hand evaluated to zero
    if #int-eq(av, int('0)) then {
      // immediately jump to the end with zero (false)
      after@([int('0)])
    } else {
      // else, evaluate the right-hand and use that as result
      bv <- evalExp(b)
      after@([bv])
    }
  } label after/1:
```

We first evaluate the left-hand and store the result in `av`. We then compare this result with the integer literal 0. In the case where it is equal (i.e. the left-hand value evaluated to false), we immediately perform a CPS tail-call to `after` with 0. In the case where it is anything else (i.e. the left-hand value evaluated to true), we evaluate the right-hand side and tail call `after` using the result of this evaluation. Dynamix's type system *requires* us to always tail-call within the branches of the if-statement (lending to the fact that the Tim grammar expects us to specify two full CPS terms for a conditional), so it would be a type error to omit either of the calls to the after continuation. Note that both branches will be present in the compiled binary since the if-expression must be evaluated at runtime (as also indicated by the orange syntax highlighting).

The `after` continuation that both branches jump to originates from the `label` syntax. In particular, the syntax `{ a* } label x/1: b` introduces a new continuation function with `b` as body. This continuation function is in scope for the computation of `a*`, such that it can refer to the continuation. The `/1` indicates that the continuation should accept a single argument (there is also a `/0` variant). Figure 6.2 shows an example of how the label construct is compiled to Tim. We discuss the label syntax, as well as its underlying abstraction, in more detail in Section 6.2.

```
result <- {
  a <- int('0)
  if #int-eq(a, int('1)) then {
    after@([str('"What?")])
  } else {
    after@([str('"Ok!")])
  }
} label after/1:

tmp0 <- #print(result)
#exit()
```

```
fix {
  fun after0(a0) =
    #print(a0) => tmp0;
    #exit()
} in
  if #int-eq(0, 1) then
    after0("What?")
  else
    after0("Ok!")
```

Figure 6.2: An example of how the `label` syntax in Dynamix abstracts away the process of creating a continuation. The Dynamix specification (left) compiles to the given Tim program (right). Sections with a matching background color indicate a correspondence between the Dynamix specification and the Tim output. Note that the meta-variable `result` is automatically bound to the argument `a0` of the continuation. We discuss the label abstraction in more detail in Section 6.2.

6.1.4 Variables

Let us continue by implementing variables. This is no simple task. In order to fully implement Tiger's variables, we will need to implement let-bindings, variable reference expressions, and assignment expressions. Let us start simple, by implementing variable references and assignments:

```
rules
  // variable references are lvalues in Tiger, and lvalues are exps
  evalExp(ExpLValue(LValueVar(Var(x)))) = #ref-fetch(var(x))

  // we will only consider assignment to variables for now
  evalExp(Assign(LValueVar(Var(x)), exp)) = {
    v <- evalExp(exp)
    #ref-store(var(x), v)
  }
```

The `var(...)` construct will convert the given constant string into a direct reference to a variable in Tim. This allows us to construct a corresponding Tim variable for every Tiger variable. Since Tim's scoping rules are similar to Tiger's, representing each Tiger variable as a Tim variable yields us correct shadowing behavior for free. Since Tim variables are immutable, we need to wrap each variable in a reference using the `#ref-` family of primitives, which provide access to mutability by storing the value on the heap through an indirection.

Note that Tiger is slightly odd in that certain expressions (e.g. assignment, loops, break) produce no value. Even though static analysis will ensure that the "values" produced by these expressions can never be read, our definition of `evalExp` still requires us to return a value. For this implementation, we simply return whatever `#ref-store` returns, which happens to be the value stored in the ref (`v`).

Before we implement let-bindings, we first need to implement some helper functions. The body of a let-binding may contain any number of expressions. We will need to evaluate all of them, but we are only interested in the result of the last expression. For this we will use a helper rule, `evalExps`, that will recursively invoke itself and yield only the last result.

```
rules
  // evalExps :: [type signature omitted]
  evalExps([a]) = evalExp(a)
  evalExps([hd|tl]) = {
    evalExp(hd) // eval head
    evalExps(tl) // and recursively eval tail
  }
```

Recursive invocations are the main method for implementing operations on lists in Dynamix. Pattern matching allows us to unfold the list of expressions one at a time until we reach the list with a single element left. Note that this implementation will fail during compilation if we attempt to compile an empty list of expressions, as there is no implementation of `evalExps` that matches an empty list. This is fine for our use case, as we will assume that the preceding static analysis phase of compilation rejects such invalid programs.

A Tiger let-binding allows for defining multiple `Dec`(laration)s at the same time. Let us also define a helper `evalDecs`, which simply delegates to `evalDec` (which we will define later):

```
rules
  // evalDecs :: [type signature omitted]
  evalDecs([]) = hole
  evalDecs([hd|tl]) = {
    evalDec(hd)
    evalDecs(tl)
  }
```

New in this snippet is the `hole` syntax. Roughly speaking, this syntax represents a *lack of computation*, or a placeholder. Using `hole` effectively tells Dynamix "please fill this with the computation that you have determined should come after the current computation". For this snippet, it means that after we have evaluated all declarations, we should continue at whichever operation comes after the invocation of `evalDecs` (presumably the body of the let-binding). We discuss the `hole` abstraction in more detail in Section 6.2.

We can now go ahead and implement `evalDec`. Since we are only concerned with variable declarations, we will opt to ignore the function declaration construct for now. For each variable declaration, we will evaluate its value, then introduce a new Tim let-binding with the value:

```
rules
  // evalDec :: [type signature omitted]
  evalDec(VarDec(n, _, exp)) = evalVarDec(n, exp)
  evalDec(VarDecNoType(n, exp)) = evalVarDec(n, exp)

  // evalVarDec :: [type signature omitted]
  evalVarDec(name, exp) = {
    v <- evalExp(exp)
    ref <- #ref-new(v)
    let var(name) = ref in
      hole
  }
```

Note that we again use the `hole` syntax. This allows us to "emit" a let-binding as a standalone operation, without needing to know what should be the body of the binding[5]. Note that we wrap the value inside a ref, as discussed earlier.

With all helper functions implemented, the rule for the let-expression itself becomes trivial:

```
rules
  evalExp(Let(decs, body)) = {
    evalDecs(decs)
    evalExps(body)
  }
```

[5]We're talking about a Tim let-binding here, not to be confused with a Tiger let-binding. Note the orange syntax highlighting.

6.1.5 Functions

Let us now consider the other form of declaration that can appear in a let-binding: functions. Since Tiger functions are allowed to be mutually recursive when defined in the same let-binding, we have to take a slightly different approach than what we did with variable declarations. Let us first define some boilerplate:

```
rules
  // evalFuns :: [type signature omitted]
  evalFuns([]) = []
  evalFuns([h|t]) = {
    hh <- evalFun(h)
    tt <- evalFuns(t)
    [hh|tt]
  }

  // evalFun :: [type signature omitted]
  evalFun(FunDec(name, args, _, body)) = evalFunImpl(name, args, body)
  evalFun(ProcDec(name, args, body)) = evalFunImpl(name, args, body)
```

The highlighted line shows our first use of *meta-lists*. As the name suggests, these are lists that exist purely during the Dynamix specification evaluation. Meta-lists are mainly used for working with unknown quantities of constructs, and are accepted by Dynamix features like function definitions, continuation calls, and calls to certain primitives. For our use case, we will build a meta-list of function definitions, so that we may insert them all inside a single Tim `fix` block (Tim supports mutual recursion between functions defined in the same fix block).

Before we can consider the implementation of `evalFunImpl`, we will need to define some more boilerplate. In particular, we will need to be able to turn a list of `FArgs`, the AST node representing a function argument, into a list of Tim variables (for declaring the function). In order to allow assignment to function arguments, we will also need to wrap each `FArg` in a `#ref`.

```
rules
  // fargsToArgNs :: [type signature omitted]
  fArgsToVars([]) = []
  fArgsToVars([FArg(n, _)|t]) = [var(n)|fArgsToVars(t)]

  // wrapFArgsInRefs :: [type signature omitted]
  wrapFArgsInRefs([]) = hole
  wrapFArgsInRefs([FArg(n, _)|t]) = {
    ref <- #ref-new(var(n))
    // shadow the formal argument with a new ref
    let var(n) = ref in
      wrapFArgsInRefs(t)
  }
```

Note that we again use the `hole` construct to avoid having to pass the body of our function as argument. With these helper functions defined, we can now define `evalFunImpl`:

```
rules
  // evalFunImpl :: [type signature omitted]
  evalFunImpl(name, args, body) = {
    n <- var(name)
    return <- fresh-var(return)
    argNames <- fArgsToVars(args)

    fun n(argNames ++ [return]) = {
      wrapFArgsInRefs(args)
      v <- evalExp(body)
      return@([v])
    }
  }
```

Since Tim is a CPS language, we need to introduce an explicit return continuation. To do so, we allocate a fresh (i.e. guaranteed to be unique) return variable, add it to the list of arguments, and explicitly invoke it as tail call with the final result of the function body.

Now we can implement our function definition handling for let-bindings. Our grammar is set up such that sequences of functions within a let-binding are a single `FunDecs` node. We simply need to compile each function declaration with the list, then place them all in a single `fix` block in order to support mutual recursion:

```
rules
  evalDec(FunDecs(fundecs)) = {
    funs <- evalFuns(fundecs)
    fix {
      funs
    }
  }
```

This is all we need. The remainder of the let-binding logic was already taken care of when we implemented the variable declarations, so by just implementing `evalDec` for a `FunDecs` instance is enough to extend our let-binding support to functions.

To finalize our let-declaration rule, we need to add an additional `evalDec` implementation to handle type declarations. Recall that the Dynamix interpreter will abort compilation if a rule is invoked with arguments that do not match any implementation. Even though type declarations do not compile to anything, we still need to handle them in a `evalDec` implementation to avoid failing compilation when we encounter one. We simply add an implementation that accepts the wildcard pattern _:

```
rules
  evalDec(_) = hole
```

As mentioned in Section 6.1.1, Dynamix evaluates rule patterns in order of *specificity*. The wildcard pattern, _, is the least specific pattern. As a result, this fallback rule will only be evaluated if none of the other rules match, regardless of where it is located in the file. For the Tiger AST, this comes out to only type declarations.

We finish our function implementation by implementing calls:

```
rules
  // evalExpList :: [type signature omitted]
  evalExpList([]) = []
  evalExpList([h|t]) = {
    v <- evalExp(h)
    tv <- evalExpList(t)
    [v|tv]
  }

  evalExp(Call(id, args)) = {
    {
      argvs <- evalExpList(args)
      tgt <- var(id)
      tgt@(argvs ++ [after])
    } label after/1:
  }
```

We again use the `label` construct to introduce a new continuation, and pass this as the return continuation to the function we call. The meta-list of arguments passed to our tail-call will correspond to formal parameters when compiled to Tim.

6.1.6 Control flow

Let us now consider the implementation of control flow constructs in Dynamix. For the sake of brevity, we will only discuss the implementation of the `while` statement and the accompanying `break` expression.

Let us start by declaring a new *scoped meta-global*, `$break`. Meta-globals allow us to share data with other rule implementations, without manually passing it around. For our purposes, we will be using the meta-global to store the continuation used to break out of the loop, so that we can tail-call it when we implement the break expression.

```
rules
  // Meta-globals start with a $ and must be declared before use.
  // '@cval' is the type of the global. We can ignore it for now.
  $break :: @cval
```

We implement the while-loop by introducing a new function that contains the loop condition. When a non-zero (i.e. truthy) value is encountered, we execute the body and unconditionally tail-call the same function again. If the value is zero (i.e. falsy), we instead tail-call the continuation representing the end of the loop:

```
rules
  evalExp(While(cond, body)) = {
    {
      bdy <- fresh-var(body)

      fix {
        fun bdy([]) = {
          v <- evalExp(cond)
          if #int-neq(v, int('0)) then {
            with $break = after do
              evalExp(body)
            bdy@([])
          } else {
            after@([])
          }
        }
      }

      bdy@([])
    } label after/0:

    // Return a dummy value.
    int('0)
  }
```

We assign a value to the scoped meta-global `$break` we previously declared using the `with` syntax, updating the value of the meta-global within the body of the binding. After the body finishes execution, the value will be restored to the value it had prior. This allows us to support nested loops and still invoke the appropriate continuation in a `break` expression, and is the reason why we refer to Dynamix's meta-globals as "scoped".

We can now trivially implement our break expression by simply tail calling the `$break` meta-global, which we have bound to the appropriate `after` continuation. Note that no other language constructs need to concern themselves with the possibility that a break may occur, even though the expression might be arbitrarily nested.

```
rules
  evalExp(Break()) = $break@([])
```

6.1.7 Records

The final Tiger language feature we will discuss is records. Perhaps surprisingly, we will not need any new Dynamix features to implement them. The Tim `#record-` family of primitives, which we can trivially call in our Dynamix specification, are enough to implement records. Let us first consider the implementation of record literals:

```
rules
  // evalFields :: [type signature omitted]
  evalFields([]) = []
  evalFields([InitField(name, val)|rest]) = {
    v <- evalExp(val)
    r <- evalFields(rest)
    [str(name), v|r]
  }

  evalExp(Record(_, fields)) = {
    fv <- evalFields(fields)
    #record-new(fv)
  }
```

This is a straightforward implementation which calls the `#record-new` primitive with alternating field names and their values, transforming the expression `Foo{ a = 1, b = 2 }` into `#record-new("a", 1, "b", 2)`. The `#record-new` primitive is defined to take a meta-list of arguments, which is transformed into a concrete list when compiled to Tim (similarly to how e.g. tail calls accept a meta-list of arguments). It is worth pointing out that meta-lists in Dynamix are homogeneous. Both the `str(x)` literals, as well as the evaluated values of each field, are Tim `cval` instances, and thus are allowed to be combined in the same meta-list[6]. We ignore the type name specified in the record literal. Since there are no language features in Tiger that support querying the type of a value at runtime, we are under no obligation to carry this information around at runtime.

Our implementation of field reading and writing is trivial. We just need to refactor some things to account for the fact that we now also support field lvalues:

```
rules
  // evalLValue :: [type signature omitted]
  evalLValue(LValueVar(Var(x))) = #ref-fetch(var(x))
  evalLValue(FieldVar(lv, member)) = {
    v <- compileLValue(lv)
    #record-read(v, str(member))
  }

  evalExp(Assign(FieldVar(lv, member), exp)) = {
    indexee <- evalLValue(lv)
    v <- evalExp(exp)
    #record-write(indexee, str(member), v)
  }

  // we replace the previous definition for variable reading:
  //   evalExp(ExpLValue(LValueVar(Var(x)))) = #ref-fetch(var(x))
  // in favor of
  evalExp(ExpLValue(lv)) = evalLValue(lv)
```

[6]In other words, the Dynamix type system keeps track of whether something represents a Tim cval, but not the exact type of the value produced by this cval.

Note that we do not wrap record fields in `#ref`s, as we did with variables. This is because records provide interior mutability for their fields. Generally, Tim data types are mutable if their data is stored behind an indirection (e.g. a record, an array, or a heap-allocated reference). We explicitly wrap local variables and arguments in `#ref`s to introduce this indirection.

6.1.8 Interacting with static analysis

Finally, let us consider a situation where we need to interact with the results of static analysis. Tiger lacks any language features that *require* the use of static analysis, but there are other ways in which we can use Dynamix's ability to interact with static analysis results. Currently, we unconditionally wrap local variables in `#ref`s in order to provide mutability. If we were able to determine exactly which subset of variables is actually assigned to, we would be able to limit our reference insertion to only these variables.[7] Dynamix itself is not nearly powerful enough to perform this analysis directly, but it is possible to perform this analysis as part of a Statix [5] specification. We will use Dynamix's ability to query the results of static analysis, allowing us to use this information within our specification.

We will assume that our Statix specification assigns the `ref` property (see Section 2.2.2) on every use of a variable, pointing back to the definition. We will also assume that the `writtenTo` property is assigned a value on this declaration if it is written to at any point in the program. An absence of this property will indicate that it is never written to. Let us first update `evalVarDec` accordingly:

```
signature
  constraint-analyzer
    // Declare the properties.
    property ref :: 'string
    property writtenTo :: 'int

rules
  // we replace our previous implementation of evalVarDec with
  evalVarDec(name, exp) = {
    v <- evalExp(exp)
    ref <- wrapInRefIfNeeded(writtenTo(name), v)
    let var(name) = ref in
      hole
  }

  // wrapInRefIfNeeded :: [type signature omitted]
  wrapInRefIfNeeded([], x) = x
  wrapInRefIfNeeded(_, x) = #ref-new(x)
```

When we declare a new constraint analyzer property, it will introduce a function with the same name that accepts any source AST term. Invoking the function will attempt to look up the Statix analysis results for the appropriate node, and retrieve the property value. The resulting value is an empty meta-list if the property did not appear on the node, or a non-empty list if the node was assigned one (through `:=`) or multiple (through `+=`) values. For this implementation, we use the pattern matching facilities of a helper rule to distinguish

[7]This is (intentionally) a somewhat silly example. An optimizing compiler for Tim would likely have no problem optimizing away extraneous references, which would make this a classic case of premature optimization.

between the case where no `writtenTo` property was on the node (and therefore no `#ref` is needed), and the case where there was a value (and therefore a `#ref` is needed). We will also need to adjust `wrapFArgsInRefs` in a similar way, but we will omit that implementation here for the sake of brevity.

For variable references, we will use the `ref` property to resolve the declaring node. We will then query the `writtenTo` on this property to determine whether we need to "unwrap" the value of the variable or not.

```
rules
  // we replace our previous implementation of evalLValue(LValueVar(...)) with
  evalLValue(LValueVar(Var(x))) = unwrapVarHelper(ref(x), var(x))

  // unwrapVarHelper :: [type signature omitted]
  unwrapVarHelper([refNode], x) = unwrapRefIfNeeded(writtenTo(refNode), x)

  // unwrapRefIfNeeded :: [type signature omitted]
  unwrapRefIfNeeded([], x) = x
  unwrapRefIfNeeded(_, x) = #ref-fetch(x)
```

Since the only method of conditionally invoking expressions in Dynamix is rule pattern matching, we need to introduce two helper functions here. The first, `unwrapVarHelper`, unwraps the result of the `ref` property lookup (we assume that a variable always has a single ref). This then delegates to the second helper, which conditionally returns either the input, or fetches the value from the input ref. This is everything we need in order to only conditionally wrap variables in `#ref` values.

6.2 Accessible CPS through abstractions

In our interactive introduction to Dynamix, we came across the `hole` expression several times. In this section, we will take a look at how this expression, alongside several others, allows us to greatly simplify the process of writing a specification that targets a CPS language like Tim.

6.2.1 Creating CPS fragments with holes

One of the core difficulties of working with a CPS target language is that constructing a target expression requires us to specify the entire remainder of the execution, up until a tail call. After all, it is impossible for us to construct a Tim expression that does not eventually tail call, as its grammar was designed to enforce the properties of CPS (see also Section 5.2). This property makes it impossible for a compiler to consider the compilation of a single expression *in isolation*.

To illustrate this, consider the following pseudo-code snippet for the compilation of integer addition in a hypothetical compiler that uses Tim as target language:

```
function compileAdd(left: Exp, right: Exp) =
  TPrimitiveExprCall(
    "int-add",
    compileExp(left),
    compileExp(right),
    newUniqueName(),
    cont // what to put here?
  )
```

We cannot create a call to `#int-add`, as we have no suitable value for `cont`. This is by design. After all, `cont` represents the continuation of the program, something which we explicitly do not know since we are compiling the expression in isolation.

The easiest solution to this issue would simply be to not compile expressions in isolation. By instead modeling our compiler as a right fold, we can ensure that we compile later expressions before we compile earlier ones. However, this approach becomes more complex when we consider that the bindings of (intermediate) values must flow from left to right, completely opposite the direction of a right fold[8]. Not to mention that traditional (formal) language specifications usually define the semantics of language features in isolation, which means that a folding approach to compilation is a significant departure from the formal specification. It would be beneficial if we could somehow retain the ability to consider the compilation of terms in isolation, without having to give up our CPS target language.

The way Dynamix solves this issue is by deferring the continuation part of a Tim term. It introduces a new expression, `hole`, which acts as a valid continuation when inserted in a Tim term. Through this expression, we are able to create fragments of Tim terms that, instead of ending in a tail call, end with a placeholder. As long as we ensure that we fill this hole with a proper CPS term before emitting the output program, we never violate the rules of the continuation-passing style[9]. Terms that contain a hole are referred to as *pluggable terms*, referring to the fact that we can "plug" the hole in the term by providing it a proper continuation later. Similarly, we refer to CPS terms that do not have a hole in them (and therefore adhere to all the rules of the CPS) as *finalized terms*, referring to the fact that they require no further manipulation. Pluggable terms may only contain a single hole, but this hole does not necessarily need to be in the continuation slot of an expression (e.g. it may also be in the body of a function). Examples of pluggable terms can be seen in Figure 6.3.

Pluggable terms are used throughout the Dynamix meta-language, although they are largely transparent to the user. Calls to primitives for example, implicitly produce a Tim primitive call expression with a hole as continuation. Some of Dynamix's abstractions, such as the `label` syntax briefly discussed in Section 6.1.3, are also simply syntactic sugar over the hole expression, as can be seen in Figure 6.4.

It is important to restate that the `hole` expression is an abstraction that purely exists on the meta-level. Since `hole` allows us to trivially violate the CPS-by-construction guarantees that

[8]In order to compile an earlier expression, we must have already compiled the later ones so that we can pass them as continuation (they flow right-to-left). However, to compile later expressions we must know the names of the variables to which the results of earlier expressions are bound (they flow left-to-right). This means that a trivial left- or right-fold is generally not enough to implement compilation with Tim as target. There are ways to make this work of course, but they generally involve quite a bit more boilerplate.

[9]This approach of leaving the continuation unspecified and filling it in later is similar to the technique of backpatching used in many compilers to defer the exact addresses of jump targets. An excellent introduction to backpatching can be found in the "Dragon book" [2].

```
if #int-eq(a0, 10) then
  #print("Illegal argument.") => tmp0;
  #exit()
else
  hole
```

```
fix {
  fun c0(a0) =
    #print(a0) => tmp0;
    hole
} in
  c0(10)
```

Figure 6.3: Two examples of pluggable terms, representing a fragment of a Tim program with a continuation left unspecified. The hole expressions are highlighted.

