FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

MSc thesis

Petur Andrias Højgaard Jacobsen

Design of a Reversible Functional Programming Language

And its Type System

Academic advisor: Michael Kirkedal Thomsen

Academic co-advisor: Robin Kaarsgaard Jensen

Submitted: August 14, 2018

Design of a Reversible Functional Programming Language

And its Type System

Petur Andrias Højgaard Jacobsen
DIKU, Department of Computer Science,
University of Copenhagen, Denmark

August 14, 2018

MSc thesis

Author:	Petur Andrias Højgaard Jacobsen
Affiliation:	DIKU, Department of Computer Science, University of Copenhagen, Denmark
Title:	Design of a Reversible Functional Programming Language / And its Type System
Academic advisor:	Michael Kirkedal Thomsen
Academic co-advisor:	Robin Kaarsgaard Jensen
Submitted:	August 14, 2018

Abstract

Reversible programming languages are languages which exhibit both forward and backward determinism. The theory of reversible flowchart languages is relatively well understood, but studies of reversible functional languages are few and far between. In this thesis, we introduce a garbage-free, reversible, function language, inspired by RFun, which we call CoreFun. We provide a formalization of its semantics and extend it with a type system based on relevance logic. The type system also has support for recursive and polymorphic types. With the type system, we are able to add the use of ancillae variables through an unrestricted fragment. Backwards determinism of of branching is achieved with a First Match Policy, but we investigate the possibility of ensuring backwards determinism with exit assertions or static guarantees of orthogonality as alternatives. As program inversion of non-flowchart languages generally is hard, we present a formalization of its inverse semantics instead. Finally, we describe how to lift the core language into a syntactically lighter language via a sequence of translation schemes, making it more amenable for modern style programming.

Contents

Introduction

1

Reversible computing is the study of computational models in which individual computation steps can be uniquely and unambiguously inverted. For programming languages, this means languages in which programs can be run *backward* and get a unique result (the exact input). In this thesis, we restrict ourselves to *garbage-free* reversible programming languages, which guarantee not only that all programs are reversible, but also that no hidden duplication of data is required in order to make this guarantee.

In this thesis, we present a simple, but *r-Turing complete* [4], reversible typed functional programming language, CoreFun. Functional languages and programming constructs are currently quite successful; this includes both applications in special domains, e.g. Erlang, and functional constructs introduced in mainstream programming languages, such as Java and C++. We believe that functional languages also provide a suitable environment for studying reversible programs and computations, as recently shown in [42]. However, the lack of a type system exposed the limitations of the original RFun language, which has motivated this work. A carefully designed type system can provide better handling of static information through the introduction of *ancillae typed* variables, which are guaranteed to be unchanged across function calls. Further, it can often be used to statically verify the *first match policy* that is essential to reversibility of partially defined functions. It should be noted that this type system is not meant to guarantee reversibility of well-typed programs (rather, guaranteeing reversibility is a job for the *semantics*). Instead, the type system aids in the clarity of expression for programs, provides fundamental well-behavedness guarantees, and is a source of additional static information which can enable static checking of certain properties, such as the aforementioned *first-match policy*.

An implementation of the work in this thesis can be found at:

https://github.com/diku-dk/coreFun/

1.1 Background

Initial studies of reversible (or information lossless) computation date back to the years around 1960. These studies were based on quite different computation models and motivations: Huffman studied information lossless finite state machines for their applications in data transmission [21], Landauer came to study reversible logic in his quest to determine the sources of energy dissipation in a computing system [27], and Lecerf studied reversible Turing machines for their theoretical properties [28].

Although the field is often motivated by a desire for energy and entropy preservation though the work of Landauer [27], we are more interested in the possibility to use reversibility as a property that can aid in the execution of a system, an approach which can be credited to Huffman [21]. It has since been used in areas like programming languages for quantum computation [17], parallel computing [39], and even robotics [40]. This diversity motivates studying reversible functional programming (and other paradigms) independently, such that we can get a better understanding of how to improve reversible programming in these diverse areas.

The earliest reversible programming language (to the authors' knowledge) is Janus, an imperative language invented in the 1980's, and later rediscovered [29, 48] as interest in reversible computation spread. Janus and languages deriving from it have since been studied in detail, so that we today have a reasonably good understanding of these kinds of reversible flowchart languages [15, 47].

Reversible functional programming languages are still at an early stage of development, and today only a few proof-of-concept languages exist. This work is founded on the initial work on RFun [42, 46], while another notable example of a reversible functional language is Theseus [23], which has recently been further developed towards a language for quantum computations [38].

The type system formulated here is based on relevance logic (originally introduced in [1], see also [12]), a substructural logic similar to linear logic [14, 44] which (unlike linear logic) permits the duplication of data. In reversible functional programming, linear type systems (see e.g. [23]) have played an important role in ensuring reversibility, but they also appear in modern languages like the Rust programming language [31]. To support ancillary variables at the type level, we adapt a type system inspired by Polakow's combined reasoning system of ordered, linear, and unrestricted intuitionistic logic [36].

The rest of this thesis is organised in the following way: In Sect. 2 we will first introduce CoreFun followed by the type system and operational semantics. We also discuss type polymorphism and show that the language is indeed reversible. In Sect. 3 we will show how the type system in some cases can be used to statically verify the first match policy. In Sect. 4 we discuss program inversion and present inverse operational semantics. Sect. 5 we show how syntactic sugar can be used to design a more modern style functional language from CoreFun. In Sect. 6 we briefly introduce a reference implementation. In Sect. 7 we discuss language design and future work. Finally in Sect. 8 we conclude.

Formalisation of CoreFun

2

The following section will present the formalisation of CoreFun. The language is intended to be minimal, but it will still accommodate future extensions to a modern style functional language. We first present a core language syntax, which will work as the base of all formal analysis. Subsequently we present typing rules and operational semantics over this language. The following is built on knowledge about implementation of type systems as explained in [35].

2.1 Grammar

A program is a collection of zero or more function definitions. Each definition must be defined over some number of input variables as constant functions are not interesting in a reversible setting. All function definitions will in interpretation be available though a static context. A typing of a program is synonymous with a typing of each function. A function is identified by a name f and takes 0 or more type parameters, and 1 or more formal parameters as inputs. Each formal parameter x is associated with a typing term τ at the time of definition for each function, which may be one of the type variables given as type parameter. The grammar is given in Fig. 2.1.

2.2 Type System

Linear logic is the foundation for linear type theory. In linear logic, each hypothesis must be used exactly once. Likewise, values which belong to a linear type must be used exactly once, and may not be duplicated nor destroyed. However, if we accept that functions may be partial (a necessity for *r-Turing completeness* [4]), first-order data may be duplicated reversibly. For this reason, we may relax the linearity constraint to relevance, that is that all available variables *must* be used at least once.

A useful concept in reversible programming is access to ancillae, i.e. values that remain unchanged across function calls. Such values are often used as a means to guarantee reversibility in a straightforward manner. To support such ancillary variables at the type level, a type system inspired by Polakow's combined reasoning system of ordered, linear, and unrestricted intuitionistic logic [36] is used. The type system splits the typing contexts into two parts: a static one (containing ancillary variables and other static parts of the environment), and a dynamic one (containing variables not considered ancillary). This gives a typing judgment of $\Sigma; \Gamma \vdash e : \tau$, where Σ is the static context and Γ is the dynamic context.

We discern between two sets of typing terms: primitive types and arrow types. This is motivated by a need to be careful about how we allow manipulation of functions, as we will treat all functions as statically known.

The grammar for typing terms can be seen in Fig. 2.2: τ_f denotes arrow types, τ primitive types, and τ_a ancillary types (i.e., types of data that may be given as ancillary data).

$q ::= d^*$	Program definition
$d ::= f\ \alpha^*\ v^+ = e$	Function definition
$e ::= x$	Variable name
$\mid ()$	Unit term
$\mid \mathbf{inl}(e)$	Left of sum term
$\mid \mathbf{inr}(e)$	Right of sum term
$\mid (e, e)$	Product term
$\mid \mathbf{let}\ l = e\ \mathbf{in}\ e$	Let-in expression
$\mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e, \mathbf{inr}(y) \Rightarrow e$	Case-of expression
$\mid f\ \alpha^*\ e^+$	Function application
$\mid \mathbf{roll}\ [\tau]\ e$	Recursive-type construction
$\mid \mathbf{unroll}\ [\tau]\ e$	Recursive-type deconstruction
$l ::= x$	Definition of variable
$\mid (x, x)$	Definition of product
$v ::= x : \tau_a$	Variable declaration

Figure 2.1: Grammar of CoreFun. Program variables are denoted by x, and type variables by α.

$$\begin{aligned}\tau_f &::= \tau_f \rightarrow \tau_f' \mid \tau \rightarrow \tau_f' \mid \tau \leftrightarrow \tau' \mid \forall X.\tau_f \\ \tau &::= 1 \mid \tau \times \tau' \mid \tau + \tau' \mid X \mid \mu X.\tau \\ \tau_a &::= \tau \mid \tau \leftrightarrow \tau'\end{aligned}$$

Figure 2.2: Typing terms. Note that X in this figure denotes any type variable.

Arrow types are types assigned to functions. For arrow types, we discern between primitive types and arrow types in the right component of unidirectional application. We only allow primitive types in bidirectional application. This is to restrict functions to only being able to be bound to ancillary parameters. This categorizes CoreFun as a restricted higher-order language in that functions may be bound to variables, but only if they are immediately known. It is ill-formed for a type bound in the dynamic context to be of an arrow type — if it were well-formed, we would be allowing the creation of new functions, which would break our assumption that all functions are statically known. We will detail the motivation for this restriction in Sect. 7.3.

Primitive types are types assigned to expressions which evaluate to canonical values by the big step semantics. These are distinctly standard, containing sum types and product types, as well as (rank-1) parametric polymorphic types[1] and a fix point operator for recursive data types (see [35] for an introduction to the latter two concepts).

Throughout this thesis, we will write $\tau_1 + \cdots + \tau_n$ for the nested sum type $\tau_1 + (\tau_2 + (\cdots + (\tau_{n-1} + \tau_n) \cdots))$ and equivalently for product types $\tau_1 \times \cdots \times \tau_n$. Similarly, as is usual, we will let arrows associate to the right.

2.2.1 Type Rules for Expressions.

The typing rules for expressions are shown in Fig. 2.3. A combination of features of the typing rules enforces relevant typing:

[1]A rank-1 polymorphic system may not instantiate type variables with polymorphic types.

(1) *Variable Typing Restriction:* The restriction on the contents of the dynamic context during certain type rules.

(2) *Dynamic Context Union:* A union operator for splitting the dynamic contexts in most type rules with more than one premise.

(3) *Context Update:* The assignment to the static context with new information instead of the dynamic when the dynamic context is empty.

The rules for application are split into three different rules, corresponding to application of dynamic parameters (T-App1), application of static parameters (T-App2), and type instantiation for polymorphic functions (T-PApp). Notice further the somewhat odd T-Unit-Elm rule. Since relevant type systems treat variables as resources that must be consumed, specific rules are required when data *can* safely be discarded (such as the case for data of unit type). What this rule essentially states is that *any* expression of unit type can be safely discarded; this is dual to the T-Unit rule, which states that the unique value () of unit type can be freely produced (i.e. in the empty context).

Variable Typing Restriction When applying certain axiomatic type rules (T-Var1 and T-Unit), we require the dynamic context to be empty. This is necessary to inhibit unused parts of the dynamic context from being accidentally "spilled" through the use of these rules. Simultaneously, we require that when we do use a variable from the dynamic context, the dynamic context contains exactly this variable and nothing else. This requirement prohibits the same sort of spilling.

Dynamic Context Union The union of the dynamic context is a method for splitting up the dynamic context into named parts, which can then be used separately in the premises of the rule. In logical derivations, splitting the known hypotheses is usually written as $\Gamma, \Gamma' \vdash \dots$, but we deliberately introduce a union operator to signify that we allow an overlap in the splitting of the hypotheses. Were we not to allow overlapping, typing would indeed be linear. For example, a possible split is:

$$\frac{\begin{array}{c}\vdots\\ \emptyset; x \mapsto 1, y \mapsto 1 \vdash \dots\end{array} \qquad \begin{array}{c}\vdots\\ \emptyset; y \mapsto 1, z \mapsto 1 \vdash \dots\end{array}}{\emptyset; x \mapsto 1, y \mapsto 1, z \mapsto 1 \vdash \dots}$$

Here y is part of the dynamic context in both premises.

Context Update We overload the rules for let and case-expressions depending on which context we are going to update with the variable assignments in these rules. This is motivated by what the form of the expression e we are assigning to the variable names is. If the dynamic context is empty, e is necessarily one of two things:

(1) A closed term. A canonical value constructed by a closed term is free as no information is consumed in its creation, allowing us to assign it to the static context instead.

(2) An open term with free variables from the static context. Since the static context may grow arbitrarily, and we have not consumed a dynamic variable in e, no information is lost by assigning the resulting canonical value to the static context instead.

Therefore we bind variables introduced in let and case-expressions to the static context when the dynamic context is empty for the derivation of e. The three instances of overloaded rules can be seen by T-Sum versus T-SumSt, T-Let1 versus T-Let1St and T-Let2 versus T-Let2St.

Alternatively we could have introduced a restricted notion of *weakening*. Weakening is not regularly supported in linear or relevant logic systems as it is not resource sensitive. But relevant logic dictates that we must only not forget information before use, otherwise we may use it freely. With restricted weakening, we say that if we already know a variable from the static environment, we may freely forget it in the dynamic environment, as the information is not lost.

This is more in line with expressing the idea that ancillae are static directly in a well typed program, as we explicitly require that each ancilla is built up again after we used it. However, it makes programs more long-winded. The proposed weakening rule is:

$$\text{T-Weakening:}\ \frac{\Sigma, x \mapsto \tau; \Gamma, x \mapsto \tau \vdash e : \tau'}{\Sigma, x \mapsto \tau; \Gamma \vdash e : \tau'}$$

2.2.2 Type Rules For Function Declarations.

The type rules for function declarations are shown in Fig. 2.4. Here T-PFun generalizes the type arguments, next T-Fun1 consumes the ancillary variables, and finally T-Fun2 handles the last dynamic variable by applying the expression typing.

We implicitly assume that pointers to all defined functions are placed in the static context Σ as an initial step. For example, when typing an expression e in a program where a function $f\ x = e$ is defined, and we have been able to establish that $\Sigma \Vdash f\ x = e : \tau \leftrightarrow \tau'$ for some types τ, τ', we assume that a variable $f : \tau \leftrightarrow \tau'$ is placed in the static context in which we will type e ahead of time. This initial step amounts to a typing rule for the full program.

Note that we write two very similar application rules T-App1 and T-App2. This discerns between function application of ancillary and dynamic data, corresponding to the two different arrow types. In particular, as shown in T-App1, application in the dynamic variable of a function is only possible when that function is of type $\tau \leftrightarrow \tau'$, where τ and τ' are non-arrow types: This specifically disallows higher-order functions. Also note that the dynamic context must be empty for application of the T-App2 rule — otherwise, it is treated as static in the evaluation of f, and we have no guarantee how it is used.

2.3 Recursive and Polymorphic Types

The type system of CoreFun supports both recursive types as well as rank-1 parametrically polymorphic types. To support both of these, type variables, which serve as holes that may be plugged by other types, are used.

For recursive types, we employ a standard treatment of iso-recursive types in which explicit **roll** and **unroll** constructs are added to witness the isomorphism between $\mu X.\tau$ and $\tau[\mu X.\tau/X]$ for a given type τ (which, naturally, may contain the type variable X). For a type τ, we let $\mathrm{TV}(\tau)$ denote the set of type variables that appear free in τ. For example, the type of lists of a given type τ can be expressed as the recursive type $\mu X.1 + (\tau \times X)$, and $\mathrm{TV}(1 + (\tau \times X)) = \{X\}$ when the type τ contains no free type variables. We define TV on contexts as $\mathrm{TV}(\Sigma) = \{v \in \mathrm{TV}(\tau) \mid x : \tau \in \Sigma\}$.

For polymorphism, we use an approach similar to System F, restricted to rank-1 polymorphism. In a polymorphic type system with rank-1 polymorphism, type variables themselves cannot be instantiated with polymorphic types, but must be instantiated with concrete types instead. While this approach

$$\boxed{\text{Judgement: } \Sigma;\Gamma \vdash e : \tau}$$

$$\text{T-Var1: } \frac{}{\Sigma;\emptyset \vdash x : \tau}\Sigma(x) = \tau \qquad \text{T-Var2: } \frac{}{\Sigma;(x \mapsto \tau) \vdash x : \tau}$$

$$\text{T-Unit: } \frac{}{\Sigma;\emptyset \vdash () : 1} \qquad \text{T-Unit-Elm: } \frac{\Sigma;\Gamma \vdash e : 1 \qquad \Sigma;\Gamma' \vdash e' : \tau}{\Sigma;\Gamma \cup \Gamma' \vdash e' : \tau}$$

$$\text{T-Inl: } \frac{\Sigma;\Gamma \vdash e : \tau}{\Sigma;\Gamma \vdash \mathbf{inl}(e) : \tau + \tau'} \qquad \text{T-Inr: } \frac{\Sigma;\Gamma \vdash e : \tau}{\Sigma;\Gamma \vdash \mathbf{inr}(e) : \tau' + \tau}$$

$$\text{T-Prod: } \frac{\Sigma;\Gamma \vdash e_1 : \tau \qquad \Sigma;\Gamma' \vdash e_2 : \tau'}{\Sigma;\Gamma \cup \Gamma' \vdash (e_1, e_2) : \tau \times \tau'}$$

$$\text{T-App1: } \frac{\Sigma;\Gamma \vdash f : \tau \leftrightarrow \tau' \qquad \Sigma;\Gamma' \vdash e : \tau}{\Sigma;\Gamma \cup \Gamma' \vdash f\ e : \tau'}$$

$$\text{T-App2: } \frac{\Sigma;\Gamma \vdash f : \tau_a \to \tau_f \qquad \Sigma;\emptyset \vdash e : \tau_a}{\Sigma;\Gamma \vdash f\ e : \tau_f}$$

$$\text{T-PApp:} \frac{\Sigma;\Gamma \vdash f : \forall\alpha.\tau_f}{\Sigma;\Gamma \vdash f\ \tau_a : \tau_f[\tau_a/\alpha]} \qquad \text{T-Let1: } \frac{\Sigma;\Gamma \vdash e_1 : \tau' \qquad \Sigma;\Gamma', x : \tau' \vdash e_2 : \tau}{\Sigma;\Gamma \cup \Gamma' \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\text{T-Let1St: } \frac{\Sigma;\emptyset \vdash e_1 : \tau' \qquad \Sigma, x : \tau';\Gamma, \vdash e_2 : \tau}{\Sigma;\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\text{T-Let2: } \frac{\Sigma;\Gamma \vdash e_1 : \tau' \times \tau'' \qquad \Sigma;\Gamma', x : \tau', y : \tau'' \vdash e_2 : \tau}{\Sigma;\Gamma \cup \Gamma' \vdash \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\text{T-Let2St: } \frac{\Sigma;\emptyset \vdash e_1 : \tau' \times \tau'' \qquad \Sigma, x : \tau', y : \tau'';\Gamma \vdash e_2 : \tau}{\Sigma;\Gamma \vdash \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\text{T-Sum: } \frac{\Sigma;\Gamma \vdash e_1 : \tau' + \tau'' \qquad \Sigma;\Gamma', x : \tau' \vdash e_2 : \tau \qquad \Sigma;\Gamma', y : \tau'' \vdash e_3 : \tau}{\Sigma;\Gamma \cup \Gamma' \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3 : \tau}$$

$$\text{T-SumSt: } \frac{\Sigma;\emptyset \vdash e_1 : \tau' + \tau'' \qquad \Sigma, x : \tau';\Gamma \vdash e_2 : \tau \qquad \Sigma, y : \tau'';\Gamma \vdash e_3 : \tau}{\Sigma;\Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3 : \tau}$$

$$\text{T-Roll: } \frac{\Sigma;\Gamma \vdash e : \tau'[\mu X.\tau'/X]}{\Sigma;\Gamma \vdash \mathbf{roll}\ [\mu X.\tau']\ e : \mu X.\tau'} \qquad \text{T-Unroll: } \frac{\Sigma;\Gamma \vdash e : \mu X.\tau}{\Sigma;\Gamma \vdash \mathbf{unroll}\ [\mu X.\tau]\ e : \tau'[\mu X.\tau/X]}$$

Figure 2.3: Expression typing.

is significantly more restrictive than the full polymorphism of System F, it is expressive enough that many practical polymorphic functions may be expressed (e.g. ML and Haskell both employ a form of rank-1 polymorphism based on the Hindley-Milner type system [33]), while being simple enough that type inference is often both decidable and feasible in practice.

$$\boxed{\text{Judgement: } \Sigma \Vdash d : \tau}$$

$$\text{T-Fun1: } \frac{\Sigma, x : \tau_a \Vdash f\ v^+ = e : \tau_f}{\Sigma \Vdash f\ x{:}\tau_a\ v^+ = e : \tau_a \to \tau_f} \qquad \text{T-Fun2: } \frac{\Sigma; (x \mapsto \tau) \vdash e : \tau'}{\Sigma \Vdash f\ x{:}\tau = e : \tau \leftrightarrow \tau'}$$

$$\text{T-PFun:} \frac{\Sigma \Vdash f\ \beta^*\ v^+ = e : \tau_f}{\Sigma \Vdash f\ \alpha\ \beta^*\ v^+ = e : \forall \alpha . \tau_f} \alpha \notin \text{TV}(\Sigma)$$

Figure 2.4: Function typing.

$$c ::= () \mid \textbf{inl}(c) \mid \textbf{inr}(c) \mid (c_1, c_2) \mid \textbf{roll}\ [\tau]\ c \mid x$$

Figure 2.5: Canonical forms.

