Using haskell to Implement Syntactic Control of Interference

by

Jared Warren

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
June, 2008

Abstract

Interference makes reasoning about imperative programs difficult but it can be controlled syntactically by a language's type system, such as Syntactic Control of Interference (SCI). HASKELL is a purely-functional, statically-typed language with a rich type system including algebraic datatypes and type classes. It is popular as a defining language for definitional interpreters of domain-specific languages, making it an ideal candidate for implementation of definitional interpreters for SCI and Syntactic Control of Interference Revisited (SCIR), a variant that improves on SCI. Inference rules and denotational semantics functions are presented for PCF, Idealized ALGOL, SCI, and SCIR. An extension to HASKELL98 is used to define HASKELL functions for those languages' semantics and to define type constructions to statically check their syntax. The results in applied programming language theory demonstrate the suitability and techniques of HASKELL for definitional interpretation of languages with rich type systems.

Acknowledgments

I would like to thank my parents.

Table of Contents

Abstract **i**

Acknowledgments **ii**

Table of Contents **iii**

Chapter 1:

Introduction 1

1.1 Interference 3

1.2 Languages 8

1.3 Definitional Interpreters 10

1.4 Thesis Outline 15

Chapter 2:

Extensions to Haskell 98 18

2.1 Haskell Extensions 19

2.2 HList Types 20

2.3 Functions on Records 22

2.4 Summary 27

Chapter 3:

Implementation of PCF in Haskell 28

3.1 Core Syntax 29

3.2 Core Semantics 36

3.3 Constants 38

3.4 Programs 42

3.5 Example 42

3.6 Validation 49

3.7 Summary 52

Chapter 4:

Implementation of Idealized ALGOL in Haskell 53

4.1 Types 54

4.2 State 58

4.3 Constant Expressions 66

4.4 Commands 69

4.5 Conditional 72

4.6 Programs 76

4.7 Example 76

4.8 Validation 81

4.9 Summary 86

Chapter 5:

Implementation of SCI in Haskell 87

5.1 Interference 88

5.2 Syntax 88
5.3 Semantics 95
5.4 Parallelism 95
5.5 Examples 98
5.6 Validation 102
5.7 Summary 104

Chapter 6:

Syntactic Control of Interference Revisited 105

6.1 Original SCIR 106
6.2 SCIR Without Ambiguity 109
6.3 Summary 117

Chapter 7:

Implementation of SCIR$_K$ in Haskell 118

7.1 Passivity 119
7.2 Phrase Syntax 121
7.3 Syntax Rules 124
7.4 Example 129
7.5 Validation 135
7.6 Summary 138

Chapter 8:

Conclusion 139

8.1 Summary of Contributions 140
8.2 Future Research 141

Bibliography **143**

Appendix A:

Code Listings **147**

A.1 HList 147

A.2 PCF 150

A.3 IA 157

A.4 SCI 167

A.5 SCIRK 171

Chapter 1

Introduction

This dissertation demonstrates the implementation of a programming language with syntactic control of interference. The implementation is a definitional interpreter and the language is incrementally implemented out of simpler fragments and variants of that language. This dissertation presents the previously existing syntax and semantics of the implemented language as well as the original code used for the interpreter.

Imperative programming languages, sometimes considered FORTRAN-like languages, are those in which the execution, assignment to, or call of a command, variable, or function, respectively, may affect the outcome of a term. This is known as *interference*. For example, the command $x := x + 1$ interferes with the expression x. If it is possible for variables to be aliases or for functions to access non-local variables, then interference may be *covert* and such programs are more difficult to understand. For example, if y is an alias for x, then the assignment $x := x + 1$ unexpectedly interferes with y. The problems with interference are discussed in depth in section 1.1 on page 3.

Type systems that control interference syntactically can be created for ALGOL-like

languages. This dissertation is concerned with the type system known as Syntactic Control of Interference (SCI) presented in [18], redesigned as Syntactic Control of Interference Revisited (SCIR) in [12], and reformulated for ease of implementation as SCIR_K in [4]. Section 1.2 on page 8 introduces the relationship between these languages. Chapter 6 on page 105 details the relationship between them and discusses implementation.

The method chosen for implementation of SCIR_K is a definitional interpreter, where statements in the implemented ('defined') language are represented by statements in another language ('defining'). In an expressive defining language, a definitional interpreter is a straight-forward but powerful technique. The nature of definitional interpreters and the value of implementing SCIR_K is examined in section 1.3 on page 10.

Implementation of SCIR_K's complex type system in a definitional interpreter requires a defining language with a sophisticated type system of its own. HASKELL has such a type system; it is considered a good platform for definitional interpreters, and enjoys a general popularity in the programming language theory community due to its advanced type system. Section 1.3.3 on page 14 provides further justification for the choice of HASKELL. Extensions and libraries for HASKELL that aid in the implementation of definitional interpreters are discussed in chapter 2 on page 18.

The implementation of SCIR_K is done incrementally. The Programming language for Computable Functions (PCF), considered as the purely-functional fragment of SCIR_K, is the first stage of implementation in chapter 3 on page 28. Next, chapter 4 on page 53 presents Idealized ALGOL, an imperative programming language with *no*

control of interference that establishes the semantics and base type system for subsequent languages. A simplistic approach to adding syntactic control of interference to Idealized ALGOL is presented in chapter 5 on page 87. Finally, chapter 7 on page 118 presents the sophisticated type system of SCIR_K as another Idealized ALGOL variant.

1.1 Interference

1.1.1 Definition

A term P *interferes* with a term Q when exercising P has an effect on the outcome of Q. P could be a command, variable, or function, that is exercised by execution, assignment, or calling, respectively. The outcome of Q could be a final state, value, or return value if Q is a command, expression (including variable dereferencing), or function, respectively. More formally, a phrase P interferes with a phrase Q if exercising Q before exercising P and exercising Q after P yields different outcomes for Q.

Interference is necessary for imperative programming – if execution order did not matter then the state would be useless. And reasoning about simple imperative programs is not difficult. It is covert interference that confuses reasoning and is therefore undesirable. Interference becomes covert in the presence of aliased variable-identifiers. If x and y are aliases, then assertions about x must also be assertions about y, and phrases that appear to interfere with x also interfere with y.

For example, it is obvious that $x := 1$ interferes with x. But if y is an alias for x, then $x := 1$ interferes with y and $y := 0$ interferes with x. Aliasing occurs when parameters are passed to functions and the function uses identifiers from a scope

wider than the function body (in some imperative languages these must be global identifiers):

$$f = \lambda x.x := 0; y := 1$$

When applied to an identifier for any other variable, z, $f(z)$ behaves as expected (assigning 0 to z and 1 to y). But when applied to the variable it already perturbs, the function acts differently. It is not enough for the programmer to know *what* a function does to reason about interference, they must also know the order it does it in. For example, at first glance, f and g should be equivalent:

$$g = \lambda x.y := 1; x := 0$$

Interference also makes it impossible to give a meaningful semantics for parallel evaluation. If the order of evaluation is significant for P and Q, then what does it mean to say that P and Q should be evaluated simultaneously? In practice, there must be some order to their evaluation (perhaps in accessing the memory location that stores a variable), and that order must be known to reason about the evaluation. For example, what is the resulting value of x if P and Q are evaluated in arbitrary order:

$$P = x := 0$$

$$Q = x := 1$$

A more complex example is a function that traverses a linked list, applying a function to each node:

$$\mathit{traverse}(l, f) = \textbf{new } x \textbf{ in } x := \mathit{first}(l);\ \textbf{while not}(x = \textbf{nil})\ \textbf{do}\ f(x); x := \mathit{next}(x)$$

If the parameter f is a function that interferes with the node it is applied to, for example deleting it or inserting a new node, then the call $x := \mathit{next}(x)$ may not have the correct outcome (including divergence). A more efficient traversal would process the current node while fetching the next node in parallel:

$$\mathit{traverse}(l, f) = \textbf{new } x \textbf{ in } x := \mathit{first}(l);\ \textbf{while not}(x = \textbf{undef})\ \textbf{do}\ f(x) \parallel x := \mathit{next}(x)$$

If f interferes with $\mathit{next}(x)$, application of this function is not even meaningful.

1.1.2 Control

Syntactic control of interference usually starts from the assumption that all channels of interference are syntactically represented by identifiers, so control must be achieved by restrictions on the use of identifiers. This dissertation focuses on the restrictions employed in the languages SCI and SCIR. Alternative approaches used in other languages are briefly discussed below.

Occam

OCCAM is a programming language based on a formal process calculus [10]. Interference is controlled between processes by only using designated channels for message

passing. It is also controlled within procedures (functions with side-effects) by mandating that a variable can be referred to by exactly one identifier. For example, the following ALGOL-like code would be invalid with a OCCAM-like type system:

$$(\lambda x.\pi_0(x) := \mathbf{deref}\,(\pi_1(x)))\langle y, y\rangle$$

This is usually verified at compile-time, although the OCCAM standard allows the possibility of runtime checks for ambiguous cases. OCCAM uses call-by-reference evaluation and has no higher-order functions. This kind of interference control is not obviously applicable to ALGOL-like languages.

Euclid

EUCLID is designed for writing verifiable programs: interference is controlled to aid verification [9]. Given a EUCLID procedure (function with side-effects) body, the only identifiers in-scope are those passed as arguments or declared as globals ('pervasive'), which are constant in the function body. For example, in the following phrase of an ALGOL-like language with a EUCLID-like type system, x can only be used passively in P, and y is fully in-scope:

$$\mathbf{global}\ x\ \mathbf{in}\ \lambda y.P$$

Higher-order functions with arguments must have all identifiers used within the arguments in-scope. To facilitate this, EUCLID includes a mechanism to explicitly import identifiers from the containing context. For example, functions can only be polymorphic on types with specific free identifiers:

$$(\lambda x.x)(0)$$

$$(\lambda x.\textbf{import } y \textbf{ in } x)(y := 0)$$

$$(\lambda x.\textbf{import } y \textbf{ in import } z \textbf{ in } x)(y := \textbf{deref } z)$$

EUCLID's constraints make interference explicit without forbidding any particular first-order functions (unlike SCIR). However, high-order functions are constrained and it is not obvious how to apply a EUCLID-like type system to a language with call-by-name evaluation [18].

FX

FX is designed to be automatically parallelizable [3]. This is achieved by what is called an "effects system". Every phrase is annotated with a subset of the effects it may perform: initializing variables, dereferencing variables, assigning variables. This is similar to SCI's concept of passive functions, but applied at the semantic level to variables.

FX can be extended with a "region system": each effect specifies a "region" of the store it applies to. Effects can be combined to make sophisticated annotations specifying that some regions are dereferenced, some are assigned, etc. This is a semantic version of SCIR's active and passive identifiers.

Early versions of FX required the programmer to explicitly specify effects. Inference algorithms have now been developed that calculate the effects statically [20]. The designers of FX are primarily interested in parallelization and optimization; they have not applied effects to the control of interference to our knowledge. Similarities

between FX and SCI should be explored in the future but are outside the scope of this dissertation.

SCI

A binary relation, #, is specified in [18] to denote non-interference. $P\#Q$ means 'P does not interfere with Q', and since the relation is symmetric, it also means 'Q does not interfere with P'. SCI focuses on identifiers as the channels of interference.

Reynolds observed that identifiers can appear in active or passive phrases [18]. In active phrases, distinct identifiers are guaranteed not to interfere with each other. Enforcement of this is discussed in section 5.2 on page 88. Passive phrases are those where identifiers are used in such a way as to guarantee non-interference. In passive phrases identifiers can be aliases – just how to define passive phrases is discussed in section 6.1 on page 106.

1.2 Languages

This dissertation first implements Idealized ALGOL in HASKELL, then looks at the implementation of interference-controlling syntaxes.

1.2.1 PCF

The Programming language for Computable Functions (PCF) is a purely-functional language. Originally introduced in [16], it is essentially the typed λ-calculus of [2] with arithmetic and recursion. PCF is sufficiently complex to be considered on its own, as the interplay of environment and higher-order functions is non-trivial. It

is considered before Idealized ALGOL because the implementation techniques for application, abstraction, environment, constants, and recursion are used in subsequent chapters.

1.2.2 Algol

Idealized ALGOL is a simple language that combines functional and imperative programming [11]. The functional fragment of Idealized ALGOL forms PCF. The imperative fragment of Idealized ALGOL, if considered in the absence of function abstraction, forms a purely-imperative language – this is sometimes called the language of WHILE-programs. For present purposes it is easier to consider Idealized ALGOL as PCF with imperative extensions rather than a merging of PCF with WHILE-programs.

1.2.3 SCI

Alternate type systems with various properties can be formulated for Idealized ALGOL. SCI is a type system that permits a subset of the valid phrases in Idealized Algol without covert interference (see section 1.1), and extends Idealized ALGOL with parallel programming extensions. SCIR [12] is a similar, alternate interference-controlling type system that improves on SCI, while SCIR_K [4] is a reformulation of SCIR that allows for type reconstruction and is easier to implement. Soundness and completeness of SCIR_K relative to SCIR were shown in [4].

1.2.4 Haskell

HASKELL is a statically-typed, purely-functional programming language. Static typing is inferred by a typed checking engine aided by typed declarations including typed

constraints. As of this writing, the latest standard is HASKELL98 [6].

With some widely-available extensions to HASKELL98, the type constraints can be considered as a crude logical programming language, with first-order functions and user-defined data structures. This language is capable of reifying rich typed-system concepts.

A full definition of HASKELL is outside the scope of this dissertation but descriptions of advanced HASKELL features and extensions are interspersed with implementation details in subsequent chapters.

1.3 Definitional Interpreters

Although the language as an abstraction has been much studied, to our knowledge Idealized ALGOL lacks a widespread execution environment that allows actual running programs, never mind software development. There are at least two reasons no compiler or interpreter may exist for Idealized ALGOL: first, it is not obvious that Idealized ALGOL has value as a reified programming language; and, second, it is difficult for a new execution environment to gain acceptance and thereby be worth producing.

This dissertation uses the implementation technique known as a *definitional interpreter* after [17]. In the HASKELL community these are known as *embeddings*, although to date that has normally been used for domain-specific programming languages [5].

A definitional interpreter is a simple, proof-of-concept implementation. In addition to being a good prototype implementation, it has benefits beyond a stand-alone compiler or interpreter. The choice of defining language (see below) also has implications for acceptance.

1.3.1 Definition

A definitional interpreter is the implementation of a language by defining syntax and semantics in terms of phrases in another language. The language being implemented is referred to as the defined language, the language it is implemented in is the defining language. The semantics and syntax of the defined language are implemented in the phrases and types, respectively, of the defining language. Programs in the defined language make up a subset of valid phrases in the defining language. The meaning of a phrase in the defined language is a value in the defining language. This is sufficient to implement dynamically-typed languages, as explored in [17]; in addition, statically-typed language phrases that represent defined language poorly-formed phrases should cause syntax errors.

Keywords and operators are used in a language as type and data constructors. As type constructors they map from one type to another, as data constructors from one data-type to another. The keywords and operators of a defined language must have corresponding entities in the defining language that map types and data. A type constructor has an associated syntax rule that specifies its rules for use. The type system in the defining language must be able to implement constraints representing those rules on arguments and context. In HASKELL, type constructors and data constructors are defined by the same statement. HASKELL's type system is not rich enough to have all constraints on data constructors, so instead of using them directly they are wrapped in HASKELL-functions that have type constraints.

Phrases in the defined language are 'lowered' to the defining language using a set of semantic evaluation functions (or one polymorphic function). The meaning of a phrase is either the value of a primitive type or user-defined data type in the defining

language. For example, a HASKELL semantic evaluation function could map constants to HASKELL-integers and functions in a defined language to HASKELL-functions.

Given type constructors implemented as HASKELL-functions with type constraints, if the functions are combined in such a way as to violate the syntax rules, the type constraints cannot be satisfied and the type checker reports an error. In this way, programs in a statically-typed defined language can be checked statically as defining language programs.

1.3.2 Value

Idealized ALGOL is considered a strong foundation for a variety of theoretical work. An implementation of Idealized ALGOL would act as a programming language theory laboratory, allowing researchers and students to investigate the practical impact of ideas and explore implementation details. An executable language facilitates the creation of non-trivial programs, placing theoretical concepts in a realistic context and scale, making emergent properties evident. For example, an implementation of Idealized ALGOL with syntactic control of interference [18], is one step in determining how SCI benefits large programs and affects programmer behaviour.[1]

Similar languages such as ALGOL68 lack some properties of Idealized ALGOL. For example, the considerable body of work in verification and type-system extensions for Idealized ALGOL (again, such as SCI) could be used without adaptation, making Idealized ALGOL a unique tool for high-reliability software development. Since a high-performance compiler is still a long way from development, early implementations of Idealized ALGOL are probably more suited as rapid prototyping languages. Because of

[1] e.g.: [18] briefly discusses permissible interference in objects, but the practicality of this refactoring has not been explored to our knowledge.

Idealized ALGOL's foundational place in history, and its use as a definitional language for various extensions, translation of a prototype written in Idealized ALGOL to more-efficient languages for optimization may be easier than in other rapid prototyping languages.

Just as Idealized ALGOL was valuable as a standard notation for publishing algorithms, a implemented version would allow it to be used by students and other less sophisticated computer science audiences. Students strongly resist learning languages that they cannot experiment with, and although Idealized ALGOL is far from the language du jour, having an implementation increases its educational potential. Its minimalist syntax and well-documented semantics would also make an implementation well suited for educational extensions.

In addition to the value of an implementation as a tool for programming, the process of creating an implementation produces value to future researchers as a byproduct. Every language that is implemented (especially those with significant but chronically underimplemented features like call-by-name) represents a significant use-case for the tools being used in its development. The implementation, if there were reason to make it, acts as an argument for improvements in the tools to work with similar languages and acts as an example for future implementers. Therefore, to increase the value of an implementation, it should be done using reusable tools in a manner that is accessible and easily understood. The framework should be as generic and high-level as possible.

1.3.3 Increasing Acceptance

HASKELL is, as of this writing, the one of the most popular platforms for writing definitional interpreters of domain-specific languages [5]. To our knowledge, a definitional interpreter for a general-purpose programming language has never been implemented. Although not used widely for large-scale software development, HASKELL is the most mainstream language that has a type system rich enough to implement type extensions to Idealized ALGOL, such as SCI (as will be discussed in detail later in this dissertation).

HASKELL may be the most mainstream language capable of implementing Idealized ALGOL as a definitional interpreter. HASKELL is popular in the programming language theory community, so the process of the implementation will be more valuable if done in HASKELL. It is desirable if the implementation platform can provide existing libraries, development environments, and runtime environments; a definitional interpreter in an established language provides all of these, if indirectly.

In addition to the value of an implementation itself, there is extra value to be gained by implementing Idealized ALGOL type extensions in HASKELL. This is because one of the roles HASKELL holds is as a type system laboratory: features are first implemented in HASKELL before finding their way into more industrial-strength languages – either directly into another member of the ML family such as OCAML [15], or inspirationally in the languages HASKELL programmers use in their day jobs. It is relatively easy for new ideas to gain acceptance in the HASKELL community because its users are already convinced of the value of a rich, static type system and there is no significant body of legacy code promoting stasis [13]. From the perspective of the HASKELL community, a single rich definitional interpreter paves the way for others,

first by suggesting fruitful language extensions and second by providing an example for other programmers.

1.4 Thesis Outline

This dissertation presents an implementation of SCIR$_K$ as built on the implementation of simpler languages: SCI, Idealized ALGOL, and PCF in turn. The syntax and semantics of all these languages was previously defined, the HASKELL code implementing them is original to this dissertation.

Implementation code is presented with an informal mapping (denoted by $\Longrightarrow$) from the relevant independent syntactic or semantic entity to its implementation. Language syntax is given as inference rules; semantics are given as a denotational semantic evaluation function. The style of language description is based on [22] and [12].

$$\text{Defined language specification} \Longrightarrow \text{Defining language code}$$

For brevity, trivial language features, like right-projection (π_1) and do-blocks, have been omitted. For ease of reading, HASKELL's lexical rules have been ignored, for example, the use of Greek or upper-case characters as variables (the HASKELL standard only allows lower-case Roman characters), and mathematical operators have been 'pretty-printed'. Verbatim listings of the implementing HASKELL code are in appendix A.

The implementation code is validated using unit tests that check the type constraints and instances of the evaluation function. Integration testing was not deemed

necessary due to the compositional nature of the languages' syntax and semantics, as well as the confidence of 'white-box' testing. Complex examples in each chapter demonstrate integration and more 'strenuous' testing for some language features.

Each language is covered by a separate chapter. The following outlines the implementation details presented in each chapter:

1.4.1 PCF

1. HASKELL-type and data structures representing PCF-type constructors and constants
2. HASKELL-type and data structures representing PCF-phrases
3. HASKELL-type constraint representing PCF-phrase typing
4. HASKELL-functions representing PCF-type construction rules and constants
5. HASKELL-type and data structures representing PCF-phrase types
6. HASKELL-functions that evaluate programs and checks program validity
7. Ad-hoc polymorphic HASKELL-function representing semantic evaluation

1.4.2 Idealized ALGOL

1. HASKELL-type and data structures representing Idealized ALGOL-phrase types
2. HASKELL-type constraint representing Idealized ALGOL-type meaning
3. HASKELL-type and data structures, functions, and type-constraint instances for Idealized ALGOL-constants

4. Ad-hoc polymorphic HASKELL-function for expanding state shape

1.4.3 SCI

1. HASKELL-type constraint representing a non-interference constraint on SCI-syntax rules

2. HASKELL-type and data structures representing SCI-phrase types

3. HASKELL-type and data structures, functions, and type-constraint instances for SCI-constants

4. Redefinition of HASKELL-function representing the SCI-application type construction rule

5. Redefinition of HASKELL-type constraint instance representing recursion SCI-type

1.4.4 SCIR$_K$

1. HASKELL-type constraint representing SCIR$_K$-passivity predicate

2. HASKELL-type and data structures representing SCIR$_K$-phrases

3. HASKELL-type constraints representing functions on SCIR$_K$-constraints

4. HASKELL-functions representing SCIR$_K$-type constructors and constants

Chapter 2

Extensions to Haskell 98

HASKELL's type classes exist in the language to facilitate ad-hoc polymorphic functions (functions with different behaviour depending on the type of their arguments). Over HASKELL's lifespan programming idioms have been identified that type classes are not flexible enough to support; for example, polymorphic functions on collections. The range of constraints on types that type classes are able to express is limited, preventing programmers from getting the full benefits of HASKELL's static typing [13]. Implementation of a general-purpose programming language like Idealized ALGOL requires flexibility in the type system of the defining language and nuanced constraints like those of SCIR$_K$ require expressiveness.

As of this writing, HASKELL98 is the latest standard of the language and its type system. There are a number of widely-used extensions to HASKELL98 that increase the flexibility and expressiveness of type classes. Some of these extensions make implementation of definitional interpreters easier and more elegant. The extensions are discussed in detail in section 2.1 on the following page.

In addition to the implementation code itself, extensions to HASKELL98 are needed

to support the HList library [8]. HList provides an implementation of strongly-typed, heterogeneous records, and polymorphic functions to manipulate them. These are used to implement sets of identifiers and mappings from identifiers to types or values. Section 2.2 on the next page presents the parts of HList used in this dissertation. Code based on HList's constructs used in all the implementations is listed in section 2.3 on page 22.

2.1 Haskell Extensions

Ad-hoc-polymorphic functions are implemented in HASKELL by type classes. Type classes parameterize functions on argument type. Sometimes it is necessary to parameterize on multiple types; this is permitted by the extension for multi-parameter type classes. The co-domain of a sematical evaluation function is usually determined by the phrase type and context: the co-domain is a dependent type. Dependent types are "simulated" by the extension for functional dependencies [14]. Lexically, functional dependencies appear as an annotation specifying a functional relation on type class parameters. The annotation is separated from the parameters by | and is written as a list of antecedent type parameters, an arrow ($\rightarrow$), and a list of consequent (dependent) type parameters.

HASKELL98 requires all instance type constraints to be "smaller" (less complex) than the type of the instance itself. This restrictive requirement guarantees that type inference terminates. The instances for semantic evaluation are either recursive or defined in terms of another type-parameterized function, so they violate this requirement. The requirement is relaxed, permitting these instances, with the HASKELL98 extension for "undecidable" instances [21].

2.2 HList Types

Record labels are constructed using *Label*, each one is parameterized with a type-level Peano natural (*HZero*, *HSucc*[1]) to give the label a distinct type. Labels are paired in F structures with arbitrary HASKELL types. Cons-lists (*HCons* and *HNil*) of these pairs are constructed into records using *Record*.

The HList library defines a number of types. The types *HZero*, *HSucc*, and *Label* are relevant only at the type-level – therefore the right-hand side can be omitted, using an extension to HASKELL98. These are type constructors only: without data constructors, their only value can be *undefined*. The *data* constructor F is not parameterized by a label, as this is only significant at the syntactic level.

```
data HZero
data HSucc n
data Label n
data F l v = F v
data HCons h t = HCons h t
data Record l = Record l
```

These constructors are combined to implement contexts and environments (and, later, states), for example in listing 1 on the next page.

[1]Rather than using a module namespace, most ambiguous HList entity identifiers start with the letter 'h'.

$$
\begin{array}{rcl}
 & & u :: \mathit{Record}\ (\\
\Pi = \iota_0 : \mathbf{int}, \iota_1 : \mathbf{bool} & \Longrightarrow & \quad \mathit{HCons}\ (F\ (\mathit{Label}\ \mathit{HZero})\ \mathit{Integer})\ (\\
 & & \quad \mathit{HCons}\ (F\ (\mathit{Label}\ (\mathit{HSucc}\ \mathit{HZero}))\ \mathit{Bool})\ \mathit{HNil})) \\
u = (\iota_0 \mapsto 7 \mid \iota_1 \mapsto \mathbf{true}) & \Longrightarrow & u = \mathit{Record}\ (\mathit{HCons}\ (F\ 7)\ (\mathit{HCons}\ (F\ \mathit{True})\ \mathit{HNil}))
\end{array}
$$

Listing 1: Record example

2.3 Functions on Records

The HList library uses multi-parameter type classes with functional dependencies to implement three primitive functions on records: lookup, union, and projection. Union and projection are in listing 2 on the following page. HList implements the projection function *h2projectByLabels*, type-parameterized by class *H2ProjectByLabels*.

$$\cup : \Pi_0 \to \Pi_1 \to \Pi_2 \quad \Longrightarrow \quad \textbf{class } \mathit{Union}\ \Pi_0\ \Pi_1\ \Pi_2 \mid \Pi_0\ \Pi_1 \to \Pi_2$$

$$\Pi_0, \iota : \theta \quad \Longrightarrow \quad \begin{array}{l} \textbf{class } \mathit{HLeftUnion}\ (\mathit{Record}\ (\mathit{HCons}\ (F\ \iota\ \theta)\ \mathit{HNil}))\ \Pi_0\ \Pi_1 \\ \quad \Rightarrow \mathit{Perturb}\ \iota\ \theta\ \Pi_0\ \Pi_1 \mid \iota\ \theta\ \Pi_0 \to \Pi_1\ \textbf{where} \end{array}$$

$$(u_0 \mid \iota \mapsto v) \quad \Longrightarrow \quad \begin{array}{l} \mathit{perturb} :: \iota \to \theta \to u_0 \to u_1 \\ \mathit{perturb}\ \iota\ v\ u_0 = \mathit{hLeftUnion}\ (\mathit{Record}\ (\mathit{HCons}\ (F\ v :: F\ \iota\ \theta)\ \mathit{HNil}))\ u_1 \end{array}$$

$$\begin{array}{l} A = \iota_n, \ldots, \iota_m \\ \Pi_1 = \iota_n : \theta_n, \ldots, \iota_m : \theta_m \\ \Pi_0 - \Pi_1 = \Pi_2 \end{array} \quad \Longrightarrow \quad \textbf{class } \mathit{H2ProjectByLabels}\ A\ \Pi_0\ \Pi_1\ \Pi_2 \mid A\ \Pi_0 \to \Pi_1\ \Pi_2\ \textbf{where}$$

$$\begin{array}{l} u_1 = (\iota_n \mapsto v_n \mid \cdots \mid \iota_m \mapsto v_m) \\ u_0 - u_1 = u_2 \end{array} \quad \Longrightarrow \quad \mathit{h2projectByLabels} :: A \to u_0 \to (u_1, u_2)$$

Listing 2: Functions on Records

For example, *perturb* used to change the second entry in the record u from the example in listing 1 on page 21:

$$perturb\ (Label\ (HSucc\ HZero))\ False\ u \Rightarrow Record\{HZero = 7, HSuccHZero = False\}$$

Implementation of a union function must deal with intersecting records. The HList library provides the function *hLeftUnion* (type-parameterized by class *HLeftUnion*) that gives precedence to both types and values in its first (left) argument. In listing 3 on the following page, *Union* is a constraint that requires identifiers that appear in both records to map to the same type and gives precedence to values from the first argument – it is also implemented on sets. (Note that the *Union* class does not type-parameterize any actual function, it makes such a function possible.) *Union* will be used in situations where the unioned records must be compatible or disjoint, *hLeftUnion* will be used where one record contains updates or perturbations of the other.

```
        -- Union on Records:
instance Union r (Record HNil) r
instance (
  RecordLabels r ls,
  HMember l ls b,
  UnionRecords b r (F l v) r''',
  Union (Record r''') (Record r') r''
  ) ⇒ Union (Record r) (Record (HCons (F l v) r')) r''
  -- Record logic:
class UnionRecords b r f r' | b r f → r'
instance HasField l r v ⇒ UnionRecords HTrue r (F l v) r
instance UnionRecords HFalse r f (HCons f r)
  -- Union on Sets:
instance Union s HNil s
instance (HMember h s b, UnionSets b s h s'', Union s'' t s')
   ⇒ Union s (HCons h t) s'
  -- Set logic:
class UnionSets b s e s' | b s e → s'
instance UnionSets HTrue s h s
instance UnionSets HFalse s h (HCons h s)
```

Listing 3: Definition of Union

Because of the way *hLeftUnion* deals with conflicts, a perturb function can be

defined as a union function where one argument is a record containing only the perturbing mapping. The type constaint on the *Perturb* class specifies that *Perturb* is a subclass of *HLeftUnion*: any types that have an instance for *Perturb* must also have an instance for *HLeftUnion*. The *Perturb* instance is fully polymorphic with the *HLeftUnion* constraint: this specifies that all types that have an instance for *HLeftUnion* must also have an instance for *Perturb*.[2]

For the purposes of a lookup function, a record can be treated as either a partial or total function. As a partial function, the type constraint mandates that if a record contains a mapping for the label, then a value must be of a specific type; and if the record contains no such mapping, then the constraint is satisfied. As a total function, the record must contain a mapping for the label and it must be mapped to a specific type. The partial case is implemented by the HList function *hLookupByLabel*, which is type-parameterized by class *HasField*, and the total case is implemented by the type constraint *IfHasField*.

$$\textbf{class } \mathit{HasField}\ \iota\ \Pi\ \theta \mid \iota\ \Pi \rightarrow \theta\ \textbf{where}$$
$$\quad \mathit{hLookupByLabel} :: \iota \rightarrow u \rightarrow \theta$$
$$\textbf{class } \mathit{IfHasField}\ l\ r\ v \mid l\ r \rightarrow v$$
$$\textbf{instance } (\mathit{HLeftUnion}\ r\ (\mathit{Record}\ (\mathit{HCons}\ (F\ l\ v)\ \mathit{HNil}))\ r', \mathit{HasField}\ l\ r'\ v)$$
$$\quad \Rightarrow \mathit{IfHasField}\ l\ r\ v$$

[2]Note that the *perturb* function is written using the HASKELL98 extension for scoped type variables to specify the label in the mapping $(F\ v :: F\ \iota\ \theta)$. This is done entirely for clarity and could be avoided by the use of a constructor function, such as *newF* defined in the HList library.