```
{
  if #int-eq(int('1), int('1)) then
    after@([])
  else
    #exit()
} label after/0:
```

```
fix {
  fun c0() =
    hole
} in
  if #int-eq(1, 1) then
    c0()
  else
    #exit()
```

Figure 6.4: The `label/0` abstraction from Dynamix is simply a syntactic sugar that produces a new continuation with a hole as body. Composition (Section 6.2.2) will ensure that any code after the label will be placed inside `c0`.

Tim offers, we must eliminate each occurrence of `hole` as part of the specification-guided compilation process. If the final term produced by a Dynamix specification still contains a hole, it will be plugged by a call to `#exit()` to fulfill this requirement. This operation is generally appropriate, because `hole` is a substitute for the remainder of the program. A hole in the final program should therefore be replaced with whatever comes *after* running the program, which is a graceful exit.[10]

[10]Earlier iterations of Dynamix would raise an error if compilation produced a pluggable term instead of a full CPS term. This was not very user-friendly, since understanding the error required understanding the entirety of the hole abstraction. Since the final operation is almost always `#exit()`, we opted to instead automatically plug the hole. Users can still manually plug the hole in their specification if they desire a different behavior.

6.2.2 Composing pluggable terms

Pluggable terms and the use of hole allow us to model the compilation of language features in isolation, but they produce fragments of CPS programs that have no meaning on their own. We need some way of *composing* several pluggable terms together, so that we can gradually build a target program from these individual fragments.

Composition of pluggable terms is a core concept in the Dynamix meta-language. Within a Dynamix block (`{ stmt* }`) the result of each statement is composed with the next, as if there was a right-associative[11] composition operator inserted between each pair of adjacent statement.[12]. This composition operator, denoted $a \bowtie b$, "plugs" the hole in a by replacing it with the contents of b, yielding either a larger pluggable term (if b was pluggable), or a complete CPS term (if b was already a complete CPS term). An example of composition being applied can be seen in Figure 6.5.

Let us consider this composition by taking a look at how we might write a specification for a list of statements in Dynamix:

```
rules
  evalStmts([]) = hole
  evalStmts([h|t]) = {
    evalStmt(h)
    evalStmts(t)
  }
```

[11]Technically, it does not matter whether composition is left- or right-associative, since the behavior is the same in both cases. We model it as right-associative here, as that is how Dynamix implements it.

[12]Readers familiar with functional programming languages, specificall Haskell's implementation of monadic composition, may recognize this approach as being quite similar to `do`-blocks. This similarity actually goes beyond just the implicit composition, as we will see later in this chapter.

```
fix {
  fun c0(a0) =
    hole
} in
  print("Hello, world!", c0)
```

```
let a0 = "den Berg" in
  let a1 = "van " in
    #string-add(a1, a0) => a2;
    #record-new("lastname", a2) => a3;
    hole
```

```
fix {
  fun c0(a0) =
    let a0 = "den Berg" in
    let a1 = "van " in
      #string-add(a1, a0) => a2;
      #record-new("lastname", a2) => a3;
      hole
} in
  print("Hello, world!", c0)
```

Figure 6.5: The result of composing two pluggable terms by substituting the second term in the hole of the first term. The background colors indicate from which input term the line originated. Terms are inserted directly, i.e. we do not perform capture-avoiding substitution. Note that the resulting composed term is itself a pluggable term, since it contains a `hole` expression.

We will use the symbolic input `evalStmts([a, b, c])`. First, we inline the recursive invocations to `evalStmts`, yielding the following:

```
rules
  evalStmts([a, b, c]) = {
    evalStmt(a)
    evalStmt(b)
    evalStmt(c)
    hole
  }
```

Next, we remove the block and instead explicitly insert the composition operator between each consecutive statement, remembering that this operator is right-associative:

```
rules
  evalStmts([a, b, c]) =
    evalStmt(a) ⋈ (evalStmt(b) ⋈ (evalStmt(c) ⋈ hole))
```

From this, it becomes clear that the computations for each individual statement are composed together to form a larger pluggable term. The final composition with `hole` is effectively a no-op, as it simply replaces a hole with a new hole. This means that the output of `evalStmts([a, b, c])` is equivalent to performing the operations from `evalStmt(a)`, followed by the computations for `evalStmt(b)`, and finally by those for `evalStmt(c)`.

The core benefit of the automatic composition between statements in a Dynamix block is that it makes linear control flow once again implicit. This tackles one of the core issues also identified in Chiel Bruin's Dynamix thesis [13], which is that a pure CPS target language is often cumbersome to work with because it enforces explicit control flow even in the cases where implicit control flow would suffice. At the same time, the composition approach formalizes the understanding from formal specifications that operations are performed exactly in the order in which they are listed in the specification. Automatic composition also means that the `hole` expression gives us a natural way to state that no operation should be performed, but that future control flow should continue at the position of the hole, without needing to introduce a special-cased no-op construct.

One aspect of composition worth considering in more detail is the behavior of the ⋈ operator when the left-hand side is a finalized CPS term. One would expect this operation to be illegal. After all, the left-hand side lacks a hole in which the right-hand side can be inserted. However, it turns out that it is trivial for specifications to encounter this situation. Consider the example in Figure 6.6. If a branch of an if-statement contains a return statement, then invoking `evalBlock(b)` will yield a completed CPS term. This term will then be attempted to be composed with the tail call to `after`.

The particularity here is that `evalBlock` can yield both pluggable terms (if there is no early return) and finalized CPS terms (if there is an early return). This makes it impossible to guard against the scenario where we attempt to compose two finalized CPS terms, since we simply must include the unconditional tail call to `after` for the cases in which `evalBlock` yields a pluggable term.[13] To resolve this conflict, we define composition such that the right-hand

[13]The bodies of conditional primitive calls must produce finalized terms. Allowing them to return pluggable terms could lead to a situation where a pluggable term contains two holes, which would violate our assumption that pluggable terms have exactly one hole.

```
rules
  evalStmt(If(cond, ifThen, ifElse)) = {
    condv <- evalExp(cond)
    if #int-eq(condv, int('1)) then {
      evalBlock(ifThen)
      after@([])
    } else {
      evalBlock(ifElse)
      after@([])
    }
  } label after/0:

  evalStmt(Return()) = $return@([])
```

Figure 6.6: A situation in which two finalized terms may be composed with each other. The if-statement unconditionally jumps to the code afterwards, but the body of the statement may perform an early return, therefore yielding a finalized CPS term. We must discard the tail call that occurs later to preserve correct behavior.

side of composition will be discarded if the left-hand side is already a finalized CPS term. While this may sound as if it could lead to an unintentional loss of computations, the case studies performed with Dynamix have shown that this approach almost always reflects the user's intended behavior for the specification.

This informal introduction to composition as a method for gradually building CPS terms lacks some finer details of the composition. We omit these details, which involve the behavior of the composition operator for other types of values (such as meta-level values like meta-lists) and the mechanics for type-checking composition, because they are not essential in grasping the concept of composition. They are fully discussed when we consider the formal semantics of the Dynamix meta-language in Chapter 7.

6.2.3 Attaching values to pluggable terms

The observant reader may have noticed that the `hole` expression alone is not actually sufficient to implement some of the abstractions we have used to write our Tiger specification. Consider integer addition, implemented in Tim as `#int-add(a, b) => c; d`. Note that this node does not directly produce a value (as is common behavior for expressions in CPS languages). Instead, it *binds* the result of the addition to `c`. This distinction is important, because it means that the pluggable term of the call to `#int-add` is not enough to be used on its own: anything that wants to make use of the result also needs to know to which identifier the result was bound.

Dynamix solves this issue through "paired pluggables". A paired pluggable, denoted $\langle v, t \rangle$, is a pluggable term t that has been paired with value v. What distinguishes paired pluggables from an arbitrary pair is that Dynamix will enforce that value v can only ever be used within the hole of term t. This allows the value-half of the paired pluggable to have a *dependency* on t, such as when t contains the terms necessary to compute v.[14] Our earlier example of integer addition is representable as the following paired pluggable:

[14]Within Dynamix, the majority of paired pluggables represent the result of some computation, alongside the terms needed to produce this value. But this is not a requirement. The value-half of a paired pluggable can be completely separate from the term-half.

$$\langle c, \texttt{\#int-add(a, b) => } c\texttt{; d}\rangle$$

Having a dedicated data structure for values associated with a pluggable term gives us the ability to enforce the requirement that a value may only be used in a context where the pluggable term has been "included". In particular, the only method for extracting the value from a paired pluggable is by using the `<-` operator. This operator is only valid in Dynamix blocks, and the name bound to the value is only available for subsequent statements. By virtue of the implicit composition within blocks, the statements for which the value is in scope are guaranteed to be composed with the computation for that value. Consider the following example:

```
rules
  evalExpr(Add(a, b)) = {
    av <- evalExpr(a)
    bv <- evalExpr(b)
    #int-add(a, b)
  }
```

We first desugar the `<-` operator into a binding for the value-half and an explicit composition for the term-half:

```
rules
  evalExp(Add(a, b)) = {
    ⟨tmp0, tmp1⟩ := evalExp(a)
    {
      tmp1
      av := tmp0
      ⟨tmp2, tmp3⟩ := evalExp(b)
      {
        tmp3
        bv := tmp2
        out := fresh-var(out)
        ⟨out, (#int-add(av, bv) => out; hole)⟩
      }
    }
  }
```

As we can see, each use of the `<-` operator creates a new scope for the remainder of the block. This ensures that the bindings `av` and `bv` are only visible to statements located after the arrow operator. The pluggable halves of the pairs, `tmp1` and `tmp3`, have already been composed (by virtue of residing inside a Dynamix block) before the bindings are introduced, so we are guaranteed that the values to which `av` and `bv` point have been initialized.

We can continue desugaring this by removing the blocks in favor of explicit composition. We will also simplify the program a little by eliminating the `tmp0` and `tmp2` bindings in favor of simply directly binding `av` and `bv`:

```
rules
  evalExp(Add(a, b)) =
    ⟨av, tmp1⟩ := evalExp(a);
    ⟨bv, tmp3⟩ := evalExp(b);
    out := fresh-var(out);
    ⟨out, tmp1 ⋈ tmp3 ⋈ (#int-add(av, bv) => out; hole)⟩
```

From this, we can clearly see that the `<-` operator, combined with implicit composition, allows us to cleanly combine the computations needed to produce `av` and `bv` with the call to `#int-add`. Note that the original call to `#int-add` has also been desugared into a paired pluggable, allocating a new fresh variable for the result of the addition operator. This way, the user need not concern themselves over where the results are stored, nor about correct composition of the terms.

Paired pluggables are also used to implement the `label/1` abstraction. This abstraction is very similar to that of the `label/0` construct shown in Figure 6.4, except that it yields a paired pluggable instead of just a pluggable term. This pair consists of the generated continuation (now accepting a single argument instead of none) and the argument name, so that the `<-` operator may be used to retrieve the "result" of the continuation call.

Paired pluggables are not restricted in what types of values they may contain. While Tim `cval` terms (e.g. integer literals, variable references) are most common, meta-level values like meta-lists may also be contained in paired pluggables (in such a pair, all values in the meta-list may have a dependency on the computation half of the pair). Several rules defined in the Dynamix specification for Tiger, such as `evalExpList` in Section 6.1.5, use this capability. We formally define exactly how paired pluggables interact with composition in Chapter 7.

6.2.4 Enforcing CPS using types

Finally, we will briefly discuss Dynamix's type system. Beyond asserting that name binding, rule definitions, and pattern matching operations are valid, the Dynamix type system is a core component of the meta-language that ensures that specifications will always produce complete CPS terms. In particular, the type system encodes the behavior of the composition operator and how it manipulates pluggable CPS terms.

Types in Dynamix can be grouped into three categories. *Source types* are prefixed with `'` and represent the signature of the input AST. They are derived from the algebraic signature of the input language and are generally automatically generated by Dynamix as part of the Spoofax build process. *Target types* are prefixed with `@` and represent different types of Tim AST nodes (recall that Tim terms are first-class values in Dynamix). Finally, meta-types lack a specific prefix and represent the types of values that only exist during evaluation of the specification and have no concrete representation in either the source or the target language.

Most of the Dynamix meta- and target types correspond directly with abstractions discussed earlier in this chapter. A brief overview of the different sorts and their meanings:

- `'<source-type>`: A type describing the structure of an ATerm value, such as `string`, `int`, or a user-defined algebraic signature.
- `List(T)`: A meta-list consisting of elements of type `T`.
- `Pluggable`: A Tim CPS expression term that *may* contain a hole. Unlike pluggable terms, which are guaranteed to contain exactly one hole, values of type `Pluggable` may either be pluggable terms or finalized terms.
- `Pluggable(T)`: A paired pluggable $\langle v, t\rangle$ where v is of type `T` and t is of type `Pluggable`. Using the arrow operator `<-` on a value of type `Pluggable(T)` produces a binding with type `T`.
- `@cval`: A Tim `cval` AST node, such as a variable reference, string literal, or integer literal. Allowed to be used as arguments to Tim tail calls and primitives.
- `@cexp`: A Tim `cexp` AST node. Values with this type are guaranteed not to contain any holes.
- `@cfun`: A Tim `cfun` AST node. Only valid when used in a `fix` expression. We treat Tim function terms as first-class values to allow defining an arbitrary number of functions inside a single `fix` block.

While source types are largely used to assert that pattern matching operations are correct, meta and target types are vital in ensuring that a Dynamix specification produces a Tim AST that adheres to the rules of the continuation-passing style. To see why this is the case, let us consider the type signature of a call to `#int-add`:

```
// #int-add :: @cval * @cval -> Pluggable(@cval)
#int-add(a, b)
```

The return type, `Pluggable(@cval)`, indicates that the primitive returns a paired pluggable, where the value half of the pair is of type `@cval`. This corresponds to what we already observed in Section 6.2.3: expression primitives return a paired pluggable consisting of the variable to which the result was bound, and the Tim AST `cexp` node for invoking the primitive. To enforce the continuation-passing style (CPS), we require that both arguments to `#int-add` are instances of `@cval`. Since values of type `@cval` represent Tim `cval` nodes, this ensures that the arguments to our primitive call will be side effect free.

To enforce that functions always tail call, Dynamix requires that the bodies of functions, let bindings, and the if-then-else are of type `@cexp`. This type represents a finalized CPS expression, and can only be created by performing a (primitive) tail call or by composing a pluggable term with a finalized term. An example showing how this enforces proper tail calls can be seen in Figure 6.7.

The behavior of the implicit composition between statements in a Dynamix block is modeled in the type system by performing a similar composition operation on the types of the statements. For instance, `Pluggable` $\bowtie$ `Pluggable` $=$ `Pluggable`, and `Pluggable` $\bowtie$ `@cexp` $=$ `@cexp`. The exact rules for this composition are discussed in Chapter 7.

```
rules
  evalBlock :: 'Block -> Pluggable

  evalExpr(Lambda(name, body)) = {
    return <- fresh-var(return)
    fix {
      fun var(name)([return]) =
        // Invalid: The body of a function must be @cexp. evalBlock returns
        // Pluggable, which is not guaranteed to be a completed CPS term.
        evalBlock(body)

      fun var(name)([return]) = {
        // Valid: unconditional tail calls are of type @cexp.
        evalBlock(body)
        return@([])
      }
    }
    var(name)
  }
```

Figure 6.7: An example showing how Dynamix's type system enforces that the rules of CPS are followed. The body of the declared `fun` must be of type `@cexp`, which can only be created by performing a tail call. Since `evalBlock` returns a pluggable term, which by definition is not a finalized CPS term, it is rejected by the type system.

In order to make it easier to write specifications in Dynamix, the type system offers some quality-of-life features for common situations. In particular, several types can be implicitly coerced to other types in situations where this coercion can be automatically deduced. These coercions include:

- Values of type `T` can be automatically coerced to `Pluggable(T)`, by implicitly constructing $\langle v, \texttt{hole} \rangle$. This allows us to use expressions like `var(x)` (which returns an `@cval`) in contexts expecting a `Pluggable(@cval)`.
- Values of type `Pluggable(T)` can be automatically coerced to `Pluggable`. We can simply discard the value half of the pair, without impacting the behavior of the program.
- Values of type `@cexp` can be coerced to `Pluggable`. This follows trivially from the definition of `Pluggable`, which states that it represents either a pluggable or a finalized CPS term.
- Values of type `@cexp` can be coerced to `Pluggable(T)`. This is a legal operation, because the unconditional tail call ensures that the value embedded in the paired pluggable can never be read at runtime.

These implicit coercions are frequently used across specifications. Figure 6.8 lists some example cases where these coercions are useful.

```
rules
  evalStmt :: 'Stmt -> Pluggable
  evalExpr :: 'Expr -> Pluggable(@cval)

  evalExpr(Int(i)) = int(i) // coerces @cval to Pluggable(@cval)
  evalExpr(Throw(e)) = {
    v <- evalExpr(e)
    $throw@([v]) // coerces @cexp to Pluggable(@cval)
  }

  evalStmt(Exp(e)) = evalExpr(e) // coerces Pluggable(@cval) to Pluggable
  evalStmt(Return()) = $return@([]) // coerces @cexp to Pluggable
```

Figure 6.8: An example showing how the implicit coercions in Dynamix's type system allow for more expressive specifications, and in which situations each type of coercion might apply.

Chapter 7

Formalizing Dynamix Core

After our informal introduction to Dynamix, we will provide a formal definition for the meta-language in this chapter. The complete Dynamix meta-language has a considerable number of language features, grammar productions, and expressions. This is largely due to the requirement that it must cleanly operate with both the source language (and therefore expose ATerm primitives), as well as the target language (and all of the idioms and features that come along with it). Many of these language features have straightforward semantics, so for the sake of brevity we will not discuss them.

Instead, we will define a subset of Dynamix called Dynamix Core. This subset contains all language features for which the typing rules or implementation is non-trivial, and in particular those powering the CPS abstractions discussed in Section 6.2. We formally define the grammar, dynamic semantics, and static semantics for Dynamix Core, and in some cases elaborate on how these definitions extend to the full Dynamix meta-language.

This chapter serves as a formal definition of the behavior we informally described in Chapter 6. In particular, all details glossed over in the informal introduction are fully specified in this chapter. We also assert some meta-properties of the Dynamix language, such as the exact set of guarantees provided by the type system (we make no attempt at proving these assertions, however). Readers that are particularly interested in the precise semantics of Dynamix or its abstractions will find that this chapter serves as a complete definition of them. Readers that are not directly concerned about the formal semantics of Dynamix can freely skip this chapter, as we do not introduce any new behavior or semantics not already informally discussed.

Terminology
Within this chapter, we use the term **source language** to refer to the language under compilation (i.e. the language for which a runtime semantics specification is written). Similarly, we use the term **target language** to refer to the language to which Dynamix compiles, i.e. Tim. Features and syntax unique to Dynamix (e.g. rules, patterns) may also be referred to as **meta** features, because they appear solely as part of the Dynamix meta-language.

7.1 Grammar

We will first briefly outline the AST grammar for Dynamix Core. Dynamix Core is based on the grammar of the full Dynamix meta-language (documented in Appendix A), but it is not directly compatible with the full meta-language. Instead, certain productions have been simplified so that it is simpler to discuss their semantics. We retain most of the operations that produce target constructs, those that concern the pluggable term abstraction, and

```
program    ::= decl*

decl       ::= rule-impl

rule-impl  ::= ID(⟨ pattern, ⟩*) = expr        pattern-matched rule body

pattern    ::= _                              wildcard
             | ID                             variable pattern
             | ID(⟨ pattern, ⟩*)              ATerm constructor pattern
             | STRING                         string literal pattern
             | INT                            int literal pattern
             | []                             nil meta-list pattern
             | [pattern|pattern]              cons meta-list pattern
             | ID@pattern                     bound pattern

expr       ::= { statement+ }                 block
             | $ID                            meta-global reference
             | ID                             meta-variable reference
             | with ⟨ $ID = expr, ⟩+ do expr  scoped meta-global
             | []                             nil meta-list
             | [expr|expr]                    cons meta-list
             | fresh-var(ID)                  fresh variable
             | var(expr)                      Tim variable from ATerm string
             | expr@(expr)                    Tim tail call
             | #ID(expr)                      Tim primitive call
             | fix { expr }                   Tim fix block
             | fun expr(expr) = expr          Tim function
             | expr label ID/1:               unary label
             | expr label ID/0:               nullary label
             | hole                           hole
             | ID(⟨ expr, ⟩*)                 rule invocation

statement  ::= ID <- expr                     bind term
             | expr                           expression
```

Grammar 7.1: The expression grammar for Dynamix Core.

those needed to provide baseline meta-language features (e.g. meta-variables, rules). The grammar of Dynamix Core can be seen in Grammar 7.1, using the same notation previously described in Section 5.2. Note that this grammar represents the AST structure of Dynamix Core, and that it should not be interpreted as a concrete grammar for the language.

7.2 Dynamic semantics

We first discuss the dynamic semantics of Dynamix Core. As with the definition for Tim in Section 5.3, we will use Kahn-style big-step operational semantics [31] to do so. We will use largely the same notation, but restate it here for the sake of clarity.