2.4 Operational Semantics

We present a call-by-value big step operational semantics on expressions in Fig. 2.6, with canonical forms shown in Fig. 2.5. As is customary with functional languages, we use substitution (defined as usual by structural induction on expressions) to associate free variables with values (canonical forms). Since the language does not allow for values of a function type, we instead use an environment p of function definitions in order to perform computations in a context (such as a program) with all defined functions.

A common problem in reversible programming is to ensure that branching of programs is done in such a way as to uniquely determine in the backward direction which branch was taken in the forward direction. Since case-expressions allow for such branching, we will need to define some rather complicated machinery of *leaf expressions*, *possible leaf values*, and *leaves* (the latter is similar to what is also used in [46]) in order to give their semantics.

Say that an expression e is a *leaf expression* if it does not contain any subexpression (including itself) of the form $\textbf{let}\ l = e_1\ \textbf{in}\ e_2$ or $\textbf{case}\ e_1\ \textbf{of}\ \textbf{inl}(x) \Rightarrow e_2, \textbf{inr}(y) \Rightarrow e_3$; the collection of leaf expressions form a set, LExpr. As the name suggests, a leaf expression is an expression that can be considered as a *leaf* of another expression. The set of leaves of an expression e, denoted leaves(e), is defined in Fig. 2.7.

The leaves of an expression are, in a sense, an abstract over-approximation of its possible values, save for the fact that leaves may be leaf expressions rather than mere canonical forms. We make this somewhat more concrete with the definition of the *possible leaf values* of an expression e, defined as:

$$\text{PLVal}(e) = \{e' \in \text{LExpr} \mid e'' \in \text{leaves}(e), e' \rhd e''\} \tag{2.1}$$

Where the relation $- \rhd -$ on leaf expressions is defined inductively as (the symmetric closure[2] of):

[2]The symmetric closure of a binary relation on a set is the smallest symmetric relation on that set which contains the relation.

$\boxed{\text{Judgement: } p \vdash e \downarrow c}$

$$\text{E-Unit: } \frac{}{p \vdash () \downarrow ()} \qquad \text{E-Inl: } \frac{p \vdash e \downarrow c}{p \vdash \mathbf{inl}(e) \downarrow \mathbf{inl}(c)} \qquad \text{E-Inr: } \frac{p \vdash e \downarrow c}{p \vdash \mathbf{inr}(e) \downarrow \mathbf{inr}(c)}$$

$$\text{E-Roll: } \frac{p \vdash e \downarrow c}{p \vdash \mathbf{roll}\ [\tau]\ e \downarrow \mathbf{roll}\ [\tau]\ c} \qquad \text{E-Unroll: } \frac{p \vdash e \downarrow \mathbf{roll}\ [\tau]\ c}{p \vdash \mathbf{unroll}\ [\tau]\ e \downarrow c}$$

$$\text{E-Prod: } \frac{p \vdash e_1 \downarrow c_1 \qquad p \vdash e_2 \downarrow c_2}{p \vdash (e_1, e_2) \downarrow (c_1, c_2)} \qquad \text{E-Let: } \frac{p \vdash e_1 \downarrow c_1 \qquad p \vdash e_2[c_1/x] \downarrow c}{p \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \downarrow c}$$

$$\text{E-LetP: } \frac{p \vdash e_1 \downarrow (c_1, c_2) \qquad p \vdash e_2[c_1/x, c_2/y] \downarrow c}{p \vdash \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 \downarrow c}$$

$$\text{E-CaseL: } \frac{p \vdash e_1 \downarrow \mathbf{inl}(c_1) \qquad p \vdash e_2[c_1/x] \downarrow c}{p \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3 \downarrow c}$$

$$\text{E-CaseR: } \frac{p \vdash e_1 \downarrow \mathbf{inr}(c_1) \qquad p \vdash e_3[c_1/y] \downarrow c}{p \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3 \downarrow c}\, c \notin \mathrm{PLVal}(e_2)$$

$$\text{E-App: } \frac{p \vdash e_1 \downarrow c_1 \cdots p \vdash e_n \downarrow c_n \qquad p \vdash e[c_1/x_1,\ \cdots,\ c_n/x_n] \downarrow c}{p \vdash f\ \alpha_1 \cdots \alpha_m\ e_1\ \cdots\ e_n \downarrow c}\, p(f) = f\ \alpha_1 \cdots \alpha_m\ x_1 \cdots x_n = e$$

Figure 2.6: Big step semantics of CoreFun.

$$\begin{aligned}
() &\rhd () \\
(e_1, e_2) &\rhd (e_1', e_2') && \text{if} \quad e_1 \rhd e_1' \text{ and } e_2 \rhd e_2' \\
\mathbf{inl}(e) &\rhd \mathbf{inl}(e') && \text{if} \quad e \rhd e' \\
\mathbf{inr}(e) &\rhd \mathbf{inr}(e') && \text{if} \quad e \rhd e' \\
\mathbf{roll}\ [\tau]\ e &\rhd \mathbf{roll}\ [\tau]\ e' && \text{if} \quad e \rhd e' \\
e &\rhd x \\
e &\rhd f\ e_1\ \ldots\ e_n \\
e &\rhd \mathbf{unroll}\ [\tau]\ e'
\end{aligned} \tag{2.2}$$

As such, the set PLVal(e) is the set of leaf expressions that can be unified, in a certain sense, with a leaf of e. Since variables, function applications, and unrolls do nothing to describe the syntactic form of possible results, we define that these may be unified with *any* expression. As such, using PLVal(e) is somewhat conservative in that it may reject definitions that are in fact reversible. Note also that PLVal(e) specifically includes all canonical forms that could be produced by e, since all canonical forms are leaf expressions as well.

This way, if we can ensure that a canonical form c produced by a branch in a case-expression could not possibly have been produced by a *previous* branch in the case-expression, we know, in the backward direction, that c must have been produced by the current branch. This is precisely the reason for the

$$
\begin{aligned}
\text{leaves}(()) &= \{()\} \\
\text{leaves}((e_1, e_2)) &= \{(e_1', e_2') \mid e_1' \in \text{leaves}(e_1), \\
&\qquad\qquad\qquad e_2' \in \text{leaves}(e_2)\} \\
\text{leaves}(\textbf{inl}(e)) &= \{\textbf{inl}(e') \mid e' \in \text{leaves}(e)\} \\
\text{leaves}(\textbf{inr}(e)) &= \{\textbf{inr}(e') \mid e' \in \text{leaves}(e)\} \\
\text{leaves}(\textbf{roll}\ [\tau]\ e) &= \{\textbf{roll}\ [\tau]\ e' \mid e' \in \text{leaves}(e)\} \\
\text{leaves}(\textbf{let}\ l = e_1\ \textbf{in}\ e_2) &= \text{leaves}(e_2) \\
\text{leaves}(\textbf{case}\ e_1\ \textbf{of}\ \textbf{inl}(x) \Rightarrow e_2, \textbf{inr}(y) \Rightarrow e_3) &= \text{leaves}(e_2) \cup \text{leaves}(e_3) \\
\text{leaves}(x) &= \{x\} \\
\text{leaves}(\textbf{unroll}\ [\tau]\ e) &= \{\textbf{unroll}\ [\tau]\ e' \mid e' \in \text{leaves}(e)\} \\
\text{leaves}(f\ e_1\ \ldots\ e_n) &= \{f\ e_1'\ \ldots\ e_n' \mid e_i' \in \text{leaves}(e_i)\}
\end{aligned}
$$

Figure 2.7: Definition function that computes the leaves of a program.

side condition of $c \notin \text{PLVal}(e_2)$ on E-CaseR, as this conservatively guarantees that c could not have been produced by the previous branch.

It should be noted that for iterated functions[3] this may add a multiplicative execution overhead that is equal to the size of the data structure. This effect has previously been shown in [41], where a `plus` function over Peano numbers, which was linear recursive over its input, actually had quadratic runtime.

It is immediate that should the side condition not hold for an expression e', no derivation is possible, and the expression does not evaluate to a value entirely. In Sect. 3, we will look at exactly under which conditions we can *statically* guarantee that the side condition will hold for every possible value of a function's domain.

We capture the conservative correctness of our definition of PLVal(e) with respect to the operational semantics — i.e. the property that any canonical form c arising from the evaluation of an expression e will also be "predicted" by PLVal in the sense that $c \in \text{PLVal}(e)$ — in the following theorem:

Theorem 1. *If $p \vdash e \downarrow c$ then $c \in PLVal(e)$.*

Proof. By induction on the structure of the derivation of $p \vdash e \downarrow c$. The proof is mostly straightforward: The case for E-Unit follows trivially, as do the cases for E-Unroll and E-App since leaves of **unroll** $[\tau]$ e (respectively $f\ e_1\ \cdots\ e_n$) are all of the form **unroll** $[\tau]$ e' (respectively $f\ e_1'\ \cdots\ e_n'$), and since $e'' \rhd$ **unroll** $[\tau]$ e' (respectively $e'' \rhd f\ e_1'\ \cdots\ e_n'$) for *any* choice of e'', it follows that PLVal(**unroll** $[\tau]$ e') = PLVal($f\ e_1\ \cdots\ e_n$) = LExpr. The cases for E-Inl, E-Inr, E-Roll, and E-Prod all follow straightforwardly by induction, noting that PLVal(**inl**(e)) = $\{$**inl**$(e') \mid e' \in \text{PLVal}(e)\}$, and similarly for **inr**(e), (e_1, e_2), and **roll** $[\tau]$ e. This leaves only the cases for **let** and **case** expressions, which follow using Lemma 1.

□

Lemma 1. *For any expression e, variables $x_1, \ldots, x_n$, and canonical forms $c_1, \ldots, c_n$, we have $PLVal(e[c_1/x_1, \ldots, c_n/x_n]) \subseteq PLVal(e)$.*

[3]An iterated function f^n is a function f composed with itself n times.

Proof. This lemma follows straightforwardly by structural induction on e, noting that it suffices to consider the case where e is open with free variables $x_1, \ldots, x_n$, as it holds trivially when e is closed (or when its free variables are disjoint from $x_1, \ldots, x_n$). With this lemma, showing the case for e.g. E-LET is straightforward since $c \in \text{PLVal}(e_2[c_1/x])$ by induction, and since $\text{PLVal}(e_2[c_1/x]) \subseteq \text{PLVal}(e_2)$ by this lemma, so $c \in \text{PLVal}(e_2) = \text{PLVal}(\textbf{let } x = e_1 \textbf{ in } e_2)$ by $\text{leaves}(\textbf{let } x = e_1 \textbf{ in } e_2) = \text{leaves}(e_2)$. □

2.5 Reversibility

Showing that the operational semantics are reversible amounts to showing that they exhibit both forward and backward determinism. Showing forward determinism is standard for any programming language (and holds straightforwardly in CoreFun as well), but backward determinism is unique to reversible programming languages. Before we proceed, we recall the usual terminology of *open* and *closed* expressions: Say that an expression e is closed if it contains no free (unbound) variables, and open otherwise.

Unlike imperative languages, where backward determinism is straightforwardly expressed as a property of the reduction relation $\sigma \vdash c \downarrow \sigma'$ where σ is a store, backward determinism is somewhat more difficult to express for functional languages, as the obvious analogue — that is, if $e \downarrow c$ and $e' \downarrow c$ then $e = e'$ — is much too restrictive (specifically, it is obviously *not* satisfied in all but the most trivial reversible functional languages). A more suitable notion turns out to be a *contextual* one, where rather than considering the reduction behaviour of closed expressions in themselves, we consider the reduction behaviour of canonical forms in a given *context* (in the form of an open expression) instead.

Theorem 2 (Contextual Backwards Determinism)**.** *For all open expressions e with free variables $x_1, \ldots, x_n$, and all canonical forms $v_1, \ldots, v_n$ and $w_1, \ldots, w_n$, if $p \vdash e[v_1/x_1, \ldots, v_n/x_n] \downarrow c$ and $p \vdash e[w_1/x_1, \ldots, w_n/x_n] \downarrow c$ then $v_i = w_i$ for all $1 \leq i \leq n$.*

Proof. The proof of this theorem follows by induction on the structure of e. For brevity, we sometimes denote the set $x_1, \ldots, x_n$ as X. The proof is mostly straightforward:

- Case $e = ()$. This follows immediately as there are no free variables in e.
- Case $e = x$. The only possible substitution is for the free variable x, and the only possible inference rule is E-VAR. We have $x[v/x] \downarrow v$ and $x[w/x] \downarrow w$, so $v = w = c$.
- Case $e = \textbf{inl}(e')$. Follows from using the Induction Hypothesis on e' and using the E-INL rule to construct a derivation for e. The proof is identical for when $e = \textbf{inr}(e')$, when $e = \textbf{roll}\ [\tau]\ e'$ and when $e = \textbf{unroll}\ [\tau]\ e'$.
- Case $e = (e', e'')$. The only possible rule for the derivation is E-PROD. First, we must consider which open terms occur in e' and e'' respectively. We have some subset $Y = y_1, \ldots, y_k \in X$, of open terms occurring in e', and likewise, we have some subset $Z = z_1, \ldots, z_l \in X$ occurring e'', such that $Y \cup Z = X$. Note that any y_i or z_i is simply an alias for some x_j, they are not fresh variables. By the Induction Hypothesis on e' and e'' we prove it for both premises. Now, what about any $x_k \in Y \cap Z$? Any value v_k has not been proven to be the same as w_k, but it certainly has to hold. But we have this by the inductive definition of substitution:

$$(e', e'')[c/x] = (e'[c/x], e''[c/x])$$

- Case $e = \textbf{let } x = e' \textbf{ in } e''$. We forego the argument about the partitioning of open terms as in the case before, but take note that it holds here as well. We first prove it for e' by the Induction Hypothesis on e'.

  Now, we have by assumption that the expressions $(\textbf{let } x = e_1 \textbf{ in } e_2)[v_1/x_1, \dots, v_n/x_n]$ and $(\textbf{let } x = e_1 \textbf{ in } e_2)[w_1/x_1, \dots, w_n/x_n]$ evaluate to the same canonical value c. Thus the theorem holds for the only free substitution (of x) in e'', and combined with the Induction Hypothesis on $x_1, \dots, x_n$ it holds for e as a whole.

- Case $e = f\ \alpha_1 \dots \alpha_m\ e_1 \dots e_q$. We have that $p(f) = f\ \alpha_1 \dots \alpha_m\ y_1 \dots y_q = e'$. There are $q + 1$ premises and thus $q + 1$ possible partitions of the open terms. Luckily, the theorem holds by simply applying the Induction Hypothesis on the q first premises, followed by noticing that every other free variable in e' (variables $y_1, \dots, y_q$) must be the substituted for the same set of canonical values by the theorem assumption — this paired with an application of the Induction Hypothesis gives us what we want.

- Case $e = \textbf{case } e' \textbf{ of } \textbf{inl}(x) \Rightarrow e'', \textbf{inr}(y) \Rightarrow e'''$. We may use two inference rules, based on the derivation of e'. Induction on e' derives a value c (where e' contains some subset $Y = y_1, \dots, y_k \in X$ of free variables). The possible forms of the value c are:

  - $c' = \textbf{inl}(x)$. By the Induction Hypothesis on e'' with the E-CaseL rule, we have what we want, with e'' evaluating to some c', where e'' contains some set of open terms $A \in X$.
  - $c' = \textbf{inr}(y)$. By the Induction Hypothesis on e''' with the E-CaseR rule, we have that the theorem holds for e''' with open terms $B = a_1, \dots, a_l \in x_1, \dots, x_n$, such that $Y \cup B = X$, where e''' evaluates to some c''.

    Both evaluation rules are over the same expression: This means the expression contain the same set of open terms Y in the first premise and not necessarily the same set of open terms A and B in the second premise. By the Induction Hypothesis on e''' we have contextual backwards determinism for the premise e''', but there is no way of guaranteeing that the substitutions of A and B are equivalent, and they can both evaluate to the same closed term where $c' = c''$.

    At this point we invoke the side condition of E-CaseR, which specifically states that should a derivation using the E-CaseR inference rule be possible so that $c' = c''$, then the derivation of e'' to c' must not exist.

And we have covered all the cases.

□

Corollary 1 (Injectivity)**.** *For all functions f and canonical forms v, w, if $p \vdash f\ v \downarrow c$ and $p \vdash f\ w \downarrow c$ then $v = w$.*

Proof. Let e be the open expression $f\ x$ (with free variable x). Since $(f\ x)[v/x] = f\ v$ and $(f\ x)[w/x] = f\ w$, applying Theorem 2 on e yields precisely that if $p \vdash f\ v \downarrow c$ and $p \vdash f\ w \downarrow c$ then $v = w$. □

Statically Checking the First Match Policy 3

The side condition in the E-CaseR rule is essential when ensuring reversibility of partial functions. The authors of the original RFun paper dubbed their very similar mechanism the *First Match Policy* [46], a convention we will follow. It is, unfortunately, a property that can only be fully guaranteed at run-time; from Rice's theorem we know that all non-trivial semantic properties of programs are undecidable [37]. However, with the type system, we can now in many cases resolve the first match policy statically.

Overall we may differentiate between two notions of divergence:

1. A function may have inputs that do not result in program termination.
2. A function may have inputs for which it does not have a defined behaviour; this could be the result of missing clauses.

Note that the semantics of CoreFun dictate that if a computation does not terminate in the forward direction, no input in the backwards direction may evaluate to this input. Dual for backwards computations.

Proof. Assume that termination is not a symmetric property. Consider some function $f\ x_1 \dots x_n$ which diverges in the forward direction with applied values $c_1 \dots c_n$. Now, there exists an f such that we should be able to find some input i to the inverse function such that $f^{-1}\ c_1 \dots i \downarrow c_n$ — that is, the diverging input in the forward direction. Since the inverse direction converges, we have determined a result in the forward direction, which is a contradiction. A dual argument is valid in the backward direction.

□

Termination analysis (1.) is not what we will detail here, but rather inputs for which the function is not defined (2.). Because the first match policy is enforced by the operational semantics, it follows that whenever an application of the Case-R rule fails its side condition, the expression cannot be derived, which by extension means the function is not defined for this input.

Definition 1. *The domain of a function f is the subset of the n-ary Cartesian product of canonical values which are inhabitants of the type of each parameter, for which f evaluates to a closed term:*

$$dom(f\ (x_1 : \tau_1) \dots (x_n : \tau_n)) = \{(c_1, \dots, c_n) \mid c_1 \in \tau_1, \dots, c_2 \in \tau_n, f\ c_1 \dots c_n \downarrow c\} \tag{3.1}$$

$$
\begin{array}{lll}
e ::= & & \\
& \vdots & \\
& \mid \textbf{case } e \textbf{ of inl}(x) \Rightarrow e, \textbf{inr}(y) \Rightarrow e \textbf{ safe } e & \text{Safe case-expression} \\
& \mid \textbf{case } e \textbf{ of inl}(x) \Rightarrow e, \textbf{inr}(y) \Rightarrow e \textbf{ unsafe} & \text{Unsafe case-expression}
\end{array}
$$

Figure 3.1: New case-expression syntax

Intuitively, some combination of inhabitants of the types $\tau_1 \dots \tau_n$ applied to f might fail to evaluate to a closed term because of either (1.) or (2.) (they are not in the domain of f.) We wish to investigate exactly when we can or when we cannot *guarantee* that a derivation of a case-expression in f is going to uphold the first match policy for *all* elements in the domain of f. To a certain degree this is reminiscent of arguing for the *totality* of the function, up to termination. Contrary to the argument for termination, this property of a first match policy guarantee is not symmetric: More specifically, a function f and its inverse function f^{-1} might not accommodate this restricted notion of totality on their respective domains, although it is certainly possible.

This analysis is only possible due to the type system, as we define the domain based directly on the type of the function — it formally hints us at the underlying sets of values which occur in case-expressions, something which is only possible when terms are given types.