2.4 Summary

Multi-parameter type classes with functional dependencies and undecidable instances are extensions to HASKELL98's type class mechanism. They are used in the implementations presented in subsequent chapters and in the HList library those implementations are built on. HList provides strongly-typed, heterogeneous records, which are used to implement sets of identifiers and mappings from identifiers to types or values. Constructs for record union, perturbing, projection, and lookup are defined for use throughout the subsequent implementations.

Chapter 3

Implementation of PCF in Haskell

The Programming language for Computable Functions (PCF) is a simple purely-functional language. It can be considered a syntactic sugaring of the λ-calculus with the addition of arithmetic and recursion. PCF is based on the applicative language in [22] although it was first described in [16].

PCF is presented first because it is equivalent to the expressions fragment of Idealized ALGOL and it establishes techniques used in the implementation of subsequent languages. Although the command fragment of Idealized ALGOL is where interference takes place, the functions included in PCF are what make it covert. The syntax and semantics of an idealized purely-functional language are more complex than that of an idealized purely-imperative language, therefore meriting separate consideration.

The implementation of a definitional interpreter is demonstrated by presenting PCF's syntax in section 3.1 on the next page and semantics in section 3.2 on page 36 alongside HASKELL statements that represent those constructs. Section 3.3 on page 38 adds a minimal set of constant functions to permit programming. Examples of the

HASKELL code's use are in sections 3.1.2 on page 35 and 3.5 on page 42. The implementation is validated by unit tests presented in section 3.6 on page 49.

3.1 Core Syntax

In a definitional interpreter, programs are written using HASKELL-functions to construct a HASKELL-data structure representing a PCF-phrase. Programs are evaluated using a HASKELL-function from these data structures to HASKELL-values.

PCF is implemented with strong typing by virtue of HASKELL's strong typing. Therefore, a distinct HASKELL-type is defined for each PCF-type constructor syntax rule. HASKELL-types are defined using the **data** keyword, which specifies a HASKELL-type constructor (on the left-hand side) and HASKELL-data constructor (on the right). HASKELL-constructors represent PCF-phrase constructors and are combined to create a HASKELL-data structure that represents a PCF-syntax tree.

HASKELL-types can be aliased using **type** statements. For example, the *Identifier* type statement in raw syntax 1 assumes that the HList library provides a definition like the following.[1]

```
data Label n = Label n
```

Raw syntax 1 shows the syntax for PCF and their corresponding HASKELL-types. Note that each identifier has a distinct type parameterized by HASKELL-type-level natural numbers as implemented in HList. For function introduction (*Lambda*), the

[1]The actual declaration uses **newtype** rather than **data** – these keywords are indistinguishable for most purposes.

type of the abstracted identifier must be declared (θ), but only appears on the left-hand side because it is only relevant at the type level.

$$
\begin{array}{rcl}
P\,Q & \Longrightarrow & \textbf{data}\ \mathit{Appl}\ P\ Q = \mathit{Appl}\ P\ Q \\
\lambda\iota : \theta.P & \Longrightarrow & \textbf{data}\ \mathit{Lambda}\ \iota\ \theta\ P = \mathit{Lambda}\ \iota\ P \\
\langle P, Q\rangle & \Longrightarrow & \textbf{data}\ \mathit{Pair}\ P\ Q = \mathit{Pair}\ P\ Q \\
\iota_n & \Longrightarrow & \textbf{type}\ \mathit{Identifier}\ n = \mathit{Label}\ n \\
\pi_0 P & \Longrightarrow & \textbf{data}\ \pi_0\ P = \pi_0\ P
\end{array}
$$

Raw Syntax 1: Raw Syntax

HASKELL-data constructors do not implement PCF's static type system: the constructor functions can be applied to values of any HASKELL-type. So rather than creating PCF-phrases using the constructors directly, HASKELL-functions are defined which manipulate the HASKELL-data structures. These functions implement PCF's inference rules, managing the typing context and phrase type.

Phrase constructor functions are listed in syntax rules 1 on page 33. The HASKELL-type they operate on is the *Phrase*, which encapsulates a typing context and a PCF-phrase (constructed from raw syntax 1). PCF's explicit typing permits the phrase type (θ) to be inferred from the phrase and context; the rules for this inference are implemented by the type class *HasType*. Note that the definition for *lambda* includes the HASKELL-type-variable z, which represents the unused part of the projection, and the parameter t, which is used to provide a type hint. The definition for *ident* is polymorphic on θ, which will be inferred from context.

$$
\Pi \vdash P : \theta \quad \Longrightarrow \quad
\begin{array}{l}
\textbf{data}\ \mathit{Phrase}\ \Pi\ P = \mathit{Phrase}\ P \\
\\
\textbf{class}\ \mathit{HasType}\ \Pi\ P\ \theta \mid \Pi\ P \rightarrow \theta
\end{array}
$$

Instances of *HasType* are defined in type definitions 1 on the following page.

Typing context is implemented as an HList *Record*. Weakening and contraction structural rules are implicitly implemented by the fact that contexts are combined by *Union* (defined in section 2.3), which requires compatibility but not necessarily correspondence (weakening) or disjointness (contraction). Identifiers are typed using *IfHasField*; this implicitly implements the non-trivial case of weakening. Note that *lambda* uses *H2ProjectByLabels* to remove the identifier from the context.

type *Context s* = *Record s*

Each PCF-type has a corresponding, distinct HASKELL-type. Since these entities are used at the syntactic level, the HASKELL-data constructors (right-hand side) are used only for function introduction when the programmer must declare the type of the abstracted identifier. The HASKELL-type and data constructors are defined in type definitions 2. An extension to HASKELL98 is used so that constructor identifiers beginning with a colon are operators.

$$\frac{\Pi \vdash P : \theta_0 \to \theta_1 \quad \Pi \vdash Q : \theta_0}{\Pi \vdash P\,Q : \theta_1} \Longrightarrow \textbf{instance}\ (\mathit{HasType}\ \Pi\ P\ (\theta_0 :\to: \theta_1), \mathit{HasType}\ \Pi\ Q\ \theta_0) \Rightarrow \mathit{HasType}\ \Pi\ (\mathit{Appl}\ P\ Q)\ \theta_1$$

$$\frac{\Pi_0, \iota_n : \theta_0 \vdash P : \theta_1}{\Pi_0 \vdash \lambda\iota_n : \theta_0 . P : \theta_0 \to \theta_1} \Longrightarrow \textbf{instance}\ (\mathit{Perturb}\ \iota\ \theta_0\ \Pi_0\ \Pi_1, \mathit{HasType}\ (\mathit{Context}\ \Pi_1)\ P\ \theta_1) \Rightarrow \mathit{HasType}\ (\mathit{Context}\ \Pi_0)\ (\mathit{Lambda}\ \iota\ \theta_0\ P)\ (\theta_0 :\to: \theta_1)$$

$$\frac{\Pi \vdash P : \theta_0 \quad \Pi \vdash Q : \theta_1}{\Pi \vdash \langle P, Q \rangle : \theta_0 \times \theta_1} \Longrightarrow \textbf{instance}\ (\mathit{HasType}\ \Pi\ P\ \theta_0, \mathit{HasType}\ \Pi\ Q\ \theta_1) \Rightarrow \mathit{HasType}\ \Pi\ (\mathit{Pair}\ P\ Q)\ (\theta_0 :\times: \theta_1)$$

$$\frac{}{\iota_n : \theta \vdash \iota_n : \theta} \Longrightarrow \textbf{instance}\ \mathit{IfHasField}\ (\mathit{Identifier}\ n)\ \Pi\ \theta \Rightarrow \mathit{HasType}\ \Pi\ (\mathit{Identifier}\ n)\ \theta$$

$$\frac{\Pi \vdash P : \theta_0 \times \theta_1}{\Pi \vdash \pi_0 P : \theta_0} \Longrightarrow \textbf{instance}\ \mathit{HasType}\ \Pi\ P\ (\theta_0 :\times: \theta_1) \Rightarrow \mathit{HasType}\ \Pi\ (\pi_0\ P)\ \theta_0$$

Type Definitions 1: Phrase Typing

$$\frac{\Pi_0 \vdash P : \theta_0 \to \theta_1 \qquad \Pi_1 \vdash Q : \theta_0}{\Pi_0, \Pi_1 \vdash P\,Q : \theta_1} \Longrightarrow$$

$$\begin{aligned}
&\mathit{appl} :: (\mathit{Union}\ \Pi_0\ \Pi_1\ \Pi_2, \mathit{HasType}\ \Pi_0\ P\ (\theta_0 :\to: \theta_1), \mathit{HasType}\ \Pi_1\ Q\ \theta_0)\\
&\quad \Rightarrow \mathit{Phrase}\ \Pi_0\ P \to \mathit{Phrase}\ \Pi_1\ Q \to \mathit{Phrase}\ \Pi_2\ (\mathit{Appl}\ P\ Q)\\
&\mathit{appl}\ (\mathit{Phrase}\ P)\ (\mathit{Phrase}\ Q) = \mathit{Phrase}\ (\mathit{Appl}\ P\ Q)
\end{aligned}$$

$$\frac{\Pi_0, \iota_n : \theta_0 \vdash P : \theta_1}{\Pi_1 \vdash \lambda\iota_n : \theta_0.P : \theta_0 \to \theta_1} \Longrightarrow$$

$$\begin{aligned}
&\mathit{lambda} :: (\\
&\qquad \mathit{H2ProjectByLabels}\ (\mathit{HCons}\ (\mathit{Identifier}\ n)\ \mathit{HNil})\ \Pi_0\ z\ \Pi_1,\\
&\qquad \mathit{HasType}\ (\mathit{Context}\ \Pi_0)\ (\mathit{Identifier}\ n)\ \theta_0\\
&\quad) \Rightarrow \mathit{Identifier}\ n\\
&\qquad \to \theta_0\\
&\qquad \to \mathit{Phrase}\ (\mathit{Context}\ \Pi_0)\ P\\
&\qquad \to \mathit{Phrase}\ (\mathit{Context}\ \Pi_1)\ (\mathit{Lambda}\ (\mathit{Identifier}\ n)\ \theta_0\ P)\\
&\mathit{lambda}\ \iota\ (t :: \theta_0)\ (\mathit{Phrase}\ P) = \mathit{Phrase}\ (\mathit{Lambda}\ \iota\ P)
\end{aligned}$$

$$\frac{\Pi_0 \vdash P : \theta_0 \qquad \Pi_1 \vdash Q : \theta_1}{\Pi_0, \Pi_1 \vdash \langle P, Q\rangle : \theta_0 \times \theta_1} \Longrightarrow$$

$$\begin{aligned}
&(\times) :: \mathit{Union}\ \Pi_0\ \Pi_1\ \Pi_2\\
&\quad \Rightarrow \mathit{Phrase}\ \Pi_0\ P \to \mathit{Phrase}\ \Pi_1\ Q \to \mathit{Phrase}\ \Pi_2\ (\mathit{Pair}\ P\ Q)\\
&(\mathit{Phrase}\ P) \times (\mathit{Phrase}\ Q) = \mathit{Phrase}\ (\mathit{Pair}\ P\ Q)
\end{aligned}$$

$$\frac{}{\iota_n : \theta \vdash \iota_n : \theta} \Longrightarrow$$

$$\begin{aligned}
&\mathit{ident} :: \mathit{Identifier}\ n\\
&\quad \to \mathit{Phrase}\ (\mathit{Context}\ (\mathit{HCons}\ (F\ (\mathit{Identifier}\ n)\ \theta)\ \mathit{HNil}))\ (\mathit{Identifier}\ n)\\
&\mathit{ident}\ \iota = \mathit{Phrase}\ \iota
\end{aligned}$$

$$\frac{\Pi \vdash P : \theta_0 \times \theta_1}{\Pi \vdash \pi_0 P : \theta_0} \Longrightarrow$$

$$\begin{aligned}
&\pi_0 :: \mathit{HasType}\ \Pi\ P\ (\theta_0 :\times: \theta_1) \Rightarrow \mathit{Phrase}\ \Pi\ P \to \mathit{Phrase}\ \Pi\ (\pi_0\ P)\\
&\pi_0\ (\mathit{Phrase}\ P) = \mathit{Phrase}\ (\pi_0\ P)
\end{aligned}$$

Syntax Rules 1: Phrase Constructors

$$
\begin{array}{rcl}
\theta ::= & & \\
\textbf{bool} & \Longrightarrow & \textbf{data}\ \mathit{Bool} = \mathit{Bool} \\
\textbf{int} & \Longrightarrow & \textbf{data}\ \mathit{Int} = \mathit{Int} \\
\theta_0 \to \theta_1 & \Longrightarrow & \textbf{data}\ \theta_0 :\!\to\!: \theta_1 = \theta_0 :\!\to\!: \theta_1 \\
\theta_0 \times \theta_1 & \Longrightarrow & \textbf{data}\ \theta_0 :\!\times\!: \theta_1 = \theta_0 :\!\times\!: \theta_1
\end{array}
$$

Type Definitions 2: Phrase Types

3.1.1 Programs

PCF-programs are simply PCF-phrases with an empty context. A HASKELL-function is provided to check the syntactic validity of a program without evaluating it (note that since syntactic validity is checked by the type system it will never return false but will trigger a type checking error if applied to an invalid program).

$$
\begin{array}{l}
\mathit{isValidProgram} :: \mathit{Phrase}\ (\mathit{Context}\ \mathit{HNil})\ P \to \mathit{Bool} \\
\mathit{isValidProgram}\ (\mathit{Phrase}\ p) = \mathit{True}
\end{array}
$$

3.1.2 Examples

Invalid Phrase

For convenience, assume the definition of these HASKELL constant-functions for the first three identifiers:

$$\iota_0 = ident\ \mathit{firstLabel}$$
$$\iota_1 = ident\ (nextLabel\ \mathit{firstLabel})$$
$$\iota_2 = ident\ (nextLabel\ (nextLabel\ \mathit{firstLabel}))$$

Consider the invalid PCF-like phrase below (constants are discussed in section 3.3). HASKELL cannot infer a type that meets the constraints of the corresponding HASKELL-phrase. Specifically, the inner $\times$ function is unable to create a *Union* of the contexts that map ι_0 to *Int* and $Int :\rightarrow: \theta_1$, respectively.

$$\pi_0\langle 0, \langle \mathbf{succ}(\iota_0), \iota_0(0)\rangle\rangle \Longrightarrow \pi_0\ (zero \times ((appl\ succ\ \iota_0) \times (appl\ \iota_0\ zero)))$$

Invalid Program

Valid programs must not have free identifiers. Consider trivial invalid program of a lone identifier. This phrase must be evaluated with an environment that defines a mapping from ι_0 to some HASKELL-type, but *isValidProgram* requires an empty context.

$$\iota_0 \Longrightarrow isValidProgram\ (\iota_0)$$

Programs where identifiers appear in 'dead code' are also not valid. HASKELL cannot infer a union of the contexts for *zero* and that will result in the empty set, so this HASKELL-phrase is not a syntactically valid program.

$$\pi_0\langle 0, \iota_0\rangle \Longrightarrow isValidProgram\ (\pi_0\ (zero \times \iota_0))$$

3.2 Core Semantics

PCF-phrases are evaluated by the *eval* function, which maps a phrase and an environment to a HASKELL-value. The *eval* function is type-parameterized by class *Eval*, specified in listing 4. Evaluation of a phrase is done by *eval* with recursive ad-hoc polymorphism; instances of *eval* are specified in semantic definitions 1 on the following page.

$$[\![P]\!]u \in [\![\theta]\!] \quad \Longrightarrow \quad \begin{array}{c} \textbf{class}\ Eval\ P\ u\ t \mid P\ u \to t\ \textbf{where} \\ eval :: P \to u \to t \end{array}$$

Listing 4: Evaluation function

$$
\begin{aligned}
& \qquad\qquad\qquad\qquad\quad \textbf{instance}\ (\mathit{Eval}\ P\ u\ (x \to y), \mathit{Eval}\ Q\ u\ x) \\
[\![PQ]\!]u = [\![P]\!]u([\![Q]\!]u) \implies & \quad \Rightarrow \mathit{Eval}\ (\mathit{Appl}\ P\ Q)\ u\ y\ \textbf{where} \\
& \quad \mathit{eval}\ (\mathit{Appl}\ P\ Q)\ u = (\mathit{eval}\ P\ u)\ (\mathit{eval}\ Q\ u)
\end{aligned}
$$

$$
\begin{aligned}
& \qquad\qquad\qquad\qquad\quad \textbf{instance}\ (\mathit{Eval}\ P\ u\ x, \mathit{Eval}\ Q\ u\ y) \\
[\![\langle P, Q\rangle]\!]u = \langle [\![P]\!]u, [\![Q]\!]u\rangle \implies & \quad \Rightarrow \mathit{Eval}\ (\mathit{Pair}\ P\ Q)\ u\ (x, y)\ \textbf{where} \\
& \quad \mathit{eval}\ (\mathit{Pair}\ P\ Q)\ u = (\mathit{eval}\ P\ u, \mathit{eval}\ Q\ u)
\end{aligned}
$$

$$
\begin{aligned}
& \qquad\qquad\qquad\qquad\quad \textbf{instance}\ \mathit{HasField}\ (\mathit{Identifier}\ n)\ u\ x \\
[\![\iota_n]\!]u = u(\iota_n) \implies & \quad \Rightarrow \mathit{Eval}\ (\mathit{Identifier}\ n)\ u\ x\ \textbf{where} \\
& \quad \mathit{eval}\ \iota\ u = \mathit{hLookupByLabel}\ \iota\ u
\end{aligned}
$$

$$
\begin{aligned}
& \qquad\qquad\qquad\qquad\quad \textbf{instance}\ (\mathit{Perturb}\ \iota\ a\ u\ u', \mathit{Eval}\ P\ (\mathit{Environment}\ u')\ y) \\
[\![\lambda\iota.P]\!]ua = [\![P]\!](u \mid \iota \mapsto a) \implies & \quad \Rightarrow \mathit{Eval}\ (\mathit{Lambda}\ \iota\ P)\ (\mathit{Environment}\ u)\ (x \to y)\ \textbf{where} \\
& \quad \mathit{eval}\ (\mathit{Lambda}\ \iota\ P)\ u = \lambda a \to \mathit{eval}\ P\ (\mathit{perturb}\ \iota\ a\ u)
\end{aligned}
$$

$$
\begin{aligned}
& \qquad\qquad\qquad\qquad\quad \textbf{instance}\ \mathit{Eval}\ P\ u\ x \\
[\![\pi_0\langle P, Q\rangle]\!]u = [\![P]\!]u \implies & \quad \Rightarrow \mathit{Eval}\ (\pi_0\ (\mathit{Pair}\ P\ Q))\ u\ x\ \textbf{where} \\
& \quad \mathit{eval}\ (\pi_0\ (\mathit{Pair}\ P\ Q))\ u = \mathit{eval}\ P\ u
\end{aligned}
$$

Semantic Definitions 1: Phrases

3.3 Constants

Defined constants are declared like other PCF-phrases, as in raw syntax 2. Since they are the leaves of the PCF-syntax tree, the HASKELL type and data constructors have no arguments. Note that although $\mathbf{cond}_\theta$ and $\mathbf{rec}_\theta$ are polymorphic, they do not have ad-hoc polymorphic semantics and therefore there is no need for polymorphic types.

$$
\begin{aligned}
0 &\Longrightarrow \textbf{data}\ \mathit{Zero} = \mathit{Zero} \\
\textbf{true} &\Longrightarrow \textbf{data}\ \mathit{True} = \mathit{True} \\
\textbf{succ} &\Longrightarrow \textbf{data}\ \mathit{Succ} = \mathit{Succ} \\
\textbf{pred} &\Longrightarrow \textbf{data}\ \mathit{Pred} = \mathit{Pred} \\
\textbf{not} &\Longrightarrow \textbf{data}\ \mathit{Not} = \mathit{Not} \\
\textbf{cond}_\theta &\Longrightarrow \textbf{data}\ \mathit{Cond} = \mathit{Cond} \\
\textbf{eq} &\Longrightarrow \textbf{data}\ \mathit{Eq} = \mathit{Eq} \\
\textbf{rec}_\theta &\Longrightarrow \textbf{data}\ \mathit{Rec} = \mathit{Rec}
\end{aligned}
$$

Raw Syntax 2: Constants

The types of constant phrases are defined in type definitions 3 on the next page. Constructor functions are listed in syntax rules 2 on page 40, each constructs a phrase with an empty context.

Semantics for each constant function are listed in semantic definitions 2 on page 41. The HASKELL-data structure and context are ignored by these instances of *eval* (the instance is selected by type). Meanings of each constant is a built-in HASKELL-primitive or function defined in the HASKELL Prelude. Recursion is defined using the HASKELL **let** syntax, which implicitly determines the least fixed point.

$$
\begin{aligned}
0 : \mathbf{int} &\implies \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Zero}\ \mathit{Int} \\
\mathbf{true} : \mathbf{bool} &\implies \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{True}\ \mathit{Bool} \\
\mathbf{succ} : \mathbf{int} \to \mathbf{int} &\implies \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Succ}\ (\mathit{Int} :\!\to\!: \mathit{Int}) \\
\mathbf{pred} : \mathbf{int} \to \mathbf{int} &\implies \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Pred}\ (\mathit{Int} :\!\to\!: \mathit{Int}) \\
\mathbf{not} : \mathbf{bool} \to \mathbf{bool} &\implies \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Not}\ (\mathit{Bool} :\!\to\!: \mathit{Bool}) \\
\mathbf{cond}_\theta : \mathbf{bool} \times \theta \times \theta \to \theta &\implies \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Cond}\ (\mathit{Bool} :\!\times\!: \theta :\!\times\!: \theta :\!\to\!: \theta) \\
\mathbf{eq} : \mathbf{int} \times \mathbf{int} \to \mathbf{bool} &\implies \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Eq}\ (\mathit{Int} :\!\times\!: \mathit{Int} :\!\to\!: \mathit{Bool}) \\
\mathbf{rec}_\theta : (\theta \to \theta) \to \theta &\implies \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Rec}\ ((\theta :\!\to\!: \theta) :\!\to\!: \theta)
\end{aligned}
$$

Type Definitions 3: Constant Types

$$\frac{}{\vdash 0 : \mathbf{int}} \implies \begin{array}{l} zero :: Phrase\ (Context\ HNil)\ Zero \\ zero = Phrase \end{array}$$

$$\frac{}{\vdash \mathbf{true} : \mathbf{bool}} \implies \begin{array}{l} true :: Phrase\ (Context\ HNil)\ True \\ true = Phrase \end{array}$$

$$\frac{}{\vdash \mathbf{succ} : \mathbf{int} \to \mathbf{int}} \implies \begin{array}{l} succ :: Phrase\ (Context\ HNil)\ Succ \\ succ = Phrase \end{array}$$

$$\frac{}{\vdash \mathbf{pred} : \mathbf{int} \to \mathbf{int}} \implies \begin{array}{l} pred :: Phrase\ (Context\ HNil)\ Pred \\ pred = Phrase \end{array}$$

$$\frac{}{\vdash \mathbf{not} : \mathbf{bool} \to \mathbf{bool}} \implies \begin{array}{l} not :: Phrase\ (Context\ HNil)\ Not \\ not = Phrase \end{array}$$

$$\frac{}{\vdash \mathbf{cond}_\theta : \mathbf{bool} \times \theta \times \theta \to \theta} \implies \begin{array}{l} cond :: Phrase\ (Context\ HNil)\ Cond \\ cond = Phrase \end{array}$$

$$\frac{}{\vdash \mathbf{eq} : \mathbf{int} \times \mathbf{int} \to \mathbf{bool}} \implies \begin{array}{l} eq :: Phrase\ (Context\ HNil)\ Eq \\ eq = Phrase \end{array}$$

$$\frac{}{\vdash \mathbf{rec}_\theta : (\theta \to \theta) \to \theta} \implies \begin{array}{l} rec :: Phrase\ (Context\ HNil)\ Rec \\ rec = Phrase \end{array}$$

Syntax Rules 2: Constants

$[\![0]\!]u = 0$	$\Longrightarrow$	**instance** *Eval Zero u Prelude.Integer* **where** *eval _ _* $= 0$
$[\![\mathbf{true}]\!]u = \text{true}$	$\Longrightarrow$	**instance** *Eval True u Prelude.Bool* **where** *eval _ _* $=$ *Prelude.True*
$[\![\mathbf{succ}]\!]un = n + 1$	$\Longrightarrow$	**instance** *Eval Succ u* (*Prelude.Integer* $\rightarrow$ *Prelude.Integer*) **where** *eval _ _* $= \lambda n \rightarrow n$ *Prelude.*$+ 1$
$[\![\mathbf{pred}]\!]un = n - 1$	$\Longrightarrow$	**instance** *Eval Pred u* (*Prelude.Integer* $\rightarrow$ *Prelude.Integer*) **where** *eval _ _* $= \lambda n \rightarrow n$ *Prelude.*$- 1$
$[\![\mathbf{not}]\!]ub = \begin{cases} \text{false} & \text{if } b, \\ \text{true} & \text{otherwise.} \end{cases}$	$\Longrightarrow$	**instance** *Eval Not u* (*Prelude.Bool* $\rightarrow$ *Prelude.Bool*) **where** *eval _ _* $= \lambda b \rightarrow$ **if** b **then** *Prelude.False* **else** *Prelude.True*
$[\![\mathbf{cond}_\theta]\!]u\langle\langle b, x\rangle, y\rangle = \begin{cases} x & \text{if } b, \\ y & \text{otherwise.} \end{cases}$	$\Longrightarrow$	**instance** *Eval Cond u* ((($Prelude.Bool, \theta$), θ) $\rightarrow \theta$) **where** *eval _ _* $= \lambda((b, x), y) \rightarrow$ **if** b **then** x **else** y
$[\![\mathbf{eq}]\!]u\langle x, y\rangle = \begin{cases} \text{true} & \text{if } x = y, \\ \text{false} & \text{otherwise.} \end{cases}$	$\Longrightarrow$	**instance** *Eval Eq u* (($Prelude.Integer, Prelude.Integer$) $\rightarrow$ *Prelude.Bool*) **where** *eval _ _* $= \lambda(x, y) \rightarrow x$*Prelude.* $\equiv y$
$[\![\mathbf{rec}_\theta]\!]uf =$ least fixed point of f	$\Longrightarrow$	**instance** *Eval Rec u* (($\theta \rightarrow \theta$) $\rightarrow \theta$) **where** *eval _ _* $=$ **let** $y = \lambda f \rightarrow f\ (y\ f)$ **in** y

Semantic Definitions 2: Constants

3.4 Programs

A *Phrase* HASKELL-data structure with an empty context is a program. Evaluation of a program is begun by the function *evalProgram*, which delegates evaluation to the *eval* function.

$$
\begin{aligned}
&\mathit{evalProgram} :: (\mathit{Eval}\ p\ (\mathit{Environment}\ \mathit{HNil})\ x) \\
&\quad \Rightarrow \mathit{Phrase}\ (\mathit{Context}\ \mathit{HNil})\ p\ t \rightarrow x \\
&\mathit{evalProgram}\ (\mathit{Phrase}\ p) = \mathit{eval}\ p\ (\mathit{Record}\ \mathit{hNil})
\end{aligned}
$$

3.5 Example

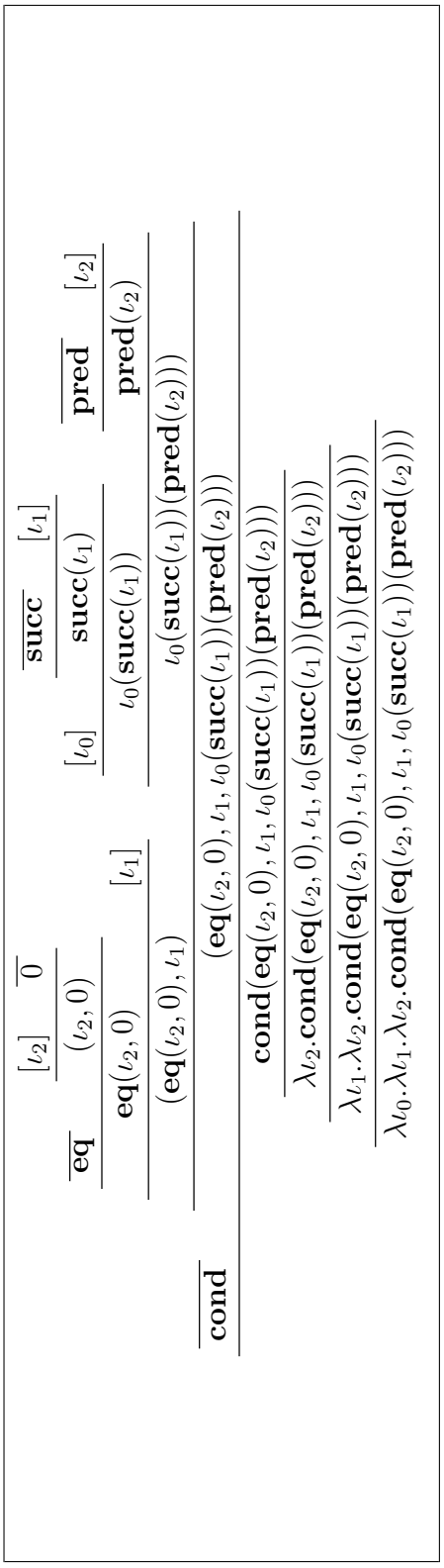
Consider a function that can be applied recursively through the parameter g, has a least fixed point that is a function that sums two integers:

$$
f\ g\ x\ y = \begin{cases} x & y = 0, \\ g\ (x+1)\ (y-1) & \mathit{otherwise}. \end{cases}
$$

The PCF-type for the function is $(\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}) \rightarrow \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$. It could be used in a phrase like:

$$\mathbf{rec}(f)(\mathbf{succ}(0))(\mathbf{succ}(0))$$

Traditional PCF-syntax is derived in derivation 1 on the following page (types and context are omitted for brevity).