```
datatype Value =
    Source of ATerm
  | Plug of CExp
  | Plug2 of CVal * CExp
  | CExp of CExp
  | CVal of CVal
  | CFun of CFun
  | List of MValue list

datatype CExp =
    CExpTailCall of CVal * CVal list
  | CExpPrimitiveTail of string * CVal list
  | CExpPrimitiveExp of string * CVal list * string * CExp
  | CExpLet of string * CVal * CExp
  | CFix of CFun list * CExp
  | Hole

datatype CVal = CValVar of string
datatype CFun = CFun of string * string list * CExp

datatype ATerm =
    AInt of int
  | AString of string
```

Figure 7.1: The signature for values in Dynamix Core, written in Standard ML notation. A `MValue` represents a (meta) value in the Dynamix Core interpreter. A `CExp` represents a Tim `cexp` AST node. A `CVal` represents a Tim `cval` AST node. A `ATerm` represents any valid ATerm node. Tim expressions in a `Plug` or `Plug`$_2$ may contain holes. Tim expressions in a `CExp` must not contain holes.

Our evaluation relation for expressions is modeled as $S, G, D \vdash e \Downarrow v$. Here, S is the store, containing values for local meta-variables. G is the global store, used for meta-globals. D is the set of rule definitions within the program, used to perform calls to rules. For S and G, we will use the notation $S' = S[x \mapsto y]$ to extend S with a new mapping from x to y, possibly shadowing an existing mapping for x. The syntax $S[x]$ retrieves the value bound to x in store S.

Similarly to our Tim specification, we will occasionally use overhead bars, $\overline{v}$, to indicate that some term represents a vector or a vector operation. For instance, $S, G, D \vdash \overline{e \Downarrow v}$ will evaluate the list of expressions $\overline{e}$ to the list of values $\overline{v}$. Similarly, $S' = S[\overline{x \mapsto y}]$ will produce a new store S' that extends S with a list of new mappings, provided that all names in $\overline{x}$ are distinct.

The semantics defined in this chapter only cover valid programs. If some case is not covered by any of the rules, it should be interpreted as being disallowed. Should a language implementation encounter such a situation at runtime (e.g. when a function call matches none of the implementation patterns), an error should be issued and evaluation should halt without producing a result.

7.2.1 Value sorts

Values in Dynamix Core, denoted v, can be one of several sorts. An algebraic signature of these sorts can be seen in Figure 7.1. We will briefly elaborate on each of the runtime value types.

`Source` values represent source AST nodes. They must be ATerm compatible, such that they can represent the full range of possible ASTs produced by an SDF3-based source language. `Source` terms are primarily used in pattern-matching operations.

`Plug` values represent pluggable target Tim AST nodes that contain exactly one `hole` expression. The representation of Tim AST nodes used in this formal definition uses a representation similar to the SDF3 grammar for Tim.

$\texttt{Plug}_2$ values represent paired pluggables. They are a combination of an arbitrary runtime value and a pluggable target Tim AST node. The value half of a $\texttt{Plug}_2$ may contain bindings that are defined in the pluggable half, as the design of Dynamix will guarantee that such bindings are bound in all situations where the value can be used.

`CExp` values represent a finalized target Tim `cexp` node. Unlike `Plug` and $\texttt{Plug}_2$ values, the value of a `CExp` is guaranteed to be a legal CPS expression (and therefore does not contain any holes).

`CVal` values represent a target Tim `cval` AST node. `CVal` values in Dynamix Core only contain `CValVar` instances (i.e. an AST node representing a variable reference), although in the full Dynamix meta-language they may also contain string and integer literals. Arguments to function and primitive calls must be `CVal`s.

`List` values represent a meta-list. Meta-lists are heterogeneous and may contain any type of runtime value. They are exposed to the user as cons-nil pairs, but the implementation does not necessarily have to use this representation. To indicate that a meta-list must be homogeneous of a specific type, we will use the overhead bar notation also used for vectors (e.g. $\texttt{List}([\overline{\texttt{CVal}(v)}])$ requires a meta-list of `CVal` terms, and binds these values to $\overline{v}$).

We will refer to Tim expression nodes that contain a hole as pluggable terms. We will refer to Tim expression nodes that do not contain a hole as finalized (since they cannot be further extended).

7.2.2 Expressions

We begin by defining the dynamic semantics for each expression sort. As mentioned, the evaluation of an expression depends only on the store S, global store G and rule declarations D.

Literals

Meta-variables and meta-globals simply yield the value as declared in the appropriate store. Meta-lists can be created through the empty list literal and the cons expression, which prepends the left element to the right list. The hole literal yields a pluggable term consisting entirely of a hole.

Note that meta-globals initially lack a value, as G is initialized to an empty store. This means that it is illegal behavior to read the value of a meta-global before it has been assigned a value through the `with` statement.

$$\frac{v = G[x]}{S, G, D \vdash \texttt{\$}x \Downarrow v}\ \textsc{V-Global}$$

$$\frac{v = S[x]}{S, G, D \vdash x \Downarrow v}\ \textsc{V-Variable}$$

$$\frac{}{S, G, D \vdash \texttt{[]} \Downarrow \texttt{List}([])}\ \textsc{V-Nil}$$

$$\frac{\begin{array}{c} S, G, D \vdash e_1 \Downarrow v_1 \\ S, G, D \vdash e_2 \Downarrow \texttt{List}([\overline{v_2}]) \end{array}}{S, G, D \vdash \texttt{[}e_1\texttt{|}e_2\texttt{]} \Downarrow \texttt{List}([v_1, \overline{v_2}])}\ \textsc{V-Cons}$$

$$\frac{}{S, G, D \vdash \textbf{hole} \Downarrow \texttt{Plug}(\texttt{Hole}())}\ \textsc{V-Hole}$$

Scoped globals

The `with` expression introduces new bindings in the global store. Bindings are applied all at once, and are only visible in the body of the expression.

$$\frac{\begin{array}{c} S, G, D \vdash \overline{e \Downarrow v} \\ G' = G[\overline{x \mapsto v}] \\ S, G', D \vdash e_b \Downarrow v_b \end{array}}{S, G, D \vdash \textbf{with}\ \overline{\texttt{\$}x = e}\ \textbf{do}\ e_b \Downarrow v_b}\ \textsc{V-With}$$

Tim value expressions

The `fresh-var` expression simply creates a new Tim `cval` term with a fresh name, prefixed with the given name. The `var` expression creates a new Tim `cval` variable with the given name.

$$\frac{\begin{array}{c} n = \text{fresh unique variable name prefixed with } x \\ v = \texttt{CVal}(\texttt{CValVar}(n)) \end{array}}{S, G, D \vdash \textbf{fresh-var(}x\textbf{)} \Downarrow v}\ \textsc{V-Fresh Var}$$

$$\frac{\begin{array}{c} S, G, D \vdash e \Downarrow \texttt{Source}(\texttt{AString}(x)) \\ v = \texttt{CVal}(\texttt{CValVar}(x)) \end{array}}{S, G, D \vdash \textbf{var(}e\textbf{)} \Downarrow v}\ \textsc{V-Var}$$

Tail calls

The tail call expression creates an equivalent Tim term. The target must be a `CVal`, and the argument must be a homogeneous `List` of `CVal` terms.

$$\frac{\begin{array}{c} S, G, D \vdash e_t \Downarrow \texttt{CVal}(v_t) \\ S, G, D \vdash e_a \Downarrow \texttt{List}([\overline{\texttt{CVal}(v_a)}]) \\ v = \texttt{CExp}(\texttt{CExpTailCall}(v_t, \overline{v_a})) \end{array}}{S, G, D \vdash e_t\textbf{@(}e_a\textbf{)} \Downarrow v}\ \textsc{V-Tail Call}$$

Tim fix declarations

The fix expression creates a new Tim fix block with a hole as continuation. The body of the expression must evaluate to a meta-list of function declarations.

$$\frac{S, G, D \vdash e \Downarrow \texttt{List}([\overline{\texttt{CFun}(v)}]) \qquad v = \texttt{Plug}(\texttt{CFix}(\overline{v}, \texttt{Hole}()))}{S, G, D \vdash \textbf{fix \{ } e \textbf{ \}} \Downarrow v} \text{ V-Fix}$$

In order to create such a meta-list of function declarations, the `fun` expression evaluates to a `CFun` instance. The body of the function must evaluate to a value of type `CExp`, and therefore must be free of holes.

$$\frac{S, G, D \vdash e_n \Downarrow \texttt{CVal}(\texttt{CValVar}(v_n)) \qquad S, G, D \vdash e_a \Downarrow \texttt{List}([\overline{\texttt{CVal}(\texttt{CValVar}(v_a))}]) \qquad S, G, D \vdash e_b \Downarrow \texttt{CExp}(v_b) \qquad v = \texttt{CFun}(v_n, \overline{v_a}, v_b)}{S, G, D \vdash \textbf{fun } e_n\textbf{(}e_a\textbf{) = } e_b \Downarrow v} \text{ V-Fun}$$

Primitives

Primitive calls are implemented by generating a fresh variable to which the result will be bound, then yielding a `Plug`$_2$ value consisting of this variable alongside the pluggable term for the primitive call. The argument must be a meta-list of `cval` nodes.

$$\frac{n = \text{fresh unique variable name} \qquad S, G, D \vdash e \Downarrow \texttt{List}([\overline{\texttt{CVal}(v)}]) \qquad v = \texttt{Plug}_2(\texttt{CVal}(\texttt{CExpVar}(n)), \texttt{CExpPrimitiveExp}(x, \overline{v}, n, \texttt{Hole}()))}{S, G, D \vdash \textbf{\#}x\textbf{(}e\textbf{)} \Downarrow v} \text{ V-Primitive}$$

Labels

The label abstractions generate a new Tim function with a hole, inserting the preceding code such that the newly generated function is in scope. The unary label abstraction generates a continuation with a single argument and yields a `Plug`$_2$, whereas the nullary label abstraction generates a continuation without arguments and yields a `Plug`. The preceding term must yield a `CExp` (i.e. unconditionally tail call), such that the resulting term contains exactly one hole.

$$\frac{n_f = \text{fresh unique variable name} \qquad n_a = \text{fresh unique variable name} \qquad S' = S[x \mapsto \texttt{CVal}(\texttt{CExpVar}(n_f))] \qquad S', G, D \vdash e \Downarrow \texttt{CExp}(t) \qquad v = \texttt{Plug}_2(\texttt{CVal}(\texttt{CExpVar}(n_a)), \texttt{CFix}([\texttt{CFun}(n_f, [n_a], t)], \texttt{Hole}()))}{S, G, D \vdash e \textbf{ label } x\textbf{/1:} \Downarrow v} \text{ V-Unary Label}$$

$$\frac{n_f = \text{fresh unique variable name} \qquad S' = S[x \mapsto \texttt{CVal}(\texttt{CExpVar}(n_f))] \qquad S', G, D \vdash e \Downarrow \texttt{CExp}(t) \qquad v = \texttt{Plug}(\texttt{CFix}([\texttt{CFun}(n_f, [], t)], \texttt{Hole}()))}{S, G, D \vdash e \textbf{ label } x\textbf{/0:} \Downarrow v} \text{ V-Nullary Label}$$

Rule calls

Rule calls select the most specific implementation of a given rule that matches the argument patterns. In case no such implementation exists, a runtime error is issued. We define the semantics of pattern matching, as well as the ChooseRule function, in Section 7.2.4.

$$\dfrac{\begin{array}{c} S, G, D \vdash \overline{e_a \Downarrow v_a} \\ (S_f, e_b) = \mathsf{ChooseRule}(D, x, \overline{v_a}) \\ S_f, G, D \vdash e_b \Downarrow v \end{array}}{S, G, D \vdash x\texttt{(}\overline{e_a}\texttt{)} \Downarrow v} \text{ V-CALL}$$

Blocks

We define the semantics of blocks through an inductive definition. For a block containing only a single statement, the block simply evaluates to the result of the statement. If the single expression is an arrow operator, we ignore the binding operation.

$$\dfrac{S, G, D \vdash e \Downarrow v}{S, G, D \vdash \texttt{\{}\ x\ \texttt{<-}\ e\ \texttt{\}} \Downarrow v} \text{ V-BLOCK UNARY BIND}$$

$$\dfrac{S, G, D \vdash e \Downarrow v}{S, G, D \vdash \texttt{\{}\ e\ \texttt{\}} \Downarrow v} \text{ V-BLOCK UNARY EXP}$$

For blocks with more than one expression, the result of evaluating the block is the result of *composing* the result of the expression with the result of the remainder of the block. The composition operator, ⋈, is defined in Section 7.2.3. The bind statement extends the current store with a new binding, and uses this store to evaluate the remainder of the block.

$$\dfrac{\begin{array}{c} S, G, D \vdash e \Downarrow v_e \\ v_b = \begin{cases} v_e & \text{if } v_e \text{ is a } \texttt{Source} \\ v_e & \text{if } v_e \text{ is a } \texttt{List} \\ v_e & \text{if } v_e \text{ is a } \texttt{CVal} \\ v_e & \text{if } v_e \text{ is a } \texttt{CFun} \\ v_e' & \text{if } v_e \text{ is } \texttt{Plug}_2(v_e', t) \end{cases} \\ S' = S[x \mapsto v_b] \\ S', G, D \vdash \texttt{\{}\ \overline{s}\ \texttt{\}} \Downarrow v_s \end{array}}{S, G, D \vdash \texttt{\{}\ x\ \texttt{<-}\ e\ \overline{s}\ \texttt{\}} \Downarrow v_e \bowtie v_s} \text{ V-BLOCK BINARY BIND}$$

$$\dfrac{\begin{array}{c} S, G, D \vdash e \Downarrow v_e \\ S, G, D \vdash \texttt{\{}\ \overline{s}\ \texttt{\}} \Downarrow v_s \end{array}}{S, G, D \vdash \texttt{\{}\ e\ \overline{s}\ \texttt{\}} \Downarrow v_e \bowtie v_s} \text{ V-BLOCK BINARY EXP}$$

7.2.3 Composition

The composition operator, ⋈, is used to combine pluggable terms by substituting `hole` expressions in the term. This composition happens automatically between expressions that reside in a block.

The exact behavior of the composition operator is largely invisible to the user, as composition happens implicitly and unavoidably. It is therefore imperative that the rules of composition lead to expected and intuitive behavior for the user. As a general rule of thumb, the composition operator is designed to propagate the computation that pluggable terms represent across the execution of a specification. Accordingly, a pluggable term can never be lost or discarded, except for when it is composed with a completed CPS term. This ensures that potential computations are not discarded, which would lead to incorrect compilation. We perform hole substitution in the case where both sides of the composition operator contain pluggable terms. This generally leads to the behavior that a user of Dynamix would expect, even if they do not fully grasp the exact mechanics of the composition operator.

In order to define the behavior of the composition operator, we introduce an auxiliary operator $t_1 \oplus t_2$. This operator produces a term t by substituting the `hole` expression in pluggable term t_1 with the term t_2. If t_2 was a pluggable term, the resulting term t is also a pluggable term. If t_2 was a finalized CPS term, t is also a finalized CPS term.

For any non-pluggable values, the composition operator simply yields the right-hand side of the operator. This behavior ensures that the result of the last expression within a block is returned as value, even in cases where no pluggable terms are present:

$$\frac{v_1 \text{ is a } \texttt{Source}, \texttt{CVal}, \texttt{CFun}, \text{ or } \texttt{List}}{v_1 \bowtie v_2 = v_2}\ \textsc{Compose-RHS}$$

If the left-hand side of the composition operator is a finalized term, we discard the right-hand side. We discuss why this behavior is desirable in Section 6.2.3.

$$\frac{}{\texttt{CExp}(t) \bowtie v_2 = v_1}\ \textsc{Compose-Fin}$$

If the left-hand side is a `Plug` term, then we either form a $\texttt{Plug}_2$ if the right-hand side does not contain any (pluggable) AST term, or perform a hole substitution otherwise.

$$\frac{v_2 \text{ is a } \texttt{Source}, \texttt{CVal}, \texttt{CFun}, \text{ or } \texttt{List}}{\texttt{Plug}(t) \bowtie v_2 = \texttt{Plug}_2(v_2, t)}\ \textsc{Compose-Plug Simple}$$

$$\frac{}{\texttt{Plug}(t_1) \bowtie \texttt{Plug}(t_2) = \texttt{Plug}(t_1 \oplus t_2)}\ \textsc{Compose-Plug Plug}$$

$$\frac{}{\texttt{Plug}(t_1) \bowtie \texttt{CExp}(t_2) = \texttt{CExp}(t_1 \oplus t_2)}\ \textsc{Compose-Plug CExp}$$

$$\frac{}{\texttt{Plug}(t_1) \bowtie \texttt{Plug}_2(v_2, t_2) = \texttt{Plug}_2(v_2, t_1 \oplus t_2)}\ \textsc{Compose-Plug Pair}$$

Finally, if the left-hand side is a $\texttt{Plug}_2$ term, then we substitute only the value half for any non-CPS right-hand side. If the right-hand side contains a CPS term, then we substitute appropriately.

$$\frac{v_2 \text{ is a } \texttt{Source}, \texttt{CVal}, \texttt{CFun}, \text{ or } \texttt{List}}{\texttt{Plug}_2(v_1, t) \bowtie v_2 = \texttt{Plug}_2(v_2, t)}\ \textsc{Compose-Plug}_2\ \textsc{Simple}$$

$$\frac{}{\texttt{Plug}_2(v_1, t_1) \bowtie \texttt{Plug}(t_2) = \texttt{Plug}(t_1 \oplus t_2)}\ \textsc{Compose-Plug}_2\ \textsc{Plug}$$

$$\frac{}{\texttt{Plug}_2(v_1, t_1) \bowtie \texttt{CExp}(t_2) = \texttt{CExp}(t_1 \oplus t_2)}\ \textsc{Compose-Plug}_2\ \textsc{Term}$$

$$\frac{}{\texttt{Plug}_2(v_1, t_1) \bowtie \texttt{Plug}_2(v_2, t_2) = \texttt{Plug}_2(v_2, t_1 \oplus t_2)}\ \textsc{Compose-Plug}_2\ \textsc{Pair}$$

7.2.4 Rule declarations and patterns

A Dynamix Core specification consists of a list of rule implementations. A single named rule may have multiple implementations, as long as the argument patterns of these implementations do not overlap per the definition Section 7.4. When a rule is invoked with a list of concrete arguments, these arguments are tested against each implementation's argument patterns in order of specificity (see Section 7.4).

In order to express this functionality, we introduce several helper functions to perform pattern matching and rule selection.

Pattern matching

The function $\mathsf{Match}(p, v) = S$ attempts to match pattern p against value v. This is a partial function, only defined for the cases where this match is successful. In case of success, the returned value S represents the store of variables bound by this pattern. The function $\mathsf{MatchAll}(\overline{p}, \overline{v}) = S$ is a partial function that performs this match operation on a list of input patterns and values. In the case that all patterns match all values, the returned store S is the union of the stores returned by the individual pattern matching operations, as long as these stores as disjoint.

We define MatchAll inductively through two rules:

$$\frac{}{\mathsf{MatchAll}([], []) = \{\}}\ \textsc{MatchAll-Nil}$$

$$\frac{\begin{array}{c} S_1 = \mathsf{Match}(p, v) \\ S_2 = \mathsf{MatchAll}(\overline{P}, \overline{V}) \\ S_1 \bigcap S_2 = \{\} \end{array}}{\mathsf{MatchAll}([p|\overline{P}], [v|\overline{V}]) = S_1 \cup S_2}\ \textsc{MatchAll-Cons}$$

Match is defined through the following set of rules, which outline each of the cases in which a pattern matches. If none of the rules apply to a given pair of pattern v and value v, we state that pattern p does not match value v, and that the value of $\mathsf{Match}(p, v)$ is undefined.

$$\frac{}{\mathsf{Match}(\_, v) = \{\}}\ \textsc{Match-Wildcard}$$

$$\frac{}{\mathsf{Match}(x, v) = \{x \mapsto v\}}\ \textsc{Match-Id}$$

$$\frac{\begin{array}{c} v = \texttt{Source}(\texttt{AConstructor}(x, \overline{v_c})) \\ S = \mathsf{MatchAll}(\overline{p}, \overline{v_c}) \end{array}}{\mathsf{Match}(x\texttt{(}\overline{p}\texttt{)}, v) = S}\ \textsc{Match-Constructor}$$

$$\frac{v = \texttt{Source}(\texttt{AString}(s))}{\mathsf{Match}(s, v) = \{\}}\ \textsc{Match-String}$$

$$\frac{v = \texttt{Source}(\texttt{AInt}(i))}{\mathsf{Match}(i, v) = \{\}}\ \textsc{Match-Int}$$

$$\frac{v = \texttt{List}([])}{\mathsf{Match}(\texttt{[]}, v) = \{\}}\ \textsc{Match-Nil}$$

$$\frac{v = \texttt{List}([v_h|\overline{v_t}]) \quad S_1 = \mathsf{Match}(p_h, v_h) \quad S_2 = \mathsf{Match}(p_t, \texttt{List}(\overline{v_t}))}{\mathsf{Match}(\texttt{[}p_h\texttt{|}p_t\texttt{]}, v) = S_1 \cup S_2} \text{ Match-Cons}$$

$$\frac{S_i = \mathsf{Match}(p_i, v) \quad S = S_i \cup \{x \mapsto v\}}{\mathsf{Match}(x\texttt{@}p_i, v) = S} \text{ Match-Bound}$$

Rule selection

In the evaluation relation $S, G, D \vdash e \Downarrow v$, we define D to be a mapping from rule name to a list of implementations, such that $D[x]$ yields a list of all implementations for rule x. These implementations are denoted $(\overline{p}, e) \in D[x]$, representing the list of argument patterns and the body of the implementation.

The function $\mathsf{FirstMatch}(O, \overline{a})$ returns the first rule implementation $(\overline{p}, e) \in O$ such that the patterns $\overline{p}$ match the arguments $\overline{a}$. This is a partial function, only defined for the cases where such an element exists.

$$\frac{O = [(\overline{p}, e)|\overline{r}] \quad o = \begin{cases} (\overline{p}, e) & \text{if } \mathsf{MatchAll}(\overline{p}, \overline{a}) \text{ is defined} \\ \mathsf{FirstMatch}(\overline{r}, \overline{p}) & \text{otherwise} \end{cases}}{\mathsf{FirstMatch}(O, \overline{a}) = o} \text{ FirstMatch}$$

The function $\mathsf{ChooseRule}(D, x, \overline{v}) = (S, e)$ selects the most specific rule implementation for rule x that matches the arguments $\overline{v}$. It returns the body of the function e, as well as the store representing the variables bound by the patterns, S. This is a partial function, only defined if there is a matching implementation for the given rule name and arguments. We give the exact definition of rule specificity in Section 7.4.

$$\frac{O = D[x], \text{ sorted by specificity, descending} \quad (\overline{p}, e) = \mathsf{FirstMatch}(O, \overline{v}) \quad S = \mathsf{MatchAll}(\overline{p}, \overline{v})}{\mathsf{ChooseRule}(D, x, \overline{v}) = (S, e)} \text{ ChooseRule}$$

7.2.5 Specification evaluation

We define a Dynamix Core specification to consist of a list of rule implementations $(x, \overline{p}, e) \in S$ representing the name of the rule x, the argument patterns $\overline{p}$ and the body of the implementation e. In order to evaluate an entire Dynamix Core specification S, we require the "initial rule" x_i, as well as the input source AST v_i. We define the function $\mathsf{Evaluate}(S, x_i, v_i) = v$ to yield the result of evaluating the rule x_i with argument v_i in specification S. We unconditionally compose this result with a call to the `#exit` primitive, to eliminate any possible holes in the final term.

We require that every rule implementation in a specification S is unique, such that the set of values matched by their argument patterns is not equal between two different rule implementations. This ensures that there is an unambiguous pattern-matching order for all rule implementations. We refer to the CompareAll function for this, defined in Section 7.4.

$$\dfrac{\begin{array}{c}\forall(x_1, \overline{p_1}, e_1), (x_2, \overline{p_2}, e_2) \in S : (x_1 = x_2 \land e_1 \neq e_2) \rightarrow \mathsf{CompareAll}(\overline{p_1}, \overline{p_2}) \neq \texttt{Equal} \\ D = \text{all rule implementations in } S\text{, grouped by name} \\ (S, e) = \mathsf{ChooseRule}(O, x_i, [v_i]) \\ S, \{\}, D \vdash e \rightarrow v \\ v' = v \bowtie \texttt{CExp(CExpPrimitiveTail("\#exit",} [])) \end{array}}{\mathsf{Evaluate}(S, x_i, v_i) = v'} \textsc{Evaluate}$$

There are no formal requirements on the name of the initial rule, although it is conventionally named `compileProgram` in the Spoofax implementation of Dynamix. Implementations are recommended to offer a method of configuring the initial rule.

7.3 Static semantics

We now discuss the static semantics for Dynamix Core. In order to do so, we must extend the AST structure outlined in Section 7.1 with productions describing the type system and type signatures for rules. These extensions can be seen in Grammar 7.2.

The Dynamix Core type system distinguishes between integer and string source ATerm types, declared ATerm constructor sorts, different types of Tim target AST nodes, pluggable terms, paired pluggables, and meta-lists. These types correspond with the different sorts of runtime values outlined in Section 7.2. Within the formal specification of the type system, we

decl	::=	*rule-sig*	
	|	*global-decl*	
	|	*aterm-sort-decl*	
global-decl	::=	**global $***ID* **::** *type*	*meta-global type signature*
aterm-sort-decl	::=	**constructor** *ID* **::** ⟨ *type* ***** ⟩* **->** *ID*	*ATerm constructor signature*
rule-sig	::=	**rule** *ID* **::** ⟨ *type* ***** ⟩* **->** *type*	*rule type signature*
type	::=	**'int**	*ATerm integer*
	|	**'string**	*ATerm string*
	|	**'***ID*	*ATerm sort reference*
	|	**@cval**	*Tim cval node*
	|	**@cexp**	*Tim cexp node*
	|	**@cfun**	*Tim cfun node*
	|	**Pluggable**	*pluggable term*
	|	**Pluggable(***type***)**	*pluggable term with value*
	|	**List(***type***)**	*meta-list type*

Grammar 7.2: Extensions to the Dynamix Core grammar for type-checking purposes.

will use the grammar from the `type` sort in Grammar 7.2 to represent types.

The Dynamix Core type system prevents the overwhelming majority of possible runtime errors. In particular, we claim that a specification that is valid according to the static semantics outlined in this chapter, will either abort compilation due to a call to a rule with arguments for which no implementation exists, abort compilation because a meta-global is used before being assigned a value, or successfully compile a source AST to a Tim program. All other classes of potential runtime errors, such as invalid name bindings or illegal CPS operations, are prevented directly on the type system level. Unfortunately, a lack of time means that we cannot formally prove this claim.

We define the static semantics of Dynamix Core through the use of inference rules. In the case that no particular inference rule matches a given situation, it should be interpreted as a disallowed situation and an error should be issued.