3.1 Benefits of First Match Policy Assertions

A pitfall with a language using a first-match policy to guarantee reversibility is the added cost of making sure the first match policy is being met at runtime. It establishes a need to be attentive when writing programs to not write case-expressions whose side condition check adds a multiplicative overhead to the function's run time, potentially increasing the asymptotic complexity of the algorithm. This may for example occur when the side condition invocation involves traversing the full structure of the input, making each iteration linear. With a first match policy assertion provided, the side condition check may be substituted with a (hopefully) simpler assertion, increasing computational efficiency.

As an additional benefit, an assertion which is expressive can be seen as an enhancement of clarity of the behaviour of the case-expression to which it belongs. It also simplifies the process of inverse evaluation of that case expression by adding additional structure to the form of the left and right arm, as will be discussed in Sect. 4.

3.2 Adding First Match Policy Assertions to the Formal Language

The first match policy is ultimately enforced with the addition of the side condition of the Case-R type rule. The issue is that it compels a computation of the PLVal set on e_1 at every application of this rule. The present ambition is to express an alternative method of ensuring reversibility of branching. We introduce two new syntactic constructs to CoreFun: *Safe* and *unsafe* case-expressions. These are presented in the grammar in Fig. 3.1.

Safe case-expressions A safe case-expression is augmented with a *safe exit assertion.* Consider an expression **case** e_1 **of** $x \Rightarrow e_2, y \Rightarrow e_3$ **safe** e_{out}. It omits the first match policy check and instead checks the resulting value c of the forward evaluation of the case-expression against a static predicate e_{out} at run time (that is, an expression which evaluates to a Boolean type $1 + 1$.) Since it is determined by the syntax of a case-expression that the expression e_1 cased over is of a binary sum type, it suffices to define

$$\text{T-S{\scriptsize UM}U:}\ \frac{\begin{array}{cc}\Sigma;\Gamma \vdash e_1 : \tau' + \tau'' & \Sigma;\Gamma', x : \tau' \vdash e_2 : \tau \\ \Sigma;\Gamma', y : \tau'' \vdash e_3 : \tau & \Sigma; \mathit{out} : \tau \vdash e_a : 1 + 1\end{array}}{\Sigma;\Gamma \cup \Gamma' \vdash \textbf{case}\ e_1\ \textbf{of}\ \textbf{inl}(x) \Rightarrow e_2, \textbf{inr}(y) \Rightarrow e_3\ \textbf{safe}\ e_a : \tau}$$

Figure 3.2: New typing rule for a case-expression with an exit assertion.

a single predicate which should always hold after e_2 has been evaluated and by implication always *not* hold when e_3 has been evaluated. The safe exit assertions we propose here preserve reversibility much like Janus — Janus includes conditionals and loops augmented with an additional exit assertion, making its control flow constructs completely symmetric. Inverse evaluation of conditionals and loops in Janus is then as simple as exchanging the entry and exit assertions in the backwards direction (while also reversing the statement blocks contained in these.)

We require a new type rule for the case-expression to support safe case expressions. It is highly similar to the preexisting T-Sum rule, but with the added criteria that the type of the safe exit assertion is $1 + 1$. We introduce a variable *out* in the assertion, which assumes the value of the result of the case-expression — granting the programmer the possibility to make the exit assertion depend on the value produced by the case-expression. The new type rule can be seen in Fig. 3.2.

We also need two new evaluation rules for the big step semantic. These differ from the conventional E-CaseR and E-CaseL rules in that we enforce the added assertion through a premise which demands to be evaluated to either $\textbf{inr}(())$ or $\textbf{inl}(())$, depending on which branch was taken. Meanwhile, the side condition is removed. Note that with an exit assertion, matching on a left branch is stricter than the conventional behaviour exhibited by E-CaseL as it might fail.

Unsafe case-expressions An *unsafe case-expression* is augmented with the **unsafe** keyword. The result of the forward evaluation of a case-expression is not subjected to any validation check when the right branch is taken. This means it allows more programs, including programs which are not injective. It is thus only reversible if the correct branch can be discerned in the backwards direction unanimously by its syntactic form. Therefore we do not allow unsafe case-expressions into the syntax directly. Rather, we only allow statically generated unsafe case-expressions. No new type rule is needed to eliminate unsafe case-expressions as typechecking occurs before static analysis. Operationally, the only difference in behaviour from the E-CaseR rule is that the side condition is omitted. Unsafe case-expression are inserted into the program by the static analysis we will present in the next Sect. 3.3.

The operational rules for safe and unsafe case-expressions are shown in Fig. 3.3. As already hinted at, the unsafe case-expression strictly speaking is not actually reversible, so we cannot prove Theorem 2 if we adopt them as an expression form directly — but we will informally say that in any circumstance where we *do* insert them, they are safe enough to maintain reversibility.

We still want to prove reversibility for case-expressions with safe exit assertions. Below, we make sure that they actually maintain the reversibility of CoreFun by extending the proof for Theorem 2.

Theorem 2 (Continued)**.**

Proof. We first reassure ourselves that the proof for the previously shown forms of e remain valid, which is given to us by Induction Hypothesis directly. We now consider the newly added case for e, which is missing to complete the proof:

- Case $e = \textbf{case}\ e_1\ \textbf{of}\ \textbf{inl}(x') \Rightarrow e_2, \textbf{inr}(y') \Rightarrow e_3\ \textbf{safe}\ e_a$

$$\text{E-C{\scriptsize ASE}SL:}\ \frac{p \vdash e_1 \downarrow \mathbf{inl}(c_1) \qquad p \vdash e_2[c_1/x] \downarrow c \qquad p \vdash e_a[c/out] \downarrow \mathbf{inl}(())}{p \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3\ \mathbf{safe}\ e_a \downarrow c}$$

$$\text{E-C{\scriptsize ASE}SR:}\ \frac{p \vdash e_1 \downarrow \mathbf{inr}(c_1) \qquad p \vdash e_3[c_1/y] \downarrow c \qquad p \vdash e_a[c/out] \downarrow \mathbf{inr}(())}{p \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3\ \mathbf{safe}\ e_a \downarrow c}$$

$$\text{E-C{\scriptsize ASE}UL:}\ \frac{p \vdash e_1 \downarrow \mathbf{inl}(c_1) \qquad p \vdash e_2[c_1/x] \downarrow c}{p \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3\ \mathbf{unsafe}\ \downarrow c}$$

$$\text{E-C{\scriptsize ASE}UR:}\ \frac{p \vdash e_1 \downarrow \mathbf{inr}(c_1) \qquad p \vdash e_3[c_1/y] \downarrow c}{p \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3\ \mathbf{unsafe}\ \downarrow c}$$

Figure 3.3: New big step semantic rules for case-expressions with assertions.

There are two possible derivations of e:

- By E-CaseSL. By Induction Hypothesis on e_1 we get a contextually deterministic derivation for e_1, evaluating to c_1. By application of the Induction Hypothesis on $e_2[c_1/x]$ we derive a value $c' = c'' = c$. Now, as the semantics are deterministic, by the Induction Hypothesis on $e_a[c/out]$, we can only ever derive the same value c_a by the Theorem assumption — specifically $\mathbf{inl}(())$ if we are going to infer the conclusion by the E-CaseSL rule.
- By E-CaseSR. The proof is analogous to the previous case.

And we are done.

□

3.3 First Match Policy over Open Terms

In the following two sections we discuss the possibility of asserting that an unsafe static assertion may be defined for a case-expression.

Intuitively, when the range of a function call is well-defined (typed), and all the leaves are disjoint, it is clear that any evaluated term will not match any other leaf. For example, the following function performs a transformation on a sum term. It is immediately obvious that its leaves are disjoint; it either evaluates to $\mathbf{inl}(\cdot)$ or to $\mathbf{inr}(\cdot)$:

```
f x : 1 + τ = case x of
                inl(()) ⇒ inr(())
                inr(y) ⇒ inl(y)
```

In Sect. 2.4, when we defined the operational semantics (cf. Fig. 2.6), the first match policy was upheld for a case expression by checking that the closed term c of the evaluation of the second branch ($\mathbf{inr}(\cdot)$) could not be a possible leaf value of the first branch ($\mathbf{inl}(\cdot)$). However, the above example includes an open term that is defined over y.

Given the existing definition of the unification relation $- \rhd -$ (Def. 2.2), this is actually easy to alleviate with regards to the static analysis. $- \rhd -$ has already been defined to take *any* term, both open and closed terms. Thus, in the general case, all we have to ensure is that all leaves of either branch do not have a possible value in the other branch.

Said otherwise, we wish to cross compare the sets of possible leaf expressions of each branch. For a case-expression **case** e_1 **of** $x \Rightarrow e_2, y \Rightarrow e_3$, e_2 and e_3 each form a set of leaves, and by taking the Cartesian product of these sets, we can see if any leaf expressions unify pair-wise. This can all be described as:

$$\{(l_2, l_3) \mid l_2 \in \text{leaves}(e_2), l_3 \in \text{leaves}(e_3), l_2 \rhd l_3\} = \emptyset \tag{3.2}$$

Because $- \rhd -$ is a symmetric relation, we may interchange the operands and still describe the same thing. Obviously, this static analysis is quite restricted because most terms will be open, and open terms unify very broadly.

3.4 First Match Policy over Closed Terms

Yet, we have not incorporated the type system into our analysis. We now investigate an alternative method — concretely examining the domain of a function we are trying to write a first match policy guarantee for. This has a couple of benefits: Critically, we wish to avoid how the static leaves sets of each branch in a case-expression often will contain open terms (as these unify with anything, *even though* the term might always have a simple, predictable form which unifies well.)

The principle behind exhausting the domain is comparing the unions of possible leaf values of each branch for a case-expression for the complete domain of the function. Thus, the leaves sets being compared will potentially be much more populous, *but* they will exclusively consist of closed terms, which in the end will more truthfully reflect how the function may be applied. In the following, let the case-expression $e_c =$ **case** e_1 **of** $x \Rightarrow e_2, y \Rightarrow e_3$ be a subexpression of e for the function $f\ (x_1 : \tau_1) \dots (x_n : \tau_n) = e$.

We respectively define the union of the left and right leaves sets for e_c as Eq. 3.3 and Eq. 3.4. e_2' is derived from e_2 with substitutions occurring as for the application of f with the values $c_1, \dots, c_n$ up until a case-rule is to be applied to e_c (by the operational semantics.) Equivalently for e_3' with e_3. We then adopt a notion of comparability between leaf sets similar to how it was defined over open terms in Eq. 3.5.

$$\text{leaves}_l = \bigcup_{\substack{(c_1,\dots,c_n) \\ \in \text{dom}(f)}} \text{leaves}(e_2') \tag{3.3}$$

$$\text{leaves}_r = \bigcup_{\substack{(c_1,\dots,c_n) \\ \in \text{dom}(f)}} \text{leaves}(e_3') \tag{3.4}$$

$$\{(l_2, l_3) \mid l_2 \in \text{leaves}_l, l_3 \in \text{leaves}_r, l_2 \rhd l_3\} = \emptyset \tag{3.5}$$

The method is obviously only tangible when it converges, which requires two characteristics of the function: for the domain of the function to be finite and for the function to terminate on any inhabitant of its domain. We require the domain to be finite as we actually want to compute the full set of possible leaves, and with an infinite domain, we have to try infinitely many inputs. We require that any computation terminates as otherwise there is a possibility that we get stuck computing the leaves for the same instance of a function application indefinitely. These two requirements are not mutually inclusive, as can be shown by an example:

$$f\ (x:1) = \textbf{let}\ y = f\ x\ \textbf{in}\ y$$

It is immediate that the domain of a function f is finite iff. none of f's parameters comprise a recursive type. Note that a parameter of a polymorphic type is general enough to be instantiated as a recursive type and is also prohibited. This property is immediate from the fact that we can construct a recursive value of an infinitely long chain of nested **roll** $[\mu X.\tau] \cdot$ terms of type $\mu X.\tau$. Any other type must necessarily be constructed by values of strictly decreasing constituent types with 1 as the bottom type. Even function applications in f of functions which return forms of recursive types can only take on a finite set of values if the domain of f is finite due to referential transparency.

Deciding if a function will terminate on any given input requires *termination analysis.* In general this property is described as the halting problem, a famous undecidable problem [43]. A lot of effort has been spent on statically guaranteeing termination for an increasing class of functions in all paradigms of programming. These often exploit a static hint of guaranteed decreases of input size, using term rewriting [13] or calling context graphs [30]. Since we forbid infinite domains, we do not expect functions not to terminate, so we may get away with a much simpler analysis.

A method that is conservative but easily proven to be correct is constructing a directed *computation graph.* It is defined as follows: Each vertex is a function name $f, g, \ldots$ and each edge $f \to g, \ldots$ between vertices f and g implies there is a function application of g in the body of f. For each function m we wish to analyze termination for, we look at the reachability relation of every function that m is related to to see if they are reflexive. This is equivalent to checking for cycles in the computation graph. If no cycles exist, termination is guaranteed.

Finally, both of the aforementioned analyses can be performed statically, making them a relatively simple but attainable practice.

Evaluating Backwards

4

For any computable function $f : X \to Y$ and x such that $f(x) = y$, we can derive the inverse function $f' : Y \to X$ such that $f'(y) = x$, by simply trying every possible $x \in X$ for f until we find one for which f computes y. This works as long as f is decidable and is known as the Generate-and-test method [32]. It is inherently inefficient though, as we need to enumerate all possible values of $x \in X$. Alternatively, any program can be made reversible by embedding a trace of information that would otherwise be destroyed, restoring enough information so as to make any computation injective. A Landauer embedding turns a deterministic TM into a two-tape deterministic RTM and the Bennett method turns a TM into a three-tape RTM, both describing the same problem [3, 5]. Predictably however, multi-tape RTMs converted to 1-tape RTM, are less efficient [2] than their original counterparts.

The strength of reversible languages lies in their design which enforces injectivity of their computable functions. In the context of reversible languages, a *program inverter* writes the inverse program p^{-1} from a program p. Program inversion is *local* if it requires no global analysis to perform (syntactic constructs can be inverted locally) and this is usually shown by a set of inverters $\mathcal{I}_x, \ldots, \mathcal{I}_z$, one for each syntactic domain $x, \ldots, z$ of the language's grammar. As an example, if we were to present program inversion for CoreFun, it would be natural to present a program inverter $\mathcal{I}_p$, a function inverter $\mathcal{I}_f$ and an expression inverter $\mathcal{I}_e$. A pleasant property of program inversion is maintaining the program size and asymptotic complexity.

Unlike some structural reversible languages like Janus, the semantics of CoreFun are not trivially locally reversed — the central issue when presenting a program inverter for a language like CoreFun is figuring out how to write deterministic control flow for the inverse program. Writing it requires "unification" analysis of the leaves of the whole function structure, which is difficult.

General methods have been proposed (some of which were later used to define a program inverter for RFun) for writing program inverters for reversible functional languages [16, 26], but the existence of both an equality/duplication operator and constructor terms as atoms is assumed, both of which are omitted from CoreFun.

Instead of presenting a program inverter, inverse evaluation for CoreFun is going to rely on dedicated big step inverse semantics, defined over the original program's syntax. Although evaluating inverse function calls in Janus most often is achieved by calling the inverse function produced by the Janus program inverter directly, big step inverse semantics exist for Janus as well [34].

An interesting observation is that program inversion as a method for reversibility is less expressive than the inverse semantics we will present in Sect. 4.2. Specifically, we can accommodate duplication of data (in the forward direction) in the inverse semantics by the I-Bind2 rule. Meanwhile, the type system prohibits us from writing the inverse program of a function which duplicates data, as we are lacking native support for the equality/duplication operator. See:

$$
\begin{array}{ll}
e ::= & \\
\quad \vdots & \\
\quad \mid f!\ \alpha^*\ e^+ & \text{Inverse function call}
\end{array}
$$

Figure 4.1: Added inverse function call to expressions in the grammar.

$$\mathcal{I}_f[\![(x:\tau) = (x,x)]\!] \overset{\text{def}}{=} f^{-1}\ (x' : \tau \times \tau) = \mathcal{I}_e[\![x']\!]$$

How would we translate $\mathcal{I}_e[\![x']\!]$? The forward program is legal — we adopted a relevant type system precisely because we embraced the partiality of functions, so the forward function may be defined while the inverse function may not. Now, the closest program we can produce is:

$$\mathcal{I}_f[\![(x:\tau) = (x,x)]\!] \overset{\text{def}}{=} f^{-1}\ (x' : \tau \times \tau) = \textbf{let}\ (x,y) = x'\ \textbf{in}\ x$$

Wherein we attempt to destroy y after decomposing the product, as they should be equal anyway — however we may never implicitly assume that the two values in the decomposed product should be equal, so the program is not well-formed. We say that CoreFun is not *closed under program inversion* [25] when we cannot produce the inverse program from the language's grammar directly.

4.1 Supporting Inverse Function Application

Naturally, the inverse application of a function requires the full extent of information which was returned in the forward direction, and since ancillae parameters remain constant across application, the same set of values is supplied for ancillae parameters in the forward and backward direction. The relationship between a function f and its inverse can be described as:

$$f\ a_1 \dots a_n\ x = y \iff f^{-1}\ a_1 \dots a_n\ y = x \tag{4.1}$$

For typing of a program when inverse function applications are supported, we may deduce the type of the inverse function from the static context Σ, which contains the usual set of statically declared functions. The inverse function f^{-1} mirrors the type of f but with the type of the dynamic parameter τ_n and the return type τ_{out} swapped around the bidirectional arrow. For example, if a program contains the following function f, we can deduce the type of f^{-1}:

$$f : \tau_1 \to \dots \to \tau_{n-1} \to \tau_n \leftrightarrow \tau_{out} \iff f^{-1} : \tau_1 \to \dots \to \tau_{n-1} \to \tau_{out} \leftrightarrow \tau_n \tag{4.2}$$

We introduce a new piece of syntax to support inverse applications. We say that to apply the inverse of a function, the programmer should append the original function name with an exclamation mark. So in the grammar an application $f^{-1}\ a_1 \dots a_n$ is denoted as $f!\ a_1 \dots a_n$. The addition to the grammar is highlighted in Fig. 4.1.

4.1.1 Typing and Evaluation of Inverse Function Applications

Typing of an inverse application is identical to a regular application, with the signature of the inverse function stored in the static context as seen in Eq. 4.2.

The big step rule for an inverse function application uses the inverse semantics defined in Sec. 4.2, with the original function stored in p to produce a canonical term, and is presented in Fig. 4.2.

$$\text{E-InvApp: } \frac{\begin{array}{ll} p(f) = \alpha_1 \dots \alpha_m \; x_1 \dots x_n = e & p \vdash e_1 \downarrow c_1 \cdots p \vdash e_n \downarrow c_n \\ p; \emptyset \vdash^{-1} e[c_1/x_1, \dots, c_{n-1}/x_{n-1}], c_n \rightsquigarrow \sigma & \sigma(x_n) = c \end{array}}{p \vdash f! \; \alpha_1 \dots \alpha_m \; e_1 \dots e_n \downarrow c}$$

Figure 4.2: New big step rules related to inverse application.

4.2 Inverse Semantics

We now present the inverse big step semantics of CoreFun. The semantics should satisfy that, given a function f with body e, which is to be run in inverse, and a value c_{out}, which is the dynamic input to f^{-1}, we may derive a value c_{in} which was supplied as the dynamic input to f so that $f \; e_1 \dots e_n \; c_{in} = c_{out}$.

Using substitution to assign values to variables in the backwards direction can not be achieved directly as for the forward direction, as the dynamic input in the backward direction does not have a variable name we can substitute with.

Instead, for the inverse semantics, we keep a store σ with mappings of the variables to values we have thus far inferred. Continuously throughout inverse evaluation we keep a free value c which we want to *bind* over an expression. The loose idea is that whenever we happen upon a variable name x while attempting to bind c, we know that x must have been bound to c in the forward direction. We define a binding form as:

$$\langle e, c \rangle \sigma = \begin{cases} \sigma[x \mapsto c] & \text{If } e \equiv x \\ \sigma & \text{If } e \equiv c \equiv () \\ \sigma \cup \sigma' & \text{If } e \equiv x \text{ and } \sigma(x) = e' \text{ and } \langle e', c \rangle \sigma = \sigma' \\ \sigma \cup \sigma' & \text{If } e \equiv \mathbf{inl}(e') \text{ and } c \equiv \mathbf{inl}(c') \text{ and } \langle e', c' \rangle = \sigma' \\ & \vdots \\ \bot & \text{Undefined otherwise} \end{cases}$$

Thus, the form of e should match the canonical value c, which "retracts" c until a variable name is bound. The variable name may have been encountered before during inverse evaluation due to relevance. We require that if an attempt is made to map a variable which already exists in σ, the value of the previous binding and the free value must be the same. The end result is that amongst the values stored in σ, the input to the forward application is assuredly mapped to the name of the dynamic forward parameter.