Derivation 1: PCF Syntax

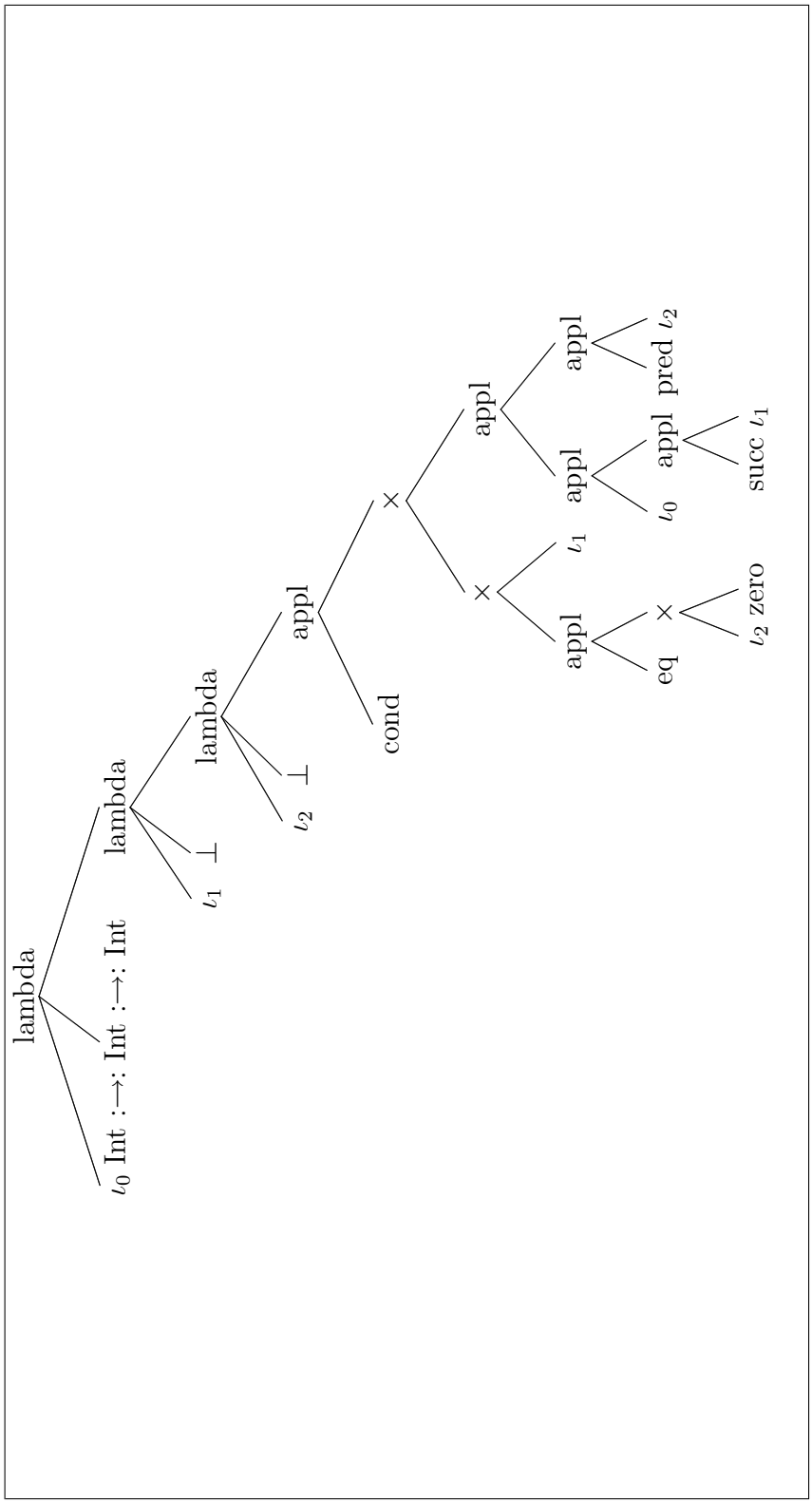
The HASKELL-code using defined PCF constructs to implement the function is in listing 5 on the next page. It could be used at the HASKELL interpreter prompt like so:

```
> let f = lambda firstLabel (Int :→: Int :→: Int) (···
> evalProgram (appl (appl (appl rec f) (appl succ zero)) (appl succ zero))
2
```

The parse tree for the code is in parse tree 1 on page 46. (ι_n is used to represent the Peano-enumerated identifiers.) The type of that HASKELL-function is in listing 6 on page 47, and parse tree 2 on page 48 shows the parsed HASKELL-type. Note that HASKELL-type variables θ_0 through θ_3 are polymorphic – semantic evaluation over any environment yields the HASKELL-type $(Integer \to Integer \to Integer) \to Integer \to Integer \to Integer$.

```
lambda firstLabel (Int :→: Int :→: Int) (
  lambda (nextLabel firstLabel) undefined (
    lambda (nextLabel (nextLabel firstLabel)) undefined (
      appl cond (
            ((appl eq (
                 ι2
               ×
                 zero))
          ×
            ι1)
        ×
          (appl (
            appl ι0 (
              appl succ
                ι1))
            (appl pred
              ι2))))))
```

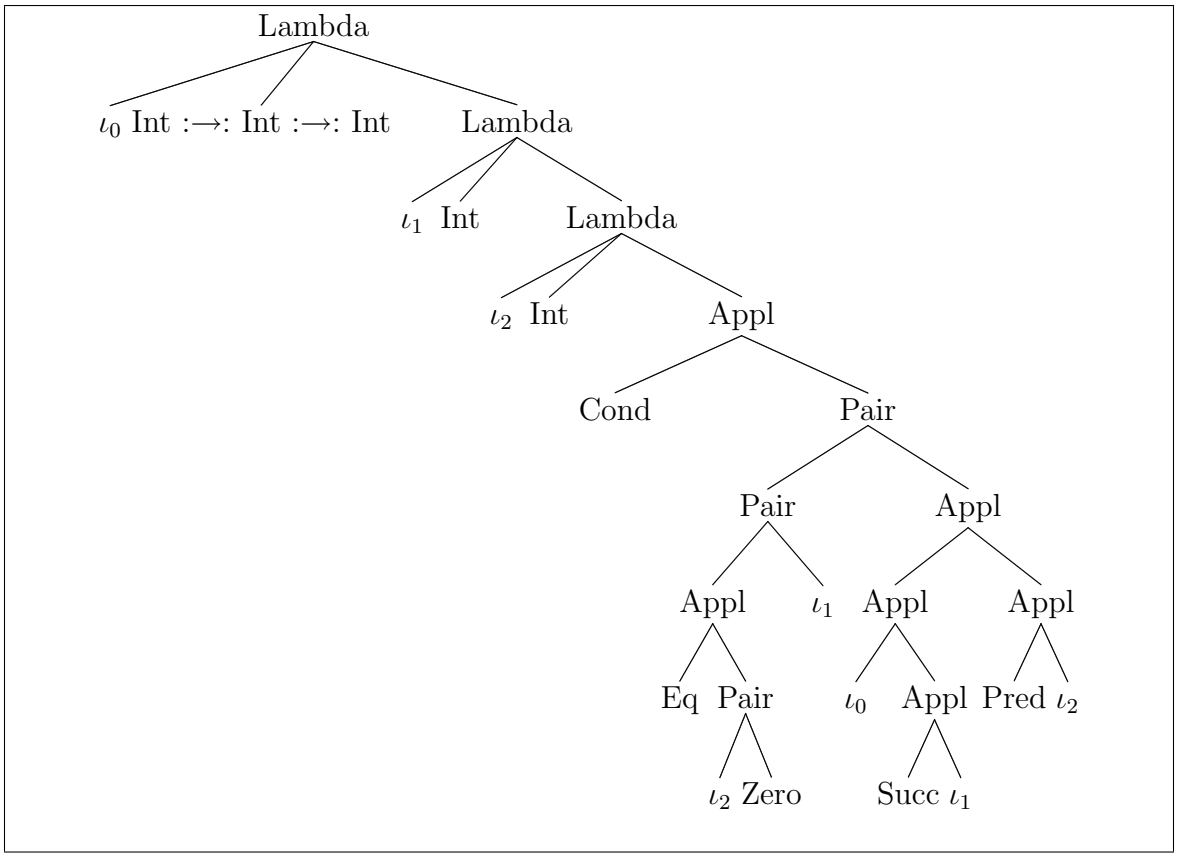
Listing 5: HASKELL Code



Parse Tree 1: HASKELL Syntax

```
Lambda (Identifier HZero) (Int :→: Int :→: Int)
  (Lambda (Identifier (HSucc HZero)) (Int)
    (Lambda (Identifier (HSucc (HSucc HZero))) (Int)
      (Appl Cond
        (Pair
          (Pair
            (Appl Eq
              (Pair
                (Identifier (HSucc (HSucc HZero)))
                Zero))
            (Identifier (HSucc HZero)))
          (Appl
            (Appl (Identifier HZero)
              (Appl Succ
                (Identifier (HSucc HZero))))
            (Appl Pred
              (Identifier (HSucc (HSucc HZero)))))))))
```

Listing 6: HASKELL Type



Parse Tree 2: HASKELL Type

3.6 Validation

In addition to the examples in section 3.1.2 on page 35 and section 3.5 on page 42, all implementation elements were unit tested. The syntax was validated by ensuring that the HASKELL type checker rejected the uses of the phrase constructing functions in listing 7. The semantics were validated by applying *evalProgram* to each of the phrases in listing 8 on the following page and ensuring the correct result as in listing 9 on page 51.

$$
\begin{aligned}
\mathbf{not}(0) &\Longrightarrow \mathit{appl}\ \mathit{not}\ \mathit{zero} \\
\mathbf{succ}(\mathbf{true}) &\Longrightarrow \mathit{appl}\ \mathit{succ}\ \mathit{true} \\
\mathbf{pred}(\mathbf{true}) &\Longrightarrow \mathit{appl}\ \mathit{pred}\ \mathit{true} \\
\mathbf{eq}(0, \mathbf{true}) &\Longrightarrow \mathit{appl}\ \mathit{eq}\ (\mathit{zero} \times \mathit{true}) \\
\pi_0\ \mathbf{true} &\Longrightarrow \pi_0\ \mathit{true} \\
\mathbf{true}(\mathbf{true}) &\Longrightarrow \mathit{appl}\ \mathit{true}\ \mathit{true} \\
\mathbf{cond}\langle 0, \mathbf{true}, \mathbf{true}\rangle &\Longrightarrow \mathit{appl}\ \mathit{cond}\ (\mathit{zero} \times \mathit{true} \times \mathit{true}) \\
\mathbf{cond}\langle \mathbf{true}, \mathbf{true}, 0\rangle &\Longrightarrow \mathit{appl}\ \mathit{cond}\ (\mathit{true} \times \mathit{true} \times \mathit{zero}) \\
\lambda\iota_0 : \mathbf{int}.\mathbf{not}(\iota_0) &\Longrightarrow \mathit{lambda}\ \mathit{firstLabel}\ \mathit{Int}\ (\mathit{appl}\ \mathit{not}\ \iota_0) \\
(\lambda\iota_0 : \mathbf{int}.\mathbf{true})(\mathbf{true}) &\Longrightarrow \mathit{appl}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{Int}\ \mathit{true})\ \mathit{true} \\
\mathbf{rec}(\mathbf{true}) &\Longrightarrow \mathit{appl}\ \mathit{rec}\ \mathit{true} \\
\mathbf{rec}(\lambda\iota_0 : \mathbf{int}.\mathbf{true}) &\Longrightarrow \mathit{appl}\ \mathit{rec}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{Int}\ \mathit{true})
\end{aligned}
$$

Listing 7: Invalid phrases

$$
\begin{array}{rcl}
\mathbf{true} & \Longrightarrow & \mathit{true} \\
\mathbf{not}(\mathbf{true}) & \Longrightarrow & \mathit{appl\ not\ true} \\
\mathbf{succ}(0) & \Longrightarrow & \mathit{appl\ succ\ zero} \\
\mathbf{pred}(\mathbf{succ}(0)) & \Longrightarrow & \mathit{appl\ pred\ (appl\ succ\ zero)} \\
\mathbf{eq}\langle 0, 0\rangle & \Longrightarrow & \mathit{appl\ eq\ (zero \times zero)} \\
\mathbf{eq}\langle \mathbf{succ}(0), 0\rangle & \Longrightarrow & \mathit{appl\ eq\ ((appl\ succ\ zero) \times zero)} \\
\pi_0\langle 0, \mathbf{true}\rangle & \Longrightarrow & \pi_0\ \mathit{(zero \times true)} \\
(\lambda\iota_0.\iota_0)(0) & \Longrightarrow & \mathit{appl\ (lambda\ firstLabel\ undefined\ \iota_0)\ zero} \\
(\lambda\iota_0.\mathbf{true})(0) & \Longrightarrow & \mathit{appl\ (lambda\ firstLabel\ undefined\ true)\ zero} \\
\mathbf{cond}\langle \mathbf{true}, \mathbf{true}, \mathbf{not}(\mathbf{true})\rangle & \Longrightarrow & \mathit{appl\ cond\ (true \times true \times (appl\ not\ true))} \\
\mathbf{cond}\langle \mathbf{not}(\mathbf{true}), 0, \mathbf{succ}(0)\rangle & \Longrightarrow & \mathit{appl\ cond\ ((appl\ not\ true) \times zero \times (appl\ succ\ zero))} \\
(\mathbf{cond}\langle \mathbf{true}, \mathbf{succ}, \mathbf{pred}\rangle)(\mathbf{succ}(0)) & \Longrightarrow & \mathit{appl\ (appl\ cond\ (true \times succ \times pred))\ (appl\ succ\ zero)} \\
\mathbf{cond}\langle \mathbf{true}, \langle 0, 0\rangle, \langle 0, 0\rangle\rangle & \Longrightarrow & \mathit{appl\ cond\ (true \times (zero \times zero) \times (zero \times zero))} \\
\mathbf{rec}(\lambda\iota_0.\mathbf{true}) & \Longrightarrow & \mathit{appl\ rec\ (lambda\ firstLabel\ undefined\ true)} \\
 & & \\
(\lambda\iota_1.(\lambda\iota_0.\iota_1)(0))(\mathbf{true}) & \Longrightarrow & \begin{array}{l}\mathit{appl\ (lambda\ (nextLabel\ firstLabel)\ undefined\ (} \\ \quad \mathit{appl\ (lambda\ firstLabel\ undefined\ \iota_1)\ zero))} \\ \mathit{(true)}\end{array} \\
 & & \\
(\lambda\iota_0.\lambda\iota_1.\iota_0(\iota_1))(\mathbf{succ})(0) & \Longrightarrow & \begin{array}{l}\mathit{appl\ (appl\ (lambda\ firstLabel\ undefined\ (} \\ \quad \mathit{lambda\ (nextLabel\ firstLabel)\ undefined\ (appl\ \iota_0\ \iota_1)))} \\ \mathit{succ)\ zero}\end{array}
\end{array}
$$

Listing 8: Semantics unit tests

$$
\begin{aligned}
&\mathit{true} \Rightarrow \mathit{True}\\
&\mathit{appl\ not\ true} \Rightarrow \mathit{False}\\
&\mathit{appl\ succ\ zero} \Rightarrow 1\\
&\mathit{appl\ pred\ (appl\ succ\ zero)} \Rightarrow 0\\
&\mathit{appl\ eq\ (zero} \times \mathit{zero)} \Rightarrow \mathit{True}\\
&\mathit{appl\ eq\ ((appl\ succ\ zero)} \times \mathit{zero)} \Rightarrow \mathit{False}\\
&\pi_0\ (\mathit{zero} \times \mathit{true}) \Rightarrow 0\\
&\mathit{appl\ (lambda\ firstLabel\ undefined}\ \iota_0)\ \mathit{zero} \Rightarrow 0\\
&\mathit{appl\ (lambda\ firstLabel\ undefined\ true)\ zero} \Rightarrow \mathit{True}\\
&\mathit{appl\ cond\ (true} \times \mathit{true} \times \mathit{(appl\ not\ true))} \Rightarrow \mathit{True}\\
&\mathit{appl\ cond\ ((appl\ not\ true)} \times \mathit{zero} \times \mathit{(appl\ succ\ zero))} \Rightarrow 1\\
&\mathit{appl\ (appl\ cond\ (true} \times \mathit{succ} \times \mathit{pred))\ (appl\ succ\ zero)} \Rightarrow 2\\
&\mathit{appl\ cond\ (true} \times \mathit{(zero} \times \mathit{zero)} \times \mathit{(zero} \times \mathit{zero))} \Rightarrow (0,0)\\
&\mathit{appl\ rec\ (lambda\ firstLabel\ undefined\ true)} \Rightarrow \mathit{True}\\
&\mathit{appl\ (lambda\ (nextLabel\ firstLabel)\ undefined\ (appl\ (lambda\ firstLabel\ undefined}\ \iota_1)\ \mathit{zero))\ (true)} \Rightarrow \mathit{True}\\
&\mathit{appl\ (appl\ (lambda\ firstLabel\ undefined\ (lambda\ (nextLabel\ firstLabel)\ undefined\ (appl}\ \iota_0\ \iota_1)))\ \mathit{succ)\ zero} \Rightarrow 1
\end{aligned}
$$

Listing 9: Expected evaluation results

3.7 Summary

A definitional interpreter for PCF can be implemented as HASKELL statements that represent PCF's syntax and semantics. This implementation can be validated using unit tests. The addition to PCF of constant functions forms a programming language and also the expressions fragment of Idealized ALGOL. Techniques in the implementation of PCF will be reused in the implementation of more complex languages in subsequent chapters.

Chapter 4

Implementation of Idealized ALGOL in Haskell

Idealized ALGOL can be thought of as an extension of PCF with imperative programming constructs, or as a combination of the applicative language PCF and the simple imperative WHILE-programs language. Higher-order functions, including recursion, operate on both expressions and commands, making Idealized ALGOL more expressive than languages purely descended from FORTRAN. The language presented here is based on [22], an idealization of the language standardized in [11].

PCF forms the functional fragment of Idealized ALGOL, it is combined with imperative features. In addition to environments, Idealized ALGOL phrases are evaluated over a state comprised of variable locations. Because the meaning of expressions can depend on (but not modify) the state, the HASKELL constructs that implement PCF must be 'lifted' to expressions: this is done by redefining the PCF definitions rather than constructing a lifting function.

The techniques established for definitional interpretation in chapter 3 on page 28

are followed closely for the implementation of Idealized ALGOL. Significant developments are the constants for manipulating the state in section 4.2 on page 58 and a polymorphic conditional constant in section 4.5 on page 72. Validation of the implementing code is performed by combining the unit tests from section 3.6 on page 49 with tests for imperative features. In chapter 5 on page 87, SCI is presented as a variant of Idealized ALGOL with substantial code reuse.

4.1 Types

The primitive data types of PCF, τ are encapsulated in expressions, and a new PCF-type, commands, support imperative operations. These are defined in type definitions 4 on the next page.

$\theta ::=$

$$\begin{aligned}
\theta_0 \to \theta_1 \quad &\Longrightarrow \quad \textbf{data}\ \theta_0 :\!\to\!: \theta_1 = \theta_0 :\!\to\!: \theta_1 \\
\theta_0 \times \theta_1 \quad &\Longrightarrow \quad \textbf{data}\ \theta_0 :\!\times\!: \theta_1 = \theta_0 :\!\times\!: \theta_1 \\
\textbf{exp}[\tau] \quad &\Longrightarrow \quad \textbf{data}\ \mathit{Exp}\ \tau = \mathit{Exp}\ \tau \\
\textbf{comm} \quad &\Longrightarrow \quad \textbf{data}\ \mathit{Comm} = \mathit{Comm}
\end{aligned}$$

$\tau ::=$

$$\begin{aligned}
\textbf{int} \quad &\Longrightarrow \quad \textbf{data}\ \mathit{Int} = \mathit{Int} \\
\textbf{bool} \quad &\Longrightarrow \quad \textbf{data}\ \mathit{Bool} = \mathit{Bool}
\end{aligned}$$

Type Definitions 4: Phrase and Primitive Types

For brevity, it is useful to define some HASKELL-type aliases. (Variables and acceptors are discussed in section 4.2 on page 58, conditionals in section 4.5 on page 72.)

$$\begin{aligned}
&\textbf{type}\ \mathit{Command}\ s = s \to s \\
&\textbf{type}\ \mathit{Expression}\ s\ \tau = s \to \tau \\
&\textbf{type}\ \mathit{Acceptor}\ s\ \tau = \mathit{Expression}\ s\ \tau \to \mathit{Command}\ s \\
&\textbf{type}\ \mathit{Variable}\ s\ \tau = (\mathit{Acceptor}\ s\ \tau, \mathit{Expression}\ s\ \tau) \\
&\textbf{type}\ \mathit{Conditional}\ s\ \theta = ((\mathit{Expression}\ s\ \mathit{Prelude.Bool}, \theta), \theta) \to \theta
\end{aligned}$$

Commands can modify states, and expressions can depend on them. The meaning

of an expression is a function from a set of states to a set of PCF-values, a command is a function from a set of states to a set of states. (States are discussed in section 4.2.) Semantic definitions 3 on the following page implements these meanings explicitly so that HASKELL can infer from the Idealized ALGOL-syntax what type of HASKELL-values are operated on by a parameterized constant (such as conditional). The inference is specified by the type constraints *EvalPhraseType*, which specifies an onto relation from Idealized ALGOL-types to HASKELL-types, and *EvalDataType*, which specifies a one-to-one relation.

$$[\![\theta]\!] : t \quad \Longrightarrow \quad \textbf{class } \mathit{EvalPhraseType}\ \theta\ t \mid \theta \to t$$

$$[\![\tau]\!] : t \quad \Longrightarrow \quad \textbf{class } \mathit{EvalDataType}\ \tau\ t \mid \tau \to t, t \to \tau$$

$$[\![P \to Q]\!] = [\![P]\!] \to [\![Q]\!] \implies \textbf{instance}\ (\mathit{EvalPhraseType}\ P\ x, \mathit{EvalPhraseType}\ Q\ y) \Rightarrow \mathit{EvalPhraseType}\ (P :\!\to\!:\ Q)\ (x \to y)$$

$$[\![P \times Q]\!] = \langle [\![P]\!], [\![Q]\!] \rangle \implies \textbf{instance}\ (\mathit{EvalPhraseType}\ P\ x, \mathit{EvalPhraseType}\ Q\ y) \Rightarrow \mathit{EvalPhraseType}\ (P :\!\times\!:\ Q)\ (x, y)$$

$$[\![\textbf{comm}]\!] = s \to s \implies \textbf{instance}\ \mathit{EvalPhraseType}\ \mathit{Comm}\ (\mathit{Command}\ s)$$

$$[\![\textbf{exp}[\tau]]\!] = s \to [\![\tau]\!] \implies \textbf{instance}\ \mathit{EvalDataType}\ \tau\ t \Rightarrow \mathit{EvalPhraseType}\ (\mathit{Exp}\ \tau)\ (\mathit{Expression}\ s\ t)$$

$$[\![\textbf{int}]\!] = \mathbb{Z} \implies \textbf{instance}\ \mathit{EvalDataType}\ \mathit{Int}\ \mathit{Prelude.Integer}$$

$$[\![\textbf{bool}]\!] = \{\text{true}, \text{false}\} \implies \textbf{instance}\ \mathit{EvalDataType}\ \mathit{Bool}\ \mathit{Prelude.Bool}$$

Semantic Definitions 3: Type Semantics

4.2 State

States store a collection of primitive values. States are implemented as HList records mapping labels to HASKELL-values.

$$\textbf{type } \mathit{State}\ s = \mathit{Record}\ s$$

Each label denotes a part of the state. Labels are encapsulated in variables, which are Idealized ALGOL-data structures consisting of an acceptor and a dereferencer, as in type definitions 5. An acceptor is a function from an expression to a command; the expression denotes the value to be stored in that part of the state, and the command, when executed, performs the storage. A dereferencer is an expression: when evaluated, it retrieves the value stored in that part of the state.

$$\begin{aligned} \mathbf{acc}[\tau] = \mathbf{exp}[\tau] \to \mathbf{comm} \quad &\Longrightarrow \quad \textbf{type } \mathit{Acc}\ \tau = \mathit{Exp}\ \tau :\!\!\to\!\!: \mathit{Comm} \\ \mathbf{var}[\tau] = \mathbf{acc}[\tau] \times \mathbf{exp}[\tau] \quad &\Longrightarrow \quad \textbf{type } \mathit{Var}\ \tau = \mathit{Acc}\ \tau :\!\!\times\!\!: \mathit{Exp}\ \tau \end{aligned}$$

Type Definitions 5: Variables

The acceptor and dereferencer parts of a variable are accessed by constant functions **deref** and :=. The raw syntax for these is listed in raw syntax 3 on the following page, the types in type definitions 6 on page 60, and the constructors in syntax rules 3 on the following page. Note that the type of τ is inferred in the implementation by HASKELL's type inference. The meanings of these constants are projections as in semantic definitions 4 on page 61.

$$\begin{aligned} \mathbf{deref}_\tau &\implies \mathbf{data}\ \mathit{Deref} = \mathit{Deref} \\ :=_\tau &\implies \mathbf{data}\ \mathit{Assign} = \mathit{Assign} \end{aligned}$$

Raw Syntax 3: Variable Constants

$$\frac{}{\vdash :=_\tau : \mathbf{var}[\tau] \times \mathbf{exp}[\tau] \to \mathbf{comm}} \implies \begin{array}{l} \mathit{assign} :: \mathit{Phrase}\ (\mathit{Context}\ \mathit{HNil})\ \mathit{Assign} \\ \mathit{assign} = \mathit{Phrase}\ \mathit{Assign} \end{array}$$

$$\frac{}{\vdash \mathbf{deref}_\tau : \mathbf{var}[\tau] \to \mathbf{exp}[\tau]} \implies \begin{array}{l} \mathit{deref} :: \mathit{Phrase}\ (\mathit{Context}\ \mathit{HNil})\ \mathit{Deref} \\ \mathit{deref} = \mathit{Phrase}\ \mathit{Deref} \end{array}$$

Syntax Rules 3: Variable Constants

$$
\begin{array}{rcl}
:=_\tau : \mathbf{var}[\tau] \times \mathbf{exp}[\tau] \to \mathbf{comm} & \Longrightarrow & \textbf{instance } \mathit{HasType} \ \Pi \ \mathit{Assign} \ (\mathit{Var}\ \tau :\times: \mathit{Exp}\ \tau :\rightarrow: \mathit{Comm}) \\
\mathbf{deref}_\tau : \mathbf{var}[\tau] \to \mathbf{exp}[\tau] & \Longrightarrow & \textbf{instance } \mathit{HasType} \ \Pi \ \mathit{Deref} \ (\mathit{Var}\ \tau :\rightarrow: \mathit{Exp}\ \tau)
\end{array}
$$

Type Definitions 6: Variable Constants

$$[\![:=_\tau]\!]u\langle v, e\rangle s = \pi_0(v)(e)(s) \quad \Longrightarrow \quad \textbf{instance } \textit{Eval Assign } u \textit{ (Variable } s\ t \rightarrow \textit{Command } s) \textbf{ where}$$
$$\textit{eval } \_\ \_ = \lambda((a, d), e) \rightarrow \lambda s \rightarrow a\ e\ s$$

$$[\![\mathbf{deref}_\tau]\!]uvs = \pi_1(v)(s) \quad \Longrightarrow \quad \textbf{instance } \textit{Eval Deref } u \textit{ (Variable } s\ t \rightarrow \textit{Expression } s\ t) \textbf{ where}$$
$$\textit{eval } \_\ \_ = \lambda(a, d) \rightarrow \lambda s \rightarrow d\ s$$

Semantic Definitions 4: Variable Constants

4.2.1 State Shape

State is implemented as a stack of mappings from labels to values, with each label referred to as a location. Expansion of state is performed by pushing a new association on the stack, referenced by a label that is the successor of the current top label.