7.3.1 Type coercion

Certain types in Dynamix Core can be implicitly converted to other types. To model this behavior, we introduce the partial helper function $\mathsf{Compatible}(T_1, T_2)$. This expression is defined if a value of type T_1 is assignable to a context expecting a value of T_2. The following inference rules describe the behavior of Compatible.

$$\frac{T_1 = T_2}{\mathsf{Compatible}(T_1, T_2)}\ \textsc{Compatible-Eq}$$

$$\frac{}{\mathsf{Compatible}(\texttt{Pluggable(}T_1\texttt{)}, \texttt{Pluggable})}\ \textsc{Compatible-Drop Value}$$

$$\frac{T_1 = T_2}{\mathsf{Compatible}(T_1, \texttt{Pluggable(}T_2\texttt{)})}\ \textsc{Compatible-Insert Hole}$$

$$\frac{}{\mathsf{Compatible}(\texttt{@cexp}, \texttt{Pluggable})}\ \textsc{Compatible-Tail to Pluggable}$$

$$\frac{}{\mathsf{Compatible}(\texttt{@cexp}, \texttt{Pluggable(}T_2\texttt{)})}\ \textsc{Compatible-Tail to Pluggable Pair}$$

When we refer to Compatible with vector arguments, e.g. $\mathsf{Compatible}(\overline{\mathsf{T_a}}, \overline{\mathsf{T_b}})$, we require that the two vectors must be of equal lengths, with their elements pairwise compatible.

7.3.2 Expressions

We use the relation $S, R, G, A \vdash e : T$ to indicate that expression e has type T when evaluated in store S and with global rule definitions R. The store S represents the types of bound identifiers, such that $S[x] = T$ retrieves the type T of meta-variable x. The notation $S' = S[y \mapsto T]$ creates a new store S' in which identifier y maps to type T, possibly shadowing an earlier binding.

R represents the set of declared rules, as an associative function mapping a rule name to a tuple consisting of the argument types and return type. We use the notation $R[x] = (\overline{T_a}, T_r)$ to fetch the argument types $\overline{T_a}$ and return type T_r of rule x. Similarly, G represents the set of declared meta-globals, such that $G[x] = T$ retrieves the type T of meta-global x.

A represents a mapping of declared ATerm sort constructors. We use the notation $A[x] = (\overline{T_a}, x_s)$ to state that constructor x accepts parameters of type $\overline{T_a}$, yielding a term of sort x_s. Unlike Dynamix, Dynamix Core does not require the separate definition of ATerm sorts.

Literals

Meta-variables and meta-globals perform a lookup in the environment for their value. The cons list literal simply asserts that the expression type matches the element type of the list operand. The empty list literal returns a list with an unbound element type. In a case where such an empty list literal is used, the type checking should succeed if and only if a valid instantiation for the unbound element type can be found that satisfies the remainder of the program. In the case that no valid instantation can be found, or in the case that multiple valid instantiations exist, the program should be rejected. This means that certain uses of the empty list literal, e.g. `x <- []`, are illegal.[1].

$$\frac{T = G[x]}{S, R, G, A \vdash \texttt{\$}x : T}\ \textsc{T-Global}$$

$$\frac{T = S[x]}{S, R, G, A \vdash x : T}\ \textsc{T-Variable}$$

$$\frac{}{S, R, G, A \vdash \texttt{[]} : \texttt{List}(T)}\ \textsc{T-Nil}$$

$$\frac{S, R, G, A \vdash e_1 : T \qquad S, R, G, A \vdash e_2 : \texttt{List(}T\texttt{)}}{S, R, G, A \vdash \texttt{[}e_1\texttt{|}e_2\texttt{]} : \texttt{List(}T\texttt{)}}\ \textsc{T-Cons}$$

Scoped globals

The `with` expression must reference existing globals, and their values must be compatible with the declared type of the global.

$$\frac{S, R, G, A \vdash \overline{e : T_a} \qquad \overline{T_b = G[x]} \qquad \mathsf{Compatible}(\overline{T_a}, \overline{T_b}) \qquad S, R, G, A \vdash e_b : T}{S, R, G, A \vdash \texttt{with}\ \overline{\texttt{\$}x = e}\ \texttt{do}\ e_b : T}\ \textsc{T-With}$$

Tim value expressions

Both `var` expressions generate a new value of type `@cval`. The `var` expression requires that its argument is an ATerm string.

$$\frac{}{S, R, G, A \vdash \texttt{fresh-var(}x\texttt{)} : \texttt{@cval}}\ \textsc{T-Fresh Var}$$

$$\frac{S, R, G, A \vdash e : \texttt{'string}}{S, R, G, A \vdash \texttt{var(}e\texttt{)} : \texttt{@cval}}\ \textsc{T-Var}$$

Tail calls

The tail call target must be a Tim `cval` term. The arguments must be a meta-list of Tim `cval` terms. Tail calls produce a completed `cexp` term.

$$\frac{S, R, G, A \vdash e_t : \texttt{@cval} \qquad S, R, G, A \vdash e_a : \texttt{List(@cval)}}{S, R, G, A \vdash e_t\texttt{@(}e_a\texttt{)} : \texttt{@cexp}}\ \textsc{T-Tail Call}$$

[1]If `x` is used elsewhere in the program, then one could conceivably infer the required list element type from the context in which it is used. Such a level of inference is not required.

Tim fix declarations

The body of a `fix` expression must be a meta-list of `@cfun` values. Values of type `@cfun` can be created through the `fun` expression, which requires that both the name and the arguments are `@cval` instances. The body of the function must be `@cexp`, to ensure that it unconditionally tail calls.

$$\frac{S, R, G, A \vdash e : \texttt{List(@cfun)}}{S, R, G, A \vdash \texttt{fix \{ } e \texttt{ \}} : \texttt{Pluggable}}\ \textsc{T-Fix}$$

$$\frac{\begin{array}{c} S, R, G, A \vdash e_n : \texttt{@cval} \\ S, R, G, A \vdash e_a : \texttt{List(@cval)} \\ S, R, G, A \vdash e_b : \texttt{@cexp} \end{array}}{S, R, G, A \vdash\vdash \texttt{fun } e_n\texttt{(}e_a\texttt{) = } e_b : \texttt{@cfun}}\ \textsc{T-Fun}$$

Primitives

Calls to primitives assert that the passed argument is a list of `@cval` terms. The expression yields a paired pluggable, `Pluggable(@cval)`. Dynamix Core does not validate whether a primitive exists, or whether the number of arguments is valid for the specific primitive invoked.

$$\frac{S, R, G, A \vdash e : \texttt{List(@cval)}}{S, R, G, A\texttt{\#}x\texttt{(}e\texttt{)} : \texttt{Pluggable(@cval)}}\ \textsc{T-Primitive}$$

Labels

Both label constructs extend the current store with a new binding for the label continuation of type `@cval`. The nullary label yields a `Pluggable` value, the unary label yields a `Pluggable(@cval)` value. The body of the label must evaluate to `@cexp`.

$$\frac{\begin{array}{c} S' = S[x \mapsto \texttt{@cval}] \\ S', R, G, A \vdash e : \texttt{@cexp} \end{array}}{S, R, G, A \vdash e \texttt{ label } x\texttt{/1:} : \texttt{Pluggable(@cval)}}\ \textsc{T-Unary Label}$$

$$\frac{\begin{array}{c} S' = S[x \mapsto \texttt{@cval}] \\ S', R, G, A \vdash e : \texttt{@cexp} \end{array}}{S, R, G, A \vdash e \texttt{ label } x\texttt{/0:} : \texttt{Pluggable}}\ \textsc{T-Nullary Label}$$

Rule calls

Calls to a meta-rule assert that the rule is defined and that the types of the arguments match the declared argument types.

$$\frac{\begin{array}{c} (\overline{T_a}, T_r) = R[x] \\ S, R, G, A \vdash \overline{e_a : T_v} \\ \mathsf{Compatible}(\overline{T_v}, \overline{T_a}) \end{array}}{S, R, G, A \vdash x\texttt{(}\overline{e_a}\texttt{)} : T_r}\ \textsc{T-Call}$$

Blocks

Blocks implicitly compose the type of any two adjacent statements, such that the final type of the block is equal to that of the composition of each of the statements within the block. Composition is performed by the type composition operator, $\bowtie_T$, defined in Section 7.3.4. Arrow statements introduce a new binding for the remainder of the block, the type of which is determined by the type of the bound expression. We define the typing rules for blocks inductively.

$$\frac{S, R, G, A \vdash e : T}{S, R, G, A \vdash \texttt{\{ } x \texttt{ <- } e \texttt{ \}} : T}\ \textsc{T-Block Unary Bind}$$

$$\frac{S, R, G, A \vdash e : T}{S, R, G, A \vdash \texttt{\{ } e \texttt{ \}} : T}\ \textsc{T-Block Unary Exp}$$

$$\frac{\begin{array}{c} S, R, G, A \vdash e : T_e \\ T_b = \begin{cases} T_h & \text{if } T_e = \texttt{Pluggable}(T_h) \\ T_e & \text{if } T_e \in \{\texttt{'int}, \texttt{'string}, \texttt{'}x, \texttt{@cval}, \texttt{@cfun}, \texttt{List}(T_l)\} \end{cases} \\ S' = S[x \mapsto T_b] \\ S', R, G, A \vdash \texttt{\{ } \overline{s} \texttt{ \}} : T_s \end{array}}{S, G, D \vdash \texttt{\{ } x \texttt{ <- } e\ \overline{s} \texttt{ \}} : T_b \bowtie_T T_s}\ \textsc{T-Block Binary Bind}$$

$$\frac{\begin{array}{c} S, R, G, A \vdash e : T_e \\ S, R, G, A \vdash \texttt{\{ } \overline{s} \texttt{ \}} : T_s \end{array}}{S, R, G, A \vdash \texttt{\{ } e\ \overline{s} \texttt{ \}} : T_e \bowtie_T T_s}\ \textsc{T-Block Binary Exp}$$

7.3.3 Declarations and patterns

We now consider the static semantics of the various type declaration sorts, as well as the rule implementation declaration.

Rule, ATerm, and global declarations

Rules, ATerm constructors, and globals must be defined before they can be used inside a specification. Their declarations reside in different namespaces, such that a global and a rule with the same name do not cause conflicts. We use the relation $R, G, A \vdash d$ to indicate that declaration d is valid in the given rule namespace R, global namespace G, and ATerm constructor namespace A.

$$\frac{R[x] = (\overline{T_a}, T_r)}{R, G, A \vdash \texttt{rule } x \texttt{ :: } \overline{T_a} \texttt{ -> } T_r}\ \textsc{D-Rule Decl}$$

$$\frac{A[x_c] = (\overline{T_a}, x_s)}{R, G, A \vdash \texttt{constructor } x_c \texttt{ :: } \overline{T_a} \texttt{ -> } x_s}\ \textsc{D-Constructor}$$

$$\frac{G[x] = T}{R, G, A \vdash \texttt{global \$}x \texttt{ :: } \overline{T}}\ \textsc{D-Global}$$

Patterns

Argument patterns assert that the argument type is compatible with the values matched by the pattern. We define the partial function $\mathsf{TMatch}(A, p, T) = S$ to indicate that pattern p is capable of matching values of type T, yielding a store S representing any local variables bound by the pattern. This function is only defined if p can successfully match values of type T. The argument A represents the set of declared constructors, as per the definition given earlier in this section.

To make the definition of TMatch more concise, we define partial function $\mathsf{TMatchAll}(A, \overline{p}, \overline{T}) = S$. This function tests that each pattern element of $\overline{p}$ is compatible with each appropriate element of $\overline{T}$. The resulting store S is the union of the stores bound by each pattern in the vector $\overline{p}$. The identifiers bound by each pattern must be distinct. We define $\mathsf{TMatchAll}$ inductively:

$$\frac{}{\mathsf{TMatchAll}(A, [], []) = \{\}}\ \textsc{TMatchAll-Nil}$$

$$\frac{\begin{array}{c} S_h = \mathsf{TMatch}(A, p_h, T_h) \\ S_t = \mathsf{TMatchAll}(A, \overline{p_t}, \overline{T_t}) \\ S_h \bigcap S_t = \{\} \end{array}}{\mathsf{TMatchAll}(A, [p_h|\overline{p_t}], [T_h|\overline{T_t}]) = S_h \cup S_t}\ \textsc{TMatchAll-Cons}$$

The individual implementations of TMatch are trivial.

$$\frac{}{\mathsf{TMatch}(A, \texttt{_}, T) = \{\}}\ \textsc{TMatch-Wildcard}$$

$$\frac{}{\mathsf{TMatch}(A, x, T) = \{x \mapsto T\}}\ \textsc{TMatch-Var}$$

$$\frac{}{\mathsf{TMatch}(A, i, \texttt{'int}) = \{\}}\ \textsc{TMatch-Int Literal}$$

$$\frac{}{\mathsf{TMatch}(A, s, \texttt{'string}) = \{\}}\ \textsc{TMatch-String Literal}$$

$$\frac{}{\mathsf{TMatch}(A, \texttt{[]}, \texttt{List(}T\texttt{)}) = \{\}}\ \textsc{TMatch-Nil}$$

$$\frac{\begin{array}{c} S_h = \mathsf{TMatch}(A, p_h, T) \\ S_t = \mathsf{TMatch}(A, p_t, \texttt{List(}T\texttt{)}) \\ S_h \bigcap S_t = \{\} \end{array}}{\mathsf{TMatch}(A, \texttt{[}p_h\texttt{|}p_t\texttt{]}, \texttt{List(}T\texttt{)}) = S_h \cup S_t}\ \textsc{TMatch-Cons}$$

$$\frac{\begin{array}{c} A[x_c] = (\overline{T_a}, x_s) \\ S = \mathsf{TMatchAll}(A, \overline{p}, \overline{T_a}) \end{array}}{\mathsf{TMatch}(A, x_c\texttt{(}\overline{p}\texttt{)}, \texttt{'}x_s) = S}\ \textsc{TMatch-Constructor}$$

$$\frac{\begin{array}{c} S_i = \mathsf{TMatch}(A, p, T) \\ x \notin S_i \\ S = S_i \cup \{x \mapsto T\} \end{array}}{\mathsf{TMatch}(A, x\texttt{@}p, T) = S}\ \textsc{TMatch-Bound}$$

Rule implementations

For a rule implementation, we assert that the rule has been declared, that the given argument patterns are valid for the declared argument types, and that the type of the body is compatible with the declared return type. The initial store used to type-check the body is derived from the bindings produced by the argument patterns.

$$\frac{\begin{array}{c} R[x] = (\overline{T_a}, T_r) \\ S = \mathsf{TMatchAll}(A, \overline{p}, \overline{T_a}) \\ S, R, G, A \vdash e : T_e \\ \mathsf{Compatible}(T_e, T_r) \end{array}}{R, G, A \vdash x\texttt{(}\overline{p}\texttt{) =}\ e}\ \text{D-RULE IMPL}$$

Specification validity

In order for a single specification unit to be valid, we simply require that all declarations must be valid. We use the notation $\vdash S$ to state that specification S is valid.

$$\frac{\begin{array}{c} R = \{\} \\ G = \{\} \\ A = \{\} \\ R, G, A \vdash \overline{d} \end{array}}{\vdash \overline{d}}\ \text{T-SPECIFICATION}$$

7.3.4 Type composition

To correspond with the dynamic composition operator $\bowtie$, we define the type composition operator $\bowtie_T$. This operator models the result of composing two values of a given type, such that $T_1 \bowtie_T T_2$ yields the type of the value created by composing a value of type T_1 with a value of type T_2. This composition operator is implicitly applied between two adjacent statements in a block.

We define the type composition operator using inference rules. If a specific combination of types is not defined in any inference rule, it is undefined and any program attempting to compose such types should be rejected.

If the left-hand side of type composition is a source type, meta-list, `@cval`, or `@cfun`, we yield the right-hand side.

$$\frac{T_1 \in \{\texttt{'int}, \texttt{'string}, \texttt{'}x, \texttt{@cval}, \texttt{@cfun}, \texttt{List}(T_l)\}}{T_1 \bowtie_T T_2 = T_2}\ \text{TCOMPOSE-RHS}$$

Combining a `Pluggable` type with a `@cexp` unconditionally turns it into a completed CPS term, regardless of whether the `Pluggable` value represented a pluggable term or a completed CPS term.

$$\frac{T_1 \in \{\texttt{Pluggable}, \texttt{Pluggable}(...)\}}{T_1 \bowtie_T \texttt{@cexp} = \texttt{@cexp}}\ \text{TCOMPOSE-FILL}$$

Composing `Pluggable(T)` and `Pluggable` yields the right-hand side of the composition.

$$\frac{}{\texttt{Pluggable(}T_1\texttt{)} \bowtie_T \texttt{Pluggable(}T_2\texttt{)} = \texttt{Pluggable(}T_2\texttt{)}}\ \text{TCOMPOSE-PLUGT PLUGT}$$

$$\frac{}{\texttt{Pluggable(}T_1\texttt{)} \bowtie_T \texttt{Pluggable} = \texttt{Pluggable}}\ \textsc{TCompose-PlugT Plug}$$

$$\frac{}{\texttt{Pluggable} \bowtie_T \texttt{Pluggable(}T_2\texttt{)} = \texttt{Pluggable(}T_2\texttt{)}}\ \textsc{TCompose-Plug PlugT}$$

$$\frac{}{\texttt{Pluggable} \bowtie_T \texttt{Pluggable} = \texttt{Pluggable}}\ \textsc{TCompose-Plug Plug}$$

Composing any pluggable term with a type that isn't pluggable itself yields a `Pluggable(T)`.

$$\frac{T_2 \notin \{\texttt{Pluggable}, \texttt{Pluggable}(...)\}}{\texttt{Pluggable} \bowtie_T T_2 = \texttt{Pluggable(}T_2\texttt{)}}\ \textsc{TCompose-Plug Other}$$

$$\frac{T_2 \notin \{\texttt{Pluggable}, \texttt{Pluggable}(...)\}}{\texttt{Pluggable(}T_1\texttt{)} \bowtie_T T_2 = \texttt{Pluggable(}T_2\texttt{)}}\ \textsc{TCompose-PlugT Other}$$

7.4 Rule specificity

In order to provide a deterministic order in which multiple implementations for the same rule are pattern-matched against their arguments, Dynamix Core defines a method of deriving the relative "specificity" of each pattern. This approach is unlike most programming languages that support pattern matching (which evaluate the patterns in order of definition), but it allows a user to define rule implementations in separate files without causing ambiguity on the order in which they should be evaluated. This same approach of rule specificity is also used in a different Spoofax meta-language, Statix [5].

We define the function $\mathsf{Compare}(p_1, p_2) = o$, where $o \in \{\texttt{Equal}, \texttt{LessSpecific}, \texttt{MoreSpecific}, \texttt{Uncomparable}\}$, to evaluate the specificity of p_1 relative to that of p_2. This function evaluates the subset of possible values matched by pattern p_1, and compares it to the subset of possible values matched by p_2. If p_1 captures exactly the same subset of values as p_2, it yields `Equal`. If the values captured by p_2 are a subset of the values captured by p_1, it yields `LessSpecific` (i.e. p_1 captures more values and as a result is less specific than p_2). Similarly, if the values captured by p_1 are a subset of the values captured by p_2, it yields `MoreSpecific`. If neither is the case, it yields `Uncomparable`. This is the case if two patterns have no overlap in the set of values that they match (e.g. one matches an integer literal, the other matches a string literal).

Note that patterns in Dynamix Core are **linear**: each variable bound by a pattern must be unique. A pattern such as `Pair(a, a)` is illegal and should be rejected. This restriction is encoded in the definition of the union of two stores, which exists if and only if the set of variables in both sets is disjoint.

Definition of CompareAll

The helper function $\mathsf{CompareAll}(\overline{p_1}, \overline{p_2}) = o$ compares each individual element of the two given lists of patterns using Compare. If each element of $\overline{p_1}$ compares equal to the corresponding element of $\overline{p_2}$, CompareAll returns `Equal`. Otherwise, it yields the first comparison result for which the two elements are not equal. This function is defined inductively:

$$\frac{}{\mathsf{CompareAll}([], []) = \texttt{Equal}}\ \textsc{CompareAll-Nil}$$

$$\frac{o_h = \mathsf{Compare}(p_1, p_2) \qquad o = \begin{cases} o_h & \text{if } o_h \neq \texttt{Equal} \\ \mathsf{CompareAll}(\overline{p_a}, \overline{p_b}) & \text{otherwise} \end{cases}}{\mathsf{CompareAll}([p_1|\overline{p_a}], [p_2|\overline{p_b}]) = o}\ \textsc{CompareAll-Cons}$$

Definition of Compare

We define the Compare(p_1, p_2) function through several inference rules. In the case that several rules match for a given input, the first rule listed should be preferred.

In the cases where we can show a trivial equality (i.e. the AST representation of the two patterns is equal), we immediately yield `Equal`:

$$\frac{p_1 = p_2}{\mathsf{Compare}(p_1, p_2) = \texttt{Equal}}\ \textsc{Compare-Trivial Eq}$$

Bound patterns have the same specificity as the pattern that they bind, and delegate their comparison to the underlying pattern:

$$\frac{}{\mathsf{Compare}(x\texttt{@}p_1, p_2) = \mathsf{Compare}(p_1, p_2)}\ \textsc{Compare-Bound 1}$$

$$\frac{}{\mathsf{Compare}(p_1, x\texttt{@}p_2) = \mathsf{Compare}(p_1, p_2)}\ \textsc{Compare-Bound 2}$$

Wildcards and variable patterns are equal in specificity, regardless of the name of the variable. They are less specific than anything else.

$$\frac{\begin{array}{c} p_1 \text{ is a variable or wildcard pattern} \\ p_2 \text{ is a variable or wildcard pattern} \end{array}}{\mathsf{Compare}(p_1, p_2) = \texttt{Equal}}\ \textsc{Compare-Var Wildcard Eq}$$

$$\frac{\begin{array}{c} p_1 \text{ is a variable or wildcard pattern} \\ p_2 \text{ is not a variable or wildcard pattern} \end{array}}{\mathsf{Compare}(p_1, p_2) = \texttt{LessSpecific}}\ \textsc{Compare-Var Other}$$

$$\frac{\begin{array}{c} p_1 \text{ is not a variable or wildcard pattern} \\ p_2 \text{ is a variable or wildcard pattern} \end{array}}{\mathsf{Compare}(p_1, p_2) = \texttt{MoreSpecific}}\ \textsc{Compare-Other Var}$$

ATerm constructor patterns are uncomparable if their constructor name does not match. Else, their specificity depends on the specificity of each of the argument patterns, compared using CompareAll.

$$\frac{x_1 \neq x_2}{\mathsf{Compare}(x_1\texttt{(}\overline{p_1}\texttt{)}, x_2\texttt{(}\overline{p_2}\texttt{)}) = \texttt{Uncomparable}}\ \textsc{Compare-Cons Neq}$$

$$\frac{x_1 = x_2}{\mathsf{Compare}(x_1\texttt{(}\overline{p_1}\texttt{)}, x_2\texttt{(}\overline{p_2}\texttt{)}) = \mathsf{CompareAll}(\overline{p_1}, \overline{p_2})}\ \textsc{Compare-Cons Eq}$$

$$\frac{p_2 \text{ is not a constructor pattern}}{\mathsf{Compare}(x_1\texttt{(}\overline{p_1}\texttt{)}, p_2) = \texttt{Uncomparable}}\ \textsc{Compare-Cons Other}$$

ATerm integer and string patterns are only equal to themselves, and uncomparable with anything else. The equality case is handled in rule Compare-Trivial Eq.

$$\frac{p_1 \text{ is an integer or string pattern}}{\mathsf{Compare}(p_1, p_2) = \texttt{Uncomparable}}\ \textsc{Compare-Literal Other}$$

The meta-list nil pattern is uncomparable with anything but itself. The equality case is handled in rule Compare-Trivial Eq.

$$\frac{}{\mathsf{Compare}(\texttt{[]}, p_2) = \texttt{Uncomparable}}\ \textsc{Compare-Nil Other}$$

The meta-list cons pattern is only comparable with other cons patterns. For these patterns, the head is compared. If the head patterns are equal, the tail pattern is compared.

$$\frac{}{\mathsf{Compare}(\texttt{[}p_a\texttt{|}p_b\texttt{]}, p_2) = \texttt{Uncomparable}}\ \textsc{Compare-Cons Other}$$

$$\frac{o_h = \mathsf{Compare}(p_a, p_x) \qquad o = \begin{cases} o_h & \text{if } o_h \neq \texttt{Equal} \\ \mathsf{Compare}(p_b, p_y) & \text{otherwise} \end{cases}}{\mathsf{Compare}(\texttt{[}p_a\texttt{|}p_b\texttt{]}, \texttt{[}p_x\texttt{|}p_y\texttt{]}) = o}\ \textsc{Compare-Cons Cons}$$

Examples

We give several examples here that show the behavior of the Compare function on different input patterns.

$$\begin{aligned}
\mathsf{Compare}(\texttt{\_}, \texttt{foo}) &= \texttt{Equal} \\
\mathsf{Compare}(\texttt{foo}, \texttt{bar}) &= \texttt{Equal} \\
\mathsf{Compare}(\texttt{[\_|1]}, \texttt{[]}) &= \texttt{Uncomparable} \\
\mathsf{Compare}(\texttt{[\_|1]}, \texttt{[\_|2]}) &= \texttt{Uncomparable} \\
\mathsf{Compare}(\texttt{[\_|1]}, \texttt{[1|\_]}) &= \texttt{LessSpecific} \\
\mathsf{Compare}(\texttt{[a|[b|c]]}, \texttt{[a|b]}) &= \texttt{MoreSpecific} \\
\mathsf{Compare}(\texttt{[a|[b|[]]]}, \texttt{[a|[]]}) &= \texttt{Uncomparable} \\
\mathsf{Compare}(\texttt{A(1)}, \texttt{A(1, 2)}) &= \texttt{Uncomparable} \\
\mathsf{Compare}(\texttt{A(1)}, \texttt{A(x)}) &= \texttt{MoreSpecific} \\
\mathsf{Compare}(\texttt{A(\_)}, \texttt{A(y)}) &= \texttt{Equal} \\
\mathsf{Compare}(\texttt{x@\_}, \texttt{y}) &= \texttt{Equal}
\end{aligned}$$

Chapter 8

Dynamix in Spoofax

While we have extensively discussed the formal semantics of the Dynamix meta-language, we have thus far largely ignored the concrete implementation of the language within the Spoofax language workbench. In this chapter, we elaborate on the concrete implementation of Dynamix, how it interacts with other parts of the Spoofax language workbench, and comment on how some of its design decisions were influenced by existing conventions within the Spoofax ecosystem. Readers that are mainly interested in the design of Dynamix and its abstractions may prefer to skip this chapter.

8.1 Project structure

The Dynamix and Tim meta-languages are integrated directly in Spoofax 3, the newest iteration of the Spoofax language workbench. While Spoofax 3 is currently still under development, the decision was made to directly integrate Dynamix into this version in order to ensure that it receives proper development and maintenance, as well as the ability to benefit from the improvements in Spoofax 3. In particular, Dynamix and Tim are able to make use of the new PIE [37] framework for incremental build tasks.

Both Dynamix and Tim are themselves languages written using the Spoofax language workbench. They use SDF3 [55] for syntax specification, Statix [5] for static analysis, and Stratego 2 [67] for the implementation of their interpreters. This choice was a natural one: the Spoofax language workbench abstracts a lot of language design work, the internal Spoofax 3 project has first-class support for adding new meta-languages that themselves are Spoofax language projects, and the overlap in technologies used between the languages allows for easy interoperation between them.

Meta-languages within Spoofax 3 essentially act as language projects that are shipped with the editor by default. This allows them to be available when a language project depends on them, without requiring the user to install them. Both Tim and Dynamix function in this manner. Beyond the implementation of the concrete language, the implementation of Dynamix within the language workbench also involves extensions to the Spoofax project configuration language and new PIE tasks for auxiliary file generation and specification compilation. We will omit the exact details for these extensions, as they are effectively implementation details.

8.2 Compiling Dynamix specifications

Dynamix has first-class support for spreading out a specification over multiple files. For this, it uses a module system that is consistent with other Spoofax meta-languages: each file constitutes a named "module", which can be imported into any other file located within