The first free value we attempt to bind is the value c_{out} applied as the dynamic input to the inverse application, and we bind it against the function body e. We informally review a motivating example of inverse evaluation:

$$f\ (x:\tau) = \textbf{let}\ y = ()\ \textbf{in}\ (y, x) \tag{4.3}$$

The input to f^{-1} must be a product of type $1 \times \tau$ (assume τ is generic). Say we evaluate $f^{-1}\ ((), ())$. This gives us an initial free value $c = ((), ())$. We will look to bind c to something. The result of evaluating a let-expression in the forward direction is an evaluation of e_2 with e_1 substituted for y, so we first want to inspect e_2 in the backward direction. Here we see that $e_2 = (x, y)$. By decomposing the products against the free value, We attain two binding forms: $\langle x, ()\rangle\sigma = \sigma'$ and $\langle y, ()\rangle\sigma' = \sigma''$. Thus far, we have that $\sigma[x \mapsto (), y \mapsto ()] \Leftrightarrow \sigma''$.

Now we move to inverse evaluate $y = ()$. We know that the let-expression resulted in an assignment to y in the forward direction and therefore look at how y was constructed. Remember that y is in σ. We indicate that y is the new free value we wish to bind (which will immediately be fetched from the store, giving us that $()$ actually is the free value) and e_1 is the expression we attempt to bind y against. In this case, we will obtain the binding form $\langle (), ()\rangle\sigma$ which does not change σ. Note that the type system ensure that the value for y in most cases will follow the syntactic form of what is given as the inverse application input. Since we are done, we should have obtained a value for the dynamic parameter x which we can directly read from the σ.

Most of the inverse semantics are presented in Fig. 4.3. Because we have a wide array of possible case expression forms, we present the inverse semantics of case-expressions on their own in Fig. 4.4. In the following we flesh out the meaning behind some of the more convoluted rules.

Binding Rules The binding rules ensure two distinct properties: Predicting a value for a variable from the forward direction (by I-Bind1) and maintaining the relevance of variables (by I-Bind2). Specifically, if the same variable name is attempted to be bound twice, we conclude that they must occur twice in the program. This is legal by relevance, but we must assure ourselves that the new value we try to bind is the same as the old one, otherwise the derivation fails.

Function Application A function application does not retract any syntactic construct atomically like other inference rules which match on the free value — we instead invoke a function application which is designed to revert the result of the forward function application. As expected, the ancillae parameters are invariant (and closed). During inverse evaluation, a regular function application dictates an inverse function application and vice versa. We want to express that the result of the forward application must have been the current free value. Hence, the inverse application designates it as the dynamic parameter — and the result of the application is matched against the dynamic parameter expression.

Let-expressions The free value must necessarily match on e_2 as that is evaluated last, with x substituted for e_1. Therefore we attempt to bind c to e_2 first. This likely gives us a binding for x. How x was computed is witnessed in e_1. So we either completely uncompute x (if e_1 was closed, which completely retracts e_1) or we figure out bindings for the variables making up x by treating x as a free value against e_1. The two inference rules I-Let and I-LetP for let-expressions function identically, but are written for each syntactic form of let-expressions.

The final inference rules for let-expression, I-LetC, is legal as when e_1 is closed, then is constant, and we are not missing any information regarding what went into deriving it in the forward direction. This rule is important for one purpose — consider the following function:

$$f\ (y:\tau) = \textbf{let}\ x = ()\ \textbf{in}\ f\ x\ y$$

$$\boxed{\text{Judgement: } p; \sigma \vdash^{-1} e, c \rightsquigarrow \sigma'}$$

$$\textsc{I-Bind}: \frac{}{p; \sigma \vdash^{-1} x, c \rightsquigarrow \sigma, x \mapsto c} \qquad \textsc{I-Bind2}: \frac{}{p; \sigma, x \mapsto c_2 \vdash^{-1} x, c_1 \rightsquigarrow \sigma, x \mapsto c} c_1 = c_2$$

$$\textsc{I-Unit}: \frac{}{p; \sigma \vdash^{-1} (), () \rightsquigarrow \sigma} \qquad \textsc{I-Prod}: \frac{p; \sigma \vdash^{-1} e_1, c_1 \rightsquigarrow \sigma' \quad p; \sigma' \vdash^{-1} e_2, c_2 \rightsquigarrow \sigma''}{p; \sigma \vdash^{-1} (e_1, e_2), (c_1, c_2) \rightsquigarrow \sigma''}$$

$$\textsc{I-Inl}: \frac{p; \sigma \vdash^{-1} e, c \rightsquigarrow \sigma'}{p; \sigma \vdash^{-1} \mathbf{inl}(e), \mathbf{inl}(c) \rightsquigarrow \sigma'} \qquad \textsc{I-Inr}: \frac{p; \sigma \vdash^{-1} e, c \rightsquigarrow \sigma'}{p; \sigma \vdash^{-1} \mathbf{inr}(e), \mathbf{inr}(c) \rightsquigarrow \sigma'}$$

$$\textsc{I-Var1}: \frac{p; \sigma \vdash^{-1} e, c \rightsquigarrow \sigma'}{p; \sigma, x \mapsto c \vdash^{-1} e, x \rightsquigarrow \sigma'} \sigma(x) = c \qquad \textsc{I-Var2}: \frac{}{p; \sigma \vdash^{-1} e, x \rightsquigarrow \sigma} x \notin \sigma$$

$$\textsc{I-Roll}: \frac{p; \sigma \vdash^{-1} e, c \rightsquigarrow \sigma'}{p; \sigma \vdash^{-1} \mathbf{roll}\ [\tau]\ e, \mathbf{roll}\ [\tau]\ c \rightsquigarrow \sigma'} \qquad \textsc{I-Unroll}: \frac{p; \sigma \vdash^{-1} e, \mathbf{roll}\ [\tau]\ c \rightsquigarrow \sigma'}{p; \sigma \vdash^{-1} \mathbf{unroll}\ [\tau]\ e, c \rightsquigarrow \sigma'}$$

$$\textsc{I-Let}: \frac{p; \sigma \vdash^{-1} e_2, c \rightsquigarrow \sigma' \quad p; \sigma' \vdash^{-1} e_1, x \rightsquigarrow \sigma''}{p; \sigma \vdash^{-1} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, c \rightsquigarrow \sigma''}$$

$$\textsc{I-LetC}: \frac{p \vdash e_1 \downarrow c' \quad p; \sigma \vdash^{-1} e_2[c'/x], c \rightsquigarrow \sigma'}{p; \sigma \vdash^{-1} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, c \rightsquigarrow \sigma'} e_1 \text{ is closed}$$

$$\textsc{I-LetP}: \frac{p; \sigma \vdash^{-1} e_2, c \rightsquigarrow \sigma' \quad \sigma' \vdash^{-1} e_1, (x, y) \rightsquigarrow \sigma''}{p; \sigma \vdash^{-1} \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2, c \rightsquigarrow \sigma''}$$

$$\textsc{I-App}: \frac{p \vdash f!\ \tau_1 \ldots \tau_m\ e_1 \ldots e_{n-1}\ c \downarrow c' \quad p; \sigma \vdash^{-1} e_n, c' \rightsquigarrow \sigma'}{p; \sigma \vdash^{-1} f\ \tau_1 \ldots \tau_m\ e_1 \ldots e_n, c \rightsquigarrow \sigma'} e_1, \ldots, e_{n-1} \text{ are closed.}$$

$$\textsc{I-InvApp}: \frac{p \vdash f\ \tau_1 \ldots \tau_m\ e_1 \ldots e_{n-1}\ c \downarrow c' \quad p; \sigma \vdash^{-1} e_n, c' \rightsquigarrow \sigma'}{p; \sigma \vdash^{-1} f!\ \tau_1 \ldots \tau_m\ e_1 \ldots e_n, c \rightsquigarrow \sigma'} e_1, \ldots, e_{n-1} \text{ are closed.}$$

Figure 4.3: The inverse semantics of CoreFun, missing case rules.

x is constant (and static) and may be used an ancilla in the application of f, but the inverse inference rules cannot discern it before it is needed, *unless* we by the I-LetC rule recognize that we do already know it.

Case-expressions We expect branching to be deterministic. This is achieved in three different ways, based on the form of the case-expression. We present particular inference rules for each form:

$$\text{I-C{\scriptsize ASE}L: } \frac{p; \sigma \vdash^{-1} e_2, c \rightsquigarrow \sigma' \quad p; \sigma' \vdash^{-1} e_1, \mathbf{inl}(x) \rightsquigarrow \sigma''}{p; \sigma \vdash^{-1} \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3, c \rightsquigarrow \sigma''}$$

$$\text{I-C{\scriptsize ASE}R: } \frac{p; \sigma \vdash^{-1} e_3, c \rightsquigarrow \sigma' \quad p; \sigma' \vdash^{-1} e_1, \mathbf{inr}(y) \rightsquigarrow \sigma''}{p; \sigma \vdash^{-1} \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3, c \rightsquigarrow \sigma'} c \notin \text{PLVal}(e_2)$$

$$\text{I-C{\scriptsize ASE}SL: } \frac{p \vdash e_a[c/out] \downarrow \mathbf{inl}(()) \quad p; \sigma \vdash^{-1} e_2, c \rightsquigarrow \sigma' \quad p; \sigma' \vdash^{-1} e_1, \mathbf{inr}(x) \rightsquigarrow \sigma''}{p; \sigma \vdash^{-1} \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3\ \mathbf{safe}\ e_a, c \rightsquigarrow \sigma'}$$

$$\text{I-C{\scriptsize ASE}SR: } \frac{p \vdash e_a[c/out] \downarrow \mathbf{inr}(()) \quad p; \sigma \vdash^{-1} e_3, c \rightsquigarrow \sigma' \quad p; \sigma' \vdash^{-1} e_1, \mathbf{inr}(y) \rightsquigarrow \sigma''}{p; \sigma \vdash^{-1} \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3\ \mathbf{safe}\ e_a, c \rightsquigarrow \sigma'}$$

$$\text{I-C{\scriptsize ASE}UL: } \frac{p; \sigma \vdash^{-1} e_2, c \rightsquigarrow \sigma' \quad p; \sigma' \vdash^{-1} e_1, \mathbf{inr}(x) \rightsquigarrow \sigma''}{p; \sigma \vdash^{-1} \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3\ \mathbf{unsafe}\ , c \rightsquigarrow \sigma'}$$

$$\text{I-C{\scriptsize ASE}UR: } \frac{p; \sigma \vdash^{-1} e_3, c \rightsquigarrow \sigma' \quad p; \sigma' \vdash^{-1} e_1, \mathbf{inr}(y) \rightsquigarrow \sigma''}{p; \sigma \vdash^{-1} \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3\ \mathbf{unsafe}\ , c \rightsquigarrow \sigma'}$$

Figure 4.4: The inverse semantics for case-expressions.

$$\dfrac{\dfrac{\dfrac{}{p; [] \vdash^{-1} x, () \rightsquigarrow [x \mapsto ()]}\text{ I-B{\scriptsize IND}} \qquad \dfrac{}{p; [x \mapsto ()] \vdash^{-1} y, () \rightsquigarrow [x \mapsto (), y \mapsto ()]}\text{ I-B{\scriptsize IND}}}{p; [] \vdash^{-1} (y, x), ((), ()) \rightsquigarrow [x \mapsto (), y \mapsto ()]}\text{ I-P{\scriptsize ROD}} \qquad \cdots}{p; [] \vdash^{-1} \mathbf{let}\ y = ()\ \mathbf{in}\ (y, x), ((), ()) \rightsquigarrow [x \mapsto ()]}\text{ I-L{\scriptsize ET}P}$$

Figure 4.5: Partial view of derivation of program Def. 4.3, by the inverse semantics with inverse input $((), ())$.

1. Determinism is achieved by the first match principle. In this case, an identical side condition, as seen in the big step forward evaluation, is added to the I-CaseR rule.

2. Determinism is achieved by an exit assertion: This is straightforward — we simply check the exit assertion against the free value, as the free value should be regarded as the result of whichever branch was taken.

3. Determinism is achieved by knowing that the free value can only uniquely match on one of the branch expressions. In this case we can freely expect the branch expression which has a binding form with c to have been taken in the forward direction without any further validity checks (but again, only if the uniqueness of the branches has been guaranteed).

With formal inference rules for inverse evaluation in place, we can derive σ for Prog. 4.3. A partial view of the derivation is presented in Fig. 4.5.

4.3 Correctness of Inverse Semantics

We now move on to prove the correctness of the inverse semantics as seen in Fig. 4.3. The proof reflects that we can recover the form of the substitution c of x by the inverse semantics from an expression e — where we have that $e[c/x] \downarrow c'$ — by knowing c'. Note that it is only relevant precisely when x is the only open term in e — but this fits well with reality: We will exploit that x is the variable name of the dynamic variable after every ancillary parameter has been substituted already, noticing that we may substitute the ancillary variable directly as they are invariant.

Lemma 2. *If $p \vdash e[c/x] \downarrow c'$ (by $\mathcal{E}$) where x is the only free variable in e, then $p; \emptyset \vdash^{-1} e, c' \rightsquigarrow \{x \mapsto c\}$ (by $\mathcal{I}$).*

Proof. By induction over the derivation $\mathcal{E}$. We have the cases:

- Case $\mathcal{E} = \dfrac{}{p \vdash ()[c/x] \downarrow ()}$

  This case is not applicable as () obviously cannot contain a free occurrence of x, so it is vacuously true.

- Case $\mathcal{E} = \dfrac{}{p \vdash x[c/x] \downarrow c'}$

  Then $c = c'$ and $e = x$, and we can construct a derivation directly by I-Bind:

$$\mathcal{I} = \frac{}{p; \emptyset \vdash^{-1} x, c \rightsquigarrow x \mapsto c}$$

- Case $\mathcal{E} = \dfrac{\overset{\mathcal{E}_0}{p \vdash e[c/x] \downarrow c'}}{p \vdash \mathbf{inl}(e)[c/x] \downarrow c'}$

  We immediately use the Induction Hypothesis on $\mathcal{E}_0$ and get a derivation $\mathcal{I}_0 = p; \emptyset \vdash^{-1} e, c' \rightsquigarrow x \mapsto c$, and by the I-Inl rule on $\mathcal{I}_0$ we have:

$$\mathcal{I} = \frac{\overset{\mathcal{I}_0}{p; \emptyset \vdash^{-1} e, c' \rightsquigarrow x \mapsto c}}{p; \emptyset \vdash^{-1} \mathbf{inl}(e), \mathbf{inl}(c') \rightsquigarrow x \mapsto c}$$

  An almost identical proof is constructed for when $\mathcal{E}$ is a derivation of $p \vdash \mathbf{inr}(e)[c/x] \downarrow c'$ and $p \vdash (\mathbf{roll}\ [\tau]\ e)[c/x] \downarrow c'$.

- Case $\mathcal{E} = \dfrac{\overset{\mathcal{E}_0}{e_1[c/x] \downarrow c'} \quad \overset{\mathcal{E}_1}{e_2[c/x] \downarrow c''}}{p \vdash (e_1, e_2)[c/x] \downarrow (c', c'')}$

  x must occur in either e_1 or e_2 or both. If x occurs in e_1 only, the Induction Hypothesis on $\mathcal{E}_0$ gives us a derivation $I_0 = p; \emptyset \vdash^{-1} e_1, c' \rightsquigarrow x \mapsto c$. Further, since e_2 by implication contains no free variables, by Lemma 3 we have a derivation $\mathcal{I}_1 = p; x \mapsto c \vdash^{-1} e_2, c'' \rightsquigarrow x \mapsto c$. By applying I-Prod with $\mathcal{I}_0$ and $\mathcal{I}_1$ we have:

$$\mathcal{I} = \cfrac{\overset{\mathcal{I}_0}{p; \emptyset \vdash^{-1} e_1, c' \rightsquigarrow x \mapsto c} \quad \overset{\mathcal{I}_1}{p; x \mapsto c \vdash^{-1} e_2, c'' \rightsquigarrow x \mapsto c}}{p; \emptyset \vdash^{-1} (e_1, e_2), (c', c'') \rightsquigarrow x \mapsto c}$$

An analogous proof is valid for if only e_2 contains x. If both e_1 and e_2 contain x, the derivation of $\mathcal{I}_1$ must include an application of I-Bind2 to show that the resulting store is in fact $x \mapsto c$.

- Case $\mathcal{E} = \cfrac{\overset{\mathcal{E}_0}{p \vdash e[c/x] \downarrow \textbf{roll}\ [\tau]\ c''}}{p \vdash (\textbf{unroll}\ [\tau]\ e)[c/x] \downarrow c''}$

  By the Induction Hypothesis on $\mathcal{E}_0$ (with $c' = \textbf{roll}\ [\tau]\ c''$) we get a derivation $\mathcal{I}_0 = p; \emptyset \vdash^{-1} e[c/x], \textbf{roll}\ [\tau]\ c'' \rightsquigarrow x \mapsto c$, and we can use it directly to construct $\mathcal{I}$ by applying I-Unroll:

$$\mathcal{I} = \cfrac{\overset{\mathcal{I}_0}{p; \emptyset \vdash^{-1} e, \textbf{roll}\ [\tau]\ c'' \rightsquigarrow x \mapsto c}}{p; \emptyset \vdash^{-1} \textbf{unroll}\ [\tau]\ e, c'' \rightsquigarrow x \mapsto c}$$

- Case $\mathcal{E} = \cfrac{\overset{\mathcal{E}_0}{p \vdash e_1[c/x] \downarrow c''} \quad \overset{\mathcal{E}_1}{\Sigma; \Gamma \vdash e_2[x'/c'', c/x] \downarrow c'}}{p \vdash (\textbf{let}\ x' = e_1\ \textbf{in}\ e_2)[c/x] \downarrow c'}$

  By Lemma 4, we may consider multiple substitutions. There are a number of cases here: x' might or might not be an open term in e_2 and x might be in e_1, e_2 or both.

  - If x' is not an open term in e_2, then x must occur in e_2. This is evident as a derivation $\mathcal{I}_0$ by the Induction Hypothesis on $\mathcal{E}_1$ would result in a store not containing a mapping for x' — and a derivation $\mathcal{I}_1$ of $\langle e_1, x' \rangle$ would immediately apply I-Var2 as there is no mapping for x' by the derivation $\mathcal{I}_0$. Thus the Induction Hypothesis on $\mathcal{E}_1$ gives us $\mathcal{I}_0 = p; \emptyset \vdash^{-1} e_2, c \rightsquigarrow x \mapsto c$ and we can construct $\mathcal{I}$ as:

$$\mathcal{I} = \cfrac{\overset{\mathcal{I}_0}{p; \emptyset \vdash^{-1} e_2, c' \rightsquigarrow x \mapsto c} \quad \cfrac{}{p; x \mapsto c \vdash^{-1} e_1, x' \rightsquigarrow x \mapsto c}\, x' \notin \{x \mapsto c\}}{p; \emptyset \vdash^{-1} \textbf{let}\ x' = e_1\ \textbf{in}\ e_2, c' \rightsquigarrow x \mapsto y}$$

  - If x' is an open term in e_2, we have three possible cases for where x occurs: in only e_1, in only e_2 or in both. If it only occurs in e_2, then by Lemma 4 we have a derivation $\mathcal{I}_1 = p; \emptyset \vdash^{-1} e_2, c' \rightsquigarrow x' \mapsto c'', x \mapsto c$. This means e_1 is necessarily a closed term, and by Lemma 3 on $\mathcal{E}_0$ we finally have:

$$\mathcal{I} = \cfrac{\overset{\mathcal{I}_0}{p; \emptyset \vdash^{-1} e_2, c' \rightsquigarrow x \mapsto c, y \mapsto c''} \quad \overset{\mathcal{I}_1}{p; x \mapsto c, y \mapsto c'' \vdash^{-1} e_1, y \rightsquigarrow x \mapsto c}}{p; \emptyset \vdash^{-1} \textbf{let}\ x' = e_1\ \textbf{in}\ e_2, c' \rightsquigarrow x \mapsto y}$$

  If x is an open term in both, a similar reasoning is used, but e_1 is not closed — rather, we know the derivation of $\langle e_1, x' \rangle$ must use an application of I-Bind2 which ensures that the only binding is $x \mapsto c$ is consistent, and we are done.