The HASKELL-function *newLabel* given in listing 10 is ad-hoc polymorphic on state shape and returns a fresh location for that shape using the HList functions *firstLabel* and *nextLabel*.

```
class NewLabel s l | s → l where
  newLabel :: s → l

instance NewLabel (State HNil) (Label HZero) where
  newLabel s = firstLabel

instance HNat n
    ⇒ NewLabel (State (HCons (F (Label n) v) t)) (Label (HSucc n)) where
  newLabel (Record (HCons h t)) = nextLabel (labelF h)
```

Listing 10: Fresh location generation

The PCF-construct $\mathbf{new}_\tau$ takes a function from a variable to a command, expands the state to include a new variable initialized to an arbitrary value depending on its type, executes the command, and returns a modified state where the new variable has been deallocated. Initialization is implemented using ad-hoc polymorphism over *eval*, the case for integers is given in semantic definitions 5 on page 65 (Booleans are very similar); to support polymorphism, syntax rules 4 on page 64 type-parameterizes the *New* HASKELL-type structure with τ (the HASKELL-data structure on the left-hand side omits τ). Records are implemented as cons lists, so pushing and popping is

performed by consing a mapping onto the head, or taking only the tail, respectively.

$$\mathbf{new}_\tau \quad \Longrightarrow \quad \mathbf{data}\ \mathit{New}\ \tau = \mathit{New}$$

$$\mathbf{new}_\tau : (\mathbf{var}[\tau] \to \mathbf{comm}) \to \mathbf{comm} \quad \Longrightarrow \quad \mathbf{instance}\ \mathit{HasType}\ \Pi\ (\mathit{New}\ \tau)\ ((\mathit{Var}\ \tau :\to: \mathit{Comm}) :\to: \mathit{Comm})$$

$$\frac{}{\vdash \mathbf{new}_\tau : (\mathbf{var}[\tau] \to \mathbf{comm}) \to \mathbf{comm}} \quad \Longrightarrow \quad \begin{array}{l} \mathit{new} :: \mathit{Phrase}\ (\mathit{Context}\ \mathit{HNil})\ (\mathit{New}\ \tau) \\ \mathit{new} = \mathit{Phrase}\ \mathit{New} \end{array}$$

Syntax Rules 4: Local allocation

$$[\![\mathbf{new_{int}}]\!]bs = \bigcup_{\forall l \in dom(s)} (l \mapsto (b\ v\ s')(l)) \Longrightarrow$$

```
instance (
    NewLabel (State s0) l,
    HasField l s' Prelude.Integer,
    HFind l ls n,
    HUpdateAtHNat n (F l Prelude.Integer) s' s',
    HList s0,
    RecordLabels s' ls
) ⇒ Eval (New Int) u (
    (Variable (State s') Prelude.Integer
     → Command (State (HCons (F l Prelude.Integer) s0)))
     → Command (State s0)
) where
eval _ _ = λb → λ(Record s) →
    let l                    = newLabel (Record s)
        v                    = mkVar l
        s'                   = Record (HCons (newF l 0) s)
        Record (HCons x s0)  = b v s'
    in Record s0
```

where

- $s' = expand(s)$
- v is a variable referring to a fresh location in s' storing type **int** initialized to 0

Semantic Definitions 5: Local allocation for **int**

4.3 Constant Expressions

Idealized ALGOL constants have the same HASKELL-constructor functions as PCF constants. The types differ: an Idealized ALGOL constant is related to a PCF constant by wrapping each primitive type in an expression, as listed in type definitions 7 on the next page. The two exceptions to this pattern are *rec*, which is fully polymorphic, and therefore identical in Idealized ALGOL and PCF, and *cond*, which is discussed in section 4.5 on page 72.

Constant Idealized ALGOL-functions take expressions as arguments and execute those arguments on the current state before applying the HASKELL-function. These are listed in semantic definitions 6 on page 68.

$$
\begin{array}{rcl}
0 : \mathbf{exp}[\mathbf{int}] & \Rrightarrow & \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Zero}\ (\mathit{Exp}\ \mathit{Int}) \\
\mathbf{true} : \mathbf{exp}[\mathbf{bool}] & \Rrightarrow & \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{True}\ (\mathit{Exp}\ \mathit{Bool}) \\
\mathbf{succ} : \mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{exp}[\mathbf{int}] & \Rrightarrow & \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Succ}\ (\mathit{Exp}\ \mathit{Int} :\rightarrow: \mathit{Exp}\ \mathit{Int}) \\
\mathbf{pred} : \mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{exp}[\mathbf{int}] & \Rrightarrow & \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Pred}\ (\mathit{Exp}\ \mathit{Int} :\rightarrow: \mathit{Exp}\ \mathit{Int}) \\
\mathbf{not} : \mathbf{exp}[\mathbf{bool}] \rightarrow \mathbf{exp}[\mathbf{bool}] & \Rrightarrow & \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Not}\ (\mathit{Exp}\ \mathit{Bool} :\rightarrow: \mathit{Exp}\ \mathit{Bool}) \\
\mathbf{eq} : \mathbf{exp}[\mathbf{int}] \times \mathbf{exp}[\mathbf{int}] \rightarrow \mathbf{exp}[\mathbf{bool}] & \Rrightarrow & \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Eq}\ (\mathit{Exp}\ \mathit{Int} :\times: \mathit{Exp}\ \mathit{Int} :\rightarrow: \mathit{Exp}\ \mathit{Bool})
\end{array}
$$

Type Definitions 7: Constant Expressions

$$[\![0]\!]us = 0 \quad \Longrightarrow$$

instance *Eval Zero u* (*Expression s Prelude.Integer*) **where**
eval _ _ = $\lambda s \rightarrow 0$

$$[\![\textbf{true}]\!]us = \text{true} \quad \Longrightarrow$$

instance *Eval True u* (*Expression s Prelude.Bool*) **where**
eval _ _ = $\lambda s \rightarrow$ *Prelude.True*

$$[\![\textbf{succ}]\!]uns = n + 1 \quad \Longrightarrow$$

instance *Eval Succ u* ((*Expression s Prelude.Integer*)
$\rightarrow$ *Expression s Prelude.Integer*) **where**
eval _ _ = $\lambda n \rightarrow \lambda s \rightarrow$ *Prelude.succ* (*n s*)

$$[\![\textbf{pred}]\!]uns = n - 1 \quad \Longrightarrow$$

instance *Eval Pred u* ((*Expression s Prelude.Integer*)
$\rightarrow$ *Expression s Prelude.Integer*) **where**
eval _ _ = $\lambda n \rightarrow \lambda s \rightarrow$ *Prelude.pred* (*n s*)

$$[\![\textbf{not}]\!]ubs = \begin{cases} \text{false} & \text{if } [\![b]\!]s, \\ \text{true} & \text{otherwise.} \end{cases} \quad \Longrightarrow$$

instance *Eval Not u* ((*Expression s Prelude.Bool*)
$\rightarrow$ *Expression s Prelude.Bool*) **where**
eval _ _ = $\lambda b \rightarrow \lambda s \rightarrow$ *Prelude.not* (*b s*)

$$[\![\textbf{eq}]\!]u\langle x, y\rangle s = \begin{cases} \text{true} & \text{if } [\![x]\!]s = [\![y]\!]s, \\ \text{false} & \text{otherwise.} \end{cases} \quad \Longrightarrow$$

instance *Eval Eq u*
(*Expression s Prelude.Integer*, *Expression s Prelude.Integer*)
$\rightarrow$ *Expression s Prelude.Bool*
) **where**
eval _ _ = $\lambda(x, y) \rightarrow \lambda s \rightarrow$ (*x s*)*Prelude.* $\equiv$ (*y s*)

Semantic Definitions 6: Constant Expressions

4.4 Commands

As well as the PCF constants, Idealized ALGOL includes imperative programming constants: command composition and a 'no-operation' command (**skip**). Raw syntax 4 and syntax rules 5 are like other constants. Type definitions 8 on the following page show that no-operation is a simple command, while sequencing is a function on pairs of commands.

$$\begin{array}{rcl} \textbf{skip} & \Longrightarrow & \textbf{data}\ \mathit{Skip} = \mathit{Skip} \\ ; & \Longrightarrow & \textbf{data}\ \mathit{Sq} = \mathit{Sq} \end{array}$$

Raw Syntax 4: Constant Commands

$$\frac{}{\vdash \textbf{skip} : \textbf{comm}} \Longrightarrow \begin{array}{l} \mathit{skip} :: \mathit{Phrase}\ (\mathit{Context}\ \mathit{HNil})\ \mathit{Skip} \\ \mathit{skip} = \mathit{Phrase}\ \mathit{Skip} \end{array}$$

$$\frac{}{\vdash ; : \textbf{comm} \times \textbf{comm} \to \textbf{comm}} \Longrightarrow \begin{array}{l} \mathit{sq} :: \mathit{Phrase}\ (\mathit{Context}\ \mathit{HNil})\ \mathit{Sq} \\ \mathit{sq} = \mathit{Phrase}\ \mathit{Sq} \end{array}$$

Syntax Rules 5: Constant Commands

Commands are functions on state. The identity function is implemented by *skip*. Commands are composed by *sq*, which executes the first command against the state, and then the second against the modified state. These meanings are defined in semantic definitions 7 on page 71.

$$\begin{array}{rcl} \textbf{skip} : \textbf{comm} & \Longrightarrow & \textbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Skip}\ \mathit{Comm} \\ (;) : \textbf{comm} \times \textbf{comm} \to \textbf{comm} & \Longrightarrow & \textbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Sq}\ (\mathit{Comm} :\times: \mathit{Comm} :\rightarrow: \mathit{Comm}) \end{array}$$

Type Definitions 8: Constant Commands

$$[\![\mathbf{skip}]\!]us = s \quad \Longrightarrow \quad \textbf{instance } \mathit{Eval\ Skip\ u\ (Command\ s)} \textbf{ where}$$
$$\mathit{eval}\ \_\ \_\ \_ = \lambda s \rightarrow s$$

$$[\![;]\!]u\langle P, Q\rangle s = [\![Q]\!]u([\![P]\!]us) \quad \Longrightarrow \quad \textbf{instance } \mathit{Eval\ Sq\ u\ ((Command\ s, Command\ s) \rightarrow Command\ s)} \textbf{ where}$$
$$\mathit{eval}\ \_\ \_\ \_ = \lambda(P, Q) \rightarrow \lambda s \rightarrow Q\ (P\ s)$$

Semantic Definitions 7: Constant Commands

4.5 Conditional

The semantics for the conditional function vary depending on the type of the second and third arguments. This is implemented using ad-hoc polymorphism over *eval*, as defined in semantic definitions 8 to 9 on pages 74–75. To support polymorphism, the *Cond* HASKELL-type structure is parameterized by θ at the type level (the left-hand side omits θ).

$$\mathbf{cond}_\theta \quad \Longrightarrow \quad \mathbf{data}\ \mathit{Cond}\ \theta = \mathit{Cond}$$

The Idealized ALGOL-type of the conditional constant is determined by its parameterization. The HASKELL-constructor *cond* infers the parameterized type from the context. This relationship is defined in syntax rules 6 on the following page.

Semantic definitions 8 on page 74 defines instances of *eval* for expressions and commands, semantic definitions 9 on page 75 defines instances for functions and products. Expressions and commands evaluate to the HASKELL conditional statement, differing from the implementation of conditional in PCF only by addition of state. Functions and products recursively evaluate conditional statements on their structural components. Note that conditional is polymorphic over variables because they are a type alias for a combination of functions and products.

$$\mathbf{cond}_\theta : \mathbf{exp}[\mathbf{bool}] \times \theta \times \theta \to \theta \quad \Longrightarrow \quad \textbf{instance } \mathit{HasType}\ \Pi\ (\mathit{Cond}\ \theta)\ (\mathit{Exp}\ \mathit{Bool} :\times: \theta :\times: \theta :\to: \theta)$$

$$\frac{}{\vdash \mathbf{cond}_\theta : \mathbf{exp}[\mathbf{bool}] \times \theta \times \theta \to \theta} \quad \Longrightarrow \quad \begin{array}{l} \mathit{cond} :: \mathit{Phrase}\ (\mathit{Context}\ \mathit{HNil})\ (\mathit{Cond}\ \theta) \\ \mathit{cond} = \mathit{Phrase}\ \mathit{Cond} \end{array}$$

Syntax Rules 6: Conditional

$$[\![cond_{\mathbf{exp}[\tau]}]\!]u\langle\langle b, x\rangle, y\rangle s = \begin{cases} [\![x]\!]s & \text{if } [\![b]\!]s, \\ [\![y]\!]s & \text{otherwise.} \end{cases}$$

$$\Longrightarrow$$

instance *EvalPhraseType* (*Exp* τ) (*Expression* s t)
  $\Rightarrow$ *Eval* (*Cond* (*Exp* τ)) u (*Conditional* s (*Expression* s t)) **where**
  eval $\_$ $\_$ $\_$ $= \lambda((b, x), y) \rightarrow \lambda s \rightarrow$ **if** b s **then** x s **else** y s

$$[\![cond_{\mathbf{comm}}]\!]u\langle\langle b, x\rangle, y\rangle s = \begin{cases} [\![x]\!]s & \text{if } [\![b]\!]s, \\ [\![y]\!]s & \text{otherwise.} \end{cases}$$

$$\Longrightarrow$$

instance *Eval* (*Cond Comm*) u (*Conditional* s (*Command* s)) **where**
  eval $\_$ $\_$ $\_$ $= \lambda((b, x), y) \rightarrow \lambda s \rightarrow$ **if** b s **then** x s **else** y s

Semantic Definitions 8: Simple conditionals

$$[\![cond_{\theta_0 \to \theta_1}]\!] u \langle \langle b, x \rangle, y \rangle a = [\![cond_{\theta_1}]\!] u ((b, [\![x]\!] a), [\![y]\!] a)$$
$$\Longrightarrow$$

```
instance (
    Eval (Cond θ1) u (Conditional s y),
    EvalPhraseType θ0 x
  ) ⇒ Eval (Cond (θ0 :→: θ1)) u (Conditional s (x → y)) where
  eval _ u = λ((b, x), y) → λa → eval (Cond :: Cond θ1) u ((b, x a), y a)
```

$$[\![cond_{\theta_0 \times \theta_1}]\!] u \langle \langle b, x \rangle, y \rangle = \langle [\![\mathbf{cond}_{\theta_0}]\!] u \langle \langle b, \pi_0 x \rangle, \pi_0 y \rangle, [\![\mathbf{cond}_{\theta_1}]\!] u \langle \langle b, \pi_1 x \rangle, \pi_1 y \rangle \rangle$$
$$\Longrightarrow$$

```
instance (
    Eval (Cond θ0) u (Conditional s x),
    Eval (Cond θ1) u (Conditional s y)
  ) ⇒ Eval (Cond (Pair θ0 θ1)) u (Conditional s (x, y)) where
  eval _ u = λ((b, (p, q)), (r, s)) → (eval (Cond :: Cond θ0) u ((b, p), r), eval (Cond :: Cond θ1) u ((b, q), s))
```

Semantic Definitions 9: Compound conditionals

4.6 Programs

Normally, Idealized ALGOL programs must be commands. A full implementation of Idealized ALGOL would include a mechanism to perform input and output (either predefined variables that access designated state locations, or constants that work through a more abstract mechanism). These concerns are outside the scope of this dissertation, therefore programs are defined as phrases with an empty context that evaluate to either expressions or commands, which execute with an empty state.

$$
\begin{aligned}
&\mathit{evalProgram} :: (\mathit{Eval}\ p\ (\mathit{Environment}\ \mathit{HNil})\ (\mathit{State}\ \mathit{HNil} \to x)) \\
&\quad \Rightarrow \mathit{Phrase}\ (\mathit{Context}\ \mathit{HNil})\ p\ t \to x \\
&\mathit{evalProgram}\ (\mathit{Phrase}\ p) = \mathit{eval}\ p\ (\mathit{Record}\ \mathit{hNil})\ (\mathit{Record}\ \mathit{hNil})
\end{aligned}
$$

The syntactic validity of programs is determined by the same *isValidProgram* function as PCF.

4.7 Example

4.7.1 Invalid Phrase

Consider the invalid Idealized ALGOL-like phrase below. From the first usage of ι_0, HASKELL's type inference engine infers that it is a **var**[**int**]. It then tries, and fails, to infer an instance of *HasType* such that **true** is of type **exp**[**int**].

$\iota_0 := 0; \iota_0 := \mathbf{true} \quad \Longrightarrow \quad appl\ sq\ ((appl\ assign\ (\iota_0 \times zero)) \times (appl\ assign\ (\iota_0 \times true)))$

4.7.2 Invalid Program

Idealized ALGOL uses the same function as PCF to check program validity, so programs still may not include free identifiers.

```
isValidProgram (appl assign (ι0 × zero))
```

4.7.3 Valid Program

Consider a program that declares a new variable, initializes it to zero, increments it, and then finishes. The traditional Idealized ALGOL-syntax for this program is derived in derivation 2 (types and context are omitted for brevity).

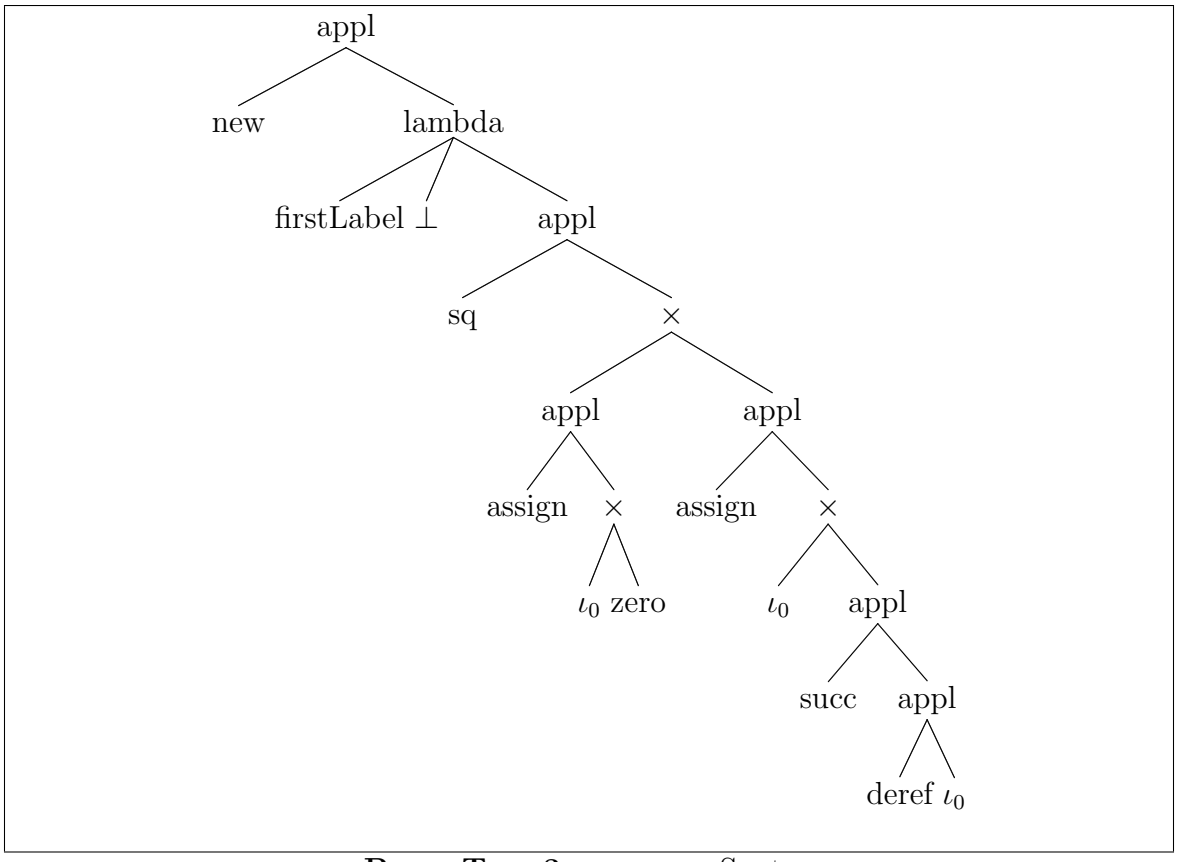
$$\dfrac{\overline{\mathbf{new_{int}}} \qquad \dfrac{\dfrac{\overline{\iota_0} \quad \overline{:=} \quad \overline{0}}{\iota_0 := 0} \quad \overline{;} \quad \dfrac{\overline{\iota_0} \quad \overline{:=} \quad \dfrac{\overline{\mathbf{succ}} \quad \dfrac{\overline{\mathbf{deref}} \quad \overline{\iota_0}}{\mathbf{deref}(\iota_0)}}{\mathbf{succ}(\mathbf{deref}(\iota_0))}}{\iota_0 := \mathbf{succ}(\mathbf{deref}(\iota_0))}}{\iota_0 := 0; \iota_0 := \mathbf{succ}(\mathbf{deref}(\iota_0))}}{\lambda\iota_0.\iota_0 := 0; \iota_0 := \mathbf{succ}(\mathbf{deref}(\iota_0))}}{\mathbf{new_{int}}(\lambda\iota_0.\iota_0 := 0; \iota_0 := \mathbf{succ}(\mathbf{deref}(\iota_0)))}$$

Derivation 2: Idealized ALGOL Syntax

The HASKELL-code using defined Idealized ALGOL constructs to implement the command is in listing 11, and the parse tree for the code is in parse tree 3 on the next page. The type of the HASKELL-function is in listing 12 on the following page, while parse tree 4 on page 80 shows the parsed HASKELL-type.

```
appl new (
    lambda firstLabel undefined (
      appl sq (
            (appl assign
                  (ι0 × zero))
             ×
            (appl assign
                  (ι0
                   ×
                  (appl succ (
                        appl deref
                              ι0)))))))
```

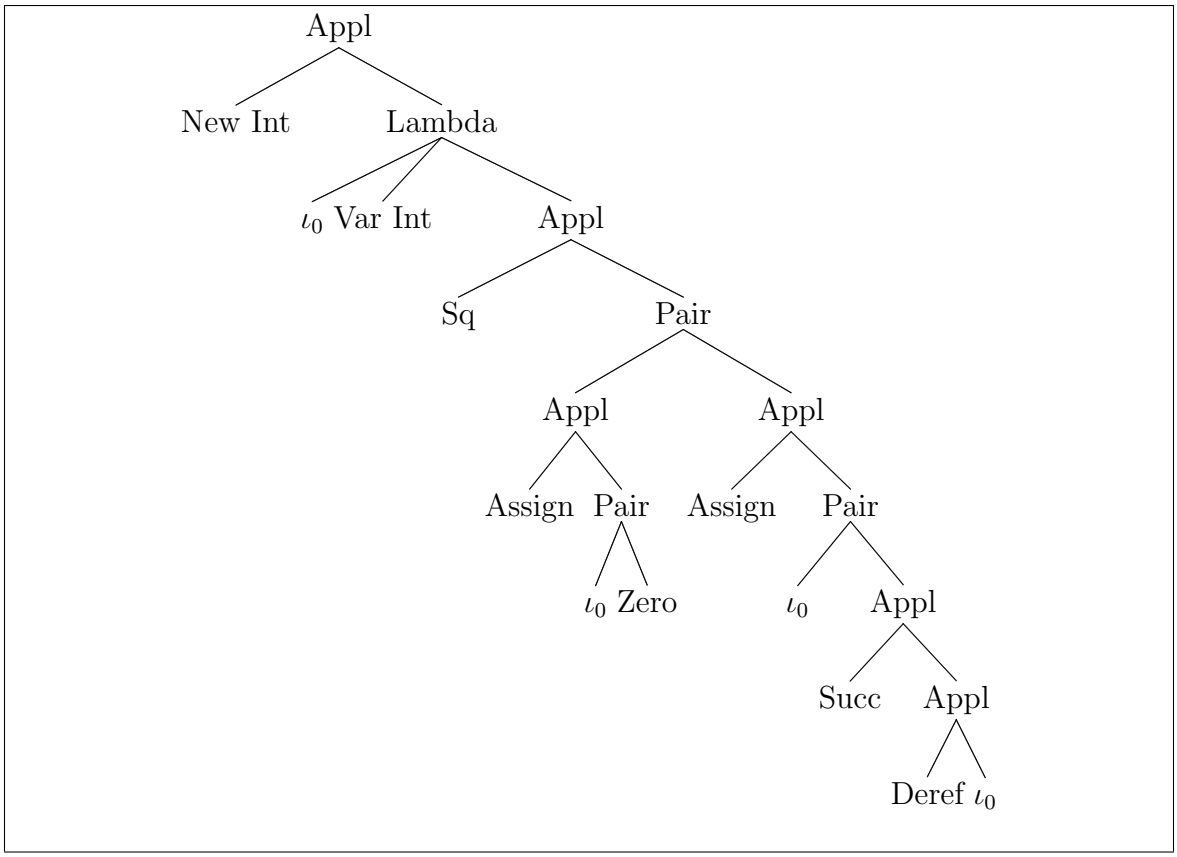
Listing 11: HASKELL Code



Parse Tree 3: HASKELL Syntax

```
Appl New
  (Lambda (Identifier HZero) (Var Int)
    (Appl Sq
      (Pair (Appl Assign
              (Pair (Identifier HZero)
                    Zero))
            (Appl Assign
              (Pair (Identifier HZero)
                    (Appl Succ
                      (Appl Deref
                        (Identifier HZero)))))))))
```

Listing 12: HASKELL Type



Parse Tree 4: HASKELL Type

4.8 Validation

Since PCF is a subset of Idealized ALGOL, the implementation of Idealized ALGOL was also validated against the unit tests in section 3.6 on page 49. The HASKELL type checker rejected all the uses of imperative phrase constructor functions in listing 13 on the next page.