```
module foo

imports bar

signature
  constructors
    Return : Exp -> Exp

rules
  evalExp :: 'Exp -> @cval
  evalExp(Int(i)) = int(i)
  evalExp(Return(e)) = {
    v <- evalExp(e)
    $return@([v])
  }
```

```
module bar

imports foo

signature
  sorts Exp
  constructors
    Int : string -> Exp
    Var : string -> Exp

rules
  $return :: @cval

  evalExp(Var(x)) = var(x)
```

Figure 8.1: An example of a Dynamix specification spread across multiple files. The `foo` module declares the `evalExp` module, using a sort imported from the `bar` module. The `bar` module adds another implementation to the imported declaration of `evalExp`.

the same project. When such an import occurs, all identifiers and declarations are imported. This is a straightforward multi-file model that works as expected in most cases, although it can occasionally cause name binding ambiguities due to the inability to only import parts of a module, as well as the lack of namespaces or other methods of qualifying an exact declaration. Figure 8.1 shows an example Dynamix specification spread out over two files.

When a language project that uses Dynamix is built, Dynamix will perform a merging step as part of compilation. This step ensures that several files are combined into a single specification, such that the Dynamix interpreter does not need to concern itself with the exact semantics of multi-file specifications. This merging step uses information assigned by the Statix specification for Dynamix. In particular, Statix keeps track of exactly which declaration any specific rule implementation, global, or rule call references. Each declaration additionally tracks the name of the module in which it was defined. As part of compilation, a Stratego strategy queries this information to transform each input specification by replacing any reference to a declaration with a fully qualified name (an example of this process can be seen in Figure 8.2). An example showing the merged form of the example from Figure 8.1 can be seen in Figure 8.3.

Static analysis information is also used for desugaring purposes. Any references to constraint analyzer property lookups, which have a syntax identical to the invocation of a rule, are deserialized into a specialized AST node that indicates that a constraint analyzer lookup should be performed. Similarly, calls to expression primitives are distinguished from calls to statement primitives, as they must generate different Tim AST nodes when evaluated. Doing this transformation at compile time ensures that the implementation does not need to resolve the definition of the rule or the primitive at runtime.

The final compilation result is a serialized form of the Dynamix specification that contains all desugared rule implementations in a single file. When a source program needs to be compiled, the Dynamix interpreter will sort these implementations by the specificity of their argument patterns (see Section 7.4), then attempt to execute the starting rule (`main!compileFile` by default) with the input AST as argument.

```
rules
  // bar(a, b, c) -> foo!bar(a, b, c)
  dx--fully-qualify:
    MExprCall(name, args) -> MExprCall(<dx--qualify-ast-name> name, args)

  // helper that takes an ast node that references a declaration,
  // with that declaration having a "declaringModule" statix prop,
  // and returns the name with that mod name prefixed to it
  dx--qualify-ast-name: name -> $[[refMod]![name]]
    with
      a := <stx-get-ast-analysis> node;
      ref := <stx-get-ast-ref(|a)> node;
      refMod := <stx-get-ast-property(|a, "declaringModule")> ref
```

Figure 8.2: An example showing how Dynamix merges multiple modules during compilation. All nodes within a module are fully qualified by interacting with the results of static analysis through the Statix API for Stratego.

```
module merged

rules
  foo!evalExp(Int(i)) = int(i)
  foo!evalExp(Return(e)) = {
    v <- foo!evalExp(e)
    $bar!return@([v])
  }
  foo!evalExp(Var(x)) = var(x)
```

Figure 8.3: The compilation result of merging the `foo` and `bar` modules from Figure 8.1. All references to rules and globals are fully qualified, and all declarations not necessary for interpretation have been removed.

8.3 Type signature generation

The Dynamix type system ensures that operations such as pattern matching and source term literals are fully typed. In order to do this, the user must specify the exact algebraic signature of their input AST, such that the type system can properly type check its use. This is an approach common in all typed Spoofax meta-languages, including Statix and Stratego 2. As source languages grow, maintaining this algebraic signature for each meta-language becomes a tiring and error-prone task. To combat this, the Spoofax meta-languages generally include the option to automatically derive algebraic ”signatures” from the SDF3 grammar of the source language. These signatures are automatically generated as part of the build process, and can be included through the multi-file specification support in those languages.

As part of the Dynamix implementation into the Spoofax language build, a similar signature generator for Dynamix was built. This signature generator supports both lexical and context-free sorts, inferring the signature of the AST from the sorts referenced in the grammar of each production. An example of an SDF3 grammar and the Dynamix source generated by the signature generator can be seen in Figure 8.4.

```
module foo

lexical sorts ID
lexical syntax
  ID = [a-zA-Z] [a-zA-Z0-9\_]*

context-free sorts Exp Literal
context-free syntax
  Exp = Literal
  Exp.Add = <<Exp> + <Exp>>

  Literal.Var = <<ID>>
```

```
module signatures/foo-sig

signature
  sorts
    ID Exp Literal

  constructors
    : string -> ID
    : Literal -> Exp
    Add : Exp * Exp -> Exp
    Var : ID -> Literal
```

Figure 8.4: An example of a simple SDF3 grammar, and the type signature automatically generated by Dynamix. Importing `signatures/foo-sig` in the specification will give access to the type information, without needing to manually define the signature. The highlighted line contains an injection, specifying that literals may be implicitly converted to expressions.

In order to properly support all SDF3 grammars, the Dynamix type system also offers rudimentary support for ”injections”: the ability to specify that some sort can be implicitly converted to some other sort. This support is on a best-effort basis and supports the majority of languages that use injections sparingly, but limitations in Statix prevent some legal programs from type checking properly[1]. This tradeoff was deemed acceptable, as the use of SDF3 grammars with injections is generally considered a bad practice.

8.4 Constraint analyzer integration

As briefly discussed in Section 6.1.8, Dynamix has first-class support for interacting with the results of static analysis. In particular, Dynamix offers the ability to read any property that has been assigned by Statix [5] during the analysis of the source program. Since Statix uses the ATerm representation for property values, Dynamix natively offers first-class support for any potential value that can be assigned by a Statix specification.

Interoperation with Statix in a Dynamix specification is done by declaring the property, as well as the type of values assigned to it. This ensures that Dynamix remains able to type check operations performed on values obtained from querying static analysis properties. If the user uses an ATerm sort not derived from the SDF3 grammar (e.g. a custom sort for representing semantic types), they will need to also define this sort in Dynamix. An example of the declaration and use of static analysis properties can be seen in Figure 8.5.

In order to model optional properties, Dynamix returns a meta-list when querying any property. An empty list indicates that the property was not present, while a non-empty list indicates that at least one value was assigned to the property (Statix allows you to assign more than one value to a property). Unfortunately, limitations in the Stratego application programming interface (API) exposed by Statix mean that it is impossible to distinguish between a list assigned as a property (`@foo.prop := [1, 2, 3]`), and multiple values assigned to the same property (`@foo.prop += 1`). This is generally not a problem, as multiple assigned prop-

[1]The Dynamix specification is generally able to ”see through” a single layer of injections. Multiple layers of injections quickly turn type-checking into a path-finding problem (”is there a sequence of injections I can take to convert type A into type B?”), which Statix is not properly equipped to handle.

```
signature
  sorts TYPE
  constructors
    INT    : TYPE
    BOOL   : TYPE
    LIST   : TYPE -> TYPE
    STRING : TYPE

constraint-analyzer
  property type :: 'TYPE

rules
  // compile a + b by switching on the type of the expression
  compileExp(a@Add(_, _)) = compileAdd(type(a), a)

  compileAdd :: List('TYPE) * 'Exp -> Pluggable(@cval)
  compileAdd([INT()], Add(a, b)) = {
    // <integer addition omitted>
  }
  compileAdd([LIST(_)], Add(a, b)) = {
    // <list concatenation omitted>
  }
  compileAdd([STRING()], Add(a, b)) = {
    // <string concatenation omitted>
  }
```

Figure 8.5: An example declaration for interacting with the results of static analysis. The `type` Statix property is declared, returning a value representing the semantic type of the node. The compilation of the addition expression uses the type information to distinguish between integer addition, list concatenation, and string concatenation.

erties are exceedingly rare and generally not mixed with direct assignments.

Internally, property lookups are done by preserving the attachments and annotations[2] present on source ATerms. By carefully ensuring that none of the transformation and evaluation steps done by the Dynamix interpreter touch these attachments, the Statix runtime is able to resolve the appropriate analysis results for a given node. Any source nodes created by the source term literal expression, `'<term>`, lack such annotations and therefore always yield empty meta-lists when a lookup is performed.

[2]Annotations are the "A" in ATerm, as discussed in Chapter 2. Attachments are invisible annotations that are only accessible by native Stratego strategies. Due to unfortunate decisions and legacy code, most APIs that resolve node information require both of them to be carefully preserved, which is not always trivial.

Chapter 9

Case studies

To evaluate the effectiveness of the Dynamix meta-language and Tim runtime, specifications were written for several different programming languages. Outside of the Tiger specification discussed as part of Section 6.1, two extensive specifications were written for the ChocoPy programming language [48] and for a subset of the Stratego programming language [67]. These two languages were chosen because of the author's familiarity with the languages, as well as the different language paradigms they represent.

We first briefly discuss the implemented specification for each language, highlighting the techniques used to implement certain language features and discussing challenges encountered while writing the specification. Afterwards, we refer back to the original design objectives for Dynamix, using the implemented case studies as a method for determining whether the objectives have been successfully achieved.

We do not discuss the full specifications for either case study in this document. Instead, we will only include relevant snippets from the Dynamix specification, foregoing the grammar and static analysis definitions for both languages entirely. Readers interested in the entire Dynamix specification can find the source code for both the ChocoPy[1] and miniStratego[2] case studies online.

9.1 ChocoPy with exceptions

We first discuss the Dynamix specification for ChocoPy with exceptions. ChocoPy [48, 49] is a typed fully specified subset of Python [63] designed for use in teaching compiler construction and language design courses. ChocoPy features include static type checking, classes with dynamic dispatch, lists, and nested functions. Within the TU Delft, ChocoPy is used as the source language for the CS4200 Compiler Construction course.

ChocoPy was chosen as a case study because it represents a "classical" fully defined imperative language with a wide range of language features. In particular, the ability to write a complete Dynamix specification for ChocoPy would indicate that Dynamix is capable of representing at least all of the features present in ChocoPy, which include features common across most popular imperative programming languages. Additionally, the typed nature of ChocoPy allows us to verify that Dynamix is capable of compiling typed languages by taking advantage of its static analysis interoperability features. Finally, because it is used as the language for compiler construction courses, we are able to reuse the large suite of tests to

[1]`https://github.com/molenzwiebel/metaborg-chocopy`

[2]`https://github.com/molenzwiebel/metaborg-ministratego`

verify that the specification is correct.

We will not discuss the exact design of the ChocoPy language in this document. Instead, we refer the reader to the excellent formal specification for ChocoPy [49], which outlines the exact static and dynamic semantics of the language. We will briefly elaborate on the extensions made to ChocoPy to support extensions in Section 9.1.1, but for the sake of brevity we will omit a formal definition for these features.

With this case study, we demonstrate that Dynamix is capable of:

- Implementing an imperative language with control flow consisting of linear fallthrough, function calls, early returns, loops, and conditional statements.
- Compiling and using composite data structures such as lists and classes.
- Performing run-time dynamic dispatch and other object-oriented features.
- Taking advantage of type analysis to implement overloaded operators and to optimize the representation of values ("autoboxing").
- Implementing a language that has exceptions in a clean manner, isolating this implementation only to the language constructs that directly throw or catch them.

9.1.1 Exceptions in ChocoPy

In order to explore the more advanced control flow features available in Dynamix, this case study extends the original ChocoPy language, as defined in the formal specification written by Padhye et. al [49], with support for Python 3.6+-style exceptions. In particular, our version of ChocoPy adds the `raise` and `try-except-finally` statements.

An example of ChocoPy with exceptions can be seen in Figure 9.1. Exceptions are unchecked, meaning that they can be raised at any time and are not part of the signature of a function. Unlike in Python, exceptions in ChocoPy do not need to inherit from an `Exception` base class, but instead can be any class instance. Every `except` block must specify the type of the exception that it wishes to catch, but unlike Python this type must match *exactly*. If the raised exception is a subclass of the declared class, it will not be handled by the except block.[3] Note that the variable bound as an exception must already be declared. This is consistent with other ChocoPy constructs, but it increases the density of the code slightly due to the numerous declarations of locals with an initial value of `None`.

The `finally` block follows the same semantics as in Python 3. In particular, the block is guaranteed to run, even in cases where the `try` or `except` block raises an exception or performs an early return. If the finally block performs an early return, or raises an exception, this will override any of the same actions performed in the try or except blocks. An example of this behavior can be seen in Figure 9.2.

Exceptions thrown propagate upwards until the nearest `except` block that accepts the specific exception type. In the case that such a handler is not found, a default exception handler is invoked, printing a message and exiting the application.

[3]This restriction exists to simplify the implementation, since it means that objects can be compared by class name instead of walking the inheritance tree.

```python
class InvalidArgumentException(object):
  pass

class NotDivisibleBy2Exception(object):
  pass

def halve_element(elements: [int], index: int):
  value: int = 0

  if index >= len(elements) or index < 0:
    raise InvalidArgumentException()

  value = elements[index]
  if value % 2 == 1:
    raise NotDivisibleBy2Exception()

  elements[index] = value // 2

elements: [int] = None
iax: InvalidArgumentException = None
ndb2x: NotDivisibleBy2Exception = None

elements = [1, 2, 3, 4]
try:
  halve_element(elements, 1) # OK
  halve_element(elements, 0) # not divisible by 2
except InvalidArgumentException as iax:
  print("Invalid index passed to halve_element")
except NotDivisibleBy2Exception as ndb2x:
  print("Element is not cleanly divisible by two")
finally:
  print("Done!")
  print(elements[0]) # prints 1
  print(elements[1]) # also prints 1
```

Figure 9.1: An example program written in our version of ChocoPy. The second invocation of `halve_element` raises an exception, which is caught by the `except` block.

```
1  class Exception(object):
2    pass
3
4  def reraise() -> int:
5    ex: Exception = None
6    try:
7      print("A")
8      raise Exception()
9      return 1
10   except Exception as ex:
11     print("B")
12     raise ex
13   finally:
14     print("C")
15     return 2 # overrides the raise from line 12
16   return 3
17
18 def hijack_return() -> int:
19   try:
20     return 1
21   finally:
22     return 2 # overrides the return from line 20
23   return 3
24
25 print(reraise()) # Prints A, B, C, 2
26 print(hijack_return()) # Prints 2
```

Figure 9.2: An example showing how the `finally` block is invoked even in the case of an uncaught exception or early return. Any raises or returns performed in the finally block take precedence over any performed prior to the block.

9.1.2 Basic ChocoPy in Dynamix

To quantify the relative effort of adding exceptions to an existing specification, an initial version of a Dynamix specification for ChocoPy was constructed that implements the entirety of the ChocoPy reference manual, but lacks any of the exception features described earlier. This implementation is fairly straightforward, with most rule implementations resembling the equivalent implementation from the Tiger specification discussed in Chapter 6 . We will limit our discussion of this specification to only those aspects of the specification that feature interesting or creative approaches to implementing a specific language feature. The full Dynamix specification can be found online[4].

Mutually recursive functions

ChocoPy functions that are declared adjacently to each other are allowed to perform mutually recursive function calls. In order to ensure that we correctly compile such cases, we must ensure that all functions within a scope are located in a single Tim `fix` expression[5]. We must also ensure that any variable definitions are compiled *before* the fix block, such that all

[4] `https://github.com/molenzwiebel/metaborg-chocopy`

[5] Recall: functions located in a fix expression are allowed to be mutually recursive.

```
def foo(a: int) -> object:
  def bar(b: int) -> object:
    return print(a + b)
  return bar(10)

foo(10)
```

```
fix {
  fun $fn_foo(a, return5) =
    fix {
      fun $fn_bar(b, return6) =
        #int-add(a, b) => y21;
        $fn_printInt(y21, return6)
    } in
      $fn_bar(10, return5)
} in
  fix {
    fun return9(a130) =
      #exit()
  } in
    $fn_foo(10, return9)
```

Figure 9.3: An example of how nested functions in ChocoPy trivially translate to nested functions in Tim. The right-hand side is the output of the Dynamix specification, although simplified to remove several indirections and features not relevant to this example. Note how the Tim function `$fn_bar` (corresponding to ChocoPy function `bar`) is nested within the Tim function `$fn_foo`, just as in the ChocoPy program.

functions can refer to them. This reordering does not change the semantics of the program.

The Dynamix specification for ChocoPy solves this by iterating over the list of definitions twice. First, variable and class definitions are processed. Afterwards, only functions are processed into a `List(@cfun)`. This approach is similar to that outlined in Section 6.1.5. In order to model the potential lack of a value, the specification uses meta-lists of size zero (to indicate None) and one (to indicate Some). This encoding is the same as commonly used in Statix specifications.

Nested functions and nonlocals

ChocoPy supports the definition of functions inside other functions. Such nested functions can access both arguments and locals defined in parent functions. Unlike full closures however, they do not need to persist beyond the call to the outer-most function (as functions are not first-class values in ChocoPy).

The implementation of nested functions with Dynamix is trivial, as Tim has first-class support for nested functions that capture their parent scope. Since the scoping rules for ChocoPy are compatible with those for Tim, simply defining Tim variables with the same name as the corresponding ChocoPy variable is enough to capture the behavior of nested functions in ChocoPy. An example of a nested function and the corresponding Tim program can be seen in Figure 9.3.

Classes and dynamic dispatch

ChocoPy has limited support for classes and dynamic dispatch. Classes can carry both data and virtual functions, and children can both inherit and redeclare functions from their parent class. However, the ability to call the parent implementation of a function, as well as the ability to downcast from a parent class to a child class, is not supported in ChocoPy. This makes the implementation of objects in the Dynamix specification considerably easier.

Object instances are implemented as records. The `$vtable` field references another record, containing function instances representing the virtual functions of the class. The `$type` field is a string representation of the type of the class, used for exception type matching and type dispatch in certain native functions (e.g. the print function). All fields of the object also live inside this record. Since fields in ChocoPy may not contain a dollar-sign character, they cannot conflict with the virtual method table and type fields. Figure 9.4 shows an example of a pair of classes and how their values are structured.

A class definition is modeled by converting it into a pair of Tim functions. The first function, `$initialize_<name>` is responsible for initializing an already allocated class object with the appropriate class member values, and the object virtual method table with the appropriate function instances. This function starts by first invoking the appropriate initialization function of the parent class, such that the resulting object already contains all fields and virtual method table entries for the parent class. Any functions that were overridden in the child class will be replaced in the virtual method table, such that a lookup will return the child implementation. The second function, `$fn_<name>`, is responsible for allocating the class record, invoking the appropriate initialization function, retrieving the `__init__` method from the virtual method table, and finally invoking it. This is the function that will be invoked when a new instance of the class is constructed by the user.

Since it is impossible to downcast values in the ChocoPy language, we do not need to keep track of the inheritance tree of objects. We do not need to insert any explicit upcasting code either, as the ″shape″ of a child object (consisting of the set of declared fields and methods) is a superset of the parent, and hence directly compatible.

Automatic boxing of primitives

Beyond a formal language specification for ChocoPy, the authors of the ChocoPy specification also provide an implementation guide [50] for compiling the language to RISC-V assembly. In this guide, the authors choose to represent integer and boolean values as native integer values such that arithmetic operations can be performed directly and efficiently. If integers or booleans are used in a context that requires an object[6], they are implicitly converted to an object representation by a process known as ″boxing″.

In order to evaluate the static analysis interoperation features of Dynamix, the decision was made to implement this boxing behavior in the Dynamix specification for ChocoPy.[7] Integers and booleans will be represented directly as Tim integers, and only converted to objects if they are passed to a context expecting an object argument. Unlike the RISC-V implementation guide, we additionally also use unboxed strings whenever possible.

Automatic boxing is implemented by assigning the `box` property in the Statix specification to any expressions used in a context where coercion from a raw primitive value to an object form is required. The Dynamix specification retrieves this property and conditionally branches on the result to potentially box the value. Figure 9.5 shows an outline of this technique.

9.1.3 Adding exceptions

[6]All values in Python are descendants of the object class, even integers, strings, and booleans.

[7]It should be noted that in the ideal case, the language author should not have to worry about the efficiency of their value representation. Even if all values are always boxed, a sufficiently advanced compiler or runtime for Tim should be able to recognize and optimize cases where a boxed representation can be elided. It is unrealistic to expect this for all possible source languages and data representations though, which is why it is good to confirm that Dynamix is able to support more complex data representations like implicit (un)boxing.