If x only occurs in e_1, then by applying the Induction Hypothesis directly on $\mathcal{E}_1$ we get a derivation $\mathcal{I}_0 = p; \emptyset \vdash^{-1} e_2, c' \rightsquigarrow x' \mapsto c''$. The only inference rule possible for $\mathcal{I}$ is I-Let, and we need to construct a derivation for $\mathcal{I}_1$. By applying I-Var1 for the second premise on $\langle e_1, x' \rangle$ we get some derivation:

$$\mathcal{I}_1 = \cfrac{\overset{\mathcal{I}_1'}{p; \emptyset \vdash^{-1} e_1, c'' \rightsquigarrow \sigma}}{p; x' \mapsto c'' \vdash^{-1} e_1, x' \rightsquigarrow \sigma}$$

For some σ. Then we may use the Induction Hypothesis on $\mathcal{E}_0$ with $\mathcal{I}_1'$, and we finalize the derivation $\mathcal{I}_1$ so that $\sigma = x \mapsto c$ and we can finally construct:

$$\mathcal{I} = \cfrac{\overset{\mathcal{I}_0}{p; \emptyset \vdash^{-1} e_2, c' \rightsquigarrow x' \mapsto c''} \quad \overset{\mathcal{I}_1}{\cfrac{p; \emptyset \vdash^{-1} e_1, c'' \rightsquigarrow x \mapsto c}{p; x' \mapsto c'' \vdash^{-1} e_1, x' \rightsquigarrow x \mapsto c}}}{p; \emptyset \vdash^{-1} \textbf{let } x' = e_1 \textbf{ in } e_2, c' \rightsquigarrow x \mapsto c}$$

An analogous proof works for when $e = \textbf{let } (x, y) = e_1 \textbf{ in } e_2$.

- Case $\mathcal{E} = \cfrac{\overset{\mathcal{E}_0', \ldots, \mathcal{E}_n'}{p \vdash e_1[c/x] \downarrow c_1' \ldots p \vdash e_n[c/x] \downarrow c_n'} \quad \overset{\mathcal{E}''}{p \vdash e[c_1'/x_1, c_n'/x_n, c/x] \downarrow c'}}{p \vdash (f\ \alpha_1 \ldots \alpha_m\ e_1 \ldots e_n \downarrow)[c/x] \downarrow c'}$

  Where $p(f) = f\ \alpha_1 \ldots \alpha_m\ x_1 \ldots x_n = e$. The only applicative rule for the derivation is I-App, so each expression $e_1, \ldots, e_{n-1}$ must be closed and x must occur in e_n. c' is also closed, as it is a canonical value. We by Theorem 3 (the correctness of inverse application,) have that $\mathcal{E}''' = f!\ \alpha_1 \ldots \alpha_m\ e_1 \ldots e_{n-1}\ c' \downarrow c''$, where $p; \emptyset \vdash^{-1} e[c_1'/x_1, \ldots, c_{n-1}'/x_{n-1}], c' \rightsquigarrow \sigma$ with $\sigma(x_n) = c''$.

  But that means that $e_n[c/x] \downarrow c''$, as a substitution of c_n' occurs for x_n in $\mathcal{E}''$, and we have $e_n[c/x] \downarrow c_n'$ by $\mathcal{E}_n'$. By the Induction Hypothesis on $\mathcal{E}_n'$ we have $\mathcal{I}_0 = p; \emptyset \vdash^{-1} e_n, c'' \rightsquigarrow x \mapsto c$, and we construct the full derivation:

$$\mathcal{I} = \cfrac{\overset{\mathcal{E}''''}{f!\ \alpha_1 \ldots \alpha_m\ e_1 \ldots e_{n-1}\ c' \downarrow c''} \quad \overset{\mathcal{I}_0}{p; \emptyset \vdash^{-1} e_n, c'' \rightsquigarrow x \mapsto c}}{p; \emptyset \vdash^{-1} f\ \alpha_1 \ldots \alpha_m\ e_1 \ldots e_n, c' \downarrow c''}$$

  An analogous proof occurs when $\mathcal{E}$ is a derivation of an inverse function call.

- Case $\mathcal{E} = \cfrac{\overset{\mathcal{E}_0}{p \vdash e_1[c/x] \downarrow \textbf{inl}(c'')} \quad \overset{\mathcal{E}_1}{p \vdash e_2[c''/x', c/x]}}{p \vdash (\textbf{case } e_1 \textbf{ of } x' \Rightarrow e_2, x'' \Rightarrow e_3)[c/x] \downarrow c'}$.

  Proving this case is very similar to the proof for let-expressions, with completely similar sub-cases for which expression contains which open term. The only difference is we cannot directly apply I-Var on e_1 but must start with matching the sum-term by I-Inl. The same is true for E-CaseR, E-CaseSL and E-CaseSR.

And we are done.

$\square$

Lemma 3. *If $p \vdash e \downarrow c$ (by $\mathcal{E}$) where there are no free variable in e, then $p; \sigma \vdash^{-1} e, c \rightsquigarrow \sigma$ (by $\mathcal{I}$).*

Proof. By induction over the derivation $\mathcal{E}$. The proof is highly similar to the one for Lemma 2, so we only show exemplary cases, especially ones which are relevant:

- Case $\mathcal{E} = \dfrac{}{p \vdash () \downarrow ()}$
  We can construct a derivation directly using the I-Unit rule as:

$$\mathcal{I} = \frac{}{p; \emptyset \vdash^{-1} (), () \rightsquigarrow \emptyset}$$

- Cases for when e is $\mathbf{inl}(e'), \mathbf{inr}(e'), \mathbf{roll}\ [\tau]\ e', (e', e'')$ and $\mathbf{unroll}\ [\tau]\ e'$ follow almost immediately by the Induction Hypothesis on the premises of $\mathcal{E}$. The precise method was exercised in the proof of Lemma 2.

- Case $\mathcal{E} = \dfrac{\overset{\mathcal{E}_0}{p \vdash e_1 \downarrow c'} \quad \overset{\mathcal{E}_1}{p \vdash e_2[c'/x] \downarrow c}}{p \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \downarrow c}$
  There are two cases: e_2 contains the open term x or it does not. If it does not, then by the Induction Hypothesis on $\mathcal{E}_1$ we get a derivation $\mathcal{I}_0 = p; \emptyset \vdash^{-1} e_2, c \rightsquigarrow \emptyset$. $\mathcal{I}$ must necessarily use the I-Let rule, and a derivation of $\langle e_1, x\rangle$ must immediately use the I-Var2 rule as $\sigma = \emptyset$, and we have:

$$\mathcal{I} = \frac{\overset{\mathcal{I}_0}{p; \emptyset \vdash^{-1} e_2, c \rightsquigarrow \emptyset} \quad \dfrac{}{p; \emptyset \vdash^{-1} e_1, x \rightsquigarrow \emptyset}x \notin \emptyset}{p; \emptyset \vdash^{-1} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \emptyset}$$

  If x is an open term in e_2, then by the induction hypothesis on $\mathcal{E}_0$ we get a derivation $\mathcal{I}_0 = p; \emptyset \vdash^{-1} e_2, c \rightsquigarrow x \mapsto c'$ for some c'. Then we construct $\mathcal{I}_1$ by applying the I-Var1 if we can prove $\mathcal{I}_1' = p; \emptyset \vdash^{-1} e_1, c' \rightsquigarrow \sigma$ for some σ. But we have this by the Induction Hypothesis on $\mathcal{E}_0$ with $\mathcal{I}_1'$ concluding $\sigma = \emptyset$, allowing us to construct:

$$\mathcal{I} = \frac{\overset{\mathcal{I}_0}{p; \emptyset \vdash^{-1} e_2, c \rightsquigarrow x \mapsto c'} \quad \dfrac{\overset{\mathcal{I}_1}{p; \emptyset \vdash^{-1} e_1, c' \rightsquigarrow \emptyset}}{p; x \mapsto c \vdash^{-1} e_1, x \rightsquigarrow \emptyset}}{p; \emptyset \vdash^{-1} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \emptyset}$$

The remaining cases have been omitted, but are straightforward to include.

□

Lemma 4. *If $p \vdash e[c_1/x_1, \ldots, c_n/x_n] \downarrow c'$ (by $\mathcal{E}$) where exactly the set $x_1, \ldots, x_n$ are the free variables occurring in e, then $p; \emptyset \vdash^{-1} e, c' \rightsquigarrow x_1 \mapsto c_1, \ldots, x_n \mapsto c_n$ (by $\mathcal{I}$).*

Proof. The proof of this is a straightforward generalization of Lemma 2 and 3.

□

Theorem 3 (Correctness of Inverse Semantics). *If $p \vdash f\ c_1 \ldots c_n \downarrow c$ with $p(f) = \alpha_1 \ldots \alpha_m\ x_1 \ldots x_n = e$, then $\sigma(x_n) = c_n$ where $p; \emptyset \vdash^{-1} e[c_1/x_1, \ldots, c_{n-1}/x_{n-1}], c \rightsquigarrow \sigma$.*

Proof. Follows immediately by an application of Lemma 2 by considering $e[c_1/x_1, \ldots, c_{n-1}/x_{n-1}]$ as the expression e with exactly one open term x_n, which we substitute with c_n. Then $p; \emptyset \vdash^{-1} e[c_1/x_1, \ldots, c_{n-1}/x_{n-1}], c \rightsquigarrow x_n \mapsto c_n$, and specifically, $\{x_n \mapsto c_n\}(x_n) = c_n$.

□

Programming in CoreFun

5

Although CoreFun is a r-Turing complete language, it lacks many of the convenient features of most modern functional languages. Luckily, we can encode many language constructs directly as syntactic transformations from a notationally light language to the formal core language.

The principle encompasses that for each category of syntactic abstraction, we can show that there exists a systematic translation from the notationally light language to CoreFun. This allows us to introduce a number of practical improvements without the necessity to corroborate their correctness by either the type system or operational semantics, instead opting to present a *translation scheme*.

A translation scheme $e \overset{\text{def}}{=} e'$ replaces the program text e with e', with e being a valid expression in the light language and e' being a valid CoreFun expression. A successful transformation therefore does not entail that a well-typed program is generated, only it is grammatically correct. Often, a translation will depend on recursive descent over the expression. In these cases, we express a translation as $\langle e \rangle$ (so an inner translation is notated as $\langle e' \rangle$ with e' being some subexpression of e. Sometimes, we require some auxiliary information to be present for a translation. In these cases, we express a translation as $\langle e, s \rangle$, where the translation for e depends the value s. We will make clear for each translation what is going on.

5.1 Variants

Variants, also known as tagged unions, are related to algebraic data types. They provide a method of declaring new data types as a fixed number of named alternatives. In CoreFun they generalize recursive types, sum types, and case-expressions over these. A variant declaration is of the form:

$$\beta = \mathrm{v}_1 \mid \cdots \mid \mathrm{v}_n$$

This defines a new data type (a variant type) identified by β. Constructing a value of a variant type entails choosing exactly one of the possible *variant constructors* $\mathrm{v}_1, \ldots, \mathrm{v}_n$ and potentially grant it data to carry. Then, given a variable of a variant type, we pattern match over its possible constructor forms to unveil the data and to control program flow.

We have seen that we can generalize binary sums $\tau_1 + \tau_2$ to n-ary sums by repeated sum types in the right component τ_2, and that we can chain together case-expressions to match the correct arm of such a sum. We choose an encoding of variants which exploits this pattern. This works because the variant constructors are ordered in the declaration and will deterministically match with the respective position in the n-ary sum.

For a variant which carries no data, the translation corresponds to stripping away the variant constructor tags, leaving us with the underlying n-sum type of all unit types:

$$\beta = \mathrm{v}_1 \mid \cdots \mid \mathrm{v}_n \overset{\text{def}}{=} \beta = 1 + \cdots + 1$$

We can further extend variants to carry data by adding parameters. We allow generic type parameters by adding a type parameter to the variant declaration. The syntax for variant declarations becomes:

$$\beta\ \alpha^* = \mathrm{v}_1\ [\tau\alpha]^* \mid \cdots \mid \mathrm{v}_n\ [\tau\alpha]^*$$

Where $[\tau\alpha]^*$ signifies zero or more constructor parameters of either a type (allowing for inner variants) *or* type variables supplied to β.

If exactly one parameter τ is present for a constructor v_i, the type at position i in the n-ary sum is changed from the unit type to the type τ.

$$\beta\ \alpha = \mathrm{v}_1 \mid \mathrm{v}_2\ \tau \mid \mathrm{v}_3\ \alpha \overset{\text{def}}{=} \beta\ \alpha = 1 + \tau + \alpha$$

Notice that we may generalize any parameter-free variant constructor to one with a single parameter of type unit, which we omit from the syntax.

If a variant constructor v_i has $m \geq 2$ parameters $p_1, \ldots, p_m$, the type in the position of i in the n-ary sum is changed into a product type $p_1 \times \cdots \times p_m$.

$$\beta = \mathrm{v}_1 \mid \cdots \mid \mathrm{v}_i\ \tau_1 \ldots \tau_m \mid \ldots \overset{\text{def}}{=} \beta = 1 + \cdots + \tau_1 \times \cdots \times \tau_m + \ldots$$

There is one more case we need to take into consideration for transformation: Recursively defined variants. In the following we use the subscript i to indicate that the variant we are declaring has been indexed by i in the family of possible variant names. The principle goes that if any of the variant constructors for a variant type β_i have a self-referencing parameter (a parameter of type β_i), the translated type of β_i is recursive and a fresh variable is designated as the recursion variable.

$$\beta_i = \mathrm{v}_1 \mid \mathrm{v}_2\ \beta_i \overset{\text{def}}{=} \beta_i = \mu X.1 + X \qquad\qquad X \text{ is a fresh type variable.}$$

The above actually corresponds to an encoding of the natural numbers.

When all variant declarations have been translated, we have generated a set VarDecs of named translations of the form $\beta = \tau$ where τ is a core type. We can define a translation of occurrences of variant types inductively over the core syntax of CoreFun, expanding variant types as we go. The interesting cases are:

$$\begin{aligned} f \ldots (x : \beta) \ldots &\overset{\text{def}}{=} f \ldots (x : \tau) \ldots && \text{When } x \text{ is any parameter for } f \text{ and VarDecs}(\beta) = \tau \\ \textbf{roll}\ [\beta]\ e &\overset{\text{def}}{=} \textbf{roll}\ [\tau]\ e && \text{When VarDecs}(\beta) = \tau \\ \textbf{unroll}\ [\beta]\ e &\overset{\text{def}}{=} \textbf{unroll}\ [\tau]\ e && \text{When VarDecs}(\beta) = \tau \end{aligned}$$

The simplest variant declaration corresponds to the type `Bool` of Boolean values:

$$\texttt{Bool} = \texttt{True} \mid \texttt{False} \overset{\text{def}}{=} \texttt{Bool} = 1 + 1$$

The `Maybe` datatype is encoded as:

$$\texttt{Maybe}\ \alpha = \texttt{Nothing} \mid \texttt{Just}\ \alpha \overset{\text{def}}{=} \texttt{Maybe} = 1 + \alpha$$

While the encoding for generic lists exemplifies most of the translation rules for variant declarations simultaneously:

$$\texttt{List}\ \alpha = \texttt{Nil} \mid \texttt{Cons}\ \alpha\ (\texttt{List}\ \alpha) \overset{\text{def}}{=} \texttt{List} = \mu X.1 + \alpha \times X$$

As mentioned earlier a variant value is an encoding of a nested number of sum terms which correspond to the ordering of the variant. We can define a general translation for variant values — in the following we denote $|\beta|$ as the number of variant constructors a variant type has. We have:

$$\begin{aligned} \texttt{v}_i\ e_1 \ldots e_n &\overset{\text{def}}{=} \mathbf{inr}_1(\ldots(\mathbf{inr}_{i-1}(\mathbf{inl}_i((e_1, \ldots, e_n)))\ldots) && \text{When } i < |\beta| \\ \texttt{v}_i\ e_1 \ldots e_n &\overset{\text{def}}{=} \mathbf{inr}_1(\ldots(\mathbf{inr}_i(e_1, \ldots, e_n))\ldots) && \text{When } i = |\beta| \end{aligned}$$

A handy result to keep in mind is that if two variant definitions have the same number of alternatives and carry data of identical types, they are isomorphic and may be encoded the same, which simplifies the translation scheme.

The final critical transformation is obtaining a classical case-expression from a case over a variant value. In the light language we wish to write a case over a variant type as a single possible choice between all the variant constructors of the type of the variant value.

$$\begin{array}{l} \texttt{case}\ v\ \texttt{of} \\ \quad \texttt{v}_1\ e_{1_1} \ldots e_{1_j} \Rightarrow e_1 \\ \qquad \vdots \\ \quad \texttt{v}_m\ e_{m_1} \ldots e_{m_k} \Rightarrow e_m \end{array}$$

Where v has type β and $\texttt{v}_1, \ldots, \texttt{v}_m$ are the variant constructors of β. We observe that we can exploit the encoding of variant values to unpack the data for each constructor. The construction is a bit more complex as we need to case over a chain of fresh variables. We therefore introduce the notation $\langle e, w\rangle$ to signify that e is to be translated and w is a fresh variable which should be cased over. Then the translation is defined recursively as a nested structure of binary case-expressions:

$$\begin{aligned} &\langle \texttt{v}_i\ e_{i_1}, \ldots, e_{i_j} \Rightarrow e_i, w\rangle \overset{\text{def}}{=} \mathbf{case}\ w\ \mathbf{of}\ \mathbf{inl}((e_{i_1}, \ldots, e_{i_j})) \Rightarrow e_i, \mathbf{inr}(w') \Rightarrow e' \\ &\qquad \text{When } i < m \text{ and } w' \text{ is a fresh variable and } e' = \langle w', v_{i+1}\ e_{i_1}, \ldots, e_{i_k} \Rightarrow e_{i+1}\rangle \\ &\langle \texttt{v}_i\ e_{i_1}, \ldots, e_{i_j} \Rightarrow e_i, w\rangle \overset{\text{def}}{=} \mathbf{let}\ (e_{i_1}, \ldots, e_{i_j}) = w\ \mathbf{in}\ e_i \\ &\qquad \text{When } i = m \end{aligned}$$

A fully-fledged exampled of how variants are translated is presented in Fig. 5.1.

```
Choice = Rock | Paper | Scissors

rpsAI (c:Choice) =
  case c of
    Rock ⇒ Paper
    Paper ⇒ Scissors
    Scissors ⇒ Rock
```

(a) Light program.

```
rpsAI (c:1 + (1 + 1)) =
  case c of
    inl(()) ⇒ inr(inl(()))
    inr(w) ⇒ case w of
      inl(()) ⇒ inr(inr(()))
      inr(()) ⇒ inl(())
```

(b) Core program.

Figure 5.1: A full example of variant translations

5.2 Top-level Function Cases

A top-level function case is a generalization of inspecting the immediate form of the inputs to a function. This entails, contrary to the core language, that arguments are not necessarily named in the definition but are pattern matched on directly as expressions. A function definition is then given as a finite, strictly ordered set of m *function clauses*:

$$\{f\ e_{1_i} \dots e_{n_i} = e \mid 1 \leq i \leq m\}$$

Where we denote the parameter expressions $e_{1_i} \dots e_{n_i}$ for each clause a *clause pattern*. There are some restrictions on a function pattern:

1. We require that no clause pattern matches another clause pattern perfectly (we require the clause patterns to be orthogonal).
2. We require that the form of each expression in a clause pattern is either a sum term, a variable name or a variant constructor. This is necessary as case-expressions in the core language are only defined for sum terms (and we saw in Sec. 5.1 that variants transform into sum terms).
3. We require that there are no omitted clauses. This is to enforce totality of the branches when the program has been transformed to the core rendition, which does not support missing branches for sum terms.

We enforce a particular ordering on the clauses. This is due to how the top-level case will be unfolded in the next transformation step. This ordering can be inferred, so strictly speaking we do not have to restrict the programmer from writing them in any arbitrary order (enforcing the proper ordering might be a good design choice nonetheless, as implicit reordering of clauses might cause confusion). We define the ordering on a parameter expression as:

$\mathbf{inl}(e) \leq \mathbf{inr}(e)$

$\mathrm{v}_i \leq \mathrm{v}_j$ When v_i and v_j are variant constructors of the same type and $i \leq j$.

And we define the ordering of clause patterns as the ordering parameter-wise, associated to the left.

$$f\ e_{1_1} \dots e_{n_1} = \cdot \leq f\ e_{1_2} \dots f\ e_{n_2} = \cdot \qquad \text{if } e_{1_1} \leq e_{1_2}, \dots, e_{n_1} \leq e_{n_2}$$

With an ordering in place, we may define function application of f as evaluating the clause body of the first clause in the ordered set of clauses of f for which each parameter expression matches the values supplied to the application. In the following, f_s denotes a chain of increasing order of function clauses. We define an *ordered traversal* as:

$$(f\ e_1 \dots e_n = e \leq f_s)\ c_1 \dots c_n \begin{cases} e & \text{if } c_1 \rhd e_1, \dots, c_n \rhd c_n \\ (f_s)\ c_1 \dots c_n & \text{otherwise} \end{cases}$$

That is, if the values supplied to the function application are applicable for a clause, evaluate the clause body. Otherwise, try the next clause. Since we assumed that clause coverage is total, this will always evaluate *some* clause body.