The language defined in this chapter has no output mechanism. To verify that a phrase yields the desired result, it is put in a program that halts if the desired result was obtained and otherwise does not halt. The syntactic sugar abbreviations in listing 14 on page 83 are used: **diverge** represents infinite recursion, **assert** halts if its parameter is true and diverges otherwise, and **test** checks that the result of executing its first argument is the assignment of expected-value x to a new variable. Validation consisted of ensuring that all phrases in listings 15 on page 84 and 16 on page 85 halt.

$$
\begin{array}{l}
\mathbf{cond}_\theta\langle \mathbf{true}, \mathbf{skip}, 0\rangle \Longrightarrow appl\ cond\ (true \times skip \times zero) \\
\qquad \mathbf{dereftrue} \Longrightarrow appl\ deref\ true \\
\qquad\qquad 0 := 0 \Longrightarrow appl\ assign\ (zero \times zero) \\
\qquad\qquad \iota_0 := \mathbf{skip} \Longrightarrow appl\ assign\ (\iota_0 \times skip) \\
\qquad \mathbf{sq}\langle \mathbf{skip}, \mathbf{true}\rangle \Longrightarrow appl\ sq\ (skip \times true) \\
\qquad\qquad \mathbf{newtrue} \Longrightarrow appl\ new\ true \\
\\
\mathbf{new}_\tau(\lambda\iota_0 : \mathbf{var}[\mathbf{bool}].\iota_0 := 0) \\
\qquad \Longrightarrow appl\ new\ (lambda\ firstLabel\ (undefined :: Var\ Bool)\ (appl\ assign\ (\iota_0 \times zero))) \\
\\
\mathbf{new}_{\mathbf{int}}(\lambda\iota_0.\iota_0 := 0); \iota_0 := 0 \\
\qquad \Longrightarrow appl\ sq\ (appl\ new\ (lambda\ firstLabel\ undefined\ (appl\ assign\ (\iota_0 \times zero))) \times appl\ assign\ (\iota_0 \times zero))
\end{array}
$$

Listing 13: Invalid phrases

$$\begin{aligned}
\mathbf{diverge} &= \mathbf{rec}_{\mathbf{comm}}(\lambda\iota_0 : \mathbf{comm}.\iota_0) \\
\mathbf{assert}\ b &= \mathbf{cond}_{\mathbf{comm}}\langle b, \mathbf{skip}, \mathbf{diverge}\rangle \\
\mathbf{test}\ P\ x &= \mathbf{new}_\theta(\lambda\iota_0.P; \mathbf{assert}(\mathbf{eq}\langle\mathbf{deref}(\iota_0), x\rangle))
\end{aligned}$$

$$\mathbf{diverge} \quad \Longrightarrow \quad \mathit{diverge} = \mathit{appl}\ \mathit{rec}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{Comm}\ \iota_0)$$

$$\mathbf{assert}\ b \quad \Longrightarrow \quad \mathit{assert}\ b = \mathit{appl}\ \mathit{cond}\ (b \times \mathit{skip} \times \mathit{diverge})$$

$$\begin{aligned}
&\mathit{test}\ P\ x = \mathit{appl}\ \mathit{new}\ (\\
\mathbf{test}\ P\ x \quad \Longrightarrow \quad &\mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ (\\
&\quad \mathit{appl}\ \mathit{sq}\ (P \times \mathit{assert}\ (\mathit{appl}\ \mathit{eq}\ (\mathit{appl}\ \mathit{deref}\ \iota_0 \times x)))))
\end{aligned}$$

Listing 14: Testing functions

$$\mathbf{skip} \Longrightarrow \mathit{skip}$$

$$\mathbf{new_{int}}(\lambda\iota_0.\mathbf{skip}) \Longrightarrow \mathit{appl\ new\ (lambda\ firstLabel\ undefined\ skip)}$$

$$\mathbf{new_{int}}(\lambda\iota_0.\iota_0 := 0) \Longrightarrow \mathit{appl\ new\ (lambda\ firstLabel\ undefined\ (appl\ assign\ (\iota_0 \times zero)))}$$

$$\mathbf{new_{bool}}(\lambda\iota_0.\iota_0 := \mathbf{true}) \Longrightarrow \mathit{appl\ new\ (lambda\ firstLabel\ undefined\ (appl\ assign\ (\iota_0 \times true)))}$$

$$\mathbf{new}(\lambda\iota_0.\mathbf{assert}(\mathbf{eq}\langle\underline{\mathbf{cond_{exp[int]}}\langle\mathbf{true}, \mathbf{deref}(\iota_0), \mathbf{succ}(0)\rangle}, 0\rangle)) \Longrightarrow \mathit{appl\ new\ (lambda\ firstLabel\ undefined\ (assert\ (appl\ eq\ (appl\ cond\ (true \times appl\ deref\ \iota_0 \times appl\ succ\ zero) \times zero))))}$$

$$\mathbf{test}\ (\underline{\mathbf{cond_{comm}}(\mathbf{true}, \iota_0 := \mathbf{succ}(0), \mathbf{skip})})\ \mathbf{succ}(0) \Longrightarrow \mathit{test\ (appl\ cond\ (true \times appl\ assign\ (\iota_0 \times appl\ succ\ zero) \times skip))\ (appl\ succ\ zero)}$$

Listing 15: Semantics unit tests, part 1

$$\textbf{test}\ (\underline{\iota_0 := \textbf{succ}(0)})\ \textbf{succ}(0)$$
$$\Longrightarrow \mathit{test}\ (\lambda \mathit{underline}\{\mathit{appl}\ \mathit{assign}\ (\iota_0 \times \mathit{appl}\ \mathit{succ}\ \mathit{zero})\})\ (\mathit{appl}\ \mathit{succ}\ \mathit{zero})$$

$$\textbf{test}\ (\underline{\iota_0 := \textbf{succ}(0); \iota_0 := 0})\ 0$$
$$\Longrightarrow \mathit{test}\ (\mathit{appl}\ \mathit{sq}\ (\mathit{appl}\ \mathit{assign}\ (\iota_0 \times \mathit{appl}\ \mathit{succ}\ \mathit{zero}) \times \mathit{appl}\ \mathit{assign}\ (\iota_0 \times \mathit{zero})))\ \mathit{zero}$$

$$(\textbf{cond}_{\theta \to \textbf{comm}} \langle \textbf{true}, \lambda\iota_0.\textbf{skip}, \lambda\iota_0.\textbf{skip}\rangle)(\textbf{true})$$
$$\Longrightarrow \mathit{appl}\ (\mathit{appl}\ \mathit{cond}\ (\mathit{true} \times \mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ \mathit{skip} \times \mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ \mathit{skip}))\ \mathit{true}$$

$$\textbf{test}\ (\underline{(:=)(\textbf{cond}_{\textbf{var}[\textbf{int}] \times \textbf{exp}[\textbf{int}]} \langle \textbf{true}, \langle \iota_0, \textbf{succ}(0)\rangle, \langle \iota_0, 0\rangle\rangle)})\ \textbf{succ}(0)$$
$$\Longrightarrow \mathit{test}\ (\mathit{appl}\ \mathit{assign}\ (\mathit{appl}\ \mathit{cond}\ (\mathit{true} \times (\iota_0 \times \mathit{appl}\ \mathit{succ}\ \mathit{zero}) \times (\iota_0 \times \mathit{zero}))))\ (\mathit{appl}\ \mathit{succ}\ \mathit{zero})$$

$$\textbf{test}\ (\underline{\textbf{new}_{\textbf{int}}(\lambda\iota_1.\iota_1 := \textbf{succ}(0); \iota_0 := \textbf{deref}\ \iota_1)})\ \textbf{succ}(0)$$
$$\Longrightarrow \mathit{test}\ (\mathit{appl}\ \mathit{new}\ (\mathit{lambda}\ (\mathit{nextLabel}\ \mathit{firstLabel})\ \mathit{undefined}\ (\mathit{appl}\ \mathit{sq}\ (\mathit{appl}\ \mathit{assign}\ (\iota_1 \times \mathit{zero}) \times \mathit{appl}\ \mathit{assign}\ (\iota_0 \times \mathit{appl}\ \mathit{succ}\ \mathit{zero})))))\ (\mathit{appl}\ \mathit{succ}\ \mathit{zero})$$

Listing 16: Semantics unit tests, part 2

4.9 Summary

PCF forms the expression fragment of Idealized ALGOL. Major additions to Idealized ALGOL are manipulation of state and polymorphic conditionals. A definitional interpreter for Idealized ALGOL in HASKELL can be implemented using the techniques for implementing PCF, but code reuse is non-trivial. Unit tests can be reused for validation of the applicative part of the implementation.

Chapter 5

Implementation of SCI in Haskell

SCI is a primarily-syntactic variant of Idealized ALGOL. It includes a constraint on application to ensure non-interference and only passive functions can be recursive. In addition, given SCI without interference it is possible to define parallelism as an extension to Idealized ALGOL.

The SCI language presented here is a simplified form of the language in [18]; the language presented in this chapter has extremely conservative criteria for non-interference. Inteference is specified in section 5.1 on the next page. Changes to Idealized ALGOL to yield SCI are detailed in sections 5.2 on the following page and 5.3 on page 95. Parallel composition of commands is defined in section 5.4 on page 95. Since SCI is close to a subset of Idealized ALGOL, the implementation can be validated with examples of the new features and ensuring that the interference constraints disallow appropriate phrases that were allowed under Idealized ALGOL's type system.

Passivity is discussed in chapter 6 on page 105 in the form of the SCIR language. Implementation of SCI without passivity is valuable because it introduces the pattern of type constraints that are used heavily in chapter 7 on page 118. Parallelism is

in some ways the simplest case of non-interference, so consideration of it without passivity is a pure look at non-interference.

5.1 Interference

Two phrases that do not share free identifiers are non-interfering in SCI. In definition 1, non-interference is implemented as a type constraint that says two contexts are non-interfering if the intersection of their domains is the empty set.

$$\mathrm{dom}(\Pi_0) \cap \mathrm{dom}(\Pi_1) = \emptyset \quad \Longrightarrow$$

```
class NonInterfering Π0 Π1
instance (
      RecordLabels Π0 I0,
      RecordLabels Π1 I1,
      HTIntersect I0 I1 HNil
  ) ⇒ NonInterfering (Context Π0) (Context Π1)
```

Definition 1: Non-interference

5.2 Syntax

To Idealized ALGOL's phrase types, SCI adds two more to deal specifically with interference control: a pair of phrases that do not interfere ($\otimes$), called a disjoint product, and a kind of function that is guaranteed not to interfere with itself or any other phrase ($\underset{P}{\longrightarrow}$), called a passive function. These are listed in type definitions 9 on the next page.

$$
\begin{array}{rcl}
\theta ::= & & \\
\quad \theta_0 \to \theta_1 & \Longrightarrow & \textbf{data}\ \theta_0 :\!\to\!: \theta_1 = \theta_0 :\!\to\!: \theta_1 \\
\quad \theta_0 \times \theta_1 & \Longrightarrow & \textbf{data}\ \theta_0 :\!\times\!: \theta_1 = \theta_0 :\!\times\!: \theta_1 \\
\quad \theta_0 \underset{P}{\to} \theta_1 & \Longrightarrow & \textbf{data}\ \theta_0 :\!\underset{P}{\to}\!: \theta_1 = \theta_0 :\!\underset{P}{\to}\!: \theta_1 \\
\quad \theta_0 \otimes \theta_1 & \Longrightarrow & \textbf{data}\ \theta_0 :\!\otimes\!: \theta_1 = \theta_0 :\!\otimes\!: \theta_1 \\
\quad \textbf{exp}[\tau] & \Longrightarrow & \textbf{data}\ \mathit{Exp}\ \tau = \mathit{Exp}\ \tau \\
\quad \textbf{comm} & \Longrightarrow & \textbf{data}\ \mathit{Comm} = \mathit{Comm}
\end{array}
$$

Type Definitions 9: Phrase Types

The SCI syntax and corresponding HASKELL-types combine two phrases into a disjoint product, and thus promote an arbitrary function to a passive function. HASKELL-type definitions are listed in raw syntax 5, and their relation to SCI-types is specified in type definitions 10 on the next page.

$$
\begin{array}{rcl}
[P, Q] & \Longrightarrow & \textbf{data}\ \mathit{Disjoint}\ P\ Q = \mathit{Disjoint}\ P\ Q \\
\textbf{promote}\, P & \Longrightarrow & \textbf{data}\ \mathit{Promote}\ P = \mathit{Promote}\ P
\end{array}
$$

Raw Syntax 5: Raw Syntax

The constructors are listed in syntax rules 7 on page 92. The disjoint product is constructed the same way as a possibly-interfering product with the addition of the *NonInterfering* constraint on contexts. Passive functions are defined as those with no free identifiers, so the constructor requires a phrase with an empty context.

$$\frac{\Pi_0 \vdash P : \theta_0 \quad \Pi_1 \vdash Q : \theta_1}{\Pi_0, \Pi_1 \vdash [P, Q] : \theta_0 \otimes \theta_1} \mathrm{dom}(\Pi_0) \not\asymp \mathrm{dom}(\Pi_1) \quad \Longrightarrow \quad \textbf{instance}\ (\mathit{HasType}\ \Pi\ P\ \theta_0, \mathit{HasType}\ \Pi\ Q\ \theta_1) \Rightarrow \mathit{HasType}\ \Pi\ (\mathit{Disjoint}\ P\ Q)\ (\theta_0 :\otimes: \theta_1)$$

$$\frac{\vdash P : \theta_0 \to \theta_1}{\vdash \textbf{promote}\, P : \theta_0 \underset{P}{\longrightarrow} \theta_1} \quad \Longrightarrow \quad \textbf{instance}\ \mathit{HasType}\ \Pi\ P\ (\theta_0 :\to: \theta_1) \Rightarrow \mathit{HasType}\ \Pi\ (\mathit{Promote}\ P)\ (\theta_0 :\underset{P}{\to}: \theta_1)$$

Type Definitions 10: Phrase Typing

$$\frac{\Pi_0 \vdash P : \theta_0 \quad \Pi_1 \vdash Q : \theta_1 \quad \Pi_0 \not\frown \Pi_1}{\Pi_0, \Pi_1 \vdash [P, Q] : \theta_0 \times \theta_1} \Longrightarrow$$

$$\begin{aligned} &(\otimes) :: (Union\ \Pi_0\ \Pi_1\ \Pi_2, NonInterfering\ \Pi_0\ \Pi_1) \\ &\quad \Rightarrow Phrase\ \Pi_0\ P \to Phrase\ \Pi_1\ Q \to Phrase\ \Pi_2\ (Disjoint\ P\ Q) \\ &(Phrase\ P) \otimes (Phrase\ Q) = Phrase\ (Disjoint\ P\ Q) \end{aligned}$$

$$\frac{\vdash P : \theta_0 \longrightarrow \theta_1}{\vdash \mathbf{promote}\, P : \theta_0 \underset{P}{\longrightarrow} \theta_1} \Longrightarrow$$

$$\begin{aligned} &promote :: HasType\ (Context\ HNil)\ P\ (\theta_0 :\longrightarrow: \theta_1) \\ &\quad \Rightarrow Phrase\ (Context\ HNil)\ P \to Phrase\ (Context\ HNil)\ (Promote\ P) \\ &promote\ (Phrase\ P) = Phrase\ (Promote\ P) \end{aligned}$$

Syntax Rules 7: Phrase Constructors

5.2.1 Application

SCI prevents interference between a function and its argument. This is implemented in syntax rules 8 on the next page by adding the *NonInterference* constraint to the application constructor.

Meaningful recursion satisfies this reduction rule:

$$\mathbf{rec}(f) \Longrightarrow f(\mathbf{rec}(f))$$

But since the function f appears in both parts of the application on the right-hand side, it violates the application constraint $f\#\mathbf{rec}(f)$. If f has no free identifiers, it cannot interfere with itself. Passive functions have no free identifiers, therefore recursion is specified to be applicable only to passive functions. This is implemented in type definitions 11 by changing the type of the recursion constant.

$$\mathbf{rec} : (\theta \underset{P}{\rightarrow} \theta) \rightarrow \theta \quad \Longrightarrow \quad \mathbf{instance}\ \mathit{HasType}\ \Pi\ \mathit{Rec}\ ((\theta :\underset{P}{\rightarrow}: \theta) :\rightarrow: \theta)$$

Type Definitions 11: Recursion

$$\frac{\Pi_0 \vdash P : \theta_0 \to \theta_1 \quad \Pi_1 \vdash Q : \theta_0 \quad \Pi_0 \not\sim \Pi_1}{\Pi_0, \Pi_1 \vdash P\,Q : \theta_1} \quad \Longrightarrow$$

$$\begin{aligned}
&\mathit{appl} :: (\\
&\quad \mathit{Union}\ \Pi_0\ \Pi_1\ \Pi_2,\\
&\quad \mathit{HasType}\ \Pi_0\ P\ (\theta_0 :\!\!\rightarrow\!: \theta_1),\\
&\quad \mathit{HasType}\ \Pi_1\ Q\ \theta_0,\\
&\quad \mathit{NonInterfering}\ \Pi_0\ \Pi_1\\
&) \Rightarrow \mathit{Phrase}\ \Pi_0\ P \to \mathit{Phrase}\ \Pi_1\ Q \to \mathit{Phrase}\ \Pi_2\ (\mathit{Appl}\ P\ Q)\\
&\mathit{appl}\ (\mathit{Phrase}\ P)\ (\mathit{Phrase}\ Q) = \mathit{Phrase}\ (\mathit{Appl}\ P\ Q)
\end{aligned}$$

Syntax Rules 8: Application

5.3 Semantics

The semantic meanings of controlled product and passive functions are the same as interfering products and functions in PCF and Idealized ALGOL. This is implemented in semantic definitions 10 by recursively calling *eval* with the uncontrolled equivalent phrase.

$$[\![[P,Q]]\!]u = [\![\langle P,Q\rangle]\!]u \implies \begin{array}{l}\textbf{instance } \mathit{Eval}\ (\mathit{Pair}\ P\ Q)\ u\ (\theta_0,\theta_1)\\ \quad \Rightarrow \mathit{Eval}\ (\mathit{Disjoint}\ P\ Q)\ u\ (\theta_0,\theta_1)\ \textbf{where}\\ \quad \mathit{eval}\ (\mathit{Disjoint}\ P\ Q)\ u = \mathit{eval}\ (\mathit{Pair}\ P\ Q)\ u\end{array}$$

$$[\![\textbf{promote}\, P]\!]u = [\![P]\!]u \implies \begin{array}{l}\textbf{instance } \mathit{Eval}\ P\ u\ \theta \Rightarrow \mathit{Eval}\ (\mathit{Promote}\ P)\ u\ \theta\ \textbf{where}\\ \quad \mathit{eval}\ (\mathit{Promote}\ P)\ u = \mathit{eval}\ P\ u\end{array}$$

Semantic Definitions 10: Phrases

5.4 Parallelism

The control of interference makes it possible to give a coherent semantics for parallel composition of commands. SCI extends Idealized ALGOL with parallel composition, which is much like sequential composition at the syntactic level.

Listing 17 on page 97 shows the raw syntax, typing, syntactic constructor, and evaluation instance for the parallel composition constant. The syntax is trivial because the domain of the constant SCI-function is a non-interfering product.

Two non-interfering commands operate on disjoint parts of the state. They could be sequentially composed in either order with the same result state. Therefore, sequential composition is a semantically correct implementation of parallel composition.

(Interleaved implementation is also possible but outside the scope of this dissertation. Interleaving can be trivially extended to a multi-processor implementation using the Glasgow Parallel HASKELL extension to HASKELL98.)

```
‖ ⟹ data Pl = Pl

‖: (comm ⊗ comm) ⟶ comm ⟹ instance HasType Π Pl (Comm :⊗: Comm :⟶: Comm)

                                  pl :: Phrase (Context HNil) Pl
⊢‖: (comm ⊗ comm) ⟶ comm  ⟹⟹    pl = Phrase Pl

                                  instance Eval Pl u ((Command s, Command s)
⟦‖⟧⟨c0, c1⟩us = c0u(c1u(s))  ⟹⟹      ⟶ Command s) where
                                    eval _ _ = λ(c0, c1) ⟶ λs ⟶ c0 (c1 s)
```

Listing 17: Parallelism

5.5 Examples

5.5.1 Invalid Phrase

Consider the invalid SCI-like phrase below. The disjoint product that is the argument to parallel composition cannot be constructed because ι_2 is used in both parts. Even though interference does not occur in this particular phrase, SCI forbids it because it could occur in a similar phrase.

$$\iota_0 := \mathbf{deref}(\iota_2) \parallel \iota_1 := \mathbf{deref}(\iota_2) \Longrightarrow \begin{array}{r} \mathit{appl}\ \mathit{pl}\ ((\mathit{appl}\ \mathit{assign}\ (\iota_0 \times (\mathit{appl}\ \mathit{deref}\ \iota_2))) \\ \otimes\ (\mathit{appl}\ \mathit{assign}\ (\iota_1 \times (\mathit{appl}\ \mathit{deref}\ \iota_2)))) \end{array}$$

5.5.2 Valid Phrases

Parallelism

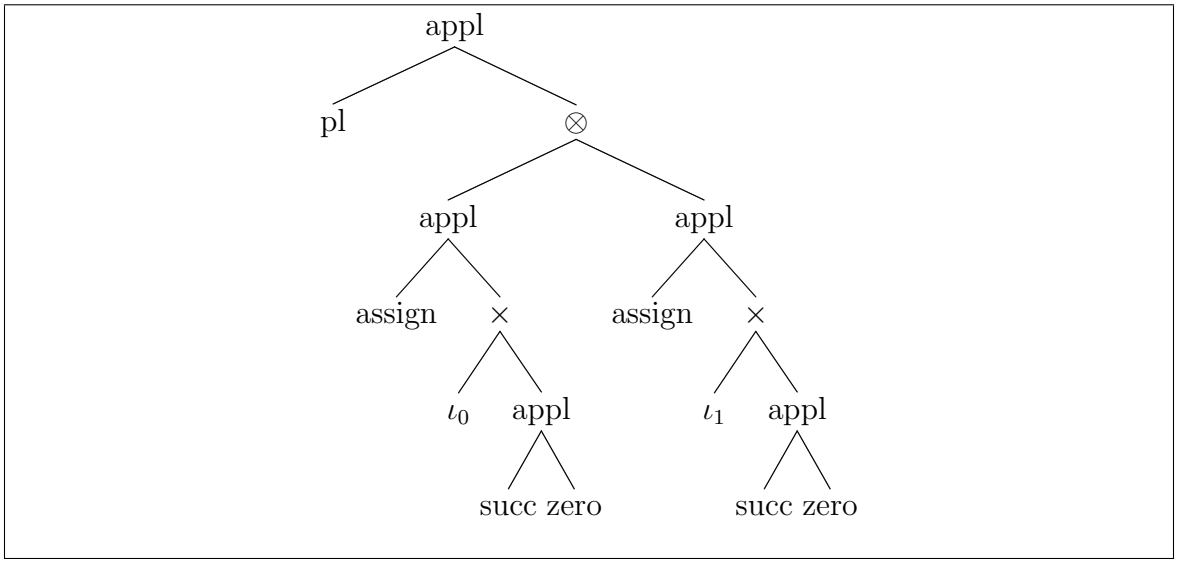
Consider a phrase that assigns 1 to ι_0 and ι_1 simultaneously. The traditional syntax is in derivation 3 on the next page and the HASKELL-code in listing 18 on page 100. The HASKELL-type of the HASKELL phrase is in listing 19 on page 101, and also parsed in parse tree 6 on page 101.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{[\iota_0] \quad \vdash \mathbf{succ}(0) : \mathbf{exp}[\mathbf{int}}}{\iota_0 : \mathbf{var}[\mathbf{int} \vdash \langle \iota_0, \mathbf{succ}(0)\rangle : \mathbf{var}[\mathbf{int}] \times \mathbf{exp}[\mathbf{int}}}{\iota_0 : \mathbf{var}[\mathbf{int} \vdash \iota_0 := \mathbf{succ}(0) : \mathbf{comm}} \qquad \dfrac{\vdots}{\iota_1 : \mathbf{var}[\mathbf{int} \vdash \iota_1 := \mathbf{succ}(0) : \mathbf{comm}}}{\iota_0, \iota_1 : \mathbf{var}[\mathbf{int} \vdash [\iota_0 := \mathbf{succ}(0), \iota_1 := \mathbf{succ}(0)] : \mathbf{comm} \otimes \mathbf{comm}}}{\iota_0, \iota_1 : \mathbf{var}[\mathbf{int} \vdash \iota_0 := \mathbf{succ}(0) \parallel \iota_1 := \mathbf{succ}(0) : \mathbf{comm}}$$

Derivation 3: SCI Syntax

```
appl pl (
        (appl assign (
                (ι0
                 ×
                appl succ
                     zero)))
         ⊗
        (appl assign (
                (ι1
                 ×
                appl succ
                     zero)))))
```

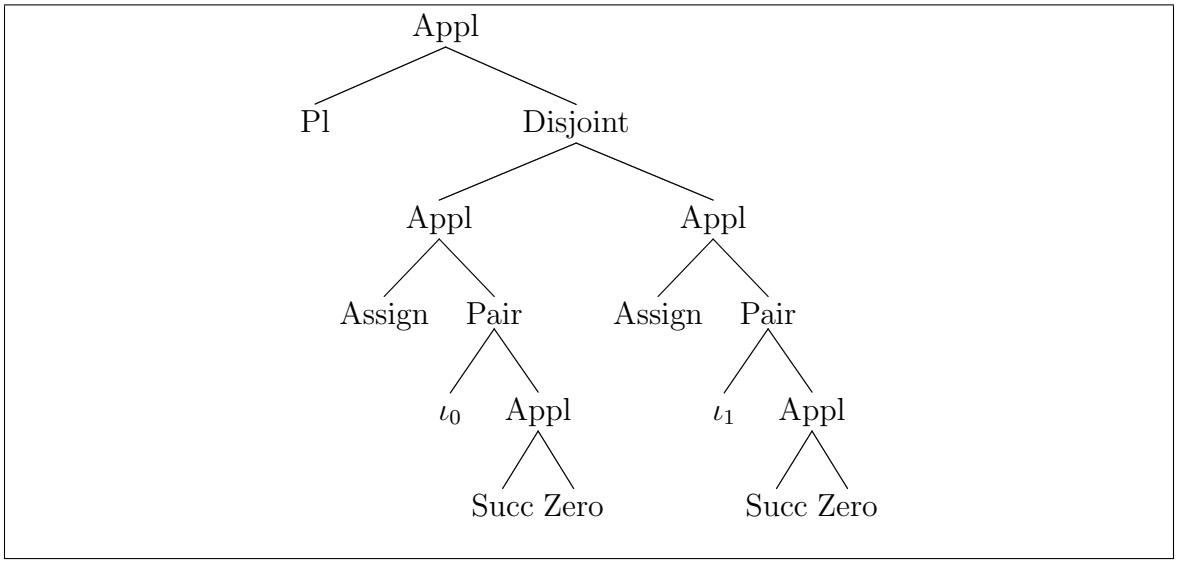
Listing 18: HASKELL Code



Parse Tree 5: HASKELL Syntax

```
Appl Pl
    (Disjoint (Appl Assign
                    (Pair (Identifier HZero)
                          (Appl Succ
                                Zero)))
              (Appl Assign
                    (Pair (Identifier (HSucc HZero))
                          (Appl Succ
                                Zero))))
```

Listing 19: HASKELL Type



Parse Tree 6: HASKELL Type

Recursion

In section 3.5, a function whose fixed point sums two integers was presented:

$$\lambda\iota_0.\lambda\iota_1.\lambda\iota_2.\mathbf{cond}(\mathbf{eq}(\iota_2, 0), \iota_1, \iota_0(\mathbf{succ}(\iota_1))(\mathbf{pred}(\iota_2)))$$

Recall that in PCF it was used like:

$$\mathbf{rec}(f)(\mathbf{succ}(0))(\mathbf{succ}(0))$$

The function is still usable in SCI but it must be promoted to a passive function in order to be used recursively:

$$\mathbf{rec}(\mathbf{promote}\ f)(\mathbf{succ}(0))(\mathbf{succ}(0))$$
$$\Longrightarrow \mathit{appl}\ (\mathit{appl}\ (\mathit{appl}\ \mathit{rec}\ (\mathit{promote}\ f))\ (\mathit{appl}\ \mathit{succ}\ \mathit{zero}))\ (\mathit{appl}\ \mathit{succ}\ \mathit{zero})$$

5.6 Validation

Since SCI is a variant of Idealized ALGOL, most of the unit tests in section 4.8 used to validate Idealized ALGOL pass for SCI. The exceptions are recursion on non-passive functions and application where the function and argument interfere. Phrases 1 – 3 in listing 13 on page 82 are valid in Idealized ALGOL but invalid in SCI, phrases 4 – 6 verify the phrase constructors for new features in SCI, and phrases 7 – 9 contain passive subphrases that make the whole phrase valid in SCIR.

The new features in SCI are unit tested by the examples in section 5.5.2 on page 98.

$$\mathbf{rec}(\lambda\iota_0.\mathbf{true}) \Longrightarrow \mathit{appl}\ \mathit{rec}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ \mathit{true}) \tag{5.1}$$
$$(\lambda\iota_0.\iota_1 := 0)(\iota_1) \Longrightarrow \mathit{appl}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ (\mathit{appl}\ \mathit{assign}\ (\iota_1 \times \mathit{zero})))\ (\iota_1) \tag{5.2}$$
$$(\lambda\iota_0.\iota_1)(\iota_1 := 0) \Longrightarrow \mathit{appl}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ \iota_1)\ (\mathit{appl}\ \mathit{assign}\ (\iota_1 \times \mathit{zero})) \tag{5.3}$$
$$\tag{5.4}$$
$$[\iota_0 := 0, \iota_0] \Longrightarrow \mathit{appl}\ \mathit{assign}\ (\iota_0 \times \mathit{zero}) \otimes \iota_0 \tag{5.5}$$
$$[\iota_0, \iota_0 := 0] \Longrightarrow \iota_0 \otimes \mathit{appl}\ \mathit{assign}\ (\iota_0 \times \mathit{zero}) \tag{5.6}$$
$$\mathbf{promote}(\lambda\iota_0.\iota_1 := 0) \Longrightarrow \mathit{promote}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ (\mathit{appl}\ \mathit{assign}\ (\iota_1 \times \mathit{zero}))) \tag{5.7}$$
$$\tag{5.8}$$
$$[\pi_0\langle 0, \iota_0 := 0\rangle, \iota_0] \Longrightarrow \pi_0\ (\mathit{zero} \times \mathit{appl}\ \mathit{assign}\ (\iota_0 \times \mathit{zero})) \otimes \iota_0 \tag{5.9}$$
$$[(\lambda\iota_0.\mathbf{true})(\iota_1 := 0), \iota_1] \Longrightarrow (\mathit{appl}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ \mathit{true})\ (\mathit{appl}\ \mathit{assign}\ (\iota_1 \times \mathit{zero}))) \otimes \iota_1 \tag{5.10}$$
$$[\pi_0\langle \iota_1, \iota_0 := 0\rangle, \iota_0] \Longrightarrow \pi_0\ (\iota_1 \times \mathit{appl}\ \mathit{assign}\ (\iota_0 \times \mathit{zero})) \otimes \iota_0 \tag{5.11}$$

Listing 20: Invalid phrases

5.7 Summary

SCI is a syntactic variant of Idealized ALGOL with a parallelism extension. The HASKELL definitional interpreter uses type constraints to control interference. Similar type constraints techniques are used in chapter 7 on page 118. To be a useful programming language, SCI needs a concept of passivity – a similar language with passivity is discussed in chapter 6 on the following page. Since SCI is more restrictive than Idealized ALGOL, some Idealized ALGOL unit tests should fail.

Chapter 6

Syntactic Control of Interference Revisited

The version of SCI presented in chapter 5 on page 87 uses conservative non-interference criteria for simplicity. This excludes a number of phrases where interference cannot actually occur because identifiers are used passively. The phenomenon of passive identifier occurrences was identified in [18] to permit some of these phrases.

The specific definition for passivity given in [18] was identified as deficient by Reynolds, so an alternative was proposed in [12] as the type system SCIR, which is discussed in section 6.1 on the following page. Unfortunately, SCIR has an ambiguous syntax that would be difficult to implement in a defintional interpreter. An equivalent type system that is easier to implement was defined in [4] as SCIR_K; implementation of the type system is discussed in section 6.2 on page 109.

6.1 Original SCIR

A subset of the types in SCI are designated passive (denoted ϕ), as defined in type definitions 12. These will be used to make specific syntax rules.

$$\theta ::= \mathbf{exp}[\tau] \mid \mathbf{comm} \mid \theta \times \theta' \mid \theta \to \theta' \mid \theta \otimes \theta' \mid \theta \underset{P}{\to} \theta'$$

$$\phi ::= \mathbf{exp}[\tau] \mid \phi \times \phi \mid \theta \to \phi \mid \phi \otimes \phi \mid \theta \underset{P}{\to} \theta'$$

Type Definitions 12: Types

SCIR's basic enhancement from SCI is to consider whether identifiers are used actively or passively. Passive identifiers cannot interfere with each other. To track whether identifiers are used passively within a specific phrase, the typing context is partitioned into a passive (left, Π) part and an active (right, Γ) part.

$$\Pi \mid \Gamma \vdash P : \theta$$

The partitioned typing context is managed by explicit structural syntax rules. These rules, listed in syntax rules 9 on the next page, manipulate the context without changing the syntax tree or phrase type. Identifiers can be "passified"[1] if the phrase type is passive, or activated in any context. The context can be weakened by adding extra identifiers to either part. Passive identifiers can be substituted for other passive identifiers of the same type, contracting the context.

[1] A neologism coined in [12].

$$\text{Passification}\frac{\Pi \mid \Gamma, \iota : \theta \vdash P : \phi}{\Pi, \iota : \theta \mid \Gamma \vdash P : \phi}$$

$$\text{Activation}\frac{\Pi, \iota : \theta_0 \mid \Gamma \vdash P : \theta_1}{\Pi \mid \Gamma, \iota : \theta_0 \vdash P : \theta_1}$$

$$\text{Weakening}\frac{\Pi_0 \mid \Gamma_0 \vdash P : \theta}{\Pi_0 \cup \Pi_1 \mid \Gamma_0 \cup \Gamma_1 \vdash P : \theta}$$

$$\text{Contraction}\frac{\Pi, \iota_0 : \theta_0, \iota_1 : \theta_0 \mid \Gamma \vdash P : \theta_1}{\Pi, \iota_0 : \theta_0 \mid \Gamma \vdash [P](\iota_1 \mapsto \iota_0) : \theta_1}$$

Syntax Rules 9: Structural Rules

The type constructors that are simple extensions from SCI are listed in syntax rules 10 on the following page. Note that identifiers are introduced in the active part. Identifiers are also abstracted out of the active part, the activation rule makes this a trivial constraint. Contexts of a pair must be weakened until they are the same.

Rules that deal with interference, listed in syntax rules 11 on the next page, must be significantly altered from SCI. Function promotion requires that only the active part of the context be empty. Disjoint-product introduction and application require that the contexts be wholly disjoint; passive identifiers that are shared between both parts of the phrases are aliased in one half and then contracted in the combined context.

$$\frac{}{\mid \iota : \theta \vdash \iota : \theta}$$

$$\frac{\Pi \mid \Gamma, \iota : \theta_0 \vdash P : \theta_1}{\Pi \mid \Gamma \vdash \lambda\iota : \theta_0.P : \theta_0 \to \theta_1}$$

$$\frac{\Pi \mid \Gamma \vdash P : \theta_0 \qquad \Pi \mid \Gamma \vdash Q : \theta_1}{\Pi \mid \Gamma \vdash \langle P, Q\rangle : \theta_0 \times \theta_1}$$

$$\frac{\Pi \mid \Gamma \vdash P : \theta_0 \times \theta_1}{\Pi \mid \Gamma \vdash \pi_0 P : \theta_0}$$

Syntax Rules 10: Trivial Type Constructors

$$\frac{\Pi \mid\vdash P : \theta_0 \to \theta_1}{\Pi \mid\vdash \textbf{promote } P : \theta_0 \underset{P}{\to} \theta_1}$$

$$\frac{\Pi_0 \mid \Gamma_0 \vdash P : \theta_0 \qquad \Pi_1 \mid \Gamma_1 \vdash Q : \theta_1}{\Pi_0 \cup \Pi_1 \mid \Gamma_0 \cup \Gamma_1 \vdash [P, Q] : \theta_0 \otimes \theta_1} \Pi_0 \cap \Pi_1 = \emptyset, \Pi_0 \cap \Gamma_1 = \emptyset, \Pi_1 \cap \Gamma_0 = \emptyset$$

$$\frac{\Pi_0 \mid \Gamma_0 \vdash P : \theta_0 \to \theta_1 \qquad \Pi_1 \mid \Gamma_1 \vdash Q : \theta_0}{\Pi_0 \cup \Pi_1 \mid \Gamma_0 \cup \Gamma_1 \vdash P(Q) : \theta_1} \Pi_0 \cap \Pi_1 = \emptyset, \Pi_0 \cap \Gamma_1 = \emptyset, \Pi_1 \cap \Gamma_0 = \emptyset$$

Syntax Rules 11: Interference Type Constructors

6.2 SCIR Without Ambiguity

The implicit structural rules cause SCIR to have an ambiguous grammar: the same phrase can have multiple valid inference trees. For example, listing 21 on the following page shows two inferences for the same phrase: passification can be done before disjoint introduction, or passification with contraction can be done after.

Direct implementation of an ambiguous grammar requires non-determinism. This is beyond the capabilities of the HASKELL type system. A bottom-up variant of SCIR does not use passification and contraction, and is therefore unambiguous and implementable. Such a type system was presented as SCIR_K in [4]: types are divided into passive and active kinds and then kind-constraints are added to identifier typings and phrase types. [4] showed equivalence between SCIR and SCIR_K.

To define the kinds, notation is abused to define a passive-type predicate:

$$\begin{aligned}
\phi(\mathbf{exp}[\tau]) &= \text{true} \\
\phi(\mathbf{comm}) &= \text{false} \\
\phi(\theta \times \theta') &= \phi(\theta) \wedge \phi(\theta') \\
\phi(\theta \to \theta') &= \phi(\theta') \\
\phi(\theta \otimes \theta') &= \phi(\theta) \wedge \phi(\theta') \\
\phi(\theta \underset{P}{\to} \theta') &= \text{true}
\end{aligned}$$

The effect of contraction, passification, and other structural rules is achieved by kind-constraints. A type context is a mapping from an identifier to a type combined with Booleans signifying passifiability (α) and contractibility (β). In the original

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mid\vdash \mathbf{deref} : \mathbf{var}[\theta] \to \mathbf{exp}[\theta]} \quad \dfrac{}{\mid \iota : \mathbf{var}[\theta] \vdash \iota : \mathbf{var}[\theta]}}{\mid \iota : \mathbf{var}[\theta] \vdash \mathbf{deref}\ \iota : \mathbf{exp}[\theta]} \quad \vdots \quad \dfrac{}{\iota : \mathbf{var}[\theta] \mid\vdash \mathbf{deref}\ \iota : \mathbf{exp}[\theta]}}{\iota : \mathbf{var}[\theta] \mid\vdash \mathbf{deref}\ \iota : \mathbf{exp}[\theta]}}{\iota : \mathbf{var}[\theta] \mid\vdash [\mathbf{deref}\ \iota, \mathbf{deref}\ \iota] : \mathbf{exp}[\theta] \otimes \mathbf{exp}[\theta]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mid\vdash \mathbf{deref} : \mathbf{var}[\theta] \to \mathbf{exp}[\theta]} \quad \dfrac{}{\mid \iota_0 : \mathbf{var}[\theta] \vdash \iota_0 : \mathbf{var}[\theta]}}{\mid \iota_0 : \mathbf{var}[\theta] \vdash \mathbf{deref}\ \iota_0 : \mathbf{exp}[\theta]} \quad \vdots \quad \dfrac{}{\mid \iota_1 : \mathbf{var}[\theta] \vdash \mathbf{deref}\ \iota_1 : \mathbf{exp}[\theta]}}{\mid \iota_0, \iota_1 : \mathbf{var}[\theta] \vdash [\mathbf{deref}\ \iota_0, \mathbf{deref}\ \iota_1] : \mathbf{exp}[\theta] \otimes \mathbf{exp}[\theta]}}{\iota_0, \iota_1 : \mathbf{var}[\theta] \mid\vdash [\mathbf{deref}\ \iota_0, \mathbf{deref}\ \iota_1] : \mathbf{exp}[\theta] \otimes \mathbf{exp}[\theta]}}{\iota : \mathbf{var}[\theta] \mid\vdash [\mathbf{deref}\ \iota, \mathbf{deref}\ \iota] : \mathbf{exp}[\theta] \otimes \mathbf{exp}[\theta]}$$

Listing 21: Ambiguous derivations

formulation given in [4], a phrase also had a "global" constraint signifying its well-formedness – that syntax has been rewritten here to disallow non-well-formed phrases.

$$\Pi(\iota) = (\theta, \alpha, \beta)$$

$$\Pi \vdash P : \theta$$

When a rule combines two type contexts, identifiers that are mapped in both are required to map to the same type. Constraints follow no such restriction, so to specify how contexts are combined it is more concise to use a notation that aggregates type mappings.

$$\vec{\iota} : (\vec{\theta}, \vec{\alpha}, \vec{\beta}) = \iota_0 : (\theta_0, \alpha_0, \beta_0), \ldots, \iota_n : (\theta_n, \alpha_n, \beta_n)$$

For example:

$$\vec{\iota} : (\vec{\theta}, \vec{\alpha} \wedge \vec{\gamma}, \vec{\beta} \vee \text{true}) = \iota_0 : (\theta_0, \alpha_0 \wedge \gamma_0, \beta_0 \vee \text{true}), \ldots, \iota_n : (\theta_n, \alpha_n \wedge \gamma_n, \beta_n \vee \text{true})$$

Type constructors for SCIR_K are listed in syntax rules 12 to 13 on pages 112–113. Identifiers are introduced as contractible, and passifiable if mapped to a passive type. If an identifier is abstracted, it must be contractible (unless the identifier is not used). Product introduction simply conjoins all constraints. Projection satisfies all identifier constraints if the projected part is passive. All identifiers in context must be passifiable to promote a function. If combined into a disjoint product, the contractibility of identifiers is determined by their passifiability. Finally, function application conjoins constraints, although all identifier constraints are satisfied if the resulting phrase is passive.

$$\frac{}{\iota : (\theta, \phi(\theta), \text{true}) \vdash \iota : \theta}$$

$$\frac{\Pi, \iota : (\theta_0, \alpha, \text{true}) \vdash P : \theta_1}{\Pi \vdash \lambda\iota : \theta_0.P : \theta_0 \to \theta_1}$$

$$\frac{\Pi \vdash P : \theta_1}{\Pi \vdash \lambda\iota : \theta_0.P : \theta_0 \to \theta_1} \iota \notin \text{dom}(\Pi)$$

$$\frac{\vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_0, \vec{\beta}_0), \Pi_0 \vdash P : \theta_0 \qquad \vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_1, \vec{\beta}_1), \Pi_1 \vdash Q : \theta_1}{\vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_0 \wedge \vec{\alpha}_1, \vec{\beta}_0 \wedge \vec{\beta}_1), \Pi_0, \Pi_1 \vdash \langle P, Q\rangle : \theta_0 \times \theta_1}$$

$$\frac{\vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_0, \vec{\beta}_0) \vdash P : \theta_0 \times \theta_1}{\vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_0 \vee \phi(\theta_0), \vec{\beta}_0 \vee \phi(\theta_0)) \vdash \pi_0 P : \theta_0}$$

Syntax Rules 12: Trivial Type Constructors for Simplified SCIR$_K$

$$\frac{\vec{\iota} : (\vec{\theta}, \mathrm{true}, \vec{\beta}) \vdash P : \theta_0 \to \theta_1}{\vec{\iota} : (\vec{\theta}, \mathrm{true}, \mathrm{true}) \vdash \mathbf{promote}\, P : \theta_0 \underset{P}{\longrightarrow} \theta_1}$$

$$\frac{\vec{\iota} : (\vec{\theta}, \vec{\alpha}_0, \vec{\beta}_0), \Pi_0 \vdash P : \theta_0 \quad \vec{\iota} : (\vec{\theta}, \vec{\alpha}_1, \vec{\beta}_1), \Pi_1 \vdash Q : \theta_1}{\vec{\iota} : (\vec{\theta}, \vec{\alpha}_0 \wedge \vec{\alpha}_1, \vec{\alpha}_0 \wedge \vec{\alpha}_1), \Pi_0, \Pi_1 \vdash [P, Q] : \theta_0 \otimes \theta_1}$$

$$\frac{\vec{\iota} : (\vec{\theta}, \vec{\alpha}_0, \vec{\beta}_0), \vec{\iota}_P : (\vec{\theta}_P, \vec{\alpha}_P, \vec{\beta}_P) \vdash P : \theta_0 \to \theta_1 \quad \vec{\iota} : (\vec{\theta}, \vec{\alpha}_1, \vec{\beta}_1), \vec{\iota}_Q : (\vec{\theta}_Q, \vec{\alpha}_Q, \vec{\beta}_Q) \vdash Q : \theta_0}{\begin{array}{l} \vec{\iota} : (\vec{\theta}, \vec{\alpha}_0 \wedge \vec{\alpha}_1 \vee \phi(\theta_1), \vec{\alpha}_0 \wedge \vec{\alpha}_1 \vee \phi(\theta_1)), \\ \vec{\iota}_P : (\vec{\theta}_P, \vec{\alpha}_P \vee \phi(\theta_1), \vec{\alpha}_P \vee \phi(\theta_1)), \\ \vec{\iota}_Q : (\vec{\theta}_Q, \vec{\alpha}_Q \vee \phi(\theta_1), \vec{\alpha}_Q \vee \phi(\theta_1)) \end{array} \vdash P(Q) : \theta_1}$$

Syntax Rules 13: Interference Type Constructors for Simplified SCIR$_K$

6.2.1 Example

Consider the phrase from section 5.5.1 that is invalid in SCI:

$$\iota_0 := \mathbf{deref}(\iota_2) \parallel \iota_1 := \mathbf{deref}(\iota_2)$$

The SCIR syntax for this program is derived in derivation 4 on the next page, while derivation 5 on page 116 shows an exerpt of the derviation of the SCIR_K syntax (non-interesting phrase types are ommitted for brevity).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mid \iota_2 : \mathbf{var}[\tau] \vdash \iota_2 : \mathbf{var}[\tau]}}{\mid \iota_2 : \mathbf{var}[\tau] \vdash \mathbf{deref}(\iota_2) : \mathbf{exp}[\tau]} \qquad \dfrac{}{\mid \iota_0 : \mathbf{var}[\tau] \vdash \iota_0}}{\iota_2 : \mathbf{var}[\tau] \mid \iota_0 : \mathbf{var}[\tau] \vdash \langle \iota_0, \mathbf{deref}(\iota_2) \rangle : \mathbf{var}[\tau] \times \mathbf{exp}[\tau]} \qquad \vdots}{\iota_2 : \mathbf{var}[\tau] \mid \iota_0 : \mathbf{var}[\tau] \vdash \iota_0 := \mathbf{deref}(\iota_2) : \mathbf{comm} \qquad \iota_2 : \mathbf{var}[\tau] \mid \iota_1 : \mathbf{var}[\tau] \vdash \iota_1 := \mathbf{deref}(\iota_2) : \mathbf{comm}}}{\iota_2 : \mathbf{var}[\tau] \mid \iota_0, \iota_1 : \mathbf{var}[\tau] \vdash [\iota_0 := \mathbf{deref}(\iota_2), \iota_1 := \mathbf{deref}(\iota_2)] : \mathbf{comm} \otimes \mathbf{comm}}}{\iota_2 : \mathbf{var}[\tau] \mid \iota_0, \iota_1 : \mathbf{var}[\tau] \vdash \iota_0 := \mathbf{deref}(\iota_2)! \mid \iota_1 := \mathbf{deref}(\iota_2) : \mathbf{comm}}$$

Derivation 4: SCIR Syntax

$$\dfrac{\dfrac{\dfrac{}{\iota_2 : (\mathbf{var}[\tau], \mathrm{false}, \mathrm{true}) \vdash \iota_2 : \mathbf{var}[\tau]}}{\vdots \quad \iota_2 : (\mathbf{var}[\tau], \mathrm{true}, \mathrm{true}) \vdash \mathbf{deref}(\iota_2) : \mathbf{exp}[\tau]} \qquad \vdots}{\dfrac{\iota_0 : (\mathbf{var}[\tau], \mathrm{false}, \mathrm{true}),\ \iota_2 : (\mathbf{var}[\tau], \mathrm{true}, \mathrm{true}) \vdash \langle \iota_0, \mathbf{deref}(\iota_2) \rangle}{\dfrac{\iota_0 : (\mathbf{var}[\tau], \mathrm{false}, \mathrm{true}),\ \iota_2 : (\mathbf{var}[\tau], \mathrm{true}, \mathrm{true}) \vdash \iota_0 := \mathbf{deref}(\iota_2)}{\dfrac{\iota_0, \iota_1 : (\mathbf{var}[\tau], \mathrm{false}, \mathrm{true}),\ \iota_2 : (\mathbf{var}[\tau], \mathrm{true}, \mathrm{true}) \vdash [\iota_0 := \mathbf{deref}(\iota_2), \iota_1 := \mathbf{deref}(\iota_2)]}{\iota_0, \iota_1 : (\mathbf{var}[\tau], \mathrm{false}, \mathrm{true}),\ \iota_2 : (\mathbf{var}[\tau], \mathrm{true}, \mathrm{true}) \vdash \iota_0 := \mathbf{deref}(\iota_2) \parallel \iota_1 := \mathbf{deref}(\iota_2) : \mathbf{comm}}}}} \qquad \vdots \qquad \dfrac{\iota_1 : (\mathbf{var}[\tau], \mathrm{false}, \mathrm{true}),\ \iota_2 : (\mathbf{var}[\tau], \mathrm{true}, \mathrm{true}) \vdash \iota_1 := \mathbf{deref}(\iota_2)}{}$$

Derivation 5: SCIR_K Syntax

6.3 Summary

To be useful for programming, the type system from chapter 5 on page 87 requires an extension for passivity. SCIR is a similar type system that recognises passive identifier occurrences. SCIR_K is an equivalent reformulation of SCIR with a non-ambiguous type system. Chapter 7 on the following page shows how to implement SCIR_K as a definitional interpreter.

Chapter 7

Implementation of SCIR_K in Haskell

SCIR_K is a syntactic variant of Idealized ALGOL that syntactically controls interference while permitting passive identifier occurrences. It is inspired by SCI, which was partially presented in chapter 5 on page 87. Good criteria for passivity were introduced by SCIR, as detailed in section 6.1 on page 106. SCIR's ambiguous syntax makes it difficult to implement directly, so an equivalent type system was formulated in [4] and called SCIR_K. Details of SCIR_K's type system are presented in section 6.2 on page 109. This chapter defines a definitional interpreter in HASKELL for SCIR_K.

The passivity criteria of SCIR_K are implemented as a HASKELL type constraint in section 7.1 on the following page. Implementation of SCIR_K rests around redefining the HASKELL-type that represents SCIR_K-phrases to include passivity 'annotations', this is presented in section 7.2 on page 121 along with type classes for manipulation of the annotations. The phrase constructors from sections 3.1 on page 29 and 5.2 on page 88 are redefined with type constraints that implement control of interference with

passivity in section 7.3 on page 124. An example demonstrates how these redefinitions make SCIR$_K$ more expressive than SCI, the extent of which is verified using unit tests on phrase validity.

7.1 Passivity

SCIR$_K$ extends SCI's concept of passive functions to include a subset of all passive phrase-types. Of the basic types, expressions are passive, commands are not. Compound types are passive depending on the passivity of their parts. This is implemented in type definitions 13 on the next page as a type constraint *IsPassive*. Rather than being a traditional type class, *IsPassive* is a functional relation from types to HList Booleans – this permits *IsPassive* to be applied to multiple elements of a type and the results combined using HList Boolean functions.

$$
\begin{array}{rcl}
\phi : \theta \to \text{boolean} & \Rrightarrow & \textbf{class } \mathit{IsPassive}\ \theta\ b \mid \theta \to b \\
\\
\phi(\textbf{exp}[\tau]) = \text{true} & \Rrightarrow & \textbf{instance } \mathit{IsPassive}\ (\mathit{Exp}\ \tau)\ \mathit{HTrue} \\
\phi(\textbf{comm}) = \text{false} & \Rrightarrow & \textbf{instance } \mathit{IsPassive}\ \mathit{Comm}\ \mathit{HFalse} \\
\\
\phi(\theta_0 \times \theta_1) = \phi(\theta_0) \wedge \phi(\theta_1) & \Rrightarrow & \textbf{instance } (\mathit{IsPassive}\ \theta_0\ b_0, \mathit{IsPassive}\ \theta_1\ b_1, \mathit{HAnd}\ b_0\ b_1\ b_2) \\
 & & \quad \Rightarrow \mathit{IsPassive}\ (\theta_0 :\times: \theta_1)\ b_2 \\
\\
\phi(\theta_0 \otimes \theta_1) = \phi(\theta_0) \wedge \phi(\theta_1) & \Rrightarrow & \textbf{instance } (\mathit{IsPassive}\ \theta_0\ b_0, \mathit{IsPassive}\ \theta_1\ b_1, \mathit{HAnd}\ b_0\ b_1\ b_2) \\
 & & \quad \Rightarrow \mathit{IsPassive}\ (\theta_0 :\otimes: \theta_1)\ b_2 \\
\\
\phi(\theta_0 \to \theta_1) = \phi(\theta_1) & \Rrightarrow & \textbf{instance } \mathit{IsPassive}\ \theta_1\ b \Rightarrow \mathit{IsPassive}\ (\theta_0 :\to: \theta_1)\ b \\
\phi(\theta_0 \underset{P}{\to} \theta_1) = \text{true} & \Rrightarrow & \textbf{instance } \mathit{IsPassive}\ (\theta_0 :\underset{P}{\to}: \theta_1)\ \mathit{HTrue}
\end{array}
$$

Type Definitions 13: Passive Types

7.2 Phrase Syntax

In SCIR$_K$, each identifier in the context is mapped to passifiability and contractability constraints. This means that the context is a mapping from identifiers to types and two Booleans. It is easier to implement this as three mappings: from identifiers to types (Π), identifiers to passifiability status (α), and identifiers to contractability status (β).

$$\vec{\iota} : (\vec{\theta}, \vec{\alpha}, \vec{\beta}) \vdash P : \theta' \quad \Longrightarrow \quad \textbf{data}\ \mathit{Phrase}\ \Pi\ P\ \alpha\ \beta = \mathit{Phrase}\ P$$

The additional mappings are implemented as HList Records.

$$\textbf{type}\ \mathit{Predicate}\ p = \mathit{Record}\ p$$

To manipulate these mappings, it is convenient to have HASKELL-constraints that deal with the complete records. Definition 2 on the following page defines a HASKELL-constraint to specify that all SCIR$_K$-constraints must be true, and to disjoin each with a given value. Conjunction is defined in definition 3 on page 123.

$$
\begin{aligned}
&\textbf{class } \mathit{AllTrue}\ v \\
&\vec{\iota} : (\vec{\theta}, \ldots \text{true} \ldots) \Longrightarrow \textbf{instance } \mathit{AllTrue}\ \mathit{HNil} \\
&\qquad\qquad \textbf{instance } \mathit{AllTrue}\ t \Rightarrow \mathit{AllTrue}\ (\mathit{HCons}\ (\mathit{F}\ l\ \mathit{HTrue})\ t) \\
&\\
&\qquad\qquad \textbf{class } \mathit{EachOr}\ v_0\ b\ v_1 \mid v_0\ b \rightarrow v_1 \\
&\vec{\iota} : (\vec{\theta}, \ldots \vee b \ldots) \Longrightarrow \textbf{instance } \mathit{EachOr}\ \mathit{HNil}\ b\ \mathit{HNil} \\
&\qquad\qquad \textbf{instance } (\mathit{HOr}\ b_0\ b_1\ b_2, \mathit{EachOr}\ t0\ b_1\ t1) \\
&\qquad\qquad\qquad \Rightarrow \mathit{EachOr}\ (\mathit{HCons}\ (\mathit{F}\ l\ b_0)\ t0)\ b_1\ (\mathit{HCons}\ (\mathit{F}\ l\ b_2)\ t1)
\end{aligned}
$$

Definition 2: Constraint vectors predicates

$$
\begin{array}{l}
\textbf{class } \mathit{UnionAnd}\ \Pi_0\ \Pi_1\ \alpha_0\ \alpha_1\ \alpha_2 \mid \Pi_0\ \Pi_1\ \alpha_0\ \alpha_1 \to \alpha_2 \\
\textbf{instance } (\\
\quad \mathit{RecordLabels}\ \Pi_0\ p_0, \\
\quad \mathit{RecordLabels}\ \Pi_1\ p_1, \\
\quad \mathit{HTIntersect}\ p_0\ p_1\ p_2, \\
\quad \mathit{UnionAnd'}\ p_2\ \alpha_0\ \alpha_1\ \alpha_2 \\
) \Rightarrow \mathit{UnionAnd}\ (\mathit{Context}\ \Pi_0)\ (\mathit{Context}\ \Pi_1) \\
\quad (\mathit{Predicate}\ \alpha_0)\ (\mathit{Predicate}\ \alpha_1)\ (\mathit{Predicate}\ \alpha_2) \\
\\
\vec{\iota} : (\vec{\theta}, \vec{\alpha}_0 \wedge \vec{\alpha}_1, \vec{\beta}_0 \wedge \vec{\beta}_1) \quad \Rrightarrow \quad \textbf{class } \mathit{UnionAnd'}\ a\ \alpha_0\ \alpha_1\ \alpha_2 \mid a\ \alpha_0\ \alpha_1 \to \alpha_2 \\
\textbf{instance } \mathit{UnionAnd'}\ \mathit{HNil}\ \alpha_0\ \alpha_1\ \mathit{HNil} \\
\textbf{instance } (\\
\quad \mathit{HasField}\ h\ \alpha_0\ b_0, \\
\quad \mathit{HasField}\ h\ \alpha_1\ b_1, \\
\quad \mathit{HAnd}\ b_0\ b_1\ b_2, \\
\quad \mathit{UnionAnd'}\ t\ \alpha_0\ \alpha_1\ \alpha_2 \\
) \Rightarrow \mathit{UnionAnd'}\ (\mathit{HCons}\ h\ t)\ \alpha_0\ \alpha_1\ (\mathit{HCons}\ (F\ h\ b_2)\ \alpha_2)
\end{array}
$$

Definition 3: Constraint vector conjunction

7.3 Syntax Rules

In syntax rules 14 on the following page, identifier introduction expresses an equivalence between *IsPassive* and the passification predicate. Potential passivity is disjuncted into each annotation on projection. Promotion sets all contractibile constraints to true.

Both abstraction rules are implemented by one HASKELL-function in syntax rules 15 on page 126 by using the *IfHasField* constraint and projecting out the abstracted identifier, whether it is in the records or not.

Note the constraint *HLeftUnion* α_2 β_2 β_3 for disjoint product introduction in syntax rules 16 on page 127: this rule specifies that contractibility is dependent on passifiability for disjoint products. But for possibly interfering products, contractibility is simply a conjunction of the contractibility of its parts, necessitating the addition of *UnionAnd* Π_0 Π_1 β_0 β_1 β_2.