```
class Foo(object):
  a: int = 1

  def foo(self: Foo) -> int:
    return self.a

  def bar(self: Foo) -> int:
    return 10

class Bar(Foo):
  b: int = 1

  def bar(self: Bar) -> int:
    return self.b
```

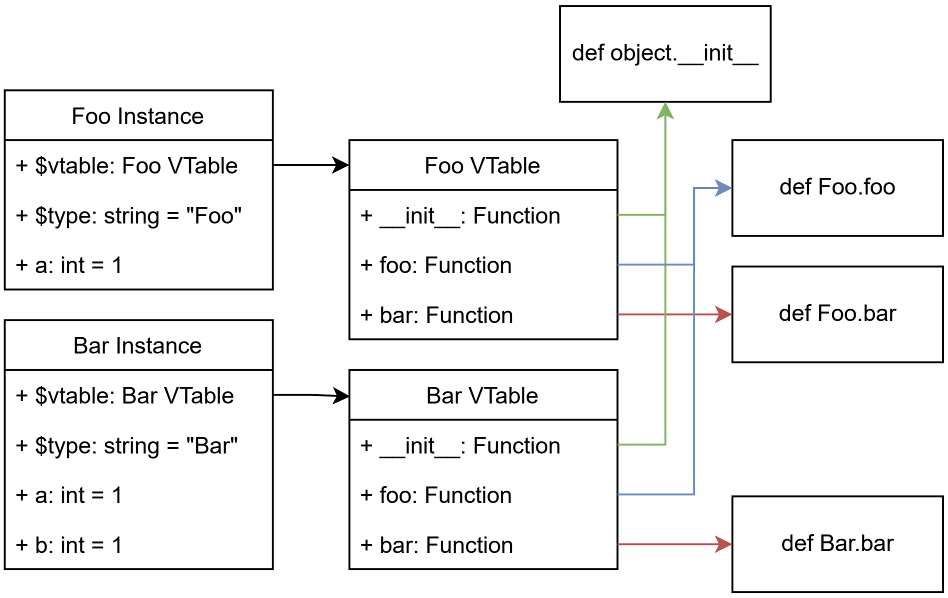


Figure 9.4: A visualization of how object instances are modeled in the Dynamix specification for ChocoPy. Objects contain a type tag, a vtable for dynamic dispatch, and the class fields. The default implementation of `__init__` simply returns immediately.

After completing an initial specification for ChocoPy that lacks any exception support, the existing specification was augmented to add exceptions. This was done intentionally, so that the effort needed to add a new form of control flow to the language can easily be quantified (the results of which we discuss in Section 9.3).

First, each function definition is extended with an additional continuation invoked when an exception is thrown. This argument works similarly to the `return` argument in the Tiger specification (see Section 6.1.5). The meta-global `$raise` is configured to always point to the nearest exception continuation, and its value is used as the exception continuation for calls to functions. This requires adjustments in the implementation of function declarations (both normal and class member functions) and (member) function calls.

The `raise` statement simply invokes the current `$raise` handler. In the case that no try-except-finally block has been defined, this defaults to a top-level function that prints a message and then exits the program. Raising an exception does not require changes in any other rules.

```
rules
  // boxExpIfNeeded(exp, given type, expected type)
  boxExpIfNeeded : Exp * TYPE * TYPE
  boxExpIfNeeded(x, INT(), OBJECT()) :- @x.box += "int".
  boxExpIfNeeded(x, BOOL(), OBJECT()) :- @x.box += "bool".
  boxExpIfNeeded(x, STRING(), OBJECT()) :- @x.box += "str".
  boxExpIfNeeded(x, _, _) :- @x.box += "other".

  stmtOk(s, _, Return(e), RT) :- {T}
    expOk(s, e) == T,
    // possibly insert box if RT is object and T is int/bool
    boxExpIfNeeded(e, T, RT),
    isSubtype(RT, T).
```

```
rules
  compileExpBoxing :: 'Exp -> Pluggable(@cval)
  compileExpBoxing(e) = {
    compiled <- compileExp(e)
    compileExpBoxingHelper(box(e), compiled)
  }

  compileExpBoxingHelper :: List('string) * @cval -> Pluggable(@cval)
  compileExpBoxingHelper(["int"|_], e) = boxInt(e)
  compileExpBoxingHelper(["str"|_], e) = boxString(e)
  compileExpBoxingHelper(["bool"|_], e) = boxBool(e)
  compileExpBoxingHelper(_, e) = e

  compileStmt(Return(e)) = {
    v <- compileExpBoxing(e)
    $return@([v])
  }
```

```
def foo() -> object:
  return 1 + 2

print(foo())
```

```
fix {
  fun $fn_foo(return14) =
    #int-add(1, 2) => y43;
    #record-new("$type", "int", "value", y43) => y44;
    return14(y44)
  fun return15(a133) =
    #exit()
  fun return16(a134) =
    $fn_print(a134, return15)
} in
  $fn_foo(return16)
```

Figure 9.5: An outline showing how the automatic boxing of primitive values for the return statement is implemented in the Dynamix specification for ChocoPy. The Statix specification (top) assigns a box property to each expression that may need to be boxed. The Dynamix specification (middle) reads this property and conditionally boxes an expression based on its value. The example ChocoPy snippet (bottom left) shows how the boxing is compiled to a Tim program (bottom right). Note how the addition operator uses normal integers, boxing only the final return value.

```
rules
  compileStmt(Try(body, excepts, NoTryFinally())) = {
    handler <- fresh-var(handler)
    raised <- fresh-var(raised)

    fix {
      fun handler([raised]) =
        compileExcepts(excepts, raised, after)
    }

    with $raise = handler do
      compileBlock(body)

    after@([])
  } label after/0:

  compileExcepts :: List('TryExcept) * @cval * @cval -> @cexp
  compileExcepts([], v, _) = $raise@([v]) // no excepts matched, re-raise
  compileExcepts([TryExcept(Type(t), id, body)|rest], val, after) = {
    exnty <- #record-read(val, str('"type"))
    if #str-eq(exnty, str(t)) then {
      vref <- #ref-new(val) // wrap exn in ref
      let var(id) = vref in {
        compileBlock(body)
        after@([])
      }
    } else {
      compileExcepts(rest, val, after)
    }
  }
```

Figure 9.6: The implementation of the try-except statement in ChocoPy, when no finally block is given. The implementation is trivial, requiring only a redefinition of the `$raise` handler and sequential comparisons for the type of the exception thrown.

If a `try` block has no `finally`, it is implemented by simply replacing the `$raise` meta-global with a handling function within the scope of the try. In case an exception is caught, it is dispatched to the correct `except` block based on the type of the exception, or bubbled upwards to the previously defined `$raise` if it did not match any of the defined `except` blocks. This implementation is trivial, as seen in Figure 9.6.

If a `try` block has a `finally` block, the implementation is more complex. Not only must the `$raise` continuation be wrapped in the body of the `try` block, but both early returns and exceptions must also be handled in both the `try` block as well as all `except` blocks. This is done by wrapping the existing handlers and invoking the finally block. In the case that the finally block does not perform an early return or exception raise, we invoke the original handler with the intercepted exception or return value. An annotated version of the try-except-finally construct from the Dynamix specification for ChocoPy can be seen in Figure 9.7.

```
compileStmt(Try(body, excepts, TryFinally(finallyBody))) = {
  finallyFromReturn <- fresh-var(finallyRet)
  finallyFromException <- fresh-var(finallyExn)
  finallyFromNormalExecution <- fresh-var(finallyCont)
  onException <- fresh-var(handler)
  val <- fresh-var(val)

  fix {
    fun onException([val]) = {
      // invoke the appropriate except block, wrapping $return and $raise
      with $return = finallyFromReturn, $raise = finallyFromException do
        compileExcepts(excepts, val, finallyFromNormalExecution)
    }

    // the continuation invoked when control flow naturally falls
    // through to the finally block (either after the end of try or
    // after the end of an except body)
    fun finallyFromNormalExecution([]) = {
      compileBlock(finallyBody)
      after@([]) // if the finally block did not return, continue
    }

    // the continuation invoked when any part of the try-except-finally
    // attempts to perform an early return. `val` is the returned value
    fun finallyFromReturn([val]) = {
      compileBlock(finallyBody)
      $return@([val]) // if the finally block did not return, propagate
    }

    // the continuation invoked when any except block attempts to raise
    // a new exception. `val` is the raised exception
    fun finallyFromException([val]) = {
      compileBlock(finallyBody)
      $raise@([val]) // if the finally block did not return, propagate
    }
  }

  // run the body of the try block, wrapping $return and replacing
  // the $raise handler with our except dispatch function
  with $return = finallyFromReturn, $raise = onException do
    compileBlock(body)

  // if we reach this, it means that the try body did not raise
  // an exception. Invoke the finally block normally
  finallyFromNormalExecution@([])
} label after/0:
```

Figure 9.7: The implementation of the try-except-finally statement in our version of ChocoPy. We must carefully handle control flow such that the finally block is guaranteed to be called, even in the presence of early returns or unhandled exceptions.

program	::=	*def**	
def	::=	*ID*`(`⟨ *ID*`,` ⟩*`|`⟨ *ID*`,` ⟩*`)` `=` *strategy*	*named strategy definition*
strategy	::=	`fail`	*unconditional failure*
	|	`id`	*unconditional success*
	|	`?`*term*	*pattern match current term*
	|	`!`*term*	*construct term from pattern*
	|	*strategy*`;` *strategy*	*sequence*
	|	*strategy* `<` *strategy* `+` *strategy*	*guarded choice*
	|	`prim(`*string*`,` ⟨ *strategy*`,` ⟩*`|`⟨ *term*`,` ⟩*`)`	*primitive strategy*
	|	`some(`*strategy*`)`	*succeed if at least 1 subterm succeeds*
	|	`one(`*strategy*`)`	*succeed if 1 subterm succeeds*
	|	`all(`*strategy*`)`	*succeed if all subterms succeed*
	|	*ID*`(`⟨ *strategy*`,` ⟩*`|`⟨ *term*`,` ⟩*`)`	*invoke strategy with args*
	|	`let` ⟨ *def* ⟩* `in` *strategy* `end`	*let binding*

Grammar 9.1: The strategy grammar for the miniStratego language. The grammar for terms, literals, and identifiers are omitted.

9.2 miniStratego

The second case study evaluates the difficulty of implementing a specification for the Stratego core language [67]. Stratego is a programming language centered around the concept of term rewriting and rule failure. Unlike conventional languages, functions (called "strategies" in Stratego) can either succeed with a value, or fail. Failure is a first-class citizen, and built-in language constructs exist for composing strategies, handling failures, and generically applying strategies to input terms. Strategies are also first-class values, allowing them to be passed as arguments to other strategies.

The full Stratego language has a large amount of constructs, but almost all of them can be desugared or rewritten to a core set of constructs as observed by Visser and Benaissa [68]. As a consequence, showing that we are able to write a specification in Dynamix for these core Stratego constructs also shows that Dynamix is capable of representing the entire Stratego language. For this case study, we implement the set of strategy primitives listed in Grammar 9.1. To avoid confusion with the core Stratego language, we name our small language miniStratego. miniStratego uses the same term syntax as ATerms, but it lacks support for annotations.

We will omit the exact semantics of miniStratego in this section. Readers not familiar with the concepts of transformation strategies are recommended to read the informal introduction to Stratego by Eelco Visser [67], as well as the formal semantics for some of the core language structures [68].

With this case study, we demonstrate that Dynamix is capable of:

- Implementing a language that performs non-linear control flow as a first-class language feature.
- Handling a dynamically typed language, such that types must be tracked at runtime.

- Implementing advanced matching features, such as arbitrarily nested non-linear pattern matching.
- Taking advantage of static analysis features to implement a language with implicit variable definitions.

9.2.1 Dynamically typed values

Since all values in Stratego are dynamically typed[8], it becomes necessary to keep track of the exact type of every value. Unlike the boxing approach discussed in the ChocoPy case study (Section 9.1.2), we cannot get away with only boxing non-primitive values. This is because Tim lacks any primitives for identifying the type of a specific value at runtime[9]. Consequently, every Stratego value is represented as a Tim record with a `type` field. The remaining fields of the record depend on the type of the data. For example, lists contain an additional `value` field with a Tim array value.

Many operations on Stratego values are implemented directly as hand-written Tim functions. These functions are comparable to standard library functions, or perhaps assembly snippets, and abstract out common logic such as value type checks, equality checks, and the implementation of the one/some/all strategies. By extracting the definitions into separate functions, the generated code for any given source program becomes less verbose. At the same time, this approach simplifies the implementation of the Dynamix specification, as it is generally more cumbersome to write a Tim function in Dynamix if there are no meta-variables or other source terms involved.

9.2.2 Pattern matching and implicit definitions

Whenever an identifier is used in a miniStratego match construct, it will either act as a binding wildcard pattern (in the case that the variable has not already been bound to a value), or as an equality test for the specific value of the variable (if it has already been bound). For instance, the strategy `?Pair(a, a)` applied to the input `Pair(1, 2)` will first *bind* a to 1 (as it does not yet have a value), and then attempt to *match* this bound value to 2 (failing in the process).

This behavior can be implemented at runtime by keeping track of the currently bound variables, and performing the appropriate behavior based on whether or not a specific identifier was already bound. However, the order in which patterns are evaluated is deterministic (depth-first, left-to-right). This means that static analysis is capable of determining which use of a variable will bind the variable, and which uses will match against an earlier bound value. By exposing this information to the Dynamix specification, variables in patterns can be turned directly into Tim variables. This avoids the overhead of having a dynamic environment of bound variables.

The match strategy is implemented by sequentially checking each component of the match pattern. In the case that any such pattern fails, it immediately invokes the failure continuation of the strategy. If a variable pattern is encountered, it either introduces a new `let` binding (if this use was a declaration), or matches the previously bound value against the current term. Any subsequent strategies (e.g. through the sequencing operator or the guarded choice operator) will have their bodies placed such that the `let` bindings from the pattern

[8]An in-development partial rewrite of the Stratego compiler, dubbed Stratego 2, adds support for incremental static type checking to the language.

[9]It would technically be possible to add such primitives to the Tim interpreter. However, we intentionally do not add such primitives as they may impede the performance of a future optimizing compiler for Tim.

```
rules
  matchTerms :: @cval * @cval * List('Term) -> Pluggable
  // matching no patterns always succeeds
  matchTerms(_, _, []) = hole

  // match index `i` of Tim array `vals` against pattern `hd`
  matchTerms(vals, i, [hd|tl]) = {
    v <- #array-read(vals, i) // read value
    matchTerm(v, hd) // match value against pattern

    // Increment the index for the next value to be matched against.
    // We would ideally do this at compile time (as these are constant
    // numbers), but Dynamix lacks the ability to perform arithmetic.
    inext <- #int-add(i, int('1))
    matchTerms(vals, inext, tl)
  }
```

Figure 9.8: The lack of compile-time arithmetic support for source integers means that we must compute pattern value indices at runtime, even though they could be determined statically.

are in scope for the remainder of the strategy declaration.

When a sequence of patterns needs to be matched against a sequence of input values (as is the case with lists or constructor arguments), this is compiled by sequentially comparing each of them. However, Dynamix lacks the ability to do compile-time arithmetic on source operators. This means that it is impossible to compute the index of each pattern at compile time. The current specification works around this by instead incrementing a Tim integer by 1 on every iteration, as seen in Figure 9.8. While this approach works, it delegates a computation to runtime that can be trivially performed at compile time. In order to properly support this, Dynamix would need to be extended with compile-time support for source term manipulation, which is not included in the version of Dynamix described in this thesis. We discuss this in more detail in Chapter 11.

9.2.3 Proper scoping through pluggable meta-globals

The miniStratego specification uses pluggable values in a somewhat unconventional manner. Specifically, the `$success` and `$failure` meta-globals used to store the current continuations for success and failure are of type `Pluggable(@cval)`, instead of `@cval`. To discuss the reason for this, let us consider the most "obvious" implementation of the sequencing strategy, shown in Figure 9.9.

We evaluate the left-hand strategy with an overridden success continuation. This new continuation evaluates the right-hand strategy with the output term. If this also succeeds, the original success continuation is called. If either of the strategies fails, the failure continuation is called. This is seemingly a completely valid implementation of the sequencing operator. However, this approach runs into scoping issues. The reason is that we compile the right-hand strategy *lexically before* the left-hand strategy. If the left-hand strategy introduces any new bindings (as is the case with variables bound by a match strategy), they will not be visible within the body of the right-hand strategy, as the body of the right-hand strategy is

```
rules
  evalStrategy(t, Seq(a, b)) = {
    success <- fresh-var(suc)
    t2 <- fresh-var(t)
    fix {
      fun success([t2]) = evalStrategy(t2, b)
    }
    with $success = success do
      evalStrategy(t, a)
  }
```

Figure 9.9: The most obvious implementation of the sequencing operator. While seemingly correct, this implementation runs into issues if strategy ‘a‘ introduces any new bindings.

```
fix {
  fun suc1(t) =
    // <implementation for !a>
    // error, `a` is not in scope
    suc0(a)
} in
  // <implementation for ?a>
  let a = t in
    suc1(a)
```

```
// <implementation for ?a>
let a = t in
  fix {
    fun suc1(t) =
      // <implementation for !a>
      // works, `a` is in scope
      suc0(a)
  } in
    suc1(a)
```

Figure 9.10: Two approaches to compiling `?a; !a`. The left Tim program is produced by the ”obvious” implementation of the sequencing operator, but runs into scoping issues because the bound variable is not in scope for the right-hand side body. The right Tim program delays the insertion of the right-hand side until the continuation is called. This ensures that any bindings introduced by the left-hand side are in scope for the right-hand side.

located in an outer scope. This is a problem that we cannot solve by changing our implementation of the sequencing operator.

The solution is to *delay* the insertion of the right-hand strategy body until just before the continuation is called. Figure 9.10 illustrates why this approach solves the scoping issues. To implement this delayed insertion of the right-hand side body, we use `Pluggable` meta-globals. Recall that such values represent Tim program snippets, that get inserted into the final program through composition. By using `Pluggable` globals, we can compose into the program only just before we need them. This approach can be seen in Figure 9.11, and is the approach used in the specification for miniStratego.

9.3 Evaluation

The case studies for ChocoPy with exceptions and miniStratego have shown that Dynamix is capable of representing specifications for two languages with widely different paradigms. However, just the ability to implement a specification for a language alone does not show that Dynamix succeeds in the goals we set out to achieve in Chapter 3. In the remainder of this chapter, we will evaluate the performance of the Dynamix project by reviewing the original design objectives and relating them back to the case studies.