The translation should unfold a series of case-expressions which respect the order of clauses as described. An observation is that we can make a distinction between the parameters which vary and those that do not. Parameters that do not vary are necessarily variable names in every clauses and can be ignored.

Another observation is that we can group the clauses by how each parameter expression is repeated to match on the subsequent parameter expression. This gives us a nested structure of sets akin a tree, where the clause bodies are the leafs. We have:

$$\{e_{1_i}\ \{e_{2_j}\ \{\dots \{e_{i+j+\dots}\} \dots\}_j\}_i \mid 1 \leq j \leq q\} \mid 1 \leq i \leq p\}$$

Where we have truncated the m occurrences of pattern expressions in the first parameter into k distinct expressions. The same characteristic holds for each parameter, and the process is then repeated recursively. The inner-most set is a singleton set containing the clause body for exactly the combination of pattern expression (which is reflected in its additive subscript). This structure tells us how to "flatten" the clauses into individual case-expressions. Each instance of a group is recursively asked to flatten. In the following, x is a fresh variable name and a and b are placeholders for the ordering of parameters. We have:

$$\begin{aligned}&\langle \{e_{a_i}\ \{e_{b_j}\ \{\dots\}\} \mid 1 \leq j \leq q\}_i \mid 1 \leq i \leq p\}\rangle \overset{\text{def}}{=} \\ &\mathbf{case}\ x\ \mathbf{of}\ e_{a_1} \Rightarrow \langle \{e_{b_j}\ \{\dots\} \mid 1 \leq j \leq q\}_1\rangle \dots e_{a_p} \Rightarrow \langle \{e_j\ \{\dots\} \mid 1 \leq j \leq q\}_p\rangle\end{aligned} \tag{5.1}$$

And the bottom case:

$$\langle \{e_{a_i}\ \{e_i\} \mid 1 \leq i \leq p\}\rangle \overset{\text{def}}{=} \mathbf{case}\ x\ \mathbf{of}\ e_{a_1} \Rightarrow e_1, \dots, e_{a_k} \Rightarrow e_p \tag{5.2}$$

A clear issue with translating the current translation scheme is the apparent loss of type information for parameters. In the core language we require that parameters are given in the form $(x : \tau)$. Transforming name-less expressions into fresh variable names is trivial, but omitting type information requires that we provide full type inference. Therefore we also require a top-clause whose only purpose is to pair up with the parameters in the core language. A complete function definition becomes

$$\begin{gathered} f :: \alpha_1 \dots \alpha_m \ . \ \tau_1 \to \dots \to \tau_n \\ \{ f \ e_{1_i} \dots e_{n_i} = e \mid 1 \leq i \leq m \} \end{gathered}$$

The clauses are translated with regards to (5.1) and (5.2), giving e_{body}, while the function as a whole is translated as:

$$\begin{array}{l} f :: \alpha_1 \dots \alpha_m \ . \ \tau_1 \to \dots \to \tau_n \\ \{ f \ e_{1_i} \dots e_{n_i} = e \mid 1 \leq i \leq m \} \end{array} \overset{\text{def}}{=} f \ \alpha_1 \dots \alpha_m \ (x_1 : \tau_1) \dots (x_n : \tau_n) = e_{body}$$

We show the translation of a top-level cases with a translation of a map function. The translation is shown in Fig. 5.2.

```
map :: $\alpha\ \beta\ (\alpha \leftrightarrow \beta) \to \mu X.1 + \alpha \times X$
map $f$ inl(()) = roll [$\mu X.1 + \beta \times X$] inl(())
map $f$ inr(($x, xs'$)) = let $x' = f\ \alpha\ x$
                              $xs''$ = map $f\ \alpha\ \beta\ xs'$
                          in roll [$\mu X.1 + \beta \times X$] inr(($x', xs''$))
```

(a) Light program.

```
map $\alpha\ \beta\ (f : \alpha \leftrightarrow \beta))\ (xs : \mu X.1 + \alpha \times X)$ = case unroll [$\mu X.1 + \alpha \times X$] $xs$ of
  inl(()) $\Rightarrow$ roll [$\mu X.1 + \beta \times X$] inl(())
  inr(($x, xs'$)) $\Rightarrow$ let $x' = f\ \alpha\ x$
                              $xs''$ = map $f\ \alpha\ \beta\ xs'$
                          in roll [$\mu X.1 + \beta \times X$] inr(($x', xs''$))
```

(b) Core program.

Figure 5.2: An example of a top-level case translation.

5.3 Guards

When we have added support for a generalized top-level function case as seen in Sect. 5.2, it is natural to consider *guards* as well. Guards are additional requirements on the form of the clause pattern $e_1 \dots e_n$ of any particular clause of a function f. Each guard is associated with one and only one clause in the form of an additional expression g where g always evaluates to $1 + 1$ (a Boolean value). A clause becomes:

$$f \ e_1 \dots e_n \mid g = e$$

During a function application f with canonical values $c_1 \cdots c_n$ substituted for the parameters of f, g has the values $c_1 \dots c_n$ in scope.

Each top-level pattern may now be repeated multiple times in the function declaration, so we lax the requirement of orthogonality of clauses, but with a new guard for each repetition. Each match of a clause pattern then in actuality matches a set of clauses, which we call a *cluster*. Again, we impose an order on a cluster of clauses. We order them by position of definition, since there is no intrinsic order of the guards, with on exception: The default guard is always the top element. We then perform an ordered traversal to discern which clause-body should be taken. Denote a pattern by p. We have:

$$\begin{array}{l} f\ p \mid g_1 = e_1 \\ \quad\vdots \\ f\ p \mid g_n = e_n \end{array} \quad \Leftrightarrow g_1 = e_1 \leq \cdots \leq g_n = e_n$$

Notice that the same pattern p occurs in every clause above. We define the ordered traversal $- \leq -$ for guards similarly as to top level function-cases by

$$g = e \leq g_s \begin{cases} e & \text{if } g \downarrow \mathbf{inl}(()) \\ g_s & \text{if } g \downarrow \mathbf{inr}(()) \end{cases}$$

To make sure that the ordered traversal is total, we require that a default guard in the form of a keyword **otherwise** is supplied for each pattern which has guards. This is simply translated directly to an $\mathbf{inl}(())$ expression. As for the restrictions on top-level cases, this ensures that *some* clause body always will be evaluated. We require totality to guarantee that we always can generate every arm in a case-expression in the core language. Note that an omitted guard for a clause $f\ p = e$ can be generalized to a clause containing a guard which is always passed (so $f\ p = e \Leftrightarrow f\ p \mid \mathbf{inl}(()) = e$).

We show a translation scheme from a top-level case function which has guards into a top-level case function which does not have guards. By this we impose an order of translations so that a translation of guards must occur before a translation of top-level case functions.

Assume we have partitioned the function clauses into clusters. We first show the translation of a particular cluster. The translation does not depend on the structure of the clause besides the guard and clause body itself, so we omit them.

$$\begin{aligned} \bot = e_1 &\overset{\text{def}}{=} e_1 \\ g_1 = e_1 \leq g_2 = e_2 \leq \cdots \leq g_n &\overset{\text{def}}{=} \mathbf{case}\ g_1\ \mathbf{of}\ \mathbf{inl}(()) \Rightarrow e_1, \mathbf{inr}(()) \Rightarrow g' \\ &\qquad \text{When } g' \text{ is the translation of } g_2 \leq \cdots \leq g_n \end{aligned}$$

$$\begin{aligned} \{f\ e_{i_1}\ \ldots\ e_{i_m} \mid g_i = e_i \mid 1 \leq i \leq n\} &\overset{\text{def}}{=} \{f\ e_{i'_1}\ \ldots\ e_{i'_m} = g'_i \mid 1 \leq i' \leq m'\} \\ &\qquad \text{When } g' \text{ is the guard-translation for the cluster } i'. \end{aligned}$$

An example of translation of guards is presented in Fig. 5.3.

5.4 Type Classes

Type classes, introduced in [45] and later popularised in Haskell [18], are aimed at solving overloading of operations by allowing types to implement or infer a type class. A type class κ is a collection of

```
tryPred :: μX.1 + X
tryPred x | case unroll [μX.1 + X] x of
              inl(()) ⇒ inl(()),
              inr(x′) ⇒ inr(()) = inl(())
tryPred x | otherwise = let x′ = unroll [μX.1 + X] x in inr(x′)
```

(a) Light program.

```
tryPred :: μX.1 + X
tryPred x = case unroll [μX.1 + X] x of
              inl(()) ⇒ inl(())
              inr(()) ⇒ let x′ = unroll [μX.1 + X] x in inr(x′)
```

(b) Core program.

Figure 5.3: A translation of guards.

function names (called *operations*) with accompanying type signatures $\{f \Rightarrow \tau_f \mid f \in \kappa\}$, which are the operations to be inferred when the type class is instantiated for a type α (where α is a type variable). τ_f may naturally contain α. The syntax for type classes is as follows

$$\textbf{class}\ \kappa\ \alpha\ \textbf{where}\ [f \Rightarrow \tau_f]^+$$

Where $[\cdot]^+$ denotes one or more instances of $\cdot$ (in this case functions and function signatures which belong to the class). Functions with generic types which apply types on operation which belong to a type class can be provided a *context* which specifies that a type class instance is written for that type.

$$f\ \kappa\ \alpha \Rightarrow \alpha\ .\ (x_1 : \tau_1) \ldots (x_n : \tau_n) = e$$

Where the types $\tau_1 \ldots \tau_n$ may include α and e may contain applications of operations from κ on terms of type α. An instance of a particular type class κ for a type τ then instantiates the operations for τ, allowing us to call f with τ substituted for α.

$$\textbf{instance}\ \kappa\ \tau\ \textbf{where}\ [f\ x^+ \Rightarrow e]^+$$

Where for each $f \in \kappa$ we have that there is a correlation between the number of parameters x in the operation implementation and the number of types in the signature for f. For simplicity we require that operation names are unique. That is, operation names are accessible through the global function scope as any other function.

A function application with an overloaded function name still requires that the type variable be manually applied, as type classes do not give any notion of type inference. This is clear from the following example, as a function's context may include the same type class for two different type variables. We show the example in the syntax given by a top level function case:

$$\begin{array}{l} f :: \kappa\ \alpha,\ \kappa\ \beta \Rightarrow \alpha\ \beta\ .\ \alpha \to \beta \\ f\ x\ y = g\ y \end{array} \qquad \text{Where } g \in \kappa$$

Which instance for g between α and β should be dispatched to? Although it is obvious that the correct instantiation is β in this case, it requires type inference for y, which is not considered in this work.

In the original introduction of type classes, instances of type classes may declare an *instance context* θ for the type τ which the instance is being declared for. An instance context dictates which instances must exist for the types which comprise τ (including τ itself), before the instance can be written. For example, this allows us to describe that we may write an instance for a type class κ of `List` α iff. α is already a member of κ — denoted $\kappa\ \alpha \Rightarrow \kappa\ (\texttt{List}\ \alpha)$. This has two functions:

1. We can describe a hierarchy of type classes although we only require one type class to be instantiated in the context of a function.
2. We can write type classes for types which have free type variables, when the type class' operations require that the type variable also is an instance of the same type class.

We will omit instance contexts in this work for simplicity. This severely hinders the usefulness of type classes as we cannot write type class instances for types which have free type parameters (like `List`), but this limitation can easily be remedied in a more extensive implementation.

We will now show a type class translation scheme. The principle is to strip away all definitions and instances of classes and treat each instance member as a top level function definition. An obvious method is to create unique functions for each instance which specialize for that instance type. We can do this as the class provides us with the function type and the instance provides us with the function implementation. We have:

$$\begin{array}{l}\textbf{class}\ \kappa\ \alpha\ \textbf{where}\ \ \Rightarrow \alpha \leftrightarrow \tau' \\ \textbf{instance}\ \kappa\ \tau\ \textbf{where}\ f\ x \Rightarrow e\end{array} \overset{\text{def}}{=} f_\tau\ (x : \tau) = e \qquad f_\tau \text{ is a fresh function name}$$

Additionally, any function g in which an application of a member function f from a class κ occurs, we need to change the application to the newly generated f_τ. There are two distinct transformations we must consider here:

1. κ is in the context of g. This means That g takes a type variable which is restricted by κ, and some class member function of κ occurs in the body of g. As we wish to remove the context, the type variable shall be removed as well, and any application of it shall instead call the member function of κ. This means g must also be made into a fresh function.

$$\begin{array}{l} g :: (\kappa\ \alpha) \Rightarrow \alpha\ .\ \alpha \\ g\ x = f\ \alpha\ x \end{array} \overset{\text{def}}{=} \begin{array}{l} g :: \tau \\ g_\tau\ x = f_\tau\ x \end{array}$$

When τ is an instance of κ and f is a member of κ

2. An application of a class member function f of κ occurs in g, but with a valid concrete type (meaning, a concrete type for which an instance exists). In this case the type variable application should just be removed and the correct member function be applied instead:

$$g\ (x : 1) = f\ 1\ x \overset{\text{def}}{=} g\ (x : 1) = f_1\ x$$

When 1 is an instance of κ and f is a member of κ

A strong benefactor of type classes is equality testing. Equality as a type class can, with variants, replace the duplication/equality operator through the following definition of equality:

```
Eq 'a = Eq | Neq 'a
```

Which should be interpreted as: Either we have that $x = y$ and thus `eq` x $y \Rightarrow$ `Eq`, or we have that $x \neq y$ and `eq` x $y \Rightarrow$ `Neq` y. Recall that since x is an ancilla, it is tacitly returned from an equality check. Thus when $x = y$, y can be destroyed as its original value is preserved in x — otherwise, we have to remember y. In the example in Fig. 5.4 we present a simpler version of an Equality type class which states equality of values as a form of attribute of two values which consumes neither value. An attribute in this fashion can always be extracted by transforming a unit value, which explains the signature of `eq`.

```
class Eq α where
  eq ⇒ α → α → 1 ↔ (1 + 1)

instance Eq (μX.1 + X) where
  eq n₀ n₁ () ⇒ case unroll [μX.1 + X] n₀ of
                  inl() = case unroll [μX.1 + X] n₁ of
                    inl() ⇒ inl(())
                    inr(n₁') ⇒ inr(())
                  inr(n₀') = case unroll [μX.1 + X] n₁ of
                    inl() ⇒ inr(())
                    inr(n₁') ⇒ eqInt n₀' n₁' ()

compare (Eq α) ⇒ α (x₁: α) (x₂: α) = eq α x₁ x₂ ()

eqNil (x: μX.1 + X) = compare (μX.1 + X) x (roll [μX.1 + X] inl(()))
```

(a) Light program.

```
eqNat (n₀: μX.1 + X) (n₁: μX.1 + X) (): 1 = case unroll [μX.1 + X] n₀ of
  inl() = case unroll [μX.1 + X] n₁ of
    inl() ⇒ inl(())
    inr(n₁') ⇒ inr(())
  inr(n₀') = case unroll [μX.1 + X] n₁ of
    inl() ⇒ inr(())
    inr(n₁') ⇒ eqNat n₀' n₁' ()

compareNat (x1: μX.1 + X) (x2: μX.1 + X) = eqNat x1 x2 ()

eqNil (x: μX.1 + X) = compareNat x (roll [μX.1 + X] inl(()))
```

(b) Core program.

Figure 5.4: Translation of type classes.

5.5 Records

Records are products of labeled elements of arbitrary types. They generalize products and product indexing by supporting component-wise projections and local updates. [8] considers *extensible records* for System F with proper subtyping and equality for records which may be extended (by a label which is a type variable) as well as *simple records* (records of a fixed label cardinality). We will adopt a notion similar to a simple record in CoreFun.

The first major deviation from the simple records of [8] we will enforce is totality of a record instantiation. Simple records as described in the literature allow records to be partially populated, using *row variables* to postpone instantiation of any number of record components. We enforce totality because we look to guarantee that we can directly translate a record into the core language by writing an ordered n-ary product, where n is the record's cardinality. Note that totality exempts us from having to define a subtyping relation between records of different "definitiveness". Combining simplicity and totality of records means we can completely omit subtyping as a necessary property of records.

Second, we disallow arbitrary projections, for the obvious reason that projections destroy information. Notice that projections were omitted in CoreFun's grammar as well, a pivotal difference from regular functional programs, necessary to maintain reversibility. What we instead want to do to support projections is to introduce a special type of scope, instantiated for a specific record γ individually, in which we may fetch and change any number of labels for γ. Any remaining labels are then automatically assumed to be invariant.

A record is defined as

$$\gamma = \{l_1 :: \tau_1, \dots, l_n :: \tau_n\}$$

Where γ ranges over record names. A record value is constructed by assigning a variable name to a record constructor amongst the records which have been defined. Each label of the record needs to be supplied a value (or expression which evaluates to a value) of the type declared for that label in the record definition.

We denote π as a permuting function over a set of assignments to labels, so that labels may be assigned in any order. The full set of permuting functions l_n for any given label cardinality n is the product of possible assignment orders. An exemplary permuting function $\pi \in l_4$ is:

$$\pi(1) = 3, \quad \pi(2) = 1, \quad \pi(4) = 2, \quad \pi(1) = 4$$

A permutation is necessarily bijective. The identity permutation id_π sends every label to itself, so $\mathrm{id}_\pi(a) = a$. We have

$$\pi(\{l_1 = c_1, \dots, l_n = c_n\}) = \{\pi(l_1 = c_1), \dots, \pi(l_n = c_n)\}$$

Because a permutation is bijective, any particular permutation has an inverse so that $\pi^{-1}\pi = \mathrm{id}_\pi$, meaning we can always order the label assignments in a record declaration by applying an inverse permutation. A full construction of a record $\gamma_i = \{l_1 :: \tau_1, \dots, l_n :: \tau_n\}$ can now be described as

$$\mathbf{let}\ r = \gamma_i\ \pi(\{l_1 = c_1, \dots, l_n = c_n\})\ \mathbf{in}\ e \qquad\qquad \text{where } c_1 : \tau_1, \dots, c_n : \tau_n \text{ and } \pi \in l_n$$

We now present a translation scheme for record construction. Recall that we denote the n-ary product $\tau_1 \times \cdots \times \tau_n$ as nested application of binary products. Now, we first need to find the π^{-1} for whatever permutation was induced for the construction of γ_i. This can easily be done as we want to invert the position of each label into the ordering of the labels in the record definition, which is statically decidable. After the fact that the product is ordered, this turns projection into a deterministic traversal into the n-ary product.

Now we introduce the *record scope*. In the record scope we may freely extract and update values within the record for which we open a record scope. The syntax is

$$\textbf{within } \gamma : e^{\rho} \textbf{ end}$$

e^{ρ} is an extended grammar for expressions where we add the following expression constructs:

- $e ::= \rho(\gamma, l_i)$
- $e ::= \textbf{let } \rho(\gamma, l_i) = e \textbf{ in } e$

Where $\rho(r, l_i)$ is a projection which returns the cell within the record γ for the label l_i. We denote $\rho(\gamma, l_i)$ as $\gamma.l_i$ in the syntax. The two expression constructs thus respectively read the value associated with a label in a record and write to a label associated with a record. We expect that the value returned from a record scope is the record itself. Because we still allow arbitrary expressions in the record scope, we want the inner expression to result in one of two things: 1) The variant itself, 2) A variant assignment. Any other expression is ill-formed.

In the transformation of a record scope we initially expose the full structure of the record at the entry point to the scope, which in itself allows projections as the variable names are now uncovered. The translation of an expression $\rho(\gamma, l_i)$ is substituted with the i'th component of the record product:

$$\textbf{within } \gamma : \gamma \textbf{ end} \overset{\text{def}}{=} \textbf{let } (x_1, \ldots, x_n) = \gamma \textbf{ in } (x_1, \ldots, x_n)$$
$$\text{When } \gamma \text{ is a variable which a record is bound to and } |\gamma| = n$$

An assignment to a record cell is translated into a regular let-expression with an assignment to a fresh variable. The exit of the record scope then reconstructs the record product with the new assignments in place of the old cells.

$$\textbf{within } \gamma : \textbf{let } \gamma.l_1 = \gamma.l_1 \textbf{ in } \gamma \textbf{ end} \overset{\text{def}}{=} \textbf{let } (x_1, \ldots, x_n) = \gamma \textbf{ in let } x_1' = x_1 \textbf{ in } (x_1', \ldots, x_n)$$
$$\text{When } \gamma \text{ is a variable which a record is bound to and } |\gamma| = n$$

We show the transformation of a declaration of three-dimensional `Vector` record and a function which uses it.