The complex application rule is implemented in syntax rules 17 on page 128. Note that α_2 consists of the identifiers that appear in both phrases, α_3 is those that appear in either, α_4 is a union of these two, and then α_5 has the disjuncted potential for passivity. (Similar usage exists for βs except that contractibility depends on passifiability.)

$$\frac{}{\iota : (\theta, \phi(\theta), \text{true}) \vdash \iota : \theta} \implies$$

```
ident :: IsPassive θ α
  ⇒ Identifier n
    → Phrase
      (Context (HCons (F (Identifier n) θ) HNil))
      (Identifier n)
      (Predicate (HCons (F (Identifier n) α) HNil))
      (Predicate (HCons (F (Identifier n) HTrue) HNil))
ident ι = Phrase ι
```

$$\frac{\vec{\iota} : (\vec{\theta}, \vec{\alpha}_0, \vec{\beta}_0) \vdash P : \theta_0 \times \theta_1}{\vec{\iota} : (\vec{\theta}, \vec{\alpha}_0 \vee \phi(\theta_0), \vec{\beta}_0 \vee \phi(\theta_0)) \vdash \pi_0 P : \theta_0} \implies$$

```
π0 :: (
      HasType Π0 P (θ0 :×: θ1),
      IsPassive θ0 b,
      EachOr α0 b α1,
      EachOr β0 b β1
  ) ⇒ Phrase Π0 P (Predicate α0) (Predicate β0)
      → Phrase Π0 (π0 P) (Predicate α1) (Predicate β1)
π0 (Phrase P) = Phrase (π0 P)
```

$$\frac{\vec{\iota} : (\vec{\theta}, \text{true}, \vec{\beta}) \vdash P : \theta_0 \underset{P}{\rightarrow} \theta_1}{\vec{\iota} : (\vec{\theta}, \text{true}, \text{true}) \vdash \textbf{promote}\ P : \theta_0 \rightarrow \theta_1} \implies$$

```
promote :: (
      HasType Π P (θ0 :→: θ1),
      AllTrue α0,
      EachOr β0 HTrue β1
      ) ⇒ Phrase Π P (Predicate α0) (Predicate β0)
      → Phrase Π (Promote P) (Predicate α0) (Predicate β1)
  promote (Phrase P) = Phrase (Promote P)
```

Syntax Rules 14: Unary Rules

$$\frac{\Pi, \iota : (\theta_0, \alpha, \text{true}) \vdash P : \theta_1}{\Pi \vdash \lambda\iota : \theta_0 . P : \theta_0 \to \theta_1} \qquad \frac{\Pi \vdash P : \theta_1}{\Pi \vdash \lambda\iota : \theta_0 . P : \theta_0 \to \theta_1} \; \iota \notin \text{dom}(\Pi)$$

$$\Longrightarrow$$

$$\begin{aligned}
&\mathit{lambda} :: (\\
&\quad \mathit{H2ProjectByLabels}\ (\mathit{HCons}\ \iota\ \mathit{HNil})\ \Pi_0\ z_0\ \Pi_1,\\
&\quad \mathit{HasType}\ (\mathit{Context}\ \Pi_0)\ \iota\ \theta,\\
&\quad \mathit{H2ProjectByLabels}\ (\mathit{HCons}\ \iota\ \mathit{HNil})\ \alpha_0\ z_1\ \alpha_1,\\
&\quad \mathit{H2ProjectByLabels}\ (\mathit{HCons}\ \iota\ \mathit{HNil})\ \beta_0\ z_2\ \beta_1,\\
&\quad \mathit{IfHasField}\ \iota\ (\mathit{Predicate}\ \beta_0)\ \mathit{HTrue}\\
&) \Rightarrow \iota\\
&\quad \to \theta\\
&\quad \to \mathit{Phrase}\ (\mathit{Context}\ \Pi_0)\ P\ (\mathit{Predicate}\ \alpha_0)\ (\mathit{Predicate}\ \beta_0)\\
&\quad \to \mathit{Phrase}\ (\mathit{Context}\ \Pi_1)\ (\mathit{Lambda}\ \iota\ \theta\ P)\ (\mathit{Predicate}\ \alpha_1)\ (\mathit{Predicate}\ \beta_1)\\
&\mathit{lambda}\ \iota\ (\theta :: \theta)\ (\mathit{Phrase}\ P) = \mathit{Phrase}\ (\mathit{Lambda}\ \iota\ P)
\end{aligned}$$

Syntax Rules 15: Abstraction

$$\frac{\vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_0, \vec{\beta}_0), \Pi_0 \vdash P : \theta_0 \qquad \vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_1, \vec{\beta}_1), \Pi_1 \vdash Q : \theta_1}{\vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_0 \wedge \vec{\alpha}_1, \vec{\beta}_0 \wedge \vec{\beta}_1), \Pi_0, \Pi_1 \vdash \langle P, Q \rangle : \theta_0 \times \theta_1} \Longrightarrow$$

$$\begin{aligned}
&(\times) :: (\\
&\quad \mathit{Union}\ \Pi_0\ \Pi_1\ \Pi_2,\\
&\quad \mathit{UnionAnd}\ \Pi_0\ \Pi_1\ \alpha_0\ \alpha_1\ \alpha_2,\\
&\quad \mathit{UnionAnd}\ \Pi_0\ \Pi_1\ \beta_0\ \beta_1\ \beta_2,\\
&\quad \mathit{HLeftUnion}\ \alpha_0\ \alpha_1\ \alpha_3,\\
&\quad \mathit{HLeftUnion}\ \beta_0\ \beta_1\ \beta_3,\\
&\quad \mathit{HLeftUnion}\ \alpha_2\ \alpha_3\ \alpha_4,\\
&\quad \mathit{HLeftUnion}\ \beta_2\ \beta_3\ \beta_4\\
&) \Rightarrow \mathit{Phrase}\ \Pi_0\ P\ \alpha_0\ \beta_0\\
&\quad \rightarrow \mathit{Phrase}\ \Pi_1\ Q\ \alpha_1\ \beta_1\\
&\quad \rightarrow \mathit{Phrase}\ \Pi_2\ (\mathit{Pair}\ P\ Q)\ \alpha_4\ \beta_4\\
&(\mathit{Phrase}\ P) \times (\mathit{Phrase}\ Q) = \mathit{Phrase}\ (\mathit{Pair}\ P\ Q)
\end{aligned}$$

$$\frac{\vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_0, \vec{\beta}_0), \Pi_0 \vdash P : \theta_0 \qquad \vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_1, \vec{\beta}_1), \Pi_1 \vdash Q : \theta_1}{\vec{\iota}\colon (\vec{\theta}, \vec{\alpha}_0 \wedge \vec{\alpha}_1, \vec{\alpha}_0 \wedge \vec{\alpha}_1), \Pi_0, \Pi_1 \vdash [P, Q] : \theta_0 \otimes \theta_1} \Longrightarrow$$

$$\begin{aligned}
&(\otimes) :: (\\
&\quad \mathit{Union}\ \Pi_0\ \Pi_1\ \Pi_2,\\
&\quad \mathit{UnionAnd}\ \Pi_0\ \Pi_1\ \alpha_0\ \alpha_1\ \alpha_2,\\
&\quad \mathit{HLeftUnion}\ \alpha_0\ \alpha_1\ \alpha_3,\\
&\quad \mathit{HLeftUnion}\ \beta_0\ \beta_1\ \beta_2,\\
&\quad \mathit{HLeftUnion}\ \alpha_2\ \alpha_3\ \alpha_4,\\
&\quad \mathit{HLeftUnion}\ \alpha_2\ \beta_2\ \beta_3\\
&) \Rightarrow \mathit{Phrase}\ \Pi_0\ P\ \alpha_0\ \beta_0\\
&\quad \rightarrow \mathit{Phrase}\ \Pi_1\ Q\ \alpha_1\ \beta_1\\
&\quad \rightarrow \mathit{Phrase}\ \Pi_2\ (\mathit{Disjoint}\ P\ Q)\ \alpha_4\ \beta_3\\
&(\mathit{Phrase}\ P) \otimes (\mathit{Phrase}\ Q) = \mathit{Phrase}\ (\mathit{Disjoint}\ P\ Q)
\end{aligned}$$

Syntax Rules 16: Product Rules

$$\frac{\vec{\iota}\colon(\vec{\theta},\vec{\alpha}_0,\vec{\beta}_0),\vec{\iota}_P : (\vec{\theta}_P,\vec{\alpha}_P,\vec{\beta}_P) \vdash P : \theta_0 \to \theta_1 \qquad \vec{\iota}\colon(\vec{\theta},\vec{\alpha}_1,\vec{\beta}_1),\vec{\iota}_Q : (\vec{\theta}_Q,\vec{\alpha}_Q,\vec{\beta}_Q) \vdash Q : \theta_0}{\vec{\iota}\colon(\vec{\theta},\vec{\alpha}_0 \wedge \vec{\alpha}_1 \vee \phi(\theta_1),\vec{\alpha}_0 \wedge \vec{\alpha}_1 \vee \phi(\theta_1)),\vec{\iota}_P : (\vec{\theta}_P,\vec{\alpha}_P\phi(\theta_1),\vec{\alpha}_P \vee \phi(\theta_1)),\vec{\iota}_Q : (\vec{\theta}_Q,\vec{\alpha}_Q \vee \phi(\theta_1),\vec{\alpha}_Q \vee \phi(\theta_1)) \vdash P(Q) : \theta_1}$$

$$\Longrightarrow$$

$$\begin{aligned}
&\mathit{appl} :: (\\
&\quad \mathit{HasType}\ \Pi_0\ P\ (\theta_0 :\!\!\to\!\!: \theta_1),\\
&\quad \mathit{HasType}\ \Pi_1\ Q\ \theta_0,\\
&\quad \mathit{Union}\ \Pi_0\ \Pi_1\ \Pi_2,\\
&\quad \mathit{UnionAnd}\ \Pi_0\ \Pi_1\ \alpha_0\ \alpha_1\ \alpha_2,\\
&\quad \mathit{HLeftUnion}\ \alpha_0\ \alpha_1\ \alpha_3,\\
&\quad \mathit{HLeftUnion}\ \beta_0\ \beta_1\ \beta_2,\\
&\quad \mathit{HLeftUnion}\ \alpha_2\ \alpha_3\ (\mathit{Predicate}\ \alpha_4),\\
&\quad \mathit{HLeftUnion}\ \alpha_2\ \beta_2\ (\mathit{Predicate}\ \beta_3),\\
&\quad \mathit{IsPassive}\ \theta_1\ b,\\
&\quad \mathit{EachOr}\ \alpha_4\ b\ \alpha_5,\\
&\quad \mathit{EachOr}\ \beta_3\ b\ \beta_4\\
&) \Rightarrow \mathit{Phrase}\ \Pi_0\ P\ \alpha_0\ \beta_0\\
&\quad \to \mathit{Phrase}\ \Pi_1\ Q\ \alpha_1\ \beta_1\\
&\quad \to \mathit{Phrase}\ \Pi_2\ (\mathit{Appl}\ P\ Q)\ (\mathit{Predicate}\ \alpha_5)\ (\mathit{Predicate}\ \beta_4)\\
&\mathit{appl}\ (\mathit{Phrase}\ P)\ (\mathit{Phrase}\ Q) = \mathit{Phrase}\ (\mathit{Appl}\ P\ Q)
\end{aligned}$$

Syntax Rules 17: Application

7.4 Example

Consider a phrase that projects the passive part out of a pair where the other part of the pair interferes. A fragment of the derivation for the SCIR$_K$ syntax is in derivation 6 on the following page.

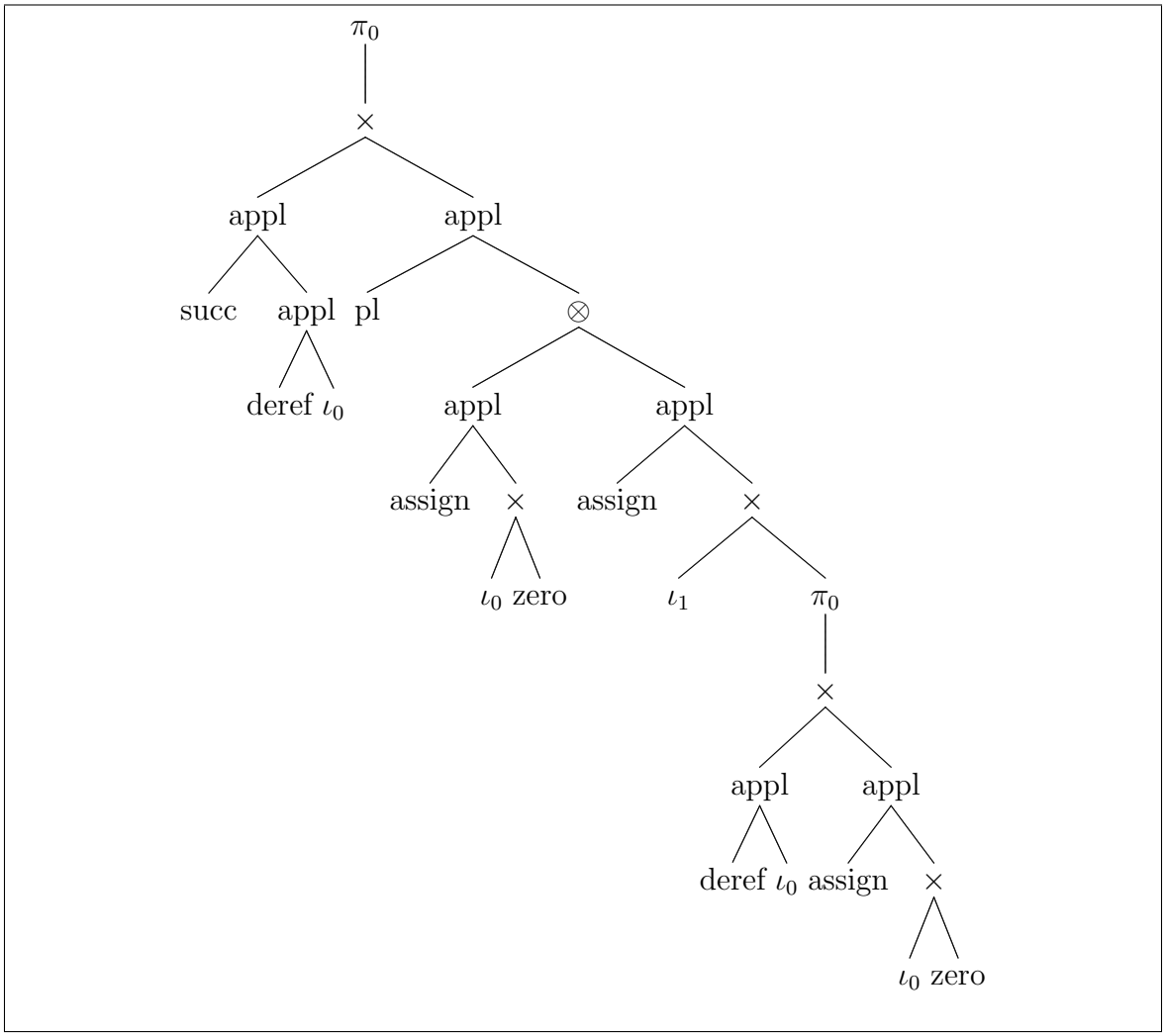
The HASKELL-code using defined SCIR$_K$ constructs to implement the command is in listing 22, and the parse tree for the code is in parse tree 7 on page 132. The type of the resulting HASKELL-data strucure is in listing 23 on page 133: note that two identifiers are in the passive part of the context, none in the active context, and the annotation is empty. Parse tree 8 shows the parsed HASKELL-type of only the phrase itself.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mid \iota_2 : \mathbf{var}[\tau] \vdash \iota_2 : \mathbf{var}[\tau]}}{\mid \iota_2 : \mathbf{var}[\tau] \vdash \mathbf{deref}(\iota_2) : \mathbf{exp}[\tau]}}{\mid \iota_0 : \mathbf{var}[\tau] \vdash \iota_0} \qquad \dfrac{}{\iota_2 : \mathbf{var}[\tau] \mid\vdash \mathbf{deref}(\iota_2) : \mathbf{exp}[\tau]}}{\iota_2 : \mathbf{var}[\tau] \mid \iota_0 : \mathbf{var}[\tau] \vdash \langle \iota_0, \mathbf{deref}(\iota_2) \rangle : \mathbf{var}[\tau] \times \mathbf{exp}[\tau]} \qquad \vdots}{\dfrac{\iota_2 : \mathbf{var}[\tau] \mid \iota_0 : \mathbf{var}[\tau] \vdash \iota_0 := \mathbf{deref}(\iota_2) : \mathbf{comm} \qquad \iota_2 : \mathbf{var}[\tau] \mid \iota_1 : \mathbf{var}[\tau] \vdash \iota_1 := \mathbf{deref}(\iota_2) : \mathbf{comm}}{\iota_2 : \mathbf{var}[\tau] \mid \iota_0, \iota_1 : \mathbf{var}[\tau] \vdash [\iota_0 := \mathbf{deref}(\iota_2), \iota_1 := \mathbf{deref}(\iota_2)] : \mathbf{comm} \otimes \mathbf{comm}}}{\iota_2 : \mathbf{var}[\tau] \mid \iota_0, \iota_1 : \mathbf{var}[\tau] \vdash \iota_0 := \mathbf{deref}(\iota_2)! \mid \iota_1 := \mathbf{deref}(\iota_2) : \mathbf{comm}}$$

Derivation 6: Example

```
π0 (
  (appl succ (
        appl deref
             ι0))
   ×
  (appl pl (
             (appl assign
                   (ι0 × zero))
              ⊗
             (appl assign (
                       ι1
                        ×
                       (π0 (
                         (appl deref
                               ι0)
                          ×
                         (appl assign
                               (ι0 × zero)))))))))
```

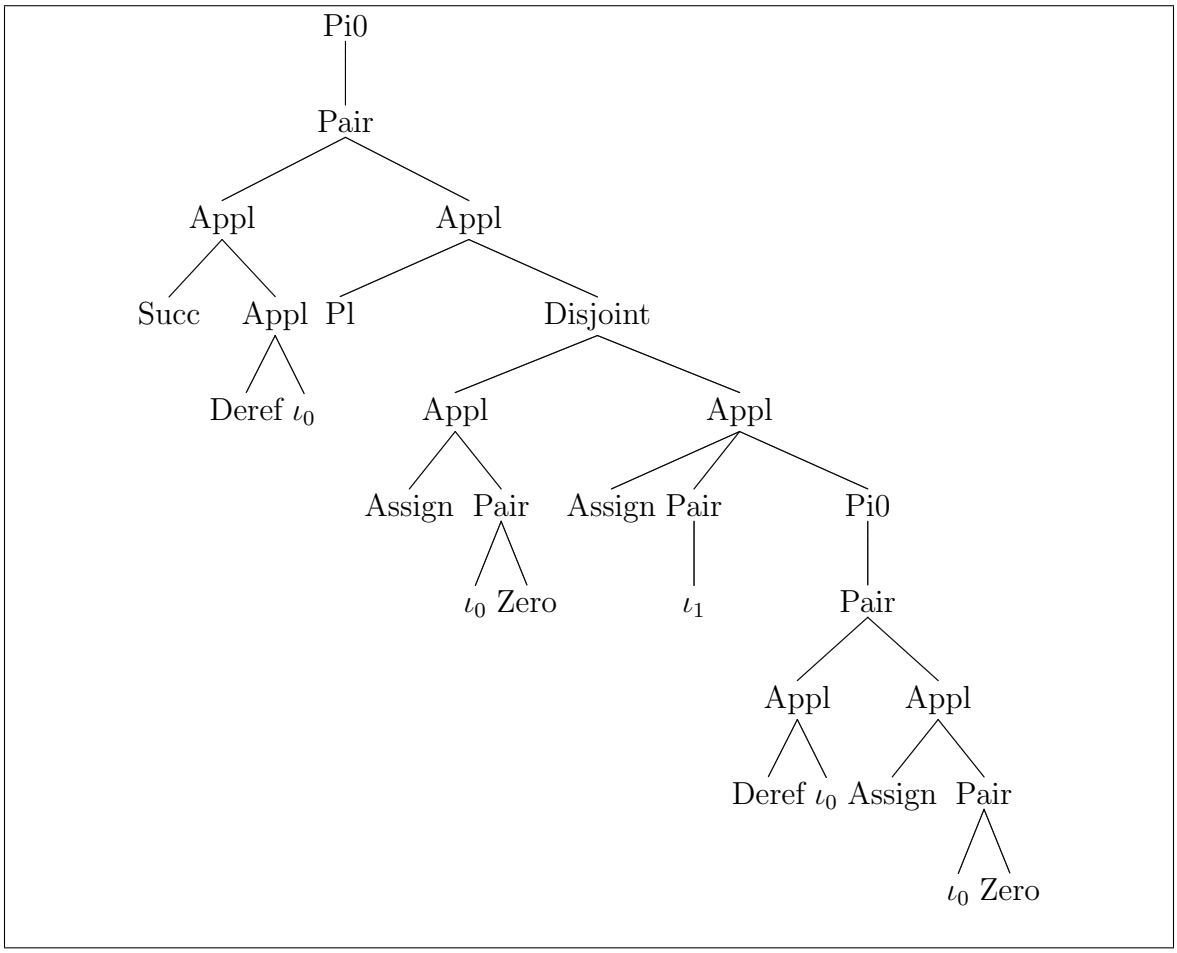
Listing 22: HASKELL Code



Parse Tree 7: HASKELL Syntax

```
Phrase
  (Record{ HSucc HZero = Var Int, HZero = Var Int })
  (Π0 (Pair (Appl Succ
                (Appl Deref
                      (Identifier HZero)))
            (Appl Pl
                  (Disjoint (Appl Assign
                                  (Pair (Identifier HZero)
                                        Zero))
                            (Appl Assign
                                  (Pair (Identifier (HSucc HZero))
                                        (Π0 (Pair (Appl Deref
                                                        (Identifier HZero))
                                                  (Appl Assign
                                                        (Pair (Identifier HZero)
                                                              Zero))))))))))
  (Predicate{ HSucc HZero = HTrue, HZero = HTrue })
  (Predicate{ HSucc HZero = HTrue, HZero = HTrue })
```

Listing 23: HASKELL Type



Parse Tree 8: HASKELL Type

7.5 Validation

Functionally, SCIR$_K$ is the extension of SCI with passivity. All phrases that were invalid in Idealized ALGOL should still be invalid in SCIR$_K$. The semantic meaning of phrases has not changed from Idealized ALGOL (besides the minor extensions in SCI). Therefore, validation of SCIR$_K$ consists of ensuring that the phrases containing passivity rejected by SCI are accepted by SCIR$_K$. Listing 24 on the next page lists phrases that were invalid in SCI and continue to be invalid in SCIR$_K$. The various forms of passivity that SCIR$_K$ accepts are in listing 25 on page 137 (and integrated in section 7.4 on page 129).

$$
\begin{aligned}
&\mathbf{rec}(\lambda\iota_0.\mathbf{true}) \Longrightarrow \mathit{appl}\ \mathit{rec}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ \mathit{true})\\
&(\lambda\iota_0.\iota_1 := 0)(\iota_1) \Longrightarrow \mathit{appl}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ (\mathit{appl}\ \mathit{assign}\ (\iota_1 \times \mathit{zero})))\ (\iota_1)\\
&(\lambda\iota_0.\iota_1)(\iota_1 := 0) \Longrightarrow \mathit{appl}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ \iota_1)\ (\mathit{appl}\ \mathit{assign}\ (\iota_1 \times \mathit{zero}))\\
&\\
&[\iota_0 := 0, \iota_0] \Longrightarrow \mathit{appl}\ \mathit{assign}\ (\iota_0 \times \mathit{zero}) \otimes \iota_0\\
&[\iota_0, \iota_0 := 0] \Longrightarrow \iota_0 \otimes \mathit{appl}\ \mathit{assign}\ (\iota_0 \times \mathit{zero})\\
&\mathbf{promote}(\lambda\iota_0.\iota_1 := 0) \Longrightarrow \mathit{promote}\ (\mathit{lambda}\ \mathit{firstLabel}\ \mathit{undefined}\ (\mathit{appl}\ \mathit{assign}\ (\iota_1 \times \mathit{zero})))
\end{aligned}
$$

Listing 24: Invalid phrases

$$
\begin{array}{l}
[\pi_0\langle 0, \iota_0 := 0\rangle, \iota_0] \Longrightarrow \pi_0\ (zero \times appl\ assign\ (\iota_0 \times zero)) \otimes \iota_0 \\
[(\lambda\iota_0.\mathbf{true})(\iota_1 := 0), \iota_1] \Longrightarrow (appl\ (lambda\ firstLabel\ undefined\ true)\ (appl\ assign\ (\iota_1 \times zero))) \otimes \iota_1 \\
[\pi_0\langle \iota_1, \iota_0 := 0\rangle, \iota_0] \Longrightarrow \pi_0\ (\iota_1 \times appl\ assign\ (\iota_0 \times zero)) \otimes \iota_0 \\
\iota_0 := (\lambda\iota_0.\mathbf{deref}(\iota_2))(\iota_2) \Longrightarrow appl\ assign\ (\iota_0 \times appl\ (lambda\ (nextLabel\ firstLabel)\ undefined\ (appl\ deref\ \iota_2))\ \iota_2)
\end{array}
$$

Listing 25: Valid phrases

7.6 Summary

Idealized ALGOL's syntax can be varied in an elegant way to syntactically control interference. This variant, SCIR$_K$, is based around annotations on phrases. The resulting language is more expressive than SCI as presented in chapter 5 on page 87.

SCIR$_K$ can be implemented as a definitional interpreter in HASKELL using combinations of type constraints that represent manipulations of the annotations. It requires substantial redefinition of the syntax rules presented in chapter 3 on page 28 but is semantically equivalent to Idealized ALGOL with parallelism. The techniques used to implement SCIR$_K$ could be used to implement other ALGOL-like languages with rich type systems.

Chapter 8

Conclusion

There are a variety of type systems providing syntactic control of interference. The SCI variant of Idealized ALGOL is one that preserves higher-order functional programming and, compared to other interference-controlling languages, puts a low burden on the programmer. SCIR is an improvement on SCI in that it is liberal in permitting passive phrases. Additionally, SCIR_K is a rewriting of SCIR that has been shown can be implemented as a definitional interpreter in HASKELL.

An implementation of SCIR_K can be incrementally built on definitional interpreters for PCF, Idealized ALGOL, and SCI-without-passification. The intrepeters consist of a combination of HASKELL-type specifications that implement type systems with HASKELL-definitions that implement syntactic constructors and semantics. Common extensions to HASKELL98 and the HList library [8] were used in implementation of the definitional interpreters.

The implementation of SCIR_K in HASKELL speaks both to the future general applicability of Idealized ALGOL type systems and the potential for richer domain-specific languages in HASKELL.

8.1 Summary of Contributions

This dissertation showed general techniques for definitional interpretation of statically-typed languages. The defining language requires a type system rich enough for user-defined entities that represent records such as contexts, environments, and states. It was shown that constructor functions may be used if built-in data constructors cannot support the required type constraints.

Specific notable techniques used to implement the syntax and semantics of each language are as follows:

8.1.1 PCF

1. Contexts are managed with union and subtraction as opposed to weakening and contraction.

2. Identifiers are encapsulations of type-level numerals.

3. Program validity is verified statically through a defining language function.

4. Semantic evaluation is implemented as an ad-hoc polymorphic function.

8.1.2 Idealized ALGOL

1. Type semantics are implemented as a type constraint.

2. The type of polymorphic functions like assignment and dereferencing can be inferred, although the ad-hoc polymorphism of conditional statements cannot.

3. State shape is managed as a stack using type constraints and functions.

4. The meaning of commands and expressions in the defined language is functions on states in the defining language.

8.1.3 SCI

SCI is implemented as a variant of Idealized ALGOL with the redefinition of only relevant constructs. In particular, type construction functions need only the addition of a non-interference constraint.

8.1.4 SCIR$_K$

1. A passive type predicate is implemented as a HASKELL-type constraint on HASKELL-types that represent SCIR$_K$-types.

2. The additional SCIR$_K$-constraints on identifiers are treated as predicates and implemented using records.

3. The predicates are managed using HASKELL-type constraints that represent functions on aggregate SCIR$_K$-constraints.

8.2 Future Research

For immediate use as a programming language, the version of SCIR$_K$ presented here would need to be extended with 'syntactic sugar'. For example, [7] has shown that the techniques used for HList can also be used to implement statically-typed objects; object-oriented SCIR$_K$ would allow some exploration of interference in objects, as outlined in [18].

Turning a definitional interpreter into a stand-alone interpreter is trivial, requiring only the implementation of a parser combinator in HASKELL. More fruitfully, a definitional interpreter can be used as a prototype and reference implementation for the implementation of a lower-level (and therefore more efficient) interpreter or compiler. General developments in the use of domain-specific languages in HASKELL will affect this implementation of SCIR$_K$.

The implementations presented here use HList quite heavily. As a result, the future direction of HList will affect this code significantly. HList's type checking error messages are cryptic even by the standards of HASKELL, so that the type checking error messages for SCIR$_K$ are similarly afflicted. Improvement of HList's error messages, such as discussed in [19], could significantly improve the usability of definitional interpreters.

The undecidable instances extension to HASKELL98 that functional dependencies and therefore HList relies on may prevent type checking from halting. At the time of this writing, the team writing the standard of the next version of HASKELL (called HASKELL$'$) is exploring alternative type system extensions [1]. If these were to gain widespread acceptence, rewriting the implementations given here would be desirable.