```
rules
  evalStrategy(t, Seq(a, b)) = {
    success <- fresh-var(suc)
    t2 <- fresh-var(t)
    with $success = {
      fix {
        fun success([t2]) = evalStrategy(t2, b)
      }
      success
    } do
      evalStrategy(t, a)
  }
```

Figure 9.11: A fixed implementation of the sequencing operator that uses pluggable meta-globals for the success and failure continuations. This ensures that the insertion of the completion handler is delayed until when it is invoked.

9.3.1 Simplicity and adoptability

One of the core design goals for the Dynamix DSL was to provide an easy to adopt meta-language, especially for existing users of the Spoofax language workbench. It is hard to quantify this aspect objectively, so instead we will highlight which design decisions for the meta-language help and harm this objective.

Generally, the Dynamix meta-language follows the same conventions set out in other Spoofax meta-languages. This includes the use of ATerms as data format, the same module system as used in SDF3, Statix, and Stratego, and similar syntax for rule and signature declarations. These similarities increase adoptability for existing users of the Spoofax workbench.

The presence of a strict type system for the meta-language helps adoption by offering instant feedback for erroneous specifications. However, the use of new abstractions, such as pluggable terms, makes the type system hard to grasp without at least a surface-level understanding of their implementation. As a result, the presence of the type system trades simplicity for adoptability.

The Dynamix DSL has a relatively small amount of language constructs, with little syntactic sugar. A basic understanding of a typical Dynamix specification can easily be used to write new specifications, especially since Dynamix specifications often tend to have similar implementations for similar features. This is exemplified by the specification for ChocoPy, which shares a lot of similarities with the specification for Tiger discussed in Chapter 6. This increases adoptability, especially if the user has access to example specifications.

A lack of syntactic sugar constructs harms the usability of the language. In particular, a lack of generics means that users are required to redefine certain operations on lists for every possible sort they wish to work with. Within the ChocoPy and miniStratego specifications alone, there are 16 instances of rules that are variations on map or reduce operations (an example of such a rule can be seen in Figure 9.12). Generics, first-class rules, or syntactic sugar such as Statix's `maps` construct, could help reduce the amount of redundant code.

Despite this verbosity, the ChocoPy and miniStratego specifications are reasonably compact. Both languages can be fully represented in less than 800 lines of Dynamix source, with the