5.6 Arbitrarily Sized Products

CoreFun only supports binary products in product construction and when pattern matching on the left-hand side of let and case-expressions, as can be noted in the language grammar. Here, we will detail a method of pattern matching on arbitrarily sized products in a single expression. There are essentially three locations we may encounter n-ary products:

1. While constructing product values. Example:

$$f\ (x : \tau) = (x, e_2, e_3)$$

```
Vector = { x: $\mu X.1+X$, y: $\mu X.1+X$, z: $\mu X.1+X$ }

$f$ ($x:\mu X.1+X$) ($y:\mu X.1+X$) ($z:\mu X.1+X$) =
  let $v$ = Vector { x = $x$, y = $y$, z = $z$ }
  in within $v$:
       $v.x$ = roll [$\mu X.1+X$] inr($v.x$)
     end
```

(a) Light program.

```
$f$ ($x:\mu X.1+X$) ($y:\mu X.1+X$) ($z:\mu X.1+X$) =
  let $v$ = $(x,(y,z))$
  in let $x'$ = roll [$\mu X.1+X$] inr($x$)
    in $(x',(y,z))$
```

(b) Core program.

Figure 5.5: Translation of records.

Translating a constructed product with n elements is as simple as repeatedly wrapping the right component into nested products where any inner product is solely binary. We define a recursive translation operation $\langle\cdot\rangle$ as:

$$\langle(e_1,e_2)\rangle \stackrel{\text{def}}{=} (e_1,e_2)$$
$$\langle(e_1,e_2\ldots,e_n)\rangle \stackrel{\text{def}}{=} (e_1,(\langle e_2,\ldots e_n\rangle))$$

2. When pattern matching on the left-hand side of a let-expression. Example:

$$g\ (x:\tau) = \textbf{let}\ (a,b,c) = f\ x\ \textbf{in}\ (a,b,c)$$

Here we need to exploit the syntax of the let-expression of the core language, which binds expressions of a product type on the left hand side of the assignment. We can unfold the full product by introducing a let-expression for each element of the product with a fresh variable name in the right component and the values of the product in the left component. We have:

$$\langle\textbf{let}\ (x_1,x_2) = e_1\ \textbf{in}\ e_2\rangle \stackrel{\text{def}}{=} \textbf{let}\ (x,y) = e_1\ \textbf{in}\ e_2$$
$$\langle\textbf{let}\ (x_1,x_2,\ldots,x_n) = e_1\ \textbf{in}\ e_2\rangle \stackrel{\text{def}}{=} \textbf{let}\ (x_1,x') = \langle\textbf{let}\ (x_2,\ldots x_n) = x'\ \textbf{in}\ e_2\rangle$$
$$\text{Where } x' \text{ is a fresh variable}$$

3. When pattern matching on the left-hand side of a case-expression. Example:

$$\textbf{case}\ h\ x\ \textbf{of}\ \textbf{inl}((a,b,c)) \Rightarrow (a,b,c), \textbf{inr}(x) \Rightarrow x$$

In this example we expect x to be of some type $\tau_1 \times \tau_2 \times \tau_3 + \alpha$. An analogous condition exists for a pattern match for the right branch.

Case-expressions actually cannot pattern match over values at all, so we introduce a pattern matching let-expression immediately in the body e when we unfold a product. This allows us to use the translation from (2.) immediately after. We introduce two rules, one for each branch. Should we need to pattern match in both branches the rules can be used in unison:

$$
\begin{aligned}
&\langle \textbf{case } x \textbf{ of inl}((e_1, \dots, e_n)) \Rightarrow e_1, \textbf{inr}(e') \Rightarrow e_2 \rangle \\
&\overset{\text{def}}{=} \textbf{ case } x \textbf{ of inl}(x') \Rightarrow \langle \textbf{let } (e_1, \dots, e_n) = x' \textbf{ in } e_1 \rangle, \textbf{inr}(e') \Rightarrow e_2 \\
&\langle \textbf{case } x \textbf{ of inl}(e') \Rightarrow e_1, \textbf{inr}((e_1, \dots, e_n)) \Rightarrow e_2 \rangle \\
&\overset{\text{def}}{=} \textbf{ case } x \textbf{ of inl}(e') \Rightarrow e_1, \textbf{inr}(x') \Rightarrow \langle \textbf{let } (e_1, \dots, e_n) = x' \textbf{ in } e_2 \rangle
\end{aligned}
$$

Where x' is a fresh variable. An example of arbitrarily sized products is presented in Fig. 5.6.

```
tripDup (x : 1) = (x, x, x)

f (x : 1) = let (y, z, w) = tripDup x
            in case inl((y, z)) of
               inl((y', z')) ⇒ (y', z', w)
               inr(()) ⇒ ((), (),  ())
```

(a) Light program.

```
tripDup (x : 1) = (x, (x, x))

f (x : 1) = let (y, t₁) = tripDup x
             in let (z, w) = t₁
                in case inl((y, z)) of
                  inl(t₂) ⇒ let (y', z')) = t₂ in (y', z', w)
                  inr(()) ⇒ ((), ((),  ()))
```

(b) Core program.

Figure 5.6: Translation of arbitrarily sized products.

5.7 Multiple Let Bindings

Multiple variable bindings in a row are currently achieved using a nested chain of let-expressions. This can without further ado be reduced to a single let-expression in which with every binding occurs first, followed by an evaluation of the final expression. We have:

$$\textbf{let } x_1 = e_1, \dots, x_n = e_n \textbf{ in } e$$

And we define a translation as:

$$\langle \textbf{let } x_1 = e_1 \textbf{ in } e\rangle \overset{\text{def}}{=} \textbf{let } x_1 = e_1 \textbf{ in } e$$

$$\langle \textbf{let } x_1 = e_1, x_2 = e_2 \ldots, x_n = e_n \textbf{ in } e\rangle \overset{\text{def}}{=} \textbf{let } x_1 = e_1 \textbf{ in } \langle \textbf{let } x_2 = e_2 \ldots, x_n = e_n \textbf{ in } e\rangle$$

We show a translation of multiple nested let-expressions as a continuation of the map function, which was shown before as an example of a translation of top-level function clauses, in Fig. 5.7.

```
map α β (f : α ↔ β) (xs : μX.1 + α × X) = case unroll [μX.1 + α × X] of
  inl(()) ⇒ roll [μX.1 + α × X] inl(()),
  inr(xs′) ⇒ let ((x, xs″)) = xs′
                 x′ = f x
                 xs‴ = map f xs″
             in cons x′ x‴
```

(a) Light program.

```
map α β (f : α ↔ β) (xs : μX.1 + α × X) = case unroll [μX.1 + α × X] of
  inl(()) ⇒ roll [μX.1 + α × X] inl(()),
  inr(xs′) ⇒ let ((x, xs″)) = xs′
             in let x′ = f x
                in let xs‴ = map f xs″
                   in cons x′ x‴
```

(b) Core program.

Figure 5.7: Translation of nested let-expressions.

Implementation 6

The language described in the work has been implemented in Haskell as a reference implementation. It includes parsers for both the core and light grammars, a typechecker implementing the base type system from Sect. 2 including every extension showed later, an interpreter implementing the big step operational semantics, an inverse interpreter implementing the inverse semantics, a first match policy analyzer and a transpiler transforming a light program into the core program.

The implementation involves a number of monads, specifically a Reader, Writer, State and Error monad. Details on these can be found in [24]. We also employ a monadic parser combinatoric approach via the `megaparsec` library. An introduction to monadic parsing in Haskell can be found in [22].

In the following we will discuss various details about the implementation, including ways in which the implementation diverges from the theoretic description.

6.1 Parsing

We implement two parsers and maintain two abstract syntax trees: one for the light language and one for the core language. Both languages are indentation oblivious but white space sensitive to some extent. Specifically, we allow line breaks only under certain conditions. Canonical terms and applications must be retained on the same line while case expressions recognize branches on separate lines and let-expressions allow the binding $l = e_1$ be separated from the body e_2, so we can write:

```
case x of
  inl(y) → y,
  inr(z) → z

let x = e
in e
```

In the core language, parsing case-expressions is deterministic without white space sensitivity or explicit delimitation of the expression. This is because we know we have to account for exactly two branches and each left hand side is uniquely given. In the light language, we do not have this insurance as a case-expression may involve any number of branches and their left hand side is unrestricted. Consider the program in Fig. 6.1.

Note the comma of `v13 → v13,` on line 5. In subfigure (a) the indentation levels seem to indicate that the first inner case-expression only has three alternatives — so the comma delimits the two outer branches, and we can begin to parse the second arm `v2 → ...`. But the meaning is ambiguous: It can also be parsed to mean that the case-expression in the first inner branch has another alternative, which is the natural interpretation in subfigure (b).

```
case x of
  v1 → case v1 of
           v11 → v11,
           v12 → v12,
           v13 → v13,
  v2 → case v2 of
           v21 → v21,
           v22 → v22
```

(a)

```
case x of
  v1 → case v1 of
           v11 → v11,
           v12 → v12,
           v13 → v13,
           v2  → case v2 of
                     v21 → v21,
                     v22 → v22
```

(b)

Figure 6.1: Ambiguous program.

Capturing the correct meaning requires indentation sensitiveness, which we do not implement. Therefore we delimit a light case expression with an additional **esac** to erase this ambiguousness. We rewrite the case-expression from Fig. 6.1 as:

```
case x of
  v1 → case v1 of
          v11 → v11,
          v12 → v12,
          v13 → v13
        esac,
  v2 → case v2 of
          v21 → v21,
          v22 → v22
        esac
  esac
```

6.2 Typechecking

The typechecker employs the State monad to maintain the set of hypotheses, mapping types to variables. We partition the hypotheses into a static and dynamic fragment to reflect Γ and Σ. Without loss of correctness we deviate slightly from the typing judgement by storing function signatures in its own environment in the form of a Reader monad. The Error monad ensures that when any error has been encountered, it will propagate to the top typechecker entry point with a designated error message. This choice of error handling promises that the first and only the first error encountered will be brought to attention — there is no tally of type errors.

Contrary to what is assumed in the type rules — where the type signature $\tau_f \to \cdots \to \tau \leftrightarrow \tau$ of a function is assumed to be fully known at the time of typing — the return type of functions is not known initially in the implementation. Obligatory annotations of types of parameters ensure that we know every parameter type, but we will have to infer return types. For this we heed to *constraint solving*. We define a special kind of type variable called a *unification variable*, denoted $\{i\}$, where i is the identity of a particular unification variable. The informal idea is to find the most general form of each unification variable by inspection of how they are used in each function body expression. A fresh unification variable is introduced for each expression we do not immediately know the full type of. In the present language, unknown types occur solely by function applications and by sum term. Specifically, **inl**(e) is typed as

$\tau + \{i'\}$ where i' is a fresh unification variable and $\Sigma; \Gamma \vdash e : \tau$. Symmetrically we have $\mathbf{inr}(e)$ as $\{i'\} + \tau$. Further, each function name f also omits a unique unification variable $\{f\}$, meaning every function application types as $\{f\}$.

A constraint on a unification variable $\{i\}$ is produced whenever we learn anything about $\{i\}$ by how an expression $e : \{i\}$ is subsequently treated, which might generate new *unification variables*. Exemplary cases are:

$\mathbf{let}\ (x, y) = e : \{i\}\ \mathbf{in}\ e'$	$\text{Cons}(i = i' \times i'')$ with $x : \{i'\}$, $y : \{i''\}$
$\mathbf{case}\ e : \{i\}\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e', \mathbf{inr}(y) \Rightarrow e''$	$\text{Cons}(i = i' + i'')$ with $x : \{i'\}$, $y : \{i''\}$

Each constraint is remarked in a global record of constraints. After the type of each function $f \in p$ has been deduced, we know that every constraint has been produced for p. For constraint solving to succeed, no two constraints may be in direct contradiction. In the reference implementation we use the Writer monad to transcribe constraints during typing, and solve them as a subsequent operation to typing.

6.3 Interpretation and Static Analysis

The implementation of forward interpretation is rather straightforward — it is implemented basically as presented in the big step semantics. It employs the Reader monad to represent p, the storage of function definitions. No state is needed as mappings between variables and values is achieved through substitution. We expect every program to be well typed. This specifically means that the only run-time error which *should* occur is a failure of the CASE-R side condition — in this case, an error is evoked via the Error monad.

Inverse interpretation introduces a state monad in addition to p, representing the store σ as seen in the inverse big step judgement. A simple first match policy analyser implementation is also made available.

6.4 Transformation

Transformations are performed serially in a pipeline fashion. In theory, transformations in Sect. 5 are (in almost all cases) defined so that each transformation is layered directly on top of the core language, something which is not feasible when they are all combined into one grammar. The exception to the rule is the transformation of guards, which are defined on top of a top-level function case. The reference implementation takes some liberties on this front and constructs the transformations on top of each other to simplify the implementation, while keeping true to the semantics of each. The order of transformations is:

$$tProducts \circ tLetIns \circ tVariants \circ tTopFunc \circ tGuards \circ tTypeClasses$$

We do not implement the translation of records, and thus do not support them in the reference implementation. The reason is that the translation scheme for records exploits that we know the full type signatures of functions to unpack record values. But we have seen how we need to infer return types via typechecking in the reference implementation. Ultimately, we cannot typecheck a program until after transformations have been applied but we apply a transformation for records until we know the return type of each function. A workaround would be to allow the programmer to to add return type annotations to each function signature.

6.4.1 Typechecking of Light Programs

There is a balance to be struck regarding the extent to which the implementation should validate the non-transformed program. A good rule of thumb is to attempt to perform the translation as long as the transformation rules are possible to apply — in the most rudimentary interpretation, we at least have to take the precautions addressed in the main text on transformations into account.

An open issue regarding typechecking a light program is the tackling the presentation of cryptic error messages which will be encountered when there is an error during typechecking of a transformation program. If the transformation is not stateful in a way such that the original program text is preserved and accessed when an error happens, the error will not reflect what the programmer wrote at all. This has been left as future work.

6.4.2 Transformation Example

As an example, we present a reversible zip function `zip` $:: \alpha\ \beta.\ [\alpha] * [\beta] \to [\alpha * \beta]$ (with $[\cdot]$ being the usual shorthand for lists) actually translated by the reference implementation. `zip` is a non-trivial example of an CoreFun function, as we have to decide the behaviour when the lists are of uneven size. We cannot omit the clauses as the case-expressions need to be total and we cannot throw away excess values from the longer list. What we do instead is constrict the function to lists of types which instantiate the `Default` type class — types for which we can define default values. The smaller list is then padded until it has the same length as the longer list. An example for lists of natural numbers, where the default value is 0:

$$\texttt{zip}\ [1,2,3,4]\ [5,6] = [(1,5),(2,6),(3,0),(4,0)]$$

The original program can be seen in Fig. 6.2 and the generated core program can be seen in Fig. 6.3. We have added line breaks and white spaces in the presentation of the transformed program where appropriate, to make the structure of the program more clear. Note that the presentation breaks the parser's rules regarding white spaces, so it is only exemplary. Also note that the transformation breaks the naming convention of every class of fresh variables it generates. These variable names are well formed as a program, but cannot be parsed! This is a simple method of ensuring uniqueness of new variables.

6.5 Default Programs and Testing

Initially, we wished to test the reference solution rigorously with something like QuickCheck [10], but soon realized that it is hardly fitting for highly structured data. Instead, we include a large array of small programs which are meant to test the limits of the parser, typechecker, interpreter and transformer. Take note that this is a heuristic approach as we have not *proven* the correctness of any part of the reference implementation. In addition to providing test programs, we include an assortment of exemplary program with numerous examples of well formed functions over standard data structures like lists and numerals.

```
type Nat = Null | Succ Nat
type List 'a = Nil | Cons 'a (List 'a)

class Default 'a where
  def ⇒ 1 ↔ 'a
instance Default Nat where
  def u ⇒ roll [Nat] Null

zip :: Default 'a, Default 'b ⇒ 'a 'b . (List 'a) * (List 'b)
zip ls = let (xs, ys) = ls
            in case unroll [List 'a] xs of
              Nil → case unroll [List 'b] ys of
                Nil → roll [List ('a * 'b)] Nil,
                Cons y ys' → let xdef = def 'a ()
                                  nil = roll [List 'a] Nil
                                  ls' = zip 'a 'b (nil, ys')
                              in roll [List ('a * 'b)] (Cons (xdef, y) ls')
                esac,
              Cons x xs' → case unroll [List 'b] ys of
                Nil → let ydef = def 'b ()
                           nil = roll [List 'b] Nil
                           ls' = zip 'a 'b (nil, xs')
                       in roll [List ('a * 'b)] (Cons (x, ydef) ls'),
                Cons y ys' → let ls' = zip 'a 'b (xs', ys')
                              in roll [List ('a * 'b)] (Cons (x, y) ls')
                esac
            esac

unzip :: Default 'a, Default 'b ⇒ 'a 'b . List ('a * 'b)
unzip ls = zip! 'a 'b ls
```

Figure 6.2: A reversible `zip` program written in the light language.

```
__zipNatNat (ls:(\A . (1 + (\B . (1 + B) ∗ A)) ∗ \A . (1 + (\B . (1 + B) ∗ A)))) =
  let (xs, ys) = ls
  in case unroll [\A . (1 + (\B . (1 + B) ∗ A))] xs of
    inl(()) → case unroll [\A . (1 + (\B . (1 + B) ∗ A))] ys of
      inl(()) → roll [\A . (1 + ((\B . (1 + B) ∗ \B . (1 + B)) ∗ A))] inl(()),
      inr(_a) →
        let (y, ys') = _a
        in let xdef = __defNat ()
        in let nil = roll [\A . (1 + (\B . (1 + B) ∗ A))] inl(())
        in let ls' = __zipNatNat (nil, ys')
        in roll [\A . (1 + ((\B . (1 + B) ∗ \B . (1 + B)) ∗ A))] inr(((xdef, y), ls')),
    inr(_a) →
      let (x, xs') = _a
      in case unroll [\A . (1 + (\B . (1 + B) ∗ A))] ys of
        inl(()) →
          let ydef = __defNat ()
          in let = nil roll [\A . (1 + (\B . (1 + B) ∗ A))] inl(())
          in let = ls' __zipNatNat (nil, xs')
          in roll [\A . (1 + ((\B . (1 + B) ∗ \B . (1 + B)) ∗ A))] inr(((x, ydef), ls')),
        inr(_b) →
          let (y, ys') = _b
          in let = ls' __zipNatNat (xs', ys')
          in roll [\A . (1 + ((\B . (1 + B) ∗ \B . (1 + B)) ∗ A))] inr(((x, y), ls'))

__unzipNatNat (ls:\A . (1 + ((\B . (1 + B) ∗ \B . (1 + B)) ∗ A))) =
  __zipNatNat! ls

__defNat (u:1) = roll [\A . (1 + A)] inl(())
```

Figure 6.3: The reversible `zip` function from Fig. 6.2 transformed by the reference implementation.

Discussion

7

7.1 Updated Grammar

We present the updated grammar of programs belonging to the light language in Fig. 7.2. We define the domains which variants, records and type classes range over in Fig. 7.1. The updated grammar is a superset of the original grammar. Specifically, any program in the core language is also a program in the updated language.

$$\kappa \in \text{Type class names} \qquad \beta \in \text{Variant names} \qquad \gamma \in \text{Record names}$$

Figure 7.1: Domains

7.2 Omitted Abstractions

Proposing additional translations which could have been considered in Sec. 5 is an indefinite process. Some lesser, mostly stylistic, propositions will be mentioned as future work in Sec. 7.4. Other translations are a bit more tricky. We present here two translations, anonymous functions and infix operators, which were considered as translations but are not as fitting to translate as they seem on the surface. They are interesting as they do not necessarily work intuitively in a reversible setting.

7.2.1 Anonymous Functions

Anonymous functions are nameless functions, modeled as abstractions from the lambda calculus invented by Church [9]. An abstraction is constructed from a variable name x and an inner term t where x is a free variable in t. An application $t\ t'$ substitutes a term t' for x in t. A term is closed when it contains no more free variables. Usually abstractions are generalized to take any number of variables as arguments.