Domain-specific programming languages have traditionally not had rich type systems. As use increases, however, it is expected to become the norm rather than the exception. SCIR$_K$ has a relatively complex type system, so its implementation could serve as an example to implementers of other languages. Other annotation-based variants of Idealized ALGOL's type system, such as security annotations, could be quite closely based on this code as well.

Bibliography

[1] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253. ACM, 2005.

[2] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[3] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole. Report on the FX-91 programming language. Technical Report MIT-LCS//MIT/LCS/TR-531, 1992. `http://psrg.lcs.mit.edu/history/publications/Papers/fx91-report.pdf`.

[4] Howard Huang and Uday Reddy. Type reconstruction for SCI. In David N. Turner, editor, *Functional Programming, Glasgow 1995*, Electronic Workshops in Computing. Springer-Verlag, 1996. `http://www.cs.bham.ac.uk/~udr/papers/sci.inference.ps.gz`.

[5] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, 1996.

[6] Simon P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. `http://haskell.org/onlinereport/`.

[7] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. Draft; Submitted for journal publication; online since 30 Sep. 2004; Full version released 10 September 2005. `http://homepages.cwi.nl/~ralf/OOHaskell/paper.pdf`, 2005.

[8] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. Technical Report SEN-E0420, CWI, Amsterdam, 2004. `http://homepages.cwi.nl/~ralf/HList/paper.pdf`.

[9] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *SIGPLAN Notices*, 12(2):1–79, 1977.

[10] INSMOS Ltd. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[11] Peter Naur et al. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, January 1963.

[12] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Theoretical Computer Science*, 228(1-2):211–252, 1999.

[13] Simon Peyton Jones. Wearing the hair shirt: a retrospective on Haskell. Invited talk at the *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. `http://research.microsoft.com/~simonpj/papers/haskell-retrospective/HaskellRetrospective.pdf`, 2003.

[14] Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: exploring the design space. In *Proceedings of the Haskell Workshop (unpublished)*, 1997. `http://research.microsoft.com/~simonpj/papers/type-class-design-space/multi.ps.gz`.

[15] Simon Peyton Jones, John Launchbury, Mark Shields, and Andrew Tolmach. Bridging the gulf: A common intermediate language for ML and Haskell. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1998.

[16] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[17] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM annual conference*, pages 717–740. ACM, 1972.

[18] John C. Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 39–46. ACM, 1978.

[19] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Improving type error diagnosis. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 80–91. ACM, 2004.

[20] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[21] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide*. 2007. `http://www.haskell.org/ghc/docs/6.6.1/html/users_guide/`.

[22] Robert D. Tennent. *Semantics of Programming Languages*. Prentice Hall, London, 1991.

Appendix A

Code Listings

A.1 HList

```
{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances #-}


module HList (
        module CommonMain,
        module Label1,
        module TypeEqGeneric1,
        --module TypeCastGeneric1,
        IfHasField,
        Union,
        Perturb, perturb
) where


import CommonMain
```

```
import Label1
import TypeEqGeneric1
import TypeCastGeneric1

-----

class IfHasField l r v
instance (
        HLeftUnion r (Record (HCons (F l v) HNil)) r',
        HasField l r' v
        ) => IfHasField l r v



class Union r r' r'' | r r' -> r''
instance Union r (Record HNil) r
instance (
        RecordLabels r ls,
        HMember l ls b,
        UnionRecords b r (F l v) r''',
        Union (Record r''') (Record r') r''
        ) => Union (Record r) (Record (HCons (F l v) r')) r''

class  UnionRecords b r f r' | b r f -> r'
instance HasField l r v => UnionRecords HTrue r (F l v) r
```

```
instance UnionRecords HFalse r f (HCons f r)



instance Union s HNil s
instance (
        HMember h s b,
        UnionSets b s h s'',
        Union s'' t s'
        ) => Union s (HCons h t) s'

class UnionSets b s e s' | b s e -> s'
instance UnionSets HTrue s h s
instance UnionSets HFalse s h (HCons h s)



class HLeftUnion (Record (HCons (F p a) HNil)) (Record u) (Record u')
        => Perturb p a u u' | p a u -> u' where
        perturb :: p -> a -> Record u -> Record u'
        perturb p a u
                = hLeftUnion (mkRecord (hCons (F a :: F p a) hNil)) u
instance HLeftUnion
                (Record (HCons (F p a) HNil))
                (Record u)
                (Record u')
```

```
        => Perturb p a u u'
```

A.2 PCF

```
{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances #-}
{-# OPTIONS -fno-monomorphism-restriction #-}

module PCF (
        Eval, eval, zero, true, succ, pred, not, eq, cond,
        evalPhrase, Cond, (:->:)((:->:)), (:><:)((:><:)),
        Phrase(Phrase), Zero(Zero), Int(Int), True(True),
        Bool(Bool), Succ(Succ), Pred(Pred), Not(Not), Eq(Eq),
        Appl(Appl), Perturb, Lambda(Lambda), perturb, Pair(Pair),
        Rec(Rec), Pi0(Pi0), lambda, appl, (><), ident, rec, pi0,
        Context, Environment, HasType, Identifier, isValidProgram
) where
import qualified Prelude
import HList

-----

type Identifier n = Label n
type Environment s = Record s
type Context c = Record c
```

```
data Appl p q = Appl p q

data Lambda l theta p = Lambda l p

data Pair p q = Pair p q

data Pi0 p = Pi0 p


data Zero = Zero

data True = True

data Succ = Succ

data Pred = Pred

data Not = Not

data Cond = Cond

data Eq = Eq

data Rec = Rec


data x :->: y = x :->: y

infixr 0 :->:

data x :><: y = x :><: y

infixl 9 :><:

data Int = Int

data Bool = Bool


data Phrase pi p = Phrase p


-----
```

```
class HasType pi p theta | pi p -> theta

instance HasType pi Zero Int

instance HasType pi True Bool

instance HasType pi Succ (Int :->: Int)

instance HasType pi Pred (Int :->: Int)

instance HasType pi Not (Bool :->: Bool)

instance HasType pi Cond (Bool :><: theta :><: theta :->: theta)

instance HasType pi Eq (Int :><: Int :->: Bool)

instance HasType pi Rec ((theta :->: theta) :->: theta)

instance HasType pi p (theta0 :><: theta1)
        => HasType pi (Pi0 p) theta0

instance (HasType pi p (theta0 :->: theta1), HasType pi q theta0)
        => HasType pi (Appl p q) theta1

instance (Perturb iota theta0 pi0 pi1, HasType (Context pi1) p theta1)
        => HasType
                (Context pi0)
                (Lambda iota theta0 p)
                (theta0 :->: theta1)

instance (HasType pi p theta0, HasType pi q theta1)
        => HasType pi (Pair p q) (theta0 :><: theta1)

instance IfHasField (Identifier n) pi theta
        => HasType pi (Identifier n) theta
```

```
-----

ident :: Identifier n
        -> Phrase
                (Context (HCons (F (Identifier n) theta) HNil))
                (Identifier n)
ident iota = Phrase iota

appl :: (
        Union pi0 pi1 pi2,
        HasType pi0 p (theta0 :->: theta1),
        HasType pi1 q theta0
        ) => Phrase pi0 p -> Phrase pi1 q -> Phrase pi2 (Appl p q)
appl (Phrase p) (Phrase q) = Phrase (Appl p q)

lambda :: (
        H2ProjectByLabels (HCons (Identifier n) HNil) pi0 z pi1,
        HasType (Context pi0) (Identifier n) theta
        ) => Identifier n -> theta
                -> Phrase (Context pi0) p
                -> Phrase
                        (Context pi1)
                        (Lambda (Identifier n) theta p)
lambda iota (t :: theta) (Phrase p) = Phrase (Lambda iota p)
```

```
(><) :: Union pi0 pi1 pi2
        => Phrase pi0 p -> Phrase pi1 q -> Phrase pi2 (Pair p q)
(Phrase p) >< (Phrase q) = Phrase (Pair p q)

pi0 :: HasType pi p (theta0 :><: theta1)
        => Phrase pi p -> Phrase pi (Pi0 p)
pi0 (Phrase p) = Phrase (Pi0 p)



zero :: Phrase (Context HNil) Zero
zero = Phrase Zero

true :: Phrase (Context HNil) True
true = Phrase True

succ :: Phrase (Context HNil) Succ
succ = Phrase Succ

pred :: Phrase (Context HNil) Pred
pred = Phrase Pred

not :: Phrase (Context HNil) Not
not = Phrase Not
```

```
cond :: Phrase (Context HNil) Cond
cond = Phrase Cond

eq :: Phrase (Context HNil) Eq
eq = Phrase Eq

rec :: Phrase (Context HNil) Rec
rec = Phrase Rec

-----

class Eval p u x | p u -> x where
        eval :: p -> u -> x

instance HasField (Identifier n) u x => Eval (Identifier n) u x where
        eval l u = hLookupByLabel l u

instance (Eval p u (x -> y), Eval q u x) => Eval (Appl p q) u y where
        eval (Appl p q) u = (eval p u) (eval q u)

instance (Perturb l x u u', Eval p (Environment u') y)
        => Eval (Lambda l theta p) (Environment u) (x -> y) where
        eval (Lambda l p) u = \a -> eval p (perturb l a u)
```

```
instance (Eval p u x, Eval q u y)
        => Eval (Pair p q) u (x,y) where
        eval (Pair p q) u = (eval p u, eval q u)

instance Eval p u (x,y) => Eval (Pi0 p) u x where
        eval (Pi0 p) u = let (x,y) = eval p u in x



instance Eval Zero u Prelude.Integer where
        eval _ _ = 0

instance Eval True u Prelude.Bool where
        eval _ _ = Prelude.True

instance Eval Succ u (Prelude.Integer -> Prelude.Integer) where
        eval _ _ = \n -> n Prelude.+ 1

instance Eval Pred u (Prelude.Integer -> Prelude.Integer) where
        eval _ _ = \n -> n Prelude.- 1

instance Eval Not u (Prelude.Bool -> Prelude.Bool) where
        eval _ _ = \b -> if b then Prelude.False else Prelude.True
```

```
instance Eval Cond u (((Prelude.Bool, theta), theta) -> theta) where
        eval _ _ = \((b, x), y) -> if b then x else y

instance Eval Eq u (
        (Prelude.Integer, Prelude.Integer) -> Prelude.Bool) where
        eval _ _ = \(x,y) -> x Prelude.== y

instance Eval Rec u ((theta -> theta) -> theta) where
        eval _ _ = let y = \f -> f (y f) in y

-----

evalProgram :: (Eval p (Environment HNil) x)
        => Phrase (Context HNil) p -> x
evalProgram (Phrase p) = eval p (Record hNil)

isValidProgram :: Phrase (Context HNil) p -> Prelude.Bool
```

A.3 IA

```
{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances #-}

module IA (
        module PCF, Comm(Comm), zero, true, succ, pred, not, eq,
        cond, Cond(Cond), deref, assign, skip, sq, new,
```

```
        Deref(Deref), Var, Exp(Exp), Assign(Assign), Skip(Skip),
        Sq(Sq), New(New), mkVar, EvalPhraseType, Zero(Zero), True(True),
        Succ(Succ), Pred(Pred), Not(Not), Eq(Eq), State, Command,
        Expression, Variable
) where
import PCF hiding (
        zero, true, succ, pred, not, eq, cond, evalPhrase,
        Cond, Zero, True, Succ, Pred, Not, Eq)
import qualified Prelude
import HList

-----

data Zero = Zero
data True = True
data Succ = Succ
data Pred = Pred
data Not = Not
data Eq = Eq

-----

type State s = Record s
type Command s = s -> s
```

```
type Expression s tau = s -> tau

type Acceptor s tau = Expression s tau -> Command s

type Variable s tau = (Acceptor s tau, Expression s tau)

type Conditional s theta
        = ((Expression s Prelude.Bool, theta), theta) -> theta


data Exp tau = Exp tau

data Comm = Comm

type Acc tau = Exp tau :->: Comm

type Var tau = Acc tau :><: Exp tau


data Deref = Deref

data Assign = Assign

data Skip = Skip

data Sq = Sq

data New tau = New

data Cond theta = Cond


-----


instance HasType pi Deref (Var tau :->: Exp tau)

instance HasType pi Assign (Var tau :><: Exp tau :->: Comm)

instance HasType pi Skip Comm

instance HasType pi Sq (Comm :><: Comm :->: Comm)
```

```
instance HasType pi (New tau) ((Var tau :->: Comm) :->: Comm)
instance HasType pi
        (Cond theta)
        (Exp Bool :><: theta :><: theta :->: theta)
instance HasType pi Zero (Exp Int)
instance HasType pi True (Exp Bool)
instance HasType pi Succ (Exp Int :->: Exp Int)
instance HasType pi Pred (Exp Int :->: Exp Int)
instance HasType pi Not (Exp Bool :->: Exp Bool)
instance HasType pi Eq (Exp Int :><: Exp Int :->: Exp Bool)

-----

deref :: Phrase (Context HNil) Deref
deref = Phrase Deref

assign :: Phrase (Context HNil) Assign
assign = Phrase Assign

skip :: Phrase (Context HNil) Skip
skip = Phrase Skip

sq :: Phrase (Context HNil) Sq
sq = Phrase Sq
```

```
new :: Phrase (Context HNil) (New tau)
new = Phrase New

cond :: Phrase (Context HNil) (Cond theta)
cond = Phrase Cond

zero :: Phrase (Context HNil) Zero
zero = Phrase Zero

true :: Phrase (Context HNil) True
true = Phrase True

succ :: Phrase (Context HNil) Succ
succ = Phrase Succ

pred :: Phrase (Context HNil) Pred
pred = Phrase Pred

not :: Phrase (Context HNil) Not
not = Phrase Not

eq :: Phrase (Context HNil) Eq
eq = Phrase Eq
```

```
-----

instance Eval Zero u (Expression s Prelude.Integer) where
        eval _ _ = \s -> 0

instance Eval True u (Expression s Prelude.Bool) where
        eval _ _ = \s -> Prelude.True

instance Eval Succ u
        (Expression s Prelude.Integer
                -> Expression s Prelude.Integer) where
        eval _ _ = \n -> \s -> Prelude.succ (n s)

instance Eval Pred u
        (Expression s Prelude.Integer
                -> Expression s Prelude.Integer) where
        eval _ _ = \n -> \s -> Prelude.pred (n s)

instance Eval Not u
        (Expression s Prelude.Bool -> Expression s Prelude.Bool) where
        eval _ _ = \b -> \s -> Prelude.not (b s)

instance Eval Eq u
```

```
        ((Expression s Prelude.Integer, Expression s Prelude.Integer)
                -> Expression s Prelude.Bool) where
        eval _ _ = \(x,y) -> \s -> (x s) Prelude.== (y s)

instance Eval Deref u (Variable s t -> Expression s t) where
        eval _ _ = \(_,e) -> e

instance Eval Assign u
        ((Variable s t, Expression s t) -> Command s) where
        eval _ _ = \((a,d), e) -> \s -> a e s

instance Eval Skip u (Command s) where
        eval _ _ = \s -> s

instance Eval Sq u ((Command s, Command s) -> Command s) where
        eval _ _ = \(c0, c1) -> \s -> c1 (c0 s)


instance (
                NewLabel (State s0) l,
                HasField l s' Prelude.Integer,
                HFind l ls n,
                HUpdateAtHNat n (F l Prelude.Integer) s' s',
                HList s0,
```

```
                RecordLabels s' ls
        ) => Eval (New Int) u (
                (Variable (State s') Prelude.Integer
                -> Command (State (HCons (F l Prelude.Integer) s0)))
                -> Command (State s0)
                ) where
        eval _ _ = \b -> \(Record s) ->
                let l                   = newLabel (Record s)
                    v                   = mkVar l
                    s'                  = Record (HCons (newF l 0) s)
                    Record (HCons x s0) = b v s'
                in Record s0


instance (
                NewLabel (State s0) l,
                HasField l s' Prelude.Bool,
                HFind l ls n,
                HUpdateAtHNat n (F l Prelude.Bool) s' s',
                HList s0,
                RecordLabels s' ls
        ) => Eval (New Bool) u (
                (Variable (State s') Prelude.Bool
                -> Command (State (HCons (F l Prelude.Bool) s0)))
                -> Command (State s0)
```

```
                ) where
        eval _ _ = \b -> \(Record s) ->
                let l                   = newLabel (Record s)
                    v                   = mkVar l
                    s'                  = Record (HCons (newF l Prelude.False) s)
                    Record (HCons x s0) = b v s'
                in Record s0



instance EvalPhraseType (Exp tau) (Expression s t)
        => Eval
                (Cond (Exp tau))
                u
                (Conditional s (Expression s t)) where
        eval _ _ = \((b, x), y) -> \s -> if b s then x s else y s

instance Eval (Cond Comm) u (Conditional s (Command s)) where
        eval _ _ = \((b, x), y) -> \s -> if b s then x s else y s

instance (
        Eval (Cond theta1) u (Conditional s y),
        EvalPhraseType theta0 x
        ) => Eval
```

```
                (Cond (theta0 :->: theta1))
                u
                (Conditional s (x -> y)) where
        eval _ u = \((b, x), y) ->
                \a -> eval (Cond :: Cond theta1) u ((b, x a), y a)

instance (
        Eval (Cond theta0) u (Conditional s x),
        Eval (Cond theta1) u (Conditional s y)
        ) => Eval
                (Cond (theta0 :><: theta1))
                u
                (Conditional s (x,y)) where
        eval _ u = \((b, (p,q)), (r,s)) ->
                (eval (Cond :: Cond theta0) u ((b, p), r),
                eval (Cond :: Cond theta1) u ((b, q), s))

-----

class EvalPhraseType theta t | theta -> t
instance (EvalPhraseType p x, EvalPhraseType q y) => EvalPhraseType (p :->: q) (x
instance (EvalPhraseType p x, EvalPhraseType q y) => EvalPhraseType (p :><: q) (x,
instance EvalDataType tau t => EvalPhraseType (Exp tau) (Expression s t)
instance EvalPhraseType Comm (Command s)
```

```
class EvalDataType tau t | tau -> t, t -> tau
instance EvalDataType Int Prelude.Integer
instance EvalDataType Bool Prelude.Bool

-----

mkVar :: (
        HasField l (State s) t,
        HFind l ls n,
        RecordLabels s ls,
        HUpdateAtHNat n (F l t) s s
        ) => l -> Variable (State s) t
```

A.4 SCI

```
{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances #-}

module SCI (
        module IA, appl, (<>), promote, rec, pl,
        Disjoint(Disjoint), Promote(Promote), Pl(Pl), (:<>:), (:~>:),
        Rec(Rec)
) where
import IA hiding (appl, rec, Rec)
import qualified Prelude
```

```
import HList

-----

class NonInterfering pi0 pi1
instance (
        RecordLabels pi0 f0,
        RecordLabels pi1 f1,
        HTIntersect f0 f1 HNil
        ) => NonInterfering (Context pi0) (Context pi1)

-----

data Disjoint p q = Disjoint p q
data Promote p = Promote p

data x :<>: y = x :<>: y
infixl 9 :<>:
data x :~>: y = x :~>: y
infixr 0 :~>:

instance (HasType pi p theta0, HasType pi q theta1)
        => HasType pi (Disjoint p q) (theta0 :<>: theta1)
instance HasType pi p (theta0 :->: theta1)
```

```
        => HasType pi (Promote p) (theta0 :~>: theta1)
instance HasType pi Rec ((theta :~>: theta) :->: theta)

-----

appl :: (
        Union pi0 pi1 pi2,
        NonInterfering pi0 pi1,
        HasType pi0 p (theta0 :->: theta1),
        HasType pi1 q theta0
        ) => Phrase pi0 p -> Phrase pi1 q -> Phrase pi2 (Appl p q)
appl (Phrase p) (Phrase q) = Phrase (Appl p q)

(<>) :: (
        Union pi0 pi1 pi2,
        NonInterfering pi0 pi1
        ) => Phrase pi0 p -> Phrase pi1 q -> Phrase pi2 (Disjoint p q)
(Phrase p) <> (Phrase q) = Phrase (Disjoint p q)

promote :: HasType (Context HNil) p (theta0 :->: theta1)
        => Phrase (Context HNil) p -> Phrase (Context HNil) (Promote p)
promote (Phrase p) = Phrase (Promote p)

-----
```

```
data Rec = Rec

rec :: Phrase (Context HNil) Rec
rec = Phrase Rec

instance Eval Rec u ((theta -> theta) -> theta) where
        eval _ _ = let y = \f -> f (y f) in y

-----

instance Eval p u theta => Eval (Promote p) u theta where
        eval (Promote p) u = eval p u

instance Eval (Pair p q) u (theta0,theta1)
        => Eval (Disjoint p q) u (theta0,theta1) where
        eval (Disjoint p q) u = eval (Pair p q) u

-----

data Pl = Pl

pl :: Phrase (Context HNil) Pl
pl = Phrase Pl
```

```
instance HasType pi Pl (Comm :<>: Comm :->: Comm)

instance Eval Pl u ((Command s, Command s) -> Command s) where
```

A.5 SCIRK

```
{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances #-}
{-# OPTIONS -fno-monomorphism-restriction #-}

module SCIRK where
import SCI hiding (
        Phrase, ident, appl, lambda, (><), zero, true, succ, pred,
        not, eq, rec, pi0, deref, assign, skip, sq, pl, new, cond,
        promote, (<>))
import qualified Prelude
import HList

-----

type Predicate p = Record p

data Phrase pi p alpha beta = Phrase p

-----
```

```
class AllTrue r
instance AllTrue HNil
instance AllTrue t => AllTrue (HCons (F l HTrue) t)

-----

evalProgram :: (
        Eval p (Environment HNil) (State HNil -> t),
        AllTrue alpha,
        AllTrue beta
        ) => Phrase (Context HNil) p (Predicate alpha) (Predicate beta)
                -> t
evalProgram (Phrase p) = eval p (Record hNil) (Record hNil)

isValidProgram :: (AllTrue alpha, AllTrue beta)
        => Phrase (Context HNil) p (Predicate alpha) (Predicate beta)
                -> Prelude.Bool
isValidProgram (Phrase p) = Prelude.True

-----

class IsPassive theta b | theta -> b
instance IsPassive (Exp tau) HTrue
```

```
instance IsPassive Comm HFalse
instance IsPassive q b => IsPassive (p :->: q) b
instance (IsPassive p b0, IsPassive q b1, HAnd b0 b1 b2)
        => IsPassive (p :><: q) b2
instance (IsPassive p b0, IsPassive q b1, HAnd b0 b1 b2)
        => IsPassive (p :<>: q) b2
instance IsPassive (p :~>: q) HTrue


-----


ident :: IsPassive theta b
        => Identifier n -> Phrase
                (Context (HCons (F (Identifier n) theta) HNil))
                (Identifier n)
                (Predicate (HCons (F (Identifier n) b) HNil))
                (Predicate (HCons (F (Identifier n) HTrue) HNil))
ident iota = Phrase iota



lambda :: (
        H2ProjectByLabels (HCons iota HNil) pi0 z0 pi1,
        HasType (Context pi0) iota theta,
        H2ProjectByLabels (HCons iota HNil) alpha0 z1 alpha1,
        H2ProjectByLabels (HCons iota HNil) beta0 z2 beta1,
```

```
        IfHasField iota (Predicate beta0) HTrue
        ) => iota
                -> theta
                -> Phrase
                        (Context pi0)
                        p
                        (Predicate alpha0)
                        (Predicate beta0)
                -> Phrase
                        (Context pi1)
                        (Lambda iota theta p)
                        (Predicate alpha1)
                        (Predicate beta1)
lambda iota (theta :: theta) (Phrase p) = Phrase (Lambda iota p)


class UnionAnd pi0 pi1 alpha0 alpha1 alpha2
        | pi0 pi1 alpha0 alpha1 -> alpha2
instance (
        RecordLabels pi0 p0,
        RecordLabels pi1 p1,
        HTIntersect p0 p1 p2,
        UnionAnd' p2 alpha0 alpha1 alpha2
        ) => UnionAnd
```

```
                (Predicate pi0)
                (Predicate pi1)
                (Predicate alpha0)
                (Predicate alpha1)
                (Predicate alpha2)

class UnionAnd' a alpha0 alpha1 alpha2 | a alpha0 alpha1 -> alpha2
instance UnionAnd' HNil alpha0 alpha1 HNil
instance (
        HasField h alpha0 b0,
        HasField h alpha1 b1,
        HAnd b0 b1 b2,
        UnionAnd' t alpha0 alpha1 alpha2
        ) => UnionAnd' (HCons h t) alpha0 alpha1 (HCons (F h b2) alpha2)


appl :: (
        HasType pi0 p (theta0 :->: theta1),
        HasType pi1 q theta0,
        Union pi0 pi1 pi2,
        UnionAnd pi0 pi1 alpha0 alpha1 alpha2,
        HLeftUnion alpha0 alpha1 alpha3,
        HLeftUnion beta0 beta1 beta2,
        HLeftUnion alpha2 alpha3 (Predicate alpha4),
```

```
        HLeftUnion alpha2 beta2 (Predicate beta3),
        IsPassive theta1 b,
        EachOr alpha4 b alpha5,
        EachOr beta3 b beta4
        ) => Phrase pi0 p alpha0 beta0
                -> Phrase pi1 q alpha1 beta1
                -> Phrase
                        pi2
                        (Appl p q)
                        (Predicate alpha5)
                        (Predicate beta4)
appl (Phrase p) (Phrase q) = Phrase (Appl p q)



(<>) :: (
        Union pi0 pi1 pi2,
        UnionAnd pi0 pi1 alpha0 alpha1 alpha2,
        HLeftUnion alpha0 alpha1 alpha3,
        HLeftUnion beta0 beta1 beta2,
        HLeftUnion alpha2 alpha3 alpha4,
        HLeftUnion alpha2 beta2 beta3
        ) => Phrase pi0 p alpha0 beta0
                -> Phrase pi1 q alpha1 beta1
```

```
                -> Phrase pi2 (Disjoint p q) alpha4 beta3
(Phrase p) <> (Phrase q) = Phrase (Disjoint p q)


(><) :: (
        Union pi0 pi1 pi2,
        UnionAnd pi0 pi1 alpha0 alpha1 alpha2,
        UnionAnd pi0 pi1 beta0 beta1 beta2,
        HLeftUnion alpha0 alpha1 alpha3,
        HLeftUnion beta0 beta1 beta3,
        HLeftUnion alpha2 alpha3 alpha4,
        HLeftUnion beta2 beta3 beta4
        ) => Phrase pi0 p alpha0 beta0
                -> Phrase pi1 q alpha1 beta1
                -> Phrase pi2 (Pair p q) alpha4 beta4
(Phrase p) >< (Phrase q) = Phrase (Pair p q)


class EachOr r0 b r1 | r0 b -> r1
instance EachOr HNil b HNil
instance (HOr b0 b1 b2, EachOr t0 b1 t1)
        => EachOr (HCons (F l b0) t0) b1 (HCons (F l b2) t1)

pi0 :: (
```

```
        HasType pi0 p (theta0 :><: theta1),
        IsPassive theta0 b,
        EachOr alpha0 b alpha1,
        EachOr beta0 b beta1
        ) => Phrase pi0 p (Predicate alpha0) (Predicate beta0)
                -> Phrase
                        pi0
                        (Pi0 p)
                        (Predicate alpha1)
                        (Predicate beta1)
pi0 (Phrase p) = Phrase (Pi0 p)



promote :: (
        HasType pi p (theta0 :->: theta1),
        AllTrue alpha0,
        EachOr beta0 HTrue beta1
        ) => Phrase pi p (Predicate alpha0) (Predicate beta0)
                -> Phrase
                        pi
                        (Promote p)
                        (Predicate alpha0)
                        (Predicate beta1)
promote (Phrase p) = Phrase (Promote p)
```

```
-----

rec :: Phrase (Context HNil) Rec (Predicate HNil) (Predicate HNil)
rec = Phrase Rec

pl :: Phrase (Context HNil) Pl (Predicate HNil) (Predicate HNil)
pl = Phrase Pl

zero :: Phrase (Context HNil) Zero (Predicate HNil) (Predicate HNil)
zero = Phrase Zero

true :: Phrase (Context HNil) True (Predicate HNil) (Predicate HNil)
true = Phrase True

succ :: Phrase (Context HNil) Succ (Predicate HNil) (Predicate HNil)
succ = Phrase Succ

pred :: Phrase (Context HNil) Pred (Predicate HNil) (Predicate HNil)
pred = Phrase Pred

not :: Phrase (Context HNil) Not (Predicate HNil) (Predicate HNil)
not = Phrase Not
```

```
eq :: Phrase (Context HNil) Eq (Predicate HNil) (Predicate HNil)
eq = Phrase Eq

deref :: Phrase (Context HNil) Deref (Predicate HNil) (Predicate HNil)
deref = Phrase Deref

assign :: Phrase (Context HNil) Assign (Predicate HNil) (Predicate HNil)
assign = Phrase Assign

skip :: Phrase (Context HNil) Skip (Predicate HNil) (Predicate HNil)
skip = Phrase Skip

sq :: Phrase (Context HNil) Sq (Predicate HNil) (Predicate HNil)
sq = Phrase Sq

new :: Phrase (Context HNil) (New tau) (Predicate HNil) (Predicate HNil)
new = Phrase New

cond :: Phrase (Context HNil) (Cond theta) (Predicate HNil) (Predicate HNil)
cond = Phrase Cond
```