```
rules
  compileExps :: List('Exp) -> Pluggable(List(@cval))
  compileExps([]) = []
  compileExps([h|tl]) = {
    eh <- compileExp(h)
    etl <- compileExps(tl)
    [eh|etl]
  }
```

Figure 9.12: Small variations on mapping and folding operations, such as the one shown in this figure, occur throughout the specifications for both case studies. Syntactic sugar or support for generics could help eliminate some of this duplicate code.

Size of specification in lines of code	
ChocoPy with exceptions	723
miniStratego	585

Table 9.13: The size of the specifications implemented. Line counts exclude generated sources such as type signatures, as well as empty lines and comments.

Dynamix specification for ChocoPy even being smaller than the Statix specification for the same language. Table 9.13 shows an overview of the size of the implemented case study specifications.

As previously mentioned, the TU Delft also uses ChocoPy for the CS4200 Compiler Construction course. The reference compiler written by course staff, which compiles ChocoPy to RISC-V assembly, is approximately 1600 lines of Stratego source. This is more than twice the length of the ChocoPy specification, and this difference increases when we consider that the reference compiler has no support for exceptions. While it is not a fair comparison to directly compare a specification to a complete compiler, it shows that extracting out a common runtime platform (i.e. the Tim interpreter) has tangible benefits in reducing the size and complexity of specifications.

9.3.2 Language paradigm applicability

The second core goal of the Dynamix project is to design a DSL that can be applied to a large variety of source languages. As with the simplicity goal, it is not trivial to ascertain this property based solely on the work done in this project.

Dynamix was intentionally built on the CPS target platform to allow low-level access to the control flow of the program. As shown in the ChocoPy with exceptions case study, this approach allows for complex control flow while at the same time isolating this control flow to just the language features that participate in it. Table 9.14 shows an overview of the overhead of adding exception support to an existing ChocoPy specification, including changes needed to existing rules. The relatively small changes needed to existing code, compared to a specification that uses small- or big-step semantics (as discussed in Section 4.3), suggests that CPS is a suitable tool for abstracting over complex control flow language features.

Adjustments to existing code	
Rules changed	5[†]
Lines added	3
Lines modified	11
New code	
Lines added	84

Table 9.14: The number of code changes needed to extend the ChocoPy specification with support for exceptions. Empty lines are omitted.

Through the specifications written for both Tiger and ChocoPy with exceptions, we have already shown that the Dynamix DSL is powerful enough to represent classical imperative programming languages. Through the case study for miniStratego, we have also shown that a less conventional programming paradigm with failure as a first-class citizen can be represented in Dynamix without major restrictions. While this does not necessarily show that Dynamix is capable of every possible language paradigm, it does show promise.

The primitive system in Dynamix and Tim, while only used for simple operations in the languages discussed in this document, also lends itself well to possibly extending the range of supported languages by Dynamix. One such example might be the development of dedicated unification primitives, such that a logic programming language like Prolog might be represented in the Dynamix language.

9.3.3 Performance

The Tim target language was designed such that it will be possible to write a performant compiler for the language as future work. While the current implementation of both the Dynamix and Tim interpreters has no focus on performance, there are no major design decisions in either of the languages that would hamper the development of a faster runtime. For completeness' sake, we will briefly discuss the performance of the current Dynamix interpreter and Tim runtime, and compare it against the reference compiler used in the CS4200 Compiler Construction course at the TU Delft, which is written in Stratego and compiles the source program to RISC-V assembly.

Performance on language conformance test-suite

We first compare the performance of the Dynamix specification against the reference compiler on the test suite that we use for student code in the CS4200 Compiler Construction course at the TU Delft. This test suite consists of 329 tests that assert the correct behavior of the compiler. Both of implementations use the same SDF3 and Statix specifications, allowing us to compare exactly the performance differences between the two for both the compilation stage, as well as executing the compiled program. The results of this benchmark can be seen in Table 9.15.

These benchmarks show that Dynamix has no performance issues running a large test suite of small programs. This is a good sign, as that is the most common use of Dynamix and the baseline Tim interpreter. One of the benefits of having Dynamix directly integrated within

[†]Adjustments were made to the implementation of class definitions, function definitions, function calls, and class member calls. These adjustments include the addition of the raise continuation as parameter, and passing the current continuation as argument.

Compiler	Compile (avg. per testcase)	Running (avg. per testcase)
Dynamix specification	11,663ms (35.4ms)	819ms (2.5ms)
Reference compiler	17,341ms (52.7ms)	2,090ms (6.4ms)

Table 9.15: The time needed to run all 329 test cases in the ChocoPy test suite for both the Dynamix specification and the reference compiler used at the TU Delft. Both implementations pass all tests. Time spent only tracks compilation and runtime performance, as both implementations use the same SDF3 and Statix specifications for parsing and static analysis.

Testcase	Compiler	Analysis time	Compilation time (l/s)	Run time
`prime.cpy`	Dynamix	932ms	67ms (448 l/s)	1,530ms
25 LOC	Reference	932ms	74ms (405 l/s)	29ms
`exp.cpy`	Dynamix	918ms	65ms (385 l/s)	386ms
30 LOC	Reference	918ms	80ms (312 l/s)	35ms
`tree.cpy`	Dynamix	1,078ms	206ms (403 l/s)	7,779ms
83 LOC	Reference	1,078ms	280ms (296 l/s)	215ms

Table 9.16: The time it takes to compile and run various ChocoPy benchmark snippets for both the Dynamix specification and the CS4200 reference compiler. Analysis time is shared between the two implementations, as both of them use the same Statix specification. Statistics in brackets indicate represent the amount of source code lines compiled per second.

the Spoofax language workbench is the ability for language designers to quickly iterate on specifications, which generally involves repeated runs of a language test suite. Indeed, the benchmarks show that both the Dynamix meta-interpreter and the Tim interpreter are fast enough to allow such workflows.

Performance on benchmark programs

The performance of the Tim runtime falls off significantly when we specifically consider programs intended for benchmarking the performance of the language. Such programs are generally much larger and perform more control flow, exposing the performance penalties incurred by the unoptimized implementation of tail calls in the Tim interpreter. Table 9.16 lists the performance of both compilers for a series of simple benchmark programs.

It is clear that the extra effort done by the reference compiler, which includes dataflow analysis and register allocation, positively benefits the runtime performance of its generated programs. Despite needing to run in a RISC-V emulator, the programs produced by the reference compiler are an order of magnitude faster than those of the naive Tim interpreter. While this shows that the performance of the current Tim interpreter is nowhere near fast enough to conform to the performance goals outlined in Chapter 3, one must consider that this Tim interpreter is not designed for performance. Future work on a faster Tim runtime can significantly improve the performance of any source languages that use Dynamix.

However, these benchmarks reaffirm that the meta-interpretation performance of the Dynamix meta-language is sufficient even for larger programs. If one only needs to inspect the generated Tim IR, even compilation of larger programs can be done in less than a second. In daily use of the Spoofax language workbench, the time spent on static analysis using the Statix constraint solver easily dwarves the time spent on meta-compilation. We therefore conclude that the performance of the meta-compiler is no cause for concern.

9.3.4 Integration within Spoofax

As the final requirement for Dynamix, we established that the language is designed specifically for use within the Spoofax workbench. As a result, we expect first-class interaction with other parts of the workbench, as well as easy integration within existing projects.

The case studies show that Dynamix passes both of these requirements. Type signatures are automatically generated from the SDF3 grammar of a language, and both the ChocoPy and miniStratego case studies make use of Dynamix's integration with the Statix constraint solver. This interaction is frictionless, requiring no additional setup on the user end. Static analysis features for Dynamix are on-par with other languages in the workbench, and the ability to execute Dynamix either manually or as part of automated testing is integrated directly within the Eclipse IDE.

The ChocoPy case study additionally made use of an existing Spoofax language workbench project, consisting of an SDF3 grammar definition and a Statix specification. Integrating a Dynamix specification for the language into this project required no changes to the grammar of the language, nor the Statix specification. Only minor changes were needed to the Statix specification to support automatic boxing of values, but this boxing approach is not necessary to write a complete and correct specification (an earlier version of a Dynamix specification for ChocoPy did not automatically box, and hence required no changes to the Statix specification).

Chapter 10

Related work

The Dynamix project is by no means the first attempt at the creation of a meta-language for dynamic semantics. In fact, a significant portion of the concepts used in Dynamix originates from earlier work done within the programming languages research community. In this chapter, we briefly discuss related work in the domain of dynamic semantics, meta-languages, and the compilation of CPS-based languages.

10.1 Meta-languages for dynamic semantics

Dynamix on the FrameVM

Chiel Bruin's Dynamix on the FrameVM [13] is the closest comparable project to Dynamix. It offers a meta-language comparable in features and expressiveness to the Dynamix from this paper, but targets the FrameVM: a virtual machine abstracting over CPS using control frames. Control frames are sequences of VM instructions that operate uniformly over control flow, akin to basic blocks in traditional compiler design. The Dynamix project presented in this document inherits its name from Bruin's project, as discussed in Section 2.3.

The primary goal of Bruin's Dynamix was to develop a minimal set of instructions for a target machine that could still support a large variety of source languages. To this extent, the FrameVM and the accompanying bytecode language Roger were developed. Control flow within the FrameVM uses CPS, with code organized in blocks called "control frames". These frames have linear control flow within them, terminating by jumping into a different frame or invoking a continuation. These frames additionally share a single set of addressable memory locations, similar to a stack frame. These memory frames are based on the concept of scopes-as-frames introduced by Bach Poulsen et al. [52]. Bruin's Dynamix is a meta-language for compiling source programs to the Roger bytecode.

While the general aim of both Bruin's FrameVM and our Tim IR is similar, the two are widely different in implementation. The hole abstraction used in Dynamix exists solely on the meta-level, whereas Bruin's use of frames is deeply ingrained in both Dynamix, Roger, and the FrameVM. Roger is also decidedly lower-level than Tim: conditionals must be modeled as jumps, and variables must be addressed using indices instead of names. The primitive operations available to Roger programs are also hardcoded, as opposed to the extensible model used in Tim.

Similar to the version of Dynamix presented in this paper, Bruin's Dynamix was designed to be integrated within the Spoofax language workbench. Its integration within this workbench is arguably deeper than this project, as it directly integrates the scope graph [4] generated by static analysis into the memory layout of the target program. However, it lacks integration with the SDF3 grammar of a language, as Bruin's Dynamix does not offer static analysis features beyond asserting name binding correctness.

DynSem

DynSem [64] was the first attempt at designing a meta-language for dynamic specifications within the Spoofax language workbench. Within DynSem, dynamic semantics are modeled as reduction rules. Users can define arbitrary reduction "arrows" and declare which input patterns should evaluate to which result. Control flow is implemented through inductive rules and conditional reduction rules.

The design of DynSem is heavily inspired by the style of formal language specifications. Each reduction rule is implicitly parameterized with both an environment and a heap, with syntactic sugar allowing the user to omit most of this notation for rules that do not require direct access to the environment (an approach similar to Scala's implicit arguments). While the reduction notation used by DynSem is closer to the formal specifications that inspired it, this similarity comes at the cost of a significantly more complex specification when non-linear control flow (e.g. exceptions, early returns) is desired. Such non-linear control flow is generally implemented by yielding an algebraic data type representing either a concrete value, or a flag indicating that some non-linear control flow should be performed. While this is a valid approach, it requires every rule to check for, and possibly propagate, non-linear control flow flags whenever it performs a recursive invocation of a reduction rule.

DynSem specifications are fully type-checked and support signature generation from SDF3 specifications. However, it is not possible to interact with the results of static analysis. It is therefore not possible to conditionally evaluate some construct based on the type of the expression, nor is it possible to access other static analysis information such as the fields of a composite type.

DynSem specifications are directly compiled to a Java AST-based interpreter. Primitive operations are modeled through "native operators", which are implemented directly as Java source code. The performance of this generated interpreter is acceptable, and later work on the project [66, 65] improves the performance to within an order of magnitude of a handwritten interpreter by running the interpreter within the Graal VM and integrating static analysis results into the memory model of the interpreter (the Scopes-describe-Frames paradigm [52]).

K Semantic Framework

The closest comparable project to Dynamix that does not use the Spoofax language workbench is the K Semantic Framework [54]. K is a framework that provides tools for defining language grammar definitions, type systems, and formal semantics. Unlike Dynamix or DynSem, K's primary focus is not on the generation of language runtimes, but rather on the systematic definition of the language semantics so that they may be tested, analyzed, or verified.

Semantics in K are defined as rewrite rules that operate within a specific *cell*. Cells are K's representation of a configuration of the program, such that a collection of all cells represents the exact state of the program at a specific point in time. K cells may contain program terms, locals, input, output, or any other data required for the execution of the language. Rewrite rules can be defined as acting on one or more cells at a time, updating the contents of these cells. One example is the `k` cell, which conventionally starts with the entire program and carries the current remainder of the program.

Complex control flow is simple to perform in K due to the cell architecture. An example is the Scheme `call/cc` construct, which is trivially implemented in K due to the access it provides to the remainder of the computation (exactly the thing that the `k` cell represents). Other control flow structures, such as loops, are defined through inductive rewriting rules, similar to how they might be represented in a formal definition of the language.

The primary use of the K Semantic Framework is for program and specification verification. While recent developments have created an LLVM [38] backend capable of yielding impressive performance, a more common use of specifications written in K is to verify certain invariants in the language using the Z3 [46] solver. K specifications exist for a large number of languages and platforms, many of which double as the most complete specification of that language. Currently maintained specifications include specifications for WebAssembly, the Ethereum Virtual Machine, and the C programming language.

PLT Redex

A project similar to the K Semantic Framework is PLT Redex [19]. PLT Redex is a DSL for the Racket [20] programming language that allows users to define grammars, type systems, and dynamic semantics using a scheme-like notation.

Dynamic semantics are defined in PLT Redex through reduction rules. The language author is able to specify a pattern that the input should match, as well as the substitution that should be performed. Unlike the pattern matching done in Dynamix and DynSem, PLT Redex allows the reduction of arbitrarily nested terms through the `in-hole` abstraction. This abstraction can identify a subterm that satisfies a given input, producing both the subterm and the original term with the subterm replaced by a hole[1]. This hole is then filled by the reduced subterm. Through the in-hole abstraction, reduction rules in PLT Redex can apply to any arbitrary sub-term of the input.

While reduction rules in PLT Redex operate solely on terms of the input grammar, this grammar can be wrapped by some state representation in order to expose this state to reduction rules. Control flow can be performed through conditional reduction rules, or by using the `in-hole` abstraction, which is powerful enough to implement Scheme's `call/cc`. Combining these abstractions allows for the simulation of a large range of language constructs, including concurrent ones.

The main objective of PLT Redex is not to provide an efficient runtime for languages, but rather the visualization, analysis, and verification of the semantics of a language. For this purpose, Redex ships with various tools that can visualize all possible reduction trees for a source program, first-class support is included for testing reductions, and automatic random test cases can be generated and verified for any given grammar.

JetBrains MPS

MPS [47] is a developer tool created by JetBrains for the development of domain-specific languagess. Unlike the other projects mentioned, DSLs created using MPS do not have a

[1]Not to be confused with the hole from Dynamix. While both conceptually have the same function, the deferment of completion of some term by acting as a placeholder, the use of holes in Dynamix is significantly different from the use of holes in PLT Redex

concrete grammar. Instead, an editor is generated that works directly on the AST of the language. A benefit of this approach is that the "syntax" of a language does not necessarily have to be text: languages developed in MPS can include tables, graphics, and other visual elements directly inside a program. However, it no longer becomes possible to develop programs outside of the editor generated by MPS.

Languages developed using MPS cannot specify dynamic semantics directly within MPS. Instead, the language author must define how each AST node "compiles" to some target language. For example, one may define how their language constructs map to Java source code, which can then be subsequently compiled and executed. This compilation process is a direct translation between the source (meta-language) AST and the target AST, although this AST can be pretty-printed to generate target program source files. While MPS has first-class support for generating Java source, it can also be used to generate JavaScript, LaTeX, XML, and more.

Attribute grammars

Beyond dedicated projects for dynamic semantics, some parser generation suites support the specification of runtime semantics through attribute grammars [36]. Such grammars allow the inline specification of how a certain language construct should be compiled or evaluated. While attribute grammars do not directly specify the semantics of the language construct, they instead rely on the semantics of the language to which they compile. Popular parser generator suites that support this feature include Yacc [30], ANTLR [51], and Bison [16].

Monadic interpreters

Dynamix's definition of the hole abstraction, alongside the use of implicit composition and the `<-` operator, heavily borrows from ideas found in the domain of monadic interpreters [70] and monadic semantics. Indeed, the block notation in Dynamix is effectively a hyper-specialized version of Haskell's `do`-notation, with pluggable term composition and the arrow operator acting similar to the monad's bind function.

One of the core benefits of monadic interpreters, as Wadler identifies in *Monads for functional programming* [70], is that interpreters written in an imperative style require substantial changes to existing code when a language is extended (exactly the same problem that denotational semantics suffers from). By parameterizing an interpreter with some monad M, it becomes possible to abstract the operations that require changing, such that the behavior of an interpreter can be extended (e.g. adding exception support) without requiring significant changes to existing code. As a downside to this approach, it generally becomes harder to implement multiple extensions (e.g. adding both exception support and non-determinism), as monads do not always easily compose with each other. Liang, Hudak, and Jones [39] resolve this issue through the introduction of "monad transformers", allowing for truly modular definitions of interpreters and their features.

Closer to the efforts described in this paper, Bernard Bot's master thesis [12] discusses the use of command trees in monadic interpreters. Command trees are an abstraction based on the concept of free monads, whose ability to represent truly modular data types was by popularized by Swierstra [61]. Bot compares the process of writing a CPS-based compiler in "plain" Haskell to an equivalent compiler that makes use of the command tree abstraction. In his thesis, he shows that command trees allow for "expressing compiler transformations typically, declaratively, and modularly". Whereas Dynamix only borrows the concept of monads and do-notation from functional languages, Bot's thesis shows how free monads and command trees can be modeled to provide succinct and type-safe compiler transformations in a language like Haskell.

10.2 CPS as a tool for compilation

The use of a CPS-based IR as part of the compilation pipeline is not new. In fact, research into the use of CPS for compiler optimizations, the lowering of CPS into target machine code, and the benefits and drawbacks of CPS as a tool for compilation has been actively ongoing ever since the first large use of CPS in the Rabbit Scheme compiler [59], more than 40 years ago. While "pure" CPS is uncommon in modern-day compilers, many compilers use techniques either derived from the CPS or shown to be isomorphic to concepts from CPS-based IRs. In Compiling with Continuations, or without? Whatever. [15], Cong et al. succinctly describe the recent research into CPS-based intermediate representations.

Most of this research does not directly apply to Dynamix at this point in time. Dynamix performs the compilation of source ASTs to a CPS-based IR, but it performs no further optimizations or lowering on this IR. However, the choice to make use of CPS was directly influenced by the fact that a large amount of research related to efficient CPS-based compilation exists. Potential future work on improving the Tim runtime is expected to make use of this research to increase the performance of code generated using a Dynamix specification.

The use of CPS for defining the mathematical semantics of a language is not new either. In fact, one of the first known formulations of the concept of continuations is by Strachey and Wadsworth [60], who use it as a tool to describe the mathematical semantics of programming languages. They observed that continuations can function as a useful primitive for abstracting over the remainder of a computation, and that this abstraction allows for a very straightforward method of describing the mathematical semantics of languages with jumps. The decision to use CPS as a target IR for Dynamix was directly inspired by this work.

Finally, the specific design of the Tim CPS IR is greatly influenced by the description of the CPS datatype in Andrew Appel's book, Compiling with Continuations [6]. While this document only defines the static and dynamic semantics of the CPS, Appel's book goes much further and also outlines the compilation techniques used to efficiently compile such a CPS IR to machine code.

Chapter 11

Future work

While the version of Dynamix presented in this paper is capable of representing a large variety of languages, it has a large number of aspects that can be improved upon. In this chapter, we discuss some of the obvious next steps for both the Dynamix meta-language, as well as the Tim intermediate representation.

11.1 The Tim intermediate representation

The current features of the Tim IR are complete enough to represent everything that the Dynamix meta-language requires. However, the performance of the interpreter is lacking and Tim programs must be a single file. We will consider some logical extensions to the Tim IR. It should be understood that many of these changes may require similar changes in the Dynamix meta-language.

11.1.1 Support for multi-file programs

Currently, a Tim program consists of a single expression in a single file. While this is sufficient for compilation from Dynamix, it becomes impossible to compile files of a multi-file source language in isolation. A lack of a Tim module system also means that any standard library included as part of a language must be embedded directly in the output Tim file, as opposed to being included in the output file.

A module system for Tim should be considered that allows for the definition of a single Tim program across multiple files. Such a system may also need to consider the ability for files to be compiled in parallel, in order to allow incremental compilation when changes only affect a single module. As part of a module system, the Tim grammar should be extended to support multiple top-level definitions beyond a single expression. A potential module system for Tim could be inspired by the module system used in Appel's Compiling with Continuations [6], as the Tim IR is inspired by the CPS language used in this book.

11.1.2 Introduction of a type system

Tim lacks any static semantics beyond name binding. An experimental Statix specification exists (see Section 5.4.2) that is able to detect several classes of errors, but it is imprecise and may reject valid code. The introduction of a type system into the language would allow static analysis to reject erroneous programs, turning a large number of runtime errors into compilation errors. The addition of a type system may additionally help improve the performance of the Tim runtime, as it allows for the shapes of objects to be known ahead of time.

If a type system is designed, careful consideration should be taken for the features present. Tim is intended as the target representation for a wide array of source languages, so the type system must not needlessly restrict which languages can be compiled using Dynamix. The type system should also consider the presence of arbitrary primitives, which may require or introduce new types. If a type system is introduced, the Dynamix meta-language should also be extended to allow it to derive and emit the appropriate Tim types for corresponding source programs.

11.1.3 Improve runtime performance

The current Tim interpreter has no special performance considerations. In order to achieve performance similar to an interpreter generated by DynSem [64], a performant (JIT-capable) interpreter or compiler for Tim should be created. Such a runtime can perhaps be based on the CPS compilation techniques discussed by Appel [6], as Tim is based on the language presented in his book. As part of an optimized runtime for Tim, it may be beneficial to introduce more specialized primitives for common operations. Similarly, a type system (as discussed earlier) would likely improve the performance of the runtime.

A compiler backend of Tim must also consider a memory model for the language. The current Tim interpreter is written in Stratego, and hence relies on the JVM garbage collector to handle memory management. A Tim compiler that compiles to machine code must either expose memory management primitives, or must include some form of garbage collection or reference counting. Special care must be taken if one wishes to support source languages where deallocation is observable (e.g. finalizers in C# or Java, the Drop trait in Rust, or destructors in C++).

11.2 The Dynamix meta-language

Beyond the future work that can be done on the Tim target language, various improvements can be made to the Dynamix meta-language to make it more approachable for users of the Spoofax language workbench. Most of this future work focuses on making the Dynamix meta-language easier to use, or allowing it to emit simpler or more performant Tim programs.

11.2.1 Decrease verbosity through syntactic sugar

As mentioned in Section 9.3, performing operations on lists of sorts is excessively verbose in Dynamix. The possibility of introducing syntactic sugar into the language should be considered, such that these operations can be simplified. One such approach may be to implement syntax similar to Statix's `maps` syntax, which automatically generates a rule that maps over a list of inputs. When introducing such syntactic sugar, the existing case study specifications should be reviewed to identify common patterns suitable for simplification through syntactic sugar.

11.2.2 Extend type system with generics or first-class meta-functions

An alternative approach to reducing the verbosity of the language is to introduce support for generic functions in the Dynamix type system. Such an extension would likely have to be paired with support for first-class meta-functions, as seen in Chiel Bruin's version of Dynamix [13]. Proper support for generics and first-class functions could greatly increase the usability of the language, without requiring dedicated syntactic sugar for common language constructs. One may additionally consider the creation of a Dynamix "standard library", containing implementations of common operations such as maps or folds.

11.2.3 Expand source term computation support

Currently, source language AST ATerms in Dynamix are largely opaque to the specification. They can be matched upon with rules and can be passed to `str`, `int`, or `var`. If the source term is a string, it can additionally be concatenated to another string using the `+` operator. However, it may be useful to expose a wider range of computations to specifications. For example, it is currently impossible to compute the length of a source list in Dynamix alone, as it is impossible to perform arithmetic on source integers. Consequently, it is impossible to conditionally compile a construct based on the length of a source list.

One possible approach could be to design a separate embedded "constant operations" language that exposes arithmetic, comparisons, and potentially even certain functions to source terms. Such a language could be similar to e.g. `constexpr` in C++ or `const` functions in Rust. However, this might be an overkill approach and one would need to be careful to ensure that it is unambiguous to users which operations are performed during compilation and which are performed at runtime.

11.2.4 Support for multi-file source languages

The current Dynamix meta-language is only able to compile a single source file at a time. While it is likely possible to compile multi-file source languages in this manner, first-class support for multi-file source languages should be considered, as multi-file languages are directly supported in the Spoofax language workbench. Adding support for multi-file source languages would likely also require a module system for Tim, as otherwise the specification would be forced to perform "merging" of several files source by itself.

11.2.5 Deeper integration with scope graphs

The current interaction that Dynamix has with the results of static analysis is limited to the ability to query properties on AST nodes. A future improvement to Dynamix could extend this support to full interaction with the scope graph [4] of the program, allowing the specification to directly query the structure of scopes and types. This concept could be extended even further into the domain of scopes-describe-frames [52], allowing composite datatypes to be based directly on the structure of the scope graph. Support for this paradigm would require similar changes to the Tim language.

11.2.6 Explore support for more exotic programming languages

While the case studies of ChocoPy and miniStratego represent two different programming paradigms, both of them are undoubtedly imperative programming languages. It is still unknown whether Dynamix is able to represent languages that depart from this paradigm. Future work could investigate whether Dynamix is capable of representing logic programming languages like Prolog [71], query languages like SQL [28], or concurrent programming languages like Erlang [10].

11.2.7 Explore using specifications for other goals than compilation

While the Dynamix DSL was designed specifically as a tool for compilation of source languages to the Tim IR, this does not necessarily need to be the only use of Dynamix. After all, a specification in Dynamix is a machine-readable specification of the dynamic semantics of the language. Future work in this space could explore the usage of a Dynamix specification for applications such as formally proving the equivalence of two source programs, or asserting that some refactoring does not alter the behavior of the program.

Chapter 12

Conclusion

Most programming languages are defined through language specifications. Specifications, commonly written in natural language or occasionally defined using formal notation, describe exactly the behavior of the language. However, one must manually verify that a given implementation of a language actually conforms to the language specification, since the two are distinct. Meta-languages for dynamic specifications attempt to solve this problem, by treating a language specification as a domain-specific language and automatically generating a language runtime from this specification.

In this thesis, we presented Tim and Dynamix. Dynamix is a new meta-language for dynamic specifications, using the technique of meta-compilation to produce programs in the Tim intermediate representation. Novel to the domain of dynamic specification meta-languages is Dynamix's use of continuation-passing style for control flow, which is an expressive method for specifying the control flow of languages while isolating this control flow only to the sections of the language that are directly affected by it. In order to avoid the traditional verbosity of building CPS terms, Dynamix introduces the hole abstraction. This abstraction allows fragments of CPS to be built in isolation, automatically combining them when needed.

Tim programs, the compilation target for Dynamix, are fully completed CPS terms. Primitives expose operations such as arithmetic, composite data types, and conditionals to the language. To increase the number of languages that can be represented, language authors can additionally implement their own primitives. Currently, Tim programs are interpreted by an interpreter written in Stratego. However, the language has been carefully designed to allow for efficient compilation as future work.

Dynamix is situated within the Spoofax language workbench. It has direct integration with the SDF3 meta-language for automatic AST type signature generation, as well as the ability to query the results of static analysis performed using Statix. Dynamix can be integrated into existing Spoofax language projects and its compilation artifacts can be automatically tested using the Spoofax testing language.

To confirm that Dynamix is capable of representing various paradigms of languages, we discussed specifications for Tiger, ChocoPy with exceptions, and miniStratego. Each of these specifications was able to represent the entirety of the language in a concise manner, indicating that Dynamix is capable of representing non-trivial language features with relative ease.

Future work on Tim can significantly increase the performance of the language by developing an optimizing compiler for the language. As part of these efforts, a type and module system for the language could also be considered. Future work on Dynamix is suggested to explore syntactic sugar or type system extensions for making the language more concise, as well as investigating whether Dynamix is able to represent languages whose paradigms are further removed from those of traditional languages, such as logic programming languages like Prolog or query languages like SQL.

Appendix A

Grammar of the Dynamix meta-language

MID	::=	**[a-zA-Z] [a-zA-Z0-9_/-]***	*module ID*
RID	::=	*LID*	*rule ID*
LID	::=	**[a-z] [a-zA-Z0-9_]***	*lowercase ID*
UID	::=	**[A-Z] [a-zA-Z0-9_-]***	*uppercase ID*
GID	::=	**$***LID*	*global ID*
PID	::=	**[-+*/A-Za-z0-9_]**$^+$	*primitive ID*
INT	::=	**-**$^?$ **[0-9]**$^+$	*integer literal*
STRING	::=	**"** *string-char** **"**	*string literal*
string-char	::=	**\n**	
	|	**\\"**	
	|	\\	
	|	*any character except " or newline*	
program	::=	**module** *MID section**	
section	::=	**imports** *MID**	*module imports*
	|	**signature** *signature-decl**	*algebraic signature declarations*
	|	**constraint-analyzer** *constraint-decl**	*constraint analyzer interop declarations*
	|	**primitives** *primitive-decl**	*target primitive declarations*
	|	**rules** *rule-decl**	*rule declarations*
rule-decl	::=	*GID* **::** *meta-type*	*global declaration*
	|	*RID* **::** *rule-signature*	*rule declaration*
	|	*RID***(**⟨ *pattern***,** ⟩***)** **=** *expr*	*rule implementation*
pattern	::=	**_**	*wildcard*
	|	*LID*	*bind variable*
	|	*UID***(**⟨ *pattern***,** ⟩***)**	*constructor*
	|	**[**⟨ *pattern***,** ⟩***]**	*list pattern*
	|	**[**⟨ *pattern***,** ⟩***|***pattern***]**	*head-tail list pattern*
	|	*STRING*	*source string literal*
	|	*INT*	*source int literal*
	|	*LID***@***pattern*	*bound pattern*
signature-decl	::=	**sorts** *UID**	*sort declarations*
	|	**constructors** *constructor-decl**	*constructor declarations*
constructor-decl	::=	*UID* **:** *UID*	*singleton constructor*
	|	*UID* **:** ⟨ *constructor-arg* ***** ⟩$^+$ **->** *UID*	*named constructor*
	|	**:** *constructor-arg* **->** *UID*	*injection*

constructor-arg	::=	*UID*	*sort reference*
	|	**string**	*source string*
	|	**int**	*source int*
	|	**List(**_constructor-arg_**)**	*list type*
constraint-decl	::=	**property** *ID* **::** *meta-type*	*constraint analyzer property*
primitive-decl	::=	**expression #**_PID_**(**⟨ *meta-type*, ⟩***)**	*expression decl*
	|	**statement #**_PID_**(**⟨ *meta-type*, ⟩***)**	*statement decl*
	|	**conditional #**_PID_**(**⟨ *meta-type*, ⟩***)**	*conditional decl*
source-type	::=	*UID*	*aterm sort reference*
	|	**string**	*aterm string type*
	|	**int**	*aterm integer*
meta-type	::=	**'**_source-type_	*source type*
	|	**@**_target-type_	*target type*
	|	**List(**_meta-type_**)**	*meta-list type*
	|	**Pluggable(**_meta-type_**)**	*pluggable type*
	|	**Pluggable**	*pluggable with value type*
target-type	::=	**value**	*Tim value*
	|	**statement**	*Tim CPS type*
	|	**fun**	*Tim function decl*
expr	::=	**'**_source-term_	*source term literal*
	|	*expr* **+** *expr*	*source string concatenation*
	|	**{** *statement*$^+$ **}**	*block*
	|	*LID*	*meta-variable reference*
	|	*GID*	*meta-global reference*
	|	*RID*(⟨ *expr*, ⟩*)	*rule call*
	|	**with** ⟨ *binding*, ⟩$^+$ **do** *expr*	*scoped meta-global binding*
	|	**[**⟨ *expr*, ⟩***]**	*meta-list literal*
	|	**[**⟨ *expr*, ⟩***|**_expr_**]**	*meta-list literal*
	|	*expr* **++** *expr*	*meta-list concatenation*
	|	**nameof(**_expr_**)**	*extract name of fun*
	|	**int(**_expr_**)**	*coerce to target int*
	|	**str(**_expr_**)**	*coerce to target string*
	|	**var(**_expr_**)**	*coerce to target var*
	|	**fresh-var(**_ID_**)**	*generate unique target var*
	|	*expr***@(**_expr_**)**	*tail-call*
	|	**#**_PID_**(**⟨ *expr*, ⟩***)**	*primitive call*
	|	**fun** *expr***(**_expr_**) =** *expr*	*function decl*
	|	**fix {** *expr** **}**	*fix decl*
	|	**let** *expr* **=** *expr* **in** *expr*	*let decl*
	|	**if #**_PID_**(**⟨ *expr*, ⟩***) then** *expr* **else** *expr*	*conditional primitive call*
	|	*expr* **label** *LID***/1:**	*unary label*
	|	*expr* **label** *LID***/0:**	*nullary label*
	|	**@**_tim-expr_	*embed Tim expression (deprecated)*
	|	**hole**	*hole*

statement	::=	*ID* `<-` *expr*	*assignment*
	|	*expr*	*expression*
binding	::=	*GID* `=` *expr*	*scoped meta-global binding*
source-term	::=	*INT*	*int literal*
	|	*STRING*	*string literal*
	|	*UID*`(`⟨ *source-term*`,` ⟩*`)`	*constructor*
	|	`[` ⟨ *source-term*`,` ⟩* `]`	*list*

Bibliography

[1] *Agda*. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN: 0-201-10088-6.

[3] Hendrik van Antwerpen. “A Constraint-based Approach to Name Binding and Type Checking using Scope Graphs”. Available at http://resolver.tudelft.nl/uuid:7a555c92-ee75-4e64-b58b-d8f09662f412. MA thesis. Delft University of Technology, Jan. 2016. URL: http://resolver.tudelft.nl/uuid:7a555c92-ee75-4e64-b58b-d8f09662f412.

[4] Hendrik van Antwerpen et al. *A Constraint Language for Static Semantic Analysis based on Scope Graphs with Proofs*. Tech. rep. TUD-SERG-2015-009. Available at http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2015-009.pdf. Software Engineering Research Group, Delft University of Technology, Sept. 2015. URL: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2015-009.pdf.

[5] Hendrik van Antwerpen et al. “Scopes as types”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018). DOI: 10.1145/3276484. URL: https://doi.org/10.1145/3276484.

[6] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN: 0-521-41695-7.

[7] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998. ISBN: 0-521-58390-X.

[8] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998. ISBN: 0-521-58388-8.

[9] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. ISBN: 0-521-58274-1.

[10] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. ISBN: 193435600X.

[11] Sandrine Blazy and Xavier Leroy. “Mechanized Semantics for the Clight Subset of the C Language”. In: *Journal of Automated Reasoning* 43.3 (2009), pp. 263–288. DOI: 10.1007/s10817-009-9148-3. URL: http://dx.doi.org/10.1007/s10817-009-9148-3.

[12] Bernard Bot. “Compiling with Command Trees”. Available at http://resolver.tudelft.nl/uuid:05092e06-d67e-404a-b1ab-74bef499d3f2. MA thesis. Delft University of Technology, May 2021. URL: http://resolver.tudelft.nl/uuid:05092e06-d67e-404a-b1ab-74bef499d3f2.

[13] Chiel Bruin. "Dynamix on the Frame VM: Declarative dynamic semantics on a VM using scopes as frames". Available at http://resolver.tudelft.nl/uuid:ddedce14-65ad-4f16-912e-6b0658eaecc0. MA thesis. Delft University of Technology, Apr. 2020. URL: http://resolver.tudelft.nl/uuid:ddedce14-65ad-4f16-912e-6b0658eaecc0.

[14] *Clang: A C language family frontend for LLVM*. URL: https://clang.llvm.org/.

[15] Youyou Cong et al. "Compiling with continuations, or without? whatever". In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019). DOI: 10.1145/3341643. URL: https://doi.org/10.1145/3341643.

[16] Robert Paul Corbett. "Static Semantics and Compiler Error Recovery". PhD thesis. EECS Department, University of California, Berkeley, June 1985. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1985/5514.html.

[17] Arie van Deursen, Jan Heering, and Paul Klint, eds. *Language Prototyping. An Algebraic Specification Approach*. Vol. 5. AMAST Series in Computing. Singapore: World Scientific, Sept. 1996.

[18] ECMA Ecma. "262: Ecmascript language specification". In: *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,* (1999).

[19] Matthias Felleisen, Robby Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. ISBN: 978-0-262-06275-6. URL: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&amp;tid=11885.

[20] Matthew Flatt. *PLT. Reference: Racket*. Tech. rep. Technical Report PLT-TR-2010-1, PLT Inc., 2010. http://racket-lang. org/tr1, 2010.

[21] *GCC, the GNU compiler collection*. URL: https://gcc.gnu.org/.

[22] James Gosling et al. *The Java Language Specification, Java SE 18 Edition*. 1st. 2022.

[23] James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 013390069X.

[24] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[25] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7". In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018), pp. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[26] ISO. *ISO/IEC 14882:1998: Programming languages — C++*. Available in electronic form for online purchase at http://webstore.ansi.org/. Sept. 1998, p. 732. URL: http://www.iso.ch/cate/d25845.html.

[27] ISO. *ISO/IEC 23270:2003: Information technology — C# Language Specification*. 2003, pp. xiii + 471. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c036768_ISO_IEC_23270_2003(E).zip;%20http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=36768.

[28] ISO. *ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL*. Available in English only. 1992, p. 587. URL: http://www.iso.ch/cate/d16663.html.

[29] ISO. *ISO/IEC 9899:1990: Programming languages — C*. 1990. URL: http://www.iso.ch/cate/d17782.html.

[30] Stephen C. Johnson and Ravi Sethi. "Yacc: A Parser Generator". In: *UNIX Vol. II: Research System (10th Ed.)* USA: W. B. Saunders Company, 1990, pp. 347–374. ISBN: 0030475295.

[31] Gilles Kahn. "Natural Semantics". In: *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*. Ed. by Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Vol. 247. Lecture Notes in Computer Science. Springer, 1987, pp. 22–39. ISBN: 3-540-17219-X.

[32] Karl Trygve Kalleberg and Eelco Visser. "Spoofax: An Interactive Development Environment for Program Transformation with Stratego/XT". In: *Proceedings of the Seventh Workshop on Language Descriptions, Tools and Applications (LDTA 2007)*. Electronic Notes in Theoretical Computer Science. Braga, Portugal: Elsevier, Mar. 2007. URL: `http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2007-018.pdf`.

[33] Lennart C. L. Kats and Eelco Visser. "The Spoofax language workbench". In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/-Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 237–238. ISBN: 978-1-4503-0240-1. DOI: `10.1145/1869542.1869592`. URL: `http://doi.acm.org/10.1145/1869542.1869592`.

[34] Lennart C. L. Kats and Eelco Visser. "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, 2010, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: `10.1145/1869459.1869497`. URL: `https://doi.org/10.1145/1869459.1869497`.

[35] James C. King. "Symbolic Execution and Program Testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[36] Donald E. Knuth. "Semantics of Context-Free Languages". In: *Theory Comput. Syst.* 2.2 (1968), pp. 127–145. URL: `http://www.springerlink.com/content/m2501m07m4666813/`.

[37] Gabriël Konat et al. "Precise, Efficient, and Expressive Incremental Build Scripts with PIE". In: *Second Workshop on Incremental Computing (IC 2019)*. 2019.

[38] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. ISBN: 0-7695-2102-9. URL: `http://csdl.computer.org/comp/proceedings/cgo/2004/2102/00/21020075abs.htm`.

[39] Sheng Liang, Paul Hudak, and Mark P. Jones. "Monad Transformers and Modular Interpreters". In: *POPL*. 1995, pp. 333–343.

[40] Chris Lilley et al. *Cascading Style Sheets, level 2 (CSS2) Specification*. W3C Recommendation. https://www.w3.org/TR/2008/REC-CSS2-20080411/. W3C, Apr. 2008.

[41] Tim Lindholm et al. *The Java Virtual Machine Specification, Java SE 18 Edition*. 1st. 2022.

[42] Jacob Matthews et al. "A Visual Environment for Developing Context-Sensitive Term Rewriting Systems". In: *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*. Ed. by Vincent van Oostrom. Vol. 3091. Lecture Notes in Computer Science. Springer, 2004, pp. 301–311. ISBN: 3-540-22153-0. URL: `http://springerlink.metapress.com/openurl.asp?genre=article&amp;issn=0302-9743&amp;volume=3091&amp;spage=301`.

[43] Antoni W. Mazurkiewicz. "Proving Algorithms by Tail Functions". In: *Inf. Comput.* 18.3 (Apr. 1971), pp. 220–226.

[44] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990. ISBN: 978-0-262-63132-7.

[45] Sangwhan Moon et al. *HTML 5.3*. WD not longer in development. https://www.w3.org/TR/2021/NO html53-20210128/. W3C, Jan. 2021.

[46] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceed*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. isbn: 978-3-540-78799-0. doi: `10.1007/978-3-540-78800-3_24`. url: `http://dx.doi.org/10.1007/978-3-540-78800-3_24`.

[47] *MPS: The domain-specific language creator by JetBrains*. url: `https://www.jetbrains.com/mps/`.

[48] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. "ChocoPy: A Programming Language for Compilers Courses". In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. SPLASH-E 2019. New York, NY, USA: Association for Computing Machinery, 2019. isbn: 9781450369893. doi: `10.1145/3358711.3361627`. url: `https://doi.org/10.1145/3358711.3361627`.

[49] Rohan Padhye et al. *ChocoPy v2.2: Language Manual and Reference*. University of California, Berkeley. Nov. 2019.

[50] Rohan Padhye et al. *ChocoPy v2.2: RISC-V Implementation Guide*. University of California, Berkeley. Oct. 2019.

[51] T.J. Parr and R. W. Quong. *ANTLR: A Predicated-LL(k) Parser Generator*. 1995.

[52] Casper Bach Poulsen et al. "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. isbn: 978-3-95977-014-9. doi: `10.4230/LIPIcs.ECOOP.2016.20`. url: `http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20`.

[53] John C. Reynolds. "The Discoveries of Continuations". In: *Higher-Order and Symbolic Computation* 6.3-4 (1993), pp. 233–248.

[54] Grigore Rosu and Traian-Florin Serbanuta. "An overview of the K semantic framework". In: *Journal of Logic and Algebraic Programming* 79.6 (2010), pp. 397–434. doi: `10.1016/j.jlap.2010.03.012`. url: `http://dx.doi.org/10.1016/j.jlap.2010.03.012`.

[55] Luis Eduardo de Souza Amorim. "Declarative Syntax Definition for Modern Language Workbenches". base-search.net (fttudelft:oai:tudelft.nl:uuid:43d7992a-7077-47ba-b38f-113f5011d07f). PhD thesis. Delft University of Technology, Netherlands, 2019. url: `https://www.base-search.net/Record/261b6c9463c1d4fe309e3c6104cd4d80fbc9d3cc8fbc66006f34130f481`

[56] Luis Eduardo de Souza Amorim and Eelco Visser. "Multi-purpose Syntax Definition with SDF3". In: *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*. Ed. by Frank S. de Boer and Antonio Cerone. Vol. 12310. Lecture Notes in Computer Science. Springer, 2020, pp. 1–23. isbn: 978-3-030-58768-0. doi: `10.1007/978-3-030-58768-0_1`. url: `https://doi.org/10.1007/978-3-030-58768-0_1`.

[57] *Spoofax: The Language Designer's Workbench*. https://www.spoofax.dev/. 2021. url: `https://www.spoofax.dev/`.

[58] *Stack overflow developer survey 2021*. url: `https://insights.stackoverflow.com/survey/2021`.

[59] Guy L. Steele. *Rabbit: A Compiler for Scheme*. Tech. rep. USA, 1978.

[60] Christopher Strachey and Christopher Wadsworth. "Continuations: A Mathematical Semantics for Handling Full Jumps". In: *Higher-Order and Symbolic Computation* 13 (Apr. 2000), pp. 135–. doi: `10.1023/A:1010026413531`.

[61] Wouter Swierstra. “Data types à la carte”. In: *Journal of Functional Programming* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758. URL: http://dx.doi.org/10.1017/S0956796808006758.

[62] *The Coq Proof Assistant*. URL: https://coq.inria.fr/.

[63] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[64] Vlad A. Vergu, Pierre Néron, and Eelco Visser. “DynSem: A DSL for Dynamic Semantics Specification”. In: *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*. Ed. by Maribel Fernández. Vol. 36. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 365–378. ISBN: 978-3-939897-85-9. DOI: 10.4230/LIPIcs.RTA.2015.365. URL: http://dx.doi.org/10.4230/LIPIcs.RTA.2015.365.

[65] Vlad A. Vergu, Andrew P. Tolmach, and Eelco Visser. “Scopes and Frames Improve Meta-Interpreter Specialization”. In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.4. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2019.4.

[66] Vlad A. Vergu and Eelco Visser. “Specializing a meta-interpreter: JIT compilation of Dynsem specifications on the Graal VM”. In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*. Ed. by Eli Tilevich and Hanspeter Mössenböck. ACM, 2018. ISBN: 978-1-4503-6424-9. DOI: 10.1145/3237009.3237018. URL: https://doi.org/10.1145/3237009.3237018.

[67] Eelco Visser. “Transformations for Abstractions”. In: *5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), 30 September - 1 October 2005, Budapest, Hungary*. IEEE Computer Society, Oct. 2005. ISBN: 0-7695-2292-0. DOI: 10.1109/SCAM.2005.26. URL: http://dx.doi.org/10.1109/SCAM.2005.26.

[68] Eelco Visser and Zine-El-Abidine Benaissa. “A core language for rewriting”. In: *Electronic Notes in Theoretical Computer Science* 15 (1998), pp. 422–441. DOI: 10.1016/S1571-0661(05)80027-1. URL: http://dx.doi.org/10.1016/S1571-0661(05)80027-1.

[69] Guido Wachsmuth, Gabriël Konat, and Eelco Visser. “Language Design with the Spoofax Language Workbench”. In: *IEEE Software* 31.5 (2014), pp. 35–43. DOI: 10.1109/MS.2014.100. URL: http://dx.doi.org/10.1109/MS.2014.100.

[70] Philip Wadler. “Monads for functional programming”. In: *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992*. Ed. by Manfred Broy. Vol. 118. NATO ASI Series. Springer, 1992, pp. 233–264. ISBN: 978-3-662-02880-3. DOI: 10.1007/978-3-662-02880-3_8. URL: https://doi.org/10.1007/978-3-662-02880-3_8.

[71] Jan Wielemaker et al. “SWI-Prolog”. In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 67–96. ISSN: 1471-0684.

Acronyms

API application programming interface

AST abstract syntax tree

ATerm annotated term format

DSL domain-specific language

CPS continuation-passing style

IR intermediate representation