

DIRK KLEEBLATT

---

# On a Strongly Normalizing STG Machine

With an Application to Dependent Type Checking



DIRK KLEEBLATT

---

Von der Fakultät IV – Elektrotechnik und Informatik –  
der Technischen Universität Berlin  
zur Erlangung des akademischen Grades  
Doktor der Ingenieurwissenschaften – Dr.-Ing. –  
genehmigte Dissertation

vorgelegt von  
Diplom-Ingenieur Dirk Kleeblatt aus Berlin

Promotionsausschuss:  
Prof. Dr. rer. nat. Sabine Glesner (Vorsitzende)  
Prof. Dr. rer. nat. Peter Pepper (Erster Gutachter)  
Prof. Dr. rer. nat. Petra Hofstedt (Zweite Gutachterin)

Tag der wissenschaftlichen Aussprache: 16. März 2011

Berlin, 2011

D 83

## On a Strongly Normalizing STG Machine

With an Application to Dependent Type Checking



## CONTENTS

---

I	INTRODUCTION	9
1	MOTIVATION	11
1.1	Dependent Types	11
1.2	Normalization	13
1.3	Normalization for Dependent Type Checking	15
1.3.1	Dependently typed languages	15
1.3.2	Proof Assistants Based on Dependent Types	16
1.4	Laziness	18
1.5	Goal of This Thesis	19
1.6	Outline	20
2	STATE OF THE ART	23
2.1	The Strongly Normalizing ZAM	23
2.2	The Strongly Normalizing Machine $\kappa$	23
2.3	The $\pi$ -Red System	24
2.4	Normalization by Evaluation	25
2.4.1	Typed Normalization by Evaluation	25
2.4.2	Untyped Normalization by Evaluation	26
II	NORMALIZATION USING COMPILED CODE	29
3	DEFINITION OF THE SEMANTICS: $s_4$	31
3.1	Language Syntax	32
3.2	Normal Forms	33
3.3	Heaps and Accumulators	35
3.4	Normalizing FUN expressions	36
3.5	Evaluation to WHNF	37
3.6	Read Back	40
3.7	Example	41
3.8	Well-formedness	43
4	LINEARIZATION: SSTG1	45
4.1	Stacks and Heaps	45
4.2	Weak Evaluation	46
4.3	Normalization and Read Back	48
4.4	Heap Correspondence	50
4.5	Equivalence of $s_4$ and SSTG1	51
4.5.1	Completeness	51
4.5.2	Soundness	52
4.5.3	Correctness	53
5	INTRODUCTION OF CLOSURES: SSTG2	55
5.1	Modifications of the Syntax	55
5.2	Environments	57
5.3	Normalization	57
5.4	Equivalence of SSTG1 and SSTG2	59

6	COMPILING TO MACHINE CODE: ISSTG	63
6.1	An Execution Environment for ISSTG	63
6.2	Compilation and Weak Evaluation	65
6.2.1	Compilation of Applications	67
6.2.2	Compilation of Local Bindings	67
6.2.3	Compilation of Case Discriminations	69
6.2.4	Preallocated Code Sequences	70
6.3	Strong Normalization and Read Back	71
6.4	Correspondence Between SSTG2 and ISSTG	71
6.5	Equivalence of SSTG2 and ISSTG	75
6.6	Correctness of Compiled Normalization	77
III	DEPENDENT TYPE CHECKING	79
7	PURE TYPE SYSTEMS	81
7.1	Basic Typing Rules	81
7.1.1	Syntax	82
7.1.2	Typing	83
7.1.3	Example: The Calculus of Constructions	85
7.2	Inductive Data Types	86
7.2.1	Syntax	87
7.2.2	Typing	87
7.2.3	Examples	88
7.3	Case Analyses	90
7.3.1	Syntax	91
7.3.2	Typing	91
7.3.3	Examples	92
7.4	Definitions by Fixed Points	93
7.4.1	Syntax	93
7.4.2	Typing	93
7.4.3	Examples	94
8	NOTES ON THE IMPLEMENTATION	95
8.1	Features of our Implementation	95
8.2	Fixed Points	97
8.2.1	New Syntax for FUN	98
8.2.2	Compilation to ISSTG	99
8.2.3	Read Back	100
8.3	Translation from PTS to FUN	101
8.3.1	Type Annotations	101
8.3.2	Constructor Saturation	102
8.4	Translation from ISSTG to x86 Machine Code	103
8.4.1	Memory Model	103
8.4.2	Translation of Instructions	104
8.5	Design Alternatives	106
9	PERFORMANCE EVALUATION	109
9.1	Benchmark Setting	109
9.2	Peano Numbers	110
9.3	Church Numbers	111

9.4	Interpretation of the Results	112
10	CONCLUSION	115
10.1	Future Work	115
IV	APPENDIX	117
A	EQUIVALENCE OF S4 AND SSTG1	119
A.1	Completeness of Weak Evaluation	119
A.2	Completeness of Read Back and Normalization	121
A.3	Soundness of Weak Evaluation	123
A.4	Soundness of Read Back and Normalization	128
B	PARTITIONING OF SSTG2 AND ISSTG REDUCTIONS	131
B.1	List of Segments	131
C	SOURCE CODE OF BENCHMARKS	137
C.1	Peano Numbers	137
C.1.1	PTS	137
C.1.2	Coq	138
C.1.3	Haskell	139
C.2	Church Numbers	139
C.2.1	PTS	139
C.2.2	Coq	140
C.2.3	Haskell	142



Part I

INTRODUCTION



## MOTIVATION

---

This thesis presents a new system for normalization of terms in functional languages. While many systems for normalization exist, our system, the strongly normalizing spineless tagless graph machine, or SSTG for short, has been designed with an emphasis on a special application area: checking of dependent types.

While several dependently typed systems have been implemented and published (see below for some references), one aspect of the type checking process has often been neglected: while there is consensus that a normalization procedure is needed for dependent type checking, in our opinion the implementation of such a procedure has not yet attracted the attention it deserves. As a consequence, most published dependent type checkers use interpreters for the normalization of terms. On the one hand, this is comprehensible, since the requirements posed by the type checker are admittedly demanding, and more easily fulfilled using an interpreted system. On the other hand, compiler construction is an established branch of computer science for quite a while now, and the widespread use of compiled languages shows that there is a need for more efficient techniques than simple interpretation of source terms.