- **Variable:** x
- **Abstraction:** $(\lambda x.\ t)$
- **Application:** $t\ c$

We would have to impose certain restrictions on where anonymous functions could be declared to maintain the restricted higher order property of CoreFun. To mirror how restricted higher order functions may be used in the core language, they should only be constructable in two places in the light language:

$q ::= d^*$	Program definition
$d ::= f$	Function definition
$\mid \textbf{class } \kappa\ \alpha \textbf{ where } [f \Rightarrow \tau_f]^+$	Type class definition
$\mid \textbf{instance } \kappa\ (\beta\ \alpha^*) \textbf{ where } [f \Rightarrow e]^+$	Type class instance
$\mid \beta\ \alpha^* = [\mathrm{v}\ [\tau\alpha]^*]^+$	Variant definition
$\mid \gamma = \{[l :: \tau]^+\}$	Record definition
$e ::= x$	Variable name
$\mid ()$	Unit term
$\mid \textbf{inl}(e)$	Left of sum term
$\mid \textbf{inr}(e)$	Right of sum term
$\mid \mathrm{v}\ e^*$	Variant term
$\mid (e_1, \ldots, e_n)$	Product term
$\mid \textbf{let } [l = e]^+ \textbf{ in } e$	let-expression
$\mid \textbf{case } e \textbf{ of } [e \Rightarrow e]^+$	case-expression
$\mid \textbf{case } e \textbf{ of } \textbf{inl}(x) \Rightarrow e, \textbf{inr}(y) \Rightarrow e \textbf{ safe } e$	Safe case-expression
$\mid \textbf{case } e \textbf{ of } \textbf{inl}(x) \Rightarrow e, \textbf{inr}(y) \Rightarrow e \textbf{ unsafe}$	Unsafe case-expression
$\mid f\ \alpha^*\ e^+$	Function application
$\mid f!\ \alpha^*\ e^+$	Inverse Function application
$\mid \textbf{roll } [\tau]\ e$	Recursive-type construction
$\mid \textbf{unroll } [\tau]\ e$	Recursive-type destruction
$\mid \textbf{within } \gamma : e^\rho \textbf{ end}$	Record scope
$e^\rho ::= \rho(\gamma, l)$	Record projection
$\mid \rho(\gamma, l) = e^\rho$	Record cell assignment
$\mid \textbf{let } \rho(\gamma, l) = e^\rho_1 \textbf{ in } e^\rho_2$	Record let-assignment
$l ::= x$	Definition of variable
$\mid (x_1, \ldots, x_n)$	Variable product
$f ::= f\ \alpha^*\ \tau^+[f\ e^+[\mid g]? = e]^+$	New Style Function Definition
$\mid f\ \alpha^*\ (x : \tau_a)^+ = e$	Old Style Function definition

Figure 7.2: Grammar for the updated language.

directly as expressions supplied as an ancillary parameter during function application or as function applications themselves.

An obvious translation of anonymous functions would be to construct a top-level function for each anonymous function, taking care of name collisions. Each newly constructed function would then be subjected to precisely one application in the full program, at the position where it was previously defined.

The main issue with anonymous functions is determining the scope of variables involved in the open term of the abstraction's body. CoreFun and its light counterpart use lexical scoping, meaning the context in which the anonymous function lives in is fully available. A contrived example could be:

```
f x = let z = inr(x)
   in let y = (\ x′ → (z, x′)) x
   in y
```

The only usage of the value z is in the abstraction's body, even though z is not supplied to the anonymous function as a parameter. In any conventional functional language this has well-defined behaviour, and we expect it to be well-defined in CoreFun as well. Alas, the abstraction body is removed from its lexical scope when the translation occurs, as it is moved into a new top-level function f_λ — meaning z is no longer available for f_λ. This indicates that we need a more thorough analysis of which values are applied to the statically generated function during translation.

7.2.2 Infix Operators and Numerals

Natural numbers can be added relatively easily as an abstraction over the unary numerals. But they are only really practical if we simultaneously add arithmetic operators. Arithmetic operators are a subset of infix binary operators. An infix operator $\odot$ is reversible if we may transform either operand while keeping the other intact, so $x \odot y = (x, x \odot y)$, and if there is an inverse operation $\odot^{-1}$ so that $x \odot^{-1} (x \odot y) = (x, y)$. The operator may be self-inverse, so $x \odot (x \odot y)) = (x, y)$.

The following are amongst the arithmetic operators which are reversible, with their inverses provided:

$$\begin{aligned}(+)^{-1} &\overset{\text{def}}{=} (-)\\ (\times)^{-1} &\overset{\text{def}}{=} (\div)\end{aligned}$$

Supporting infix operators requires non-trivial design choices because applying the inverse operation on the resulting value of the forward operation might require that we go against the intuition of how these operations work if the inverse operator is not commutative. This is because the transformed value necessarily is the right operand (because of the design of ancillae parameters leaning left). This issue is evident from both operator pairs $+/-$ and $\times/\div$, as we have:

$$\begin{aligned}2 + 3 &= 5\\ 2 +!\ 5 &= 3\end{aligned}$$

It appears as if $2 - 5 = 3$, if we naively define $(-)$ as the inverse of $(+)$. The analogous scenario crops up for $(\div)$ as the inverse of $(\times)$. A possible fix would be to instead transform the left operand for binary operations, but this really does not align well with the type system as the type system dictates that the right most parameter of a function is always transformed.

There are a couple of other options. We can forbid the declaration of new infix operators and fix the behaviour of the operators we do define as additional syntactic sugar. That is, we can define any infix operator into an equivalent operator in Polish notation as $l \odot r \overset{\text{def}}{=} (\odot)\ r\ l$, with the operands reversed (which works fine in the forward direction as well as both $+$ and $\times$ are commutative). This also makes it simple to transform the operations we *do* write, as we simply predefine the functions which correspond to infix operators.

If we do allow the generation of new infix operators, the programmer should have access to some mechanism through which they can define what precedence the operands should have in the inverse direction or if the operator is commutative.

Encoding Natural Numbers Natural numbers as numerals are generalized over the Peano numbers encoded by the recursive type $\mu X.\ 1 + X$. They are translated the obvious way by recursive descent on a numeral n as a **roll**-term on a $\mathbf{inl}(n - 1)$ expressions until we reach the bottom numeral 0, which is constructed as $\mathbf{inr}(())$. That this construction is correct can be shown with induction.

Proof. The induction hypothesis states, by the principle of well-formed induction, that any element n in the infinite domain has a non-infinite set of values which which it relates to, that is, the set $A = \{n' \mid n' \prec n\}$ is finite. If it is finite there must be a least element n_0 for which $\forall n' \in A.\ n_0 \prec n'$. For the recursive type $\mu X.1 + X$ this is $\mathbf{roll}\ [\mu X.1 + X]\ \mathbf{inl}(())$, the first unary number, which is a well-formed expression in the core language.

For the induction step we assume for any n we have a well-formed expression which denotes n in the core language and want to prove that we form a well-formed expression for $n+1$. Then we can simply take the next Peano number. Denote c the canonical representation of n. Then we take $n+1$ to be **roll** $[\mu X.1 + X]$ **inr**(c).

$\square$

7.3 Reversible Higher Order Language

We now move on to another famous advantage of functional languages: higher-order functions. Higher-order functions are functions which take functions as arguments or return functions. They are overall a sparsely discussed topic in a reversible setting — especially general higher order reversible functional programming, though some sources exist [7]. More interesting are reversible effects, which allow various interesting properties like reversible side effects and concurrency. Reversible effects have nice models in Category Theory as inverse arrows [19] and reversible monads [20].

We have seen that taking functions as arguments is unproblematic, given we can guarantee that the function is statically known. Then what is the roadblock regarding arbitrary higher-order functions in a reversible setting?

We should reiterate the fact that our ultimate goal is to design a *garbage-free* language. Being garbage-free especially entails that the amount of information we need to maintain for a program to remain reversible is minimal. We also remind the reader about the notion of a *trace*, which is a particular strategy to transform any non-reversible program into a reversible program by consistently store information critical to knowing the context in which a computation took place.

Now, let us temporarily adopt the view that higher-order functions are allowed in CoreFun. This implies that any variable is allowed to be assigned a function value. Say we have a function `twice` which takes a function f and returns a function which applies f twice, defined as `twice` $f = f \circ f$. Its type signature in an irreversible setting is `twice` : $(A \to A) \to (A \to A)$. Evidently it transfers well to a reversible setting, where we can define it as `rtwice` : $(A \leftrightarrow A) \leftrightarrow (A \leftrightarrow A)$, as the function f is transformed cleanly. Now, consider the following function:

```
g x = let f′ = rtwice f
      in f′ x
```

Where f is some static function defined for the program. We have that $g\ x$ will return $c = f(f(x))$. But given just the canonical form c, the context in which it was computed is not immediately clear. Consider computing g^{-1} — what is the function f'? In this particular instance we easily see that it is the function generated by `twice` on f, but this information is not known presently in the inverse direction. The issue becomes more evident when we consider some arbitrary f' which may have been dynamically introduced *anywhere*.

We are therefore forced to store information about *how* a value was produced if it is not clear from the syntactic construct itself, so we know how to invert it. Doing this requires that we for a function application of a dynamically introduced function return a *closure* containing the definition of the applied function f' as well as the result of $f'\ x$. The closure is, as a value, going to be treated as $f'\ x$ going forward, but storing f' becomes important for inverting the computation.

This one characteristic, which is unavoidable if we want to support full-on higher order behaviour, adds what surmounts to a pseudo-trace with heavy use of higher-order functions, making higher-order functions unattractive.

7.4 Future Work

7.4.1 Inductive First Match Policy Guarantee

The static first match policy analysis presented in Sec. 3 is only possible by investigating the open form of the program text but it also makes it somewhat limited. More specifically, it does not inspect the closed form of expression of recursive types, meaning case-expressions operating on recursively defined data structures may never generate static first match policy guarantees, even though there is a large class of functions for which it is strictly possible. Expressions of a recursive type require a more thorough analysis. What could such a method look like?

Here we adhere to an *inductive principle*, which we have to define clearly. We introduce a `plus` function to develop the subject:

```
plus $n_0$: $\mu X.1 + X$ $n_1$: $\mu X.1 + X$ =
  case unroll [$\mu X.1 + X$] $n_1$ of
    inl() ⇒ ($n_0$, $n_0$)
    inr($n'$) ⇒ let ($n'_0$, $n'_1$) = plus $n_0$ $n'$
               in let succ = roll [$\mu X.1 + X$] inr($n'_1$)
               in ($n'_0$, succ)
```

As in the well-known structural or mathematical induction, we must identify base cases for the induction hypothesis. A simple solution is to define these as the branches in which a function application to the function which is being evaluated does not occur. There might be multiple such branches without issue. Note that this does not work well with mutually recursive functions. For `plus` there is only one base case, the left arm of the main case-expression.

Analogously the inductive step is defined on each branch which contains a recursive call. For each recursive call the induction hypothesis says that, granted the arguments given to the recursive call, eventually one of the base cases will be hit. This is because any instance of the recursive type can only be finitely often folded, giving a guarantee of the finiteness of the decreasing chain. Though there is a catch which should be addressed: Inductive proofs are only valid for *strictly decreasing* chains of elements to ensure that the recursion actually halts. For example, for `plus` we need to make sure that $n' \prec n_1$. Should the chain not be strictly decreasing, we have that the evaluation is non-terminating, and the function is not defined for this input.

To tie it all together we need to show that the recursive call in the right arm of the `plus` function does indeed result in the base case in the left arm, allowing us to use the induction hypothesis to conclude that $n'_0 = n'_1$. If we are able to, we may directly treat the return value of the recursive function call as an instance of the value which the base case returns. We then continue evaluating the body in the inductive step. For `plus` we say that:

```
... ⇒ let ($n_0$, $n_0$) = plus $n_0$ $n'$
      in let succ = roll [$\mu X.1 + X$] inr($n_0$)
      in ($n'_0$, succ)
```

And now we can see that the case-arms are provably disjoint, giving us a static guarantee of the first match policy. However, generalizing an implementation of inductively derived first match policy guarantees requires usage of proof assistants, like *Coq* [6], on a growing number of identified cases and is rather complex. Further discussion of this has therefore been left for future work.

7.4.2 Possible Future Abstractions

The following is a short, informal bullet point list the authors suggest as additional possibilities for abstractions:

- Built in syntactic sugar for a number of common data structures. Lists can be further simplified in the following (ubiquitous) way: encoding [] as `Nil`, the empty list, $(x : xs)$ as `Cons` x xs, the head and tail of a list, and $[x_1, x_2]$ as `Cons` x_1 (`Cons` x_2 `Nil`), a literal list construction.

  A built-in `Char` data-type encoding characters can be wrapped in additional syntactic sugar to separate it from any regular variant. Then strings, simply encoded as lists over characters, can be represented as standard in quotation marks, encoding "$c_1c_2c_3$" as $[c_1, c_2, c_3]$

- A notion of modules, possibly inspired by the Haskell module system [11]. Code reuse and modularisation are powerful concepts which are easily transferable to a reversible setting in the form of programs spread over multiple files, as this method is strictly related to exposure of code.

Conclusion

8

Although CoreFun is a continuation of the work that was started with RFun, its abstract syntax and evaluation semantics are quite different and include more explicit primitive language constructs. However, we have also shown that CoreFun can be made lighter via syntactic sugar to mimic other functional languages.

We have presented a formal type system for CoreFun, including support for recursive types through a fix point operator and polymorphic types via parametric polymorphism. The type system is built on relevance typing, which is sufficient for reversibility if we accept that functions may be partial.

Evaluation has been presented through a big step semantics. Most evaluation rules were straightforward, but it was necessary to define a notion of leaves and a relation for "unification" as machinery to describe the side condition necessary to capture the first match policy.

An advantage offered by the type system is the ability to check the first match policy statically. A static guarantee that the first match policy should hold for a function will eliminate the run time overhead of case-expressions, often leading to more efficient evaluation. By the program syntax, we can check for orthogonality of inputs and the possible values of leaf expressions. By the type system, we can argue for the finiteness of a function's domain and exhaust the possible computational paths for a case-expression. Further, as was shown in future work, we can apply an induction principle for recursive types. However, it is difficult to detect exactly when this will yield a first match policy guarantee.

We noted that, in contrast to many reversible programming languages, the syntax of CoreFun does not easily support generation of inverse programs. This is not problematic as the relational semantics do make it possible to inverse interpret a program. We presented an inference system for inverse evaluation and showed how inverse application of functions is achieved.

Finally, we have argued that it is possible to enhance the syntax of CoreFun with high level constructs, which in turn have simple translation schemes back to the core language. We have presented numerous examples: variants, type classes (which, as an example, can be used to replace the duplication/equality operator in the original RFun language), top-level cases, records, arbitrarily sized products and abbreviated let-expressions.

Bibliography

[1] Anderson, A.R., Belnap, N.D.: Entailment: The logic of relevance and necessity, vol. 1. Princeton University Press (1975)

[2] Axelsen, H.B.: Time complexity of tape reduction for reversible turing machines. In: International Workshop on Reversible Computation. pp. 1–13. Springer (2011)

[3] Axelsen, H.B., Glück, R.: What do reversible programs compute? In: Hofmann, M. (ed.) Foundations of Software Science and Computational Structures. LNCS, vol. 6604, pp. 42–56. Springer-Verlag (2011)

[4] Axelsen, H.B., Glück, R.: On reversible turing machines and their function universality. Acta Informatica 53(5), 509–543 (2016)

[5] Bennett, C.H.: Logical reversibility of computation. IBM journal of Research and Development 17(6), 525–532 (1973)

[6] Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)

[7] Bohne, S., Widemann, B.T.: An approach for generalized reversible functional programming

[8] Cardelli, L.: Extensible records in a pure calculus of subtyping. Digital. Systems Research Center (1992)

[9] Church, A.: An unsolvable problem of elementary number theory. American journal of mathematics 58(2), 345–363 (1936)

[10] Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. Acm sigplan notices 46(4), 53–64 (2011)

[11] Diatchki, I.S., Jones, M.P., Hallgren, T.: A formal specification of the haskell 98 module system. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. pp. 17–28. ACM (2002)

[12] Dunn, J.M., Restall, G.: Relevance logic. In: Gabbay, D., Guenther, F. (eds.) Handbook of Philosophical Logic, vol. 6, pp. 1–192. Springer-Verlag, second edn. (2002)

[13] Giesl, J., Swiderski, S., Schneider-Kamp, P., Thiemann, R.: Automated termination analysis for haskell: From term rewriting to programming languages. In: International Conference on Rewriting Techniques and Applications. pp. 297–312. Springer (2006)

[14] Girard, J.Y.: Linear logic. Theoretical Computer Science 50(1), 1–101 (1987)

[15] Glück, R., Kaarsgaard, R.: A categorical foundation for structured reversible flowchart languages. In: Silva, A. (ed.) Mathematical Foundations of Programming Semantics (MFPS XXXIII). Electronic Notes in Theoretical Computer Science, vol. 336, pp. 155–171. Elsevier (2018)

[16] Glück, R., Kawabe, M.: A program inverter for a functional language with equality and constructors. In: Asian Symposium on Programming Languages and Systems. pp. 246–264. Springer (2003)

[17] Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: A scalable quantum programming language. In: Conference on Programming Language Design and Implementation, PLDI. pp. 333–342. PLDI '13, ACM (2013)

[18] Hall, C.V., Hammond, K., Peyton Jones, S.L., Wadler, P.L.: Type classes in haskell. ACM Transactions on Programming Languages and Systems (TOPLAS) 18(2), 109–138 (1996)

[19] Heunen, C., Kaarsgaard, R., Karvonen, M.: Reversible effects as inverse arrows. arXiv preprint arXiv:1805.08605 (2018)

[20] Heunen, C., Karvonen, M.: Reversible monadic computing. arXiv preprint arXiv:1505.04330 (2015)

[21] Huffman, D.A.: Canonical forms for information-lossless finite-state logical machines. IRE Transactions on Information Theory 5(5), 41–59 (1959)

[22] Hutton, G., Meijer, E.: Monadic parsing in haskell. Journal of functional programming 8(4), 437–444 (1998)

[23] James, R.P., Sabry, A.: Theseus: A high level language for reversible computing (2014), work in progress paper at RC 2014. Available at www.cs.indiana.edu/~sabry/papers/theseus.pdf

[24] Jones, M.P.: Functional programming with overloading and higher-order polymorphism. In: International School on Advanced Functional Programming. pp. 97–136. Springer (1995)

[25] Kaarsgaard, R.: The Logic of Reversible Computing: Theory and Practice. Ph.D. thesis, University of Copenhagen (2017)

[26] Kawabe, M., Glück, R.: The program inverter lrinv and its structure. In: International Workshop on Practical Aspects of Declarative Languages. pp. 219–234. Springer (2005)

[27] Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5(3), 183–191 (1961)

[28] Lecerf, Y.: Machines de Turing réversibles. Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences 257, 2597–2600 (1963)

[29] Lutz, C., Derby, H.: Janus: A time-reversible language. A letter to R. Landauer. `http://tetsuo.jp/ref/janus.pdf` (1986)

[30] Manolios, P., Vroon, D.: Termination analysis with calling context graphs. In: International Conference on Computer Aided Verification. pp. 401–414. Springer (2006)

[31] Matsakis, N.D., Klock II, F.S.: The rust language. In: ACM SIGAda Ada Letters. vol. 34, pp. 103–104. ACM (2014)

[32] McCarthy, J.: The inversion of functions defined by turing machines. Automata studies pp. 177–181 (1956)

[33] Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences 17(3), 348–375 (1978)

[34] Paolini, L., Piccolo, M., Roversi, L.: A certified study of a reversible programming language. In: LIPIcs-Leibniz International Proceedings in Informatics. vol. 69. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)

[35] Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)

[36] Polakow, J.: Ordered Linear Logic and Applications. Ph.D. thesis, Carnegie Mellon University (2001)

[37] Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society 74(2), 358–366 (1953)

[38] Sabry, A., Valiron, B., Vizzotto, J.K.: From symmetric pattern-matching to quantum control. In: Baier, C., Dal Lago, U. (eds.) Foundations of Software Science and Computation Structures. pp. 348–364. Springer International Publishing (2018)

[39] Schordan, M., Jefferson, D., Barnes, P., Oppelstrup, T., Quinlan, D.: Reverse code generation for parallel discrete event simulation. In: Krivine, J., Stefani, J.B. (eds.) Reversible Computation, pp. 95–110. 9138, Springer (2015)

[40] Schultz, U.P., Laursen, J.S., Ellekilde, L., Axelsen, H.B.: Towards a domain-specific language for reversible assembly sequences. In: Krivine, J., Stefani, J. (eds.) Reversible Computation, RC. vol. 9138, pp. 111–126 (2015)

[41] Thomsen, M.K.: A functional language for describing reversible logic. In: Specification & Design Languages, FDL 2012. pp. 135–142. IEEE (2012)

[42] Thomsen, M.K., Axelsen, H.B.: Interpretation and programming of the reversible functional language. In: Symposium on the Implementation and Application of Functional Programming Languages. pp. 8:1–8:13. IFL '15, ACM (2016)

[43] Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London mathematical society 2(1), 230–265 (1937)

[44] Wadler, P.: Linear types can change the world! In: IFIP TC 2 Working Conference on Programming Concepts and Methods. pp. 347–359. North Holland (1990)

[45] Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 60–76. ACM (1989)

[46] Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) Reversible Computation, RC '11. LNCS, vol. 7165, pp. 14–29. Springer-Verlag (2012)

[47] Yokoyama, T., Axelsen, H.B., Glück, R.: Fundamentals of reversible flowchart languages. Theoretical Computer Science (2015), dx.doi.org/10.1016/j.tcs.2015.07.046. Article in press

[48] Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Partial Evaluation and Program Manipulation. PEPM '07. pp. 144–153. ACM (2007)