This thesis wants to bring forward the connection between the two fields of dependent type checking and compiler construction, hopefully aiding further collaboration.

In this introductory chapter, we set the scene to motivate the goal of this thesis, and to give an informal impression why the application of compiled code is problematic during dependent type checking. Later chapters will detail our approach to normalization during dependent type checking, using a more formal style than this introduction.

Parts of this thesis were previously published [18, 22, 23, 24].

### 1.1 DEPENDENT TYPES

There is a very active research community around the theory, application and implementation of dependent types. To give an intuitive feeling, definition 1.1 captures this notion informally but sufficiently for the sake of this introduction.

**DEFINITION 1.1** (Dependent Type) *A dependent type is a type that depends on a value.*

---

```
1 program Arrays1;
2 var
3     i    : Integer;
4     A, B : Array [1 .. 10] of Integer;
5 begin
6     for i := 1 to 10 do
7         A[i] := i;
8     B := A;
9     writeln(B[4]);
10 end.
```

---

Listing 1.1: An Example for Arrays in Pascal

---

```
1 program Arrays2;
2 var
3     i    : Integer;
4     A    : Array [1 .. 10] of Integer;
5     B    : Array [1 .. 20] of Integer;
6 begin
7     for i := 1 to 10 do
8         A[i] := i;
9     B := A;
10    writeln(B[4]);
11 end.
```

---

Listing 1.2: An Erroneous Example of Arrays in Pascal

As a simple case of dependent types consider the array types of Pascal, introduced by Wirth in [34]. The example program in listing 1.1 declares two arrays A and B (line 4). In the declaration, not only the content type is defined, but also the array *domain* is fixed as the interval from 1 to 10. The *type* of A depends on the *values* 1 and 10.

The remainder of the program should be self explanatory, after compilation this program outputs “4” as expected.

Listing 1.2 differs only in the declaration of the second array B. This time, the array indices range from 1 to 20. As the array domain is part of the type, the arrays A and B have different types in this example. Thus, the assignment to B (line 9) is ill-typed and a Pascal compiler rejects this program.

The Java standard, published by Gosling et al. in [17], takes another approach. Listing 1.3 gives an equivalent Java program. The types given for A and B (lines 3 and 4) are `int[]` in both cases, even though they are initialized with arrays of different lengths. The array length is *not* part of the type according to the Java specification. Since A and B have the same type, the assignment (line 8) is well-typed and the program outputs “4” after compilation.

---

```

1 class Arrays {
2     public static void main(String[] args) {
3         int[] A = new int[10];
4         int[] B = new int[20];
5
6         for (int i = 0; i < A.length; i++)
7             A[i] = i;
8         B = A;
9         System.out.println(B[4]);
10    }
11 }

```

---

Listing 1.3: An Example for Arrays in Java

This shows an important property of dependent type systems: typically, more programs are rejected than in languages without dependent types. Another important consequence can be seen when the output statements in listings 1.1 and 1.3 (line 9 in both programs) are mistakenly changed to print `B[40]` instead of `B[4]`. A Pascal compiler rejects the changed program, since the array boundaries are violated. A standard-conforming Java compiler accepts the changed program, which results in a run-time error after compilation. In general, dependent type systems can ensure invariants at compile-time that can only be tested at run-time in languages without dependent types.

Dependent types are not only found in programming languages, but also play an important role in proof theory via the Curry-Howard-correspondence that relates types and propositions as well as terms and proofs. Thus, dependent type checkers can be found in proof assistants, too. We show an example in section 1.3.

## 1.2 NORMALIZATION

Again, we introduce the notion of normalization informally, postponing a formal treatment to later chapters.

**DEFINITION 1.2** (reduction system, normal form, normalization)  
*A reduction system is defined as a set of states and a relation describing state transitions. A normal form is a state that cannot do a transition to a next state. The process of finding such a normal form is called normalization.*

The semantics of functional programming languages is often given as a variant of the  $\lambda$ -calculus extended with more advanced language constructs as e.g. pattern matching and local bindings. Running a functional program, i.e. evaluating a functional expression, then means to normalize an expression.

There are two different approaches to this normalization process.

**INTERPRETERS** analyze the expression and recognize patterns that allow a state transition. If a pattern is recognized and matches some subexpression in the state, this subexpression is often called *redex*, short for reducible expression. Then, this transition is performed, and the analysis is repeated until no transitions are possible anymore.

**COMPILERS** translate the expression to code that can be executed by some machine, either a physical processor or a virtual machine. Executing this generated code gives the normal form of the translated expression.

On the one hand, the execution of the code created by a compiler is typically way more efficient than the interpretation of the source program, especially when code for physical processors is generated. On the other hand, interpreters allow quite flexible language designs that cannot easily be dealt with in compiler systems.

Reduction systems can be divided into two classes.

**STRONG REDUCTION** allows state transitions regardless of the context of a redex.

**WEAK REDUCTION** performs state transitions only in restricted contexts.

A typical example to clarify the difference is the handling of *unsaturated functions*, i. e. functions not applied to all expected arguments. Consider the following  $\lambda$ -expression  $e$ .

$$e := (\lambda f. \lambda x. f\ x)(\lambda y. y)$$

This expression can do a  $\beta$ -reduction to expression  $e'$ .

$$e' := \lambda x. (\lambda y. y)\ x$$

A typical weak reduction system would stop with this expression and would not perform any further state transitions. The syntactical criterion for stopping the evaluation at this stage is the presence of an unsaturated function: no value for the formal parameter  $x$  is given. Put in other words, weak reduction does not perform  $\beta$ -reduction under  $\lambda$ -abstractions. A normal form in a weak reduction system is often called *weak head normal form* or **WHNF** for short.

With a strong reduction system,  $e'$  can do a second transition to  $e''$ .

$$e'' := \lambda x. x$$

---

```

1  append :: (a :: #) -> (n :: Nat) -> Vector a n
2                -> (m :: Nat) -> Vector a m
3                -> Vector a (n + m)

```

---

Listing 1.4: Appending Lists With Fixed Length in Cayenne

Regardless of evaluation contexts,  $e''$  does not give any opportunities for further state transitions, so  $e''$  is a normal form, sometimes also called *strong* normal form to emphasize the difference to WHNFS.

Typically, compiled systems only perform reduction to WHNFS, which is appropriate in the most common case where normalization is needed: running functional programs. Dealing with e.g. reduction under  $\lambda$ -abstractions is one of the features more easily implemented in interpreters. The reason is that in the last transition,  $x$  has to be substituted for  $y$ , while  $x$  is not a value, but a free variable in the reduced subexpression, and therefore is not bound to a concrete value at run time.

### 1.3 NORMALIZATION FOR DEPENDENT TYPE CHECKING

At first sight, the problems of dependent type checking and normalization are not related. However, the following examples show that a normalization procedure is a required ingredient of most type checkers for dependently typed languages.

#### 1.3.1 *Dependently typed languages*

Many languages with dependent type systems allow user-defined functions from values to types that capture the dependencies. The values a type depends on may be given using arbitrary functions, even user defined ones. For example, in Cayenne, introduced by Augustsson in [4], it is possible to define a function `Vector` with type `# -> Nat -> #`, where `#` denotes the type of all types. Thus, `Vector` takes a type and a number as parameters and returns a new type, which represents lists of fixed length with a given content type. Thus, `Vector` defines a new dependent type that depends on a natural number interpreted as the length of a list. Using this definition, a function `append` to concatenate two vectors can be declared as in listing 1.4.

The type of `append` can be read as follows: for any type  $a$ , given a natural number  $n$ , and a vector containing  $n$  elements of type  $a$ , and furthermore a second natural number  $m$  together with a vector of size  $m$ , return a vector containing  $n$  plus  $m$  elements. This type gives quite a good specification of the `append` function. In the function definition (not shown here), we have to deal with

arbitrary vectors of arbitrary lengths. That means  $n$  and  $m$  occur as free variables in the types of the body. Moreover, the definition of the addition operator has to be applied to check the body, i. e. a normalization procedure is needed. And since this operator is applied to free variables here, this procedure has to be able to deal with free variables for the type checker of Cayenne. For other reasons that will be made explicit in later chapters this normalization procedure has to be strong and must not stop at WHNFS.

Thus, in a dependently typed language, expressions may be evaluated at two separate stages: during type checking, and when running the user's program proper. Care has to be taken to ensure that the same semantics is applied in both stages, otherwise type safety might be violated. On the one hand, from a software engineering standpoint it is best to use the same normalization mechanism in both stages, i. e. the same compiler or interpreter, since in this case there cannot be two deviating implementations. But on the other hand, the most widespread and efficient compilation techniques lead to weak normalization only and are thus not adequate for the type checking stage. This severely limits the number of candidates for possible evaluation implementations for dependently typed languages.

### 1.3.2 Proof Assistants Based on Dependent Types

An example of a proof assistant making use of dependent types is Coq [28]. It is based on Coquand's and Huet's calculus of constructions [10], the most complete system of Barendregt's  $\lambda$ -cube [6]. In Coq the Curry-Howard-correspondence is fully exploited: to define a proposition, one has to write down a type, and to prove a proposition, a term with this type has to be constructed. This construction may be made explicitly by giving a term of the correct type, or implicitly by constructing the term using so-called tactics. In the latter case, the term may be retrieved from the sequence of tactics after the construction is complete.

Figure 1.5 shows a session with the interactive interface of Coq. In line 1 we state (after Coq's prompt "Coq <") that we want to prove a lemma that we call `add_zero`, stating that  $x + 0 = x$  for all natural numbers  $x$ . Coq responds with repeating the goal of our proof, giving all hypotheses and assumptions we may use above a vertical line. Since we do not have any hypotheses yet, there is nothing to see.

In line 6 (the prompt having changed to "`add_zero <`" to make clear which lemma we are about to prove), we make use of the tactic `intro x`, corresponding to the natural language proof phrase "Assume  $x$  is a natural number". This results in  $x : \text{nat}$

---

```

1 Coq < Lemma add_zero : forall (x : nat), x + 0 = x.
2
3 =====
4   forall x : nat, x + 0 = x
5
6 add_zero < intro x.
7
8   x : nat
9   =====
10    x + 0 = x
11
12 add_zero < induction x.
13
14 =====
15    0 + 0 = 0
16
17 subgoal 2 is:
18   S x + 0 = S x
19
20 add_zero < simpl.
21
22 =====
23    0 = 0
24
25 subgoal 2 is:
26   S x + 0 = S x
27
28 add_zero < apply refl_equal.
29
30   x : nat
31   IHx : x + 0 = x
32   =====
33    S x + 0 = S x
34
35 add_zero < simpl.
36
37   x : nat
38   IHx : x + 0 = x
39   =====
40    S (x + 0) = S x
41
42 add_zero < rewrite IHx.
43
44   x : nat
45   IHx : x + 0 = x
46   =====
47    S x = S x
48
49 add_zero < apply refl_equal.
50 Proof completed.

```

---

Listing 1.5: A proof of  $x + 0 = x$

being added to our list of assumptions. Next we perform an induction on  $x$  in line 12. The base case now is our main goal, we have to show that  $0 + 0 = 0$ . Before returning to the prompt, Coq informs us in Line 18, that there is a second subgoal waiting for us, namely the inductive step, that we will prove after the base case.

In line 20 we advise Coq to simplify the goal. The natural numbers are no built-in structure in the implementation of Coq, but defined in a library as a data structure with constructors for zero and successor. Consequently, the addition operation has the expected definition in the library as a recursive function, performing case analysis on the first argument. This definition is used, normalizing  $0 + 0$  to  $0$ . While this is a first example of the evaluation of a user defined function, it is quite uninteresting, since there are no free variables in the goal, and computing  $0 + 0$  poses no special problems for existing compilers. Nevertheless, the simplification allows us to apply the reflexivity of equality in line 28, leaving us with the second subgoal, our inductive step, to be proved.

In the inductive step, we find the induction hypothesis  $IHx$  among our assumptions that we may use in our proof. However, the goal has not yet the right shape to apply the hypothesis, so we perform a second simplification in line 35. Again, Coq uses the definition of the addition function to perform a normalization. But observe that this time the arguments are  $S\ x$  and  $0$ , so now a computation containing a free variable has to be performed. While for this particular computation the complexity is extraordinarily low, such that using an interpreter does not lead to severe performance problems, it is easy to imagine that we might find very complicated functions in a similar situation in other proofs. It is this constellation that is the target of our compilation system introduced in the main part of this thesis.

Using the equality  $IHx$  to rewrite the goal in line 42, we can finish the proof using reflexivity of equality a second time in line 49.

#### 1.4 LAZINESS

From the  $\lambda$ -calculus, different evaluation strategies are known. A collection of a number of strategies has been assembled by Barendregt in [5]. They vary in the order in which function applications are reduced, and these variations are reflected in implementations of different functional programming languages.

Probably the most widespread evaluation strategy in programming languages is related to the *applicative order* reduction. Here, all function arguments are normalized before the function appli-

cation is evaluated. This is also called *eager evaluation* and usually permits the most efficient implementations.

Nevertheless, another evaluation strategy from  $\lambda$ -calculus can be found in some language implementations, namely *normal order* reduction. In this strategy, function applications are considered before normalizing the arguments, thus the function bodies work with unevaluated arguments. These unevaluated subexpressions get only evaluated when actually needed. A variant of this strategy avoids duplicated evaluations when arguments are used more than once by building indirections via pointers and overwriting subexpressions with their normal form. The details of this strategy called *lazy evaluation* are shown in later chapters.

While eager evaluation is more efficient in many cases, lazy evaluation has its advantages.

- Lazy evaluation allows certain programming styles that can ease the implementation of some algorithms, most prominently by infinite data structures, e. g. infinite lists. In proof assistants, this corresponds to coinductive definitions.
- Whenever a normal form exists, it can be found by lazy evaluation. This is not the case for eager evaluation, so when we are working e. g. with pure type systems, as described by Barendregt in [6], where not necessarily all terms are normalizing, we might face typings that can be checked using lazy evaluation but not using eager evaluation.

These advantages make lazy evaluation the standard evaluation strategy for some languages, as for example Haskell.

## 1.5 GOAL OF THIS THESIS

The exposition above makes it possible to state the goal of this thesis as follows.

**GOAL OF THIS THESIS** is to present a normalization system that is suitable for dependent type checking and employs compiled code using lazy evaluation.

**THIS GOAL IS WORTHWHILE** because dependently typed systems find useful applications in programming languages and proof assistants, checking dependent types needs normalization systems, compiled code is much faster than interpreted code, and the advantages of lazy evaluation are relevant to programming languages as well as proof assistants. To the best of our knowledge, this goal has not yet been reached by existing systems.

## 1.6 OUTLINE

In the following chapters, we will show how this goal can be reached. We start with an overview of former work on strongly normalizing systems (chapter 2). These systems are very well suited for their intended use cases, but do not fulfill our goal, e. g. because they employ eager evaluation instead of laziness, or their results have  $\eta$ -long normal forms, sometimes unsuitable for proof assistants, or they have a broader scope of application, thus introducing unnecessary administrative structures at run time, leading to systems more complex than required for dependent type checking.

We continue with a presentation of our new normalization system. We give a definition of a simple functional language called `FUN` (chapter 3), that can be thought of as an intermediate language between a full-fledged language intended for the programmer of a dependently typed language or the user of a proof assistant on the one hand, and on the other hand the machine instructions of the processor architecture that are executed after compilation. However, `FUN` is fairly close to the programmer's point of view, abstracting away the intricacies of individual instructions at the machine level. We give a weakly normalizing big-step semantics for `FUN`, building on the work of Encina and Peña [15], and an extension to strong normalization taking some ideas from Leroy and Grégoire [19, 20].

While a checker for dependent types can be implemented directly using these definitions, such an implementation would lead to an interpreter, not providing the efficiency we aim for. Thus we give stepwise refinements to small-step operational semantics (chapters 4 and 5), that cover more and more aspects of compiled languages. We show how these modified systems preserve normal forms.

A further step takes us to a compilation system, generating pseudo machine instructions from `FUN` expressions, and still allowing the reduction to strong normal forms (chapter 6). While we still abstract from hardware-dependent details as e. g. register allocation and some aspects of the memory model, generating machine code for physical processors out of these instructions is simple. This system still captures the normalization semantics defined in chapter 3.

We close this thesis with a case study, giving evidence that our goal of an efficient strongly normalizing system has been reached, and that it is indeed suitable for dependent type checking. We present a definition of syntax and typing rules of a prototype implementation of pure type systems, enriched with data types, case expressions and fixed points (chapter 7). The implementation is powerful enough to be used for theorem proving, but lacks the

bells and whistles that are essential for a proof assistant proper, as e.g. support for proof libraries, support for automatically searching for proofs, and an extensible tactic language. Our type checker is able to interpret the pseudo machine instructions from chapter 6, but can also translate them to machine code for x86 processors before execution. We give an overview over our implementation (chapter 8), before we finally show some performance numbers that support our claim of an efficient implementation (chapter 9).



## STATE OF THE ART

---

This chapter presents former approaches of strongly normalizing systems that either have been used for dependent type checking or at least appear to be suitable for such usage. We show, why these approaches do not fulfill the goals of this thesis.

### 2.1 THE STRONGLY NORMALIZING ZAM

Grégoire and Leroy [19, 20] describe an implementation of strong normalization using compiled code. Their system was integrated into the proof assistant Coq that is based on a dependent type theory. This integration significantly increased the efficiency of the type checker.

This normalization system is based on the Zinc abstract machine (ZAM, proposed by Leroy in [27]), an abstract machine originally used to implement weak reduction for ML compilers. With a few modifications, this machine was extended with a so-called read back phase. Strong normalization then starts with a weak normalization, followed by read back that detects further redices and restarts their weak normalization, until no redices remain.

This system performs eager evaluation, since it is based on the ZAM that was intended for the strict language ML. For an integration into Coq, this is an appropriate choice, since strict evaluation is fairly efficient in most cases, and all reduction sequences of well typed Coq terms are finite, avoiding any problems with non-termination.

However, as explained in the introduction, lazy languages have their applications, therefore we present in this thesis a similar though lazy system.

### 2.2 THE STRONGLY NORMALIZING MACHINE K

In [11], Crégut presents two extensions of Krivine's machine  $\kappa$  [26], called  $\kappa_N$  and  $\kappa_{NP}$ , that perform strong normalization.

The first variant  $\kappa_N$  uses a call-by-name semantics, thus no sharing of reductions for duplicated arguments is implemented. The machine definition is given in a style that combines small step reduction rules with big step semantical rules. To be able to implement such a heterogeneous semantics, a collection of stack marks is invented, that together with output produced as side effect of reduction allows to construct strong normal forms. An

alternative formulation uses a meta-level controller that is close to Grégoir's and Leroy's read back phase.

The second machine `KNP` does share reduction sequences, but requires run-time code generation to do so. This makes the amount of code produced for strong normalization unbounded.

Since an efficient implementation of sharing is crucial for a nonstrict language, `KN` and `KNP` do not fulfill our needs. The code management problems with `KNP` complicate matters to an extent that Crégut in [11] comes to the conclusion that compiled code for `KNP` is probably not the right solution.

In a more recent publication [12], Crégut defines further variants of `KN` and `KNP` with very similar properties.

None of these abstract machines supports algebraic data types with constructors and case discrimination, important features for functional languages and supported by our machines.

### 2.3 THE $\pi$ -RED SYSTEM

The  $\pi$ -Red system of Gärtner and Kluge [21] was developed as an aid for teaching functional languages as well as a tool for program development and debugging. It allows to reduce functional expressions stepwise, and prints the resulting expression after a user defined number of steps. The reductions are carried out even under  $\lambda$ -abstractions, so a strong normalization can be obtained by providing a sufficiently large number of reductions. When no normal form is reached in the specified number of steps,  $\pi$ -Red computes a term containing residual redices.

All reductions are performed using compiled code. Since the compiled code cannot execute the bodies of  $\lambda$ -abstractions when their arguments are not present,  $\pi$ -Red contains a so-called  $\eta$ -extension unit that generates pseudo arguments and is interleaved with code execution the same way as weak evaluation and read back of the `ZAM` normalizer.

While the original presentation of  $\pi$ -Red deals only with strict evaluation, Kluge presents in [25, chapter 10] a lazy variant of the  $\pi$ -Red system.

The broad focus of  $\pi$ -Red comes at a cost. To be able to stop computations after arbitrary numbers of reductions, and to reconstruct source code for unevaluated closures, instructions of the  $\pi$ -Red abstract machine operate in one of two possible modes. Before the number of reductions is exceeded, operation is similar to other abstract machines for functional languages. But thereafter, the machine switches to a second mode, where some instructions behave differently. E.g. the instruction for function application does not call the applied function anymore, but merely collects arguments from a stack and saves them into a graph structure. To reconstruct source code expressions from this

graph structure, all function pointers are referenced indirectly via descriptors during computations.

Since evaluation for dependent type checking has no need to stop evaluation after arbitrary numbers of reductions, we strive for a simpler approach without several operation modes and without the overhead of additional indirections for function pointers.

## 2.4 NORMALIZATION BY EVALUATION

The normalization by evaluation approach translates in a first step a syntactical term into a semantical object. That is, for example, a  $\lambda$ -abstraction is translated into a mathematical function, and a function application into the value of the (semantical meaning of the) operator expression at the (semantical meaning of the) operand expression.

In a second step, the semantical object is *reified*, i.e. a term representation is obtained. The reification procedure ensures to yield normal forms, and since this normal form has the same semantical meaning as the original syntactical term, this method is sound.

Normalization by evaluation can be applied in typed and in untyped settings.

### 2.4.1 Typed Normalization by Evaluation

For the simply typed  $\lambda$ -calculus, Berger and Schwichtenberg give a definition of normalization by evaluation [7]. It was extended to type-directed partial evaluation (TDPE) by Danvy in [14] for richer functional languages that include for instance some base types and conditionals. Later, an extension from simple types to Martin-Löf type theory was found by Abel et al. [1].

In an implementation of normalization by evaluation, the translation into semantical objects maps e.g. a  $\lambda$ -abstraction term into an abstraction, i.e. function, of the implementation language. When the original abstraction term has a simple type, it corresponds to a polymorphic function. This function can now be called with some textual representation of a free variable.

As an example, consider the  $\lambda$ -term  $(\lambda x.x)(\lambda y.y)$  with the simple type  $\mathbf{o} \rightarrow \mathbf{o}$ . If we take Haskell as an implementation language, we can translate this term into a Haskell function defined as  $f = (\backslash x \rightarrow x)(\backslash y \rightarrow y)$  with  $f :: \text{alpha} \rightarrow \text{alpha}$ . Suppose, we now want to obtain a normal form represented with the type data  $\text{NF} = \text{Var String} \mid \text{App NF NF} \mid \text{Lam String NF}$ . Then, we can evaluate  $\text{Lam "z"} (f (\text{Var "z"}))$  according to the Haskell semantics and *using an existing Haskell compiler* to the value  $\text{Lam "z"} (\text{Var "z"})$ . The decision, which Haskell term has

to be evaluated to obtain a normal form by reification is based solely on the type of the expression to be normalized.

While relying on the type of an expression in this way allows a quite efficient implementation, it has some drawbacks, too. First, it can be shown that it produces  $\eta$ -long normal forms, which may be not what is needed, as e. g. the calculus of constructions underlying Coq does not allow  $\eta$ -expansions, i. e. there is no proof of  $f = \lambda x. f x$  in Coq without additional axioms.

A second drawback gets obvious when we consider structured base types, e. g. boolean truth values or natural numbers. If we need a normal form of some function  $g :: \text{Bool} \rightarrow \text{Nat}$ , TDPE suggests to take  $\lambda b. \text{if } b \text{ then } g \text{ True else } g \text{ False}$ . The idea is simple: we evaluate  $g$  at both elements of its domain, and put both values into a conditional. While this looks good at the first moment, a closer look reveals a problem. Consider the constant Haskell function  $g = \lambda b \rightarrow 42$ , where TDPE gives  $\lambda b. \text{if } b \text{ then } 42 \text{ else } 42$ . Here, TDPE assigns a strict normal form to a non-strict function, which is quite unsatisfactory.

The situation gets even worse when we consider some function  $h :: \text{Nat} \rightarrow \text{Bool}$ . In this case, TDPE cannot deliver a normal form, since it is not possible to evaluate  $h$  on all elements of its infinite domain.

The system presented in this thesis does not suffer from these problems: it does not perform  $\eta$ -expansions, does not change the strictness properties of normalized function terms, and allows to deal with infinite function domains.

Another disadvantage of TDPE when used for dependent type checking is the software engineering problem mentioned in section 1.3. While it seems easy to use TDPE during the type checking stage for normalization, it is an uncommon approach for the final execution stage of programs. In fact, we are not aware of a programming language implementation that uses TDPE for program execution. To use TDPE at this stage, a great deal of compiler construction knowledge has to be abandoned, and lots of techniques have to be developed to reach the state of the art in areas of, for example, separate compilation, modularization, optimization, or generation of standalone executables.

#### 2.4.2 *Untyped Normalization by Evaluation*

Normalization by evaluation can be applied even when no type information is available. Implementations can be found by denotational approaches, as shown by Filinski and Rohde in [16], and operational approaches, as demonstrated by Aehlig and Joachimski in [2]. To improve efficiency, a compiled version was published by Aehlig et al. in [3], translating source terms to ML programs that compute normal forms upon execution.

Again, terms are translated to semantical objects that get reified in a second step. But since no type information can be consulted, additional administrative layers are needed to implement closures or to discriminate between functions and free variable. These layers of the normalization by evaluation implementation add to the administrative layers already inevitably introduced by the implementation language. Such additional layers are avoided by our approach.



## Part II

### NORMALIZATION USING COMPILED CODE



## DEFINITION OF THE SEMANTICS: $S_4$

---

In this chapter, we present `FUN`, a simple lazy functional language, and a strong normalization system for `FUN`. While `FUN` is an extension of pure  $\lambda$ -terms with local bindings, constructors and pattern matching, it is intended as a intermediate language due to its restrictions. Translation of functional programs to `FUN` is straightforward.

The strong normalization of `FUN` expressions is defined via two stages: weak evaluation and read back. Every strong normalization starts with a weak evaluation, reaching a `WHNF`. This `WHNF` is then read back, thereby extracting all redices still contained in the `WHNF` and reducing them in a subsequent iteration.

The original definition of `FUN` and its evaluation to `WHNF` is due to Encina and Peña [15]. They give a big-step weak evaluation semantics called  $s_3$ . Reflecting the basement on their work, we call our big-step semantics  $s_4$ .

The extension of this semantics to strong normalization is new. For this extension we make use of accumulators, a representation of free variables and other stuck redices, originally introduced by Grégoire and Leroy [19, 20] for their strict normalization system. The division of strong reduction into weak evaluation and read back is analogous to their work, too. We adopt their work to the needs of lazy evaluation with a variant of the spineless tagless graph machine, `STG` for short, originally introduced by Peyton Jones and Salkid [31]. While the original accumulators were merely data structures, scrutinized by the normalization routines, our accumulators play a more active role.

In this presentation, we do not deal with two issues that have to be addressed in product quality implementations. Both are orthogonal to the problem of strong normalization and thus can be easily added.

- We present an untyped system, but will ignore any run time errors like case discrimination on  $\lambda$ -expressions. In practice, either a type checker has to be used that can statically guarantee the absence of such errors, or some run time type checking has to be implemented.

Our case study in part III performs type checks before normalization.

- Evaluating expressions may lead to non-termination. While we integrate the use of blackholing that can detect simple loops, a complete termination check is not possible, of

course. Practical implementations have several options regarding infinite computations: the simplest solution is to ignore this problem, leaving the possibility of a non-terminating type check, which might be unsatisfactory for our users. Another approach is to introduce some kind of counter or timer that aborts normalization after some number of reduction steps or a after certain time span elapsed. Finally, it is possible to enforce termination by a type system: some variants of pure type systems are strongly normalizing, thus every well-typed term has a normal form. Coq is based on the calculus of constructions, an instance of a strongly normalizing pure type system, and hence avoids this problem. However, we lose Turing completeness when choosing this route.

Our case study in part III can be configured to check arbitrary pure type systems, depending on this configuration infinite computations are either ruled out by the type system or ignored by the implementation otherwise.

### 3.1 LANGUAGE SYNTAX

The syntax of `FUN` is given in figure 3.1. We make use of unspecified syntactical categories of variables, pointers and constructors. Variables are denoted by the meta variables  $x$  and  $y$ , pointers by  $p$ ,  $q$  and  $r$ , and constructors by  $C$ .

Here and in the following boldface font signifies sequences of objects, i. e. while  $x$  is a variable,  $\mathbf{x}$  is a sequence of variables. Indexing of sequences starts with 1, so  $p_1$  is the first pointer of the sequence  $\mathbf{p}$ . The length of sequence  $\mathbf{p}$  is written  $|\mathbf{p}|$ .

In the simplest case a `FUN` expression is a reference, i. e. either a variable  $x$  or a pointer  $p$  to the heap. A description of heaps is given below.

Function application is restricted to references in argument position, but more than one may be given as argument. Thus,  $e \mathbf{ref}$  abbreviates  $(\dots (e \mathbf{ref}_1) \dots) \mathbf{ref}_n$ .

Variables can be bound to so-called  $\lambda$ -forms  $\mathbf{lf}$  with `let`, where the right hand side might be another expression, a constructor application that has to be saturated (i. e. all arguments must be given), or a  $\lambda$ -abstraction that may abstract over several variables at once, such that  $\lambda \mathbf{x}. e$  abbreviates  $\lambda x_1 \dots \lambda x_n. e$ . Constructors and abstractions may only occur in local bindings.

Definition by cases can be done for arbitrary expressions. A sequence of alternatives `alt` relates constructors to expressions, binding the constructor arguments to variables (if any).

We use the notions of free variables, open and closed terms and substitutions as it is usual in language descriptions, without giving explicit definitions here. A substitution of the variables

---

```

e ::= ref
   | e ref
   | let x = lf in e
   | case e of alt

ref ::= x | p

lf ::= e
    | C ref
    | λx. e           where |x| > 0

alt ::= C x -> e

```

---

Figure 3.1: Syntax of FUN

$x$  with the pointers  $p$  in expression  $e$  is denoted by  $e[p/x]$ . At every occurrence of such a substitution we assume implicitly that  $|p| = |x|$  holds.

Translating unrestricted functional terms to FUN terms is easy: if the original term contains complex function or constructor arguments, bind them with  $\lambda$ -forms. Constructors and  $\lambda$ -expressions not occurring on the right hand side of local definitions are dealt with in the same way. An unsaturated constructor  $C$  can be saturated by an  $\eta$ -expansion to  $\lambda x. C x$ . If this  $\eta$ -expansion is unwanted in the resulting normal form, it is easy to mark these expansions and perform a corresponding  $\eta$ -reduction after normalization, as it has been implemented in our case study.

### 3.2 NORMAL FORMS

Strong normal forms of FUN expressions are defined in figure 3.2. A normal form is either a constructor with normal forms in argument position, or a  $\lambda$ -abstraction with a normal form in its body, or an unreducible head.

An unreducible head  $h$  is either a plain variable, or an application of an unreducible head to a normal form, or a case discrimination where the scrutinee is not a constructor but another unreducible head.

As an example consider the expression  $e$  for computing  $1 + 1$  using church numerals.

```

e := let
      plus = λx y s z. let
                    y' = y s z
                  in x s y'
      one = λs z. s z

```

---


$$\begin{aligned}
 v &::= C\ v \\
 &\quad | \lambda x. v \\
 &\quad | h \\
 h &::= x \\
 &\quad | h\ v \\
 &\quad | \text{case } h \text{ of } v \text{alt} \\
 v\text{alt} &::= C\ x \rightarrow v
 \end{aligned}$$


---

Figure 3.2: Strong Normal Forms of FUN

in plus one one

The normal form of  $e$  that can be obtained using our system is  $v$ , the church numeral for 2, as expected.

$$v := \lambda s. \lambda z. s (s z)$$

Maybe surprisingly  $v$  does not obey the same syntax as  $e$ : in normal forms, function and constructor arguments are not restricted to be variables, as is demonstrated by the complex argument  $(s z)$  given to  $s$  in  $v$  above. Furthermore,  $\lambda$ -abstractions and constructors are not restricted to right-hand sides of local bindings. While it is somewhat unusual to have normal forms outside the original language definitions, this is less of an issue when we consider the intermediate-language character of FUN. We can think of  $e$  as the translation result of  $e'$  given in another functional language without the restrictions of FUN:

$$\begin{aligned}
 e' &:= \text{let} \\
 &\quad \text{one} = \lambda s z. s z \\
 &\quad \text{in } (\lambda x y s z. x s (y s z)) \text{ one one}
 \end{aligned}$$

It is easy but unnecessary for this presentation to define such a language that is a superset of the normal forms given in figure 3.2 together with an appropriate translation.

But another restriction is added to normal forms that is not applied to FUN expressions. Abstractions may abstract over several variables at once in FUN, but not in the normal forms. While the abstract machines defined in the following chapters can enhance efficiency by considering multiple abstracted variables in one step, such considerations are not important for normal forms. Additionally, we can simplify our presentation a little bit by restricting abstractions in normal forms to single variables.

---


$$\begin{array}{lcl}
\text{hv} & ::= & \text{lf} \\
& | & \text{q} \mapsto \text{p} \\
& | & \text{k} \\
\\
\text{k} & ::= & \langle x \rangle \\
& | & \langle \text{p } \text{q} \rangle \\
& | & \langle \text{case } \text{p} \text{ of } \text{alt} \rangle
\end{array}$$


---

Figure 3.3: Heap Values and Accumulators for  $S_4$ 

## 3.3 HEAPS AND ACCUMULATORS

Since we want to employ lazy evaluation, we have to implement *sharing* to avoid duplicated evaluations of subterms. Thus, we need a *heap*, i.e. a mapping from pointers to heap values  $\text{hv}$ , according to the grammar in figure 3.3.

In our system sharing is implemented in the same way as in other implementations of lazy languages: we allocate each expression on the heap that is bound by `let`, hence the first kind of heap values are  $\lambda$ -forms.

After an unevaluated expression allocated at  $\text{p}$  on the heap (a so-called *thunk*) is finally evaluated to its WHNF at address  $\text{q}$ , the thunk is overwritten with an indirection to  $\text{q}$ , so in the resulting heap, we find  $\text{q} \mapsto \text{q}$  at address  $\text{p}$ .

In addition, we need a representation of free variables and other subexpressions that are unreducible because they contain free variables at redex positions. Free variables are inserted to evaluate expressions under  $\lambda$ -abstractions and case discriminations. When such a free variable is used in a function position or as a scrutinee of a case discrimination, it collects the context of this occurrence for later analysis. Thus, this special kind of heap value is called *accumulator*, denoted by the meta variable  $\text{k}$ .

The structure of accumulators is given in figure 3.3. An accumulator  $\text{k}$  is either a free variable, or a suspended application of another accumulator to a single argument pointer, or a suspended case discrimination with another accumulator as scrutinee. The similarity to unreducible heads from figure 3.2 is not by accident: accumulators are run time representations of not yet fully normalized unreducible heads.

To maintain sharing, we only reference accumulators contained within other accumulators by pointers. Thereby we ensure the invariant that the heap values referenced in function and scrutinee position are only accumulators: when considering all rules creating new accumulators either during weak evaluation or dur-

ing read back, we can easily see that if the function or scrutinee pointer of all accumulators reference only other accumulators in the initial heap, this is the case for all intermediate heaps and the final heap, too.

To make the difference between accumulators and usual expressions visible, we enclose accumulators in angle brackets.

We denote heaps with the meta variables  $\Gamma$ ,  $\Delta$  and  $\Theta$ , and use the following notations for expressing heap allocations.

- $\Gamma[p \mapsto hv]$  means that  $p$  is bound to the heap value  $hv$  in  $\Gamma$ . So no new allocation takes place here, this is just a proposition about a particular binding in  $\Gamma$ .
- $\Gamma \cup [p \mapsto hv]$  means that a new binding is added to  $\Gamma$ , thus  $p$  is not bound to any heap value in  $\Gamma$  itself, but is bound to  $hv$  in the resulting new heap (i. e. we use disjoint unions). Here, a new allocation is performed.
- $\Gamma[p \nrightarrow]$  means that  $p$  is unbound in  $\Gamma$ .
- The empty heap is denoted as  $\emptyset$ .

### 3.4 NORMALIZING FUN EXPRESSIONS

The normalization of FUN expressions is done with two interacting systems. An evaluator reduces an expression to WHNF, using a modified version of the STG machine. The resulting WHNF is *read back*, normalizing all redices that may be contained in such a WHNF.

Thus, the overall system is defined using three relations:

**WEAK EVALUATION** using semantics  $s_4$  is defined by the four place relation  $\cdot : \cdot \downarrow \cdot : \cdot$ , where  $\Gamma : e \downarrow \Delta : p$  means that expression  $e$  evaluates under heap  $\Gamma$  to the new heap  $\Delta$  that contains the WHNF of  $e$  at address  $p$ .

**READ BACK** is defined by the four place relation  $\cdot : \cdot \uparrow \cdot : \cdot$ , where  $\Gamma : p \uparrow \Delta : v$  means that reading back the WHNF located in  $\Gamma$  at address  $p$  yields the new heap  $\Delta$  and the strong normal form  $v$ .

**STRONG NORMALIZATION** is defined via weak evaluation and read back as the four place relation  $\cdot : \cdot \Downarrow \cdot : \cdot$ , where  $\Gamma : e \Downarrow \Delta : v$  means that the normal form of  $e$  under the heap  $\Gamma$  is  $v$ , and the normalization process yields the new heap  $\Delta$ . This is the main relation in our system.

The normalization relation is defined by the single inference rule given in figure 3.4 as the composition of weak evaluation and read back, which are defined in the following sections.

---


$$\text{Norm} \frac{\Gamma : e \downarrow \Delta : p \quad \Delta : p \uparrow \Theta : v}{\Gamma : e \Downarrow \Theta : v}$$


---

Figure 3.4: Normalization by Weak Evaluation and Read Back

While this inference rule and the rules in the following sections are strictly speaking part of inductively defined relations and of course have the usual semantics of inference rules, we will read them in a very operational manner. That is, while e. g. the two premises of Norm are independent of each other and do not imply any temporal order, we will say that  $e$  is first weakly reduced and then read back. This is justified by the determination properties of our rules – building trees of rules, i. e. finding proofs of the defined relations, is straightforward in a left to right order, but difficult in any other order. After all, our system defines a (big-step) operational semantics, and can easily be implemented by an interpreter, thus the operational reading gives a sound intuition.

One intended use of our system is to normalize closed expressions that contain no pointers. To accomplish this for a given expression  $e$ , we search for a heap  $\Delta$  and a normal form  $v$  such that  $\emptyset : e \Downarrow \Delta : v$  holds.

But it is of course possible to normalize expressions containing free variables, too. For example, to normalize  $\text{let } f = \lambda x. x \text{ in } f y$ , where  $y$  is free, we search for a heap  $\Delta$  and a normal form  $v$  such that  $[p_1 \mapsto \langle y \rangle] : \text{let } f = \lambda x. x \text{ in } f p_1 \Downarrow \Delta : v$  holds.

In both cases, the search for  $\Delta$  and  $v$  is made easy by the determination properties of our rules. In fact,  $\Delta$  and  $v$  are completely determined up to the choice of bound variable names and fresh pointers chosen to do allocations during evaluation.<sup>1</sup>

### 3.5 EVALUATION TO WHNF

The evaluation relation is defined inductively by the inference rules in figure 3.5. Rules Lam through Case<sup>(1)</sup> are the same as in the semantics S<sub>3</sub> defined by de la Encina and Peña [15], while we introduced rules Accu, App<sup>(3)</sup> and Case<sup>(2)</sup> to deal with

---

<sup>1</sup> A proof proceeds by induction on the size of inference trees built with the rules defined in figures 3.4, 3.5 and 3.6, followed by a case analysis on the rule used at the root of the tree. Two scenarios are possible: either the rule is the only rule matching to the shape of  $\Gamma$  and  $e$  of the conclusion at hand, then  $\Delta$  and  $v$  are determined as conjectured. Or there is an ambiguity between the rules App<sup>(1)</sup>, App<sup>(2)</sup> and App<sup>(3)</sup>, or between Case<sup>(1)</sup> and Case<sup>(2)</sup>. However, in each of these two ambiguous situations, the induction hypothesis for the first premise of the respective rule resolves the ambiguity.

accumulators. The latter are especially designed for cooperation with the read back phase defined in section 3.6.

Rule Lam merely states that a  $\lambda$ -abstraction is in WHNF and no evaluation is necessary. Rule Cons states the same for constructor values.

Rule Var describes the evaluation of pointers referencing unevaluated expressions in the heap. Observe, that  $p$  is bound neither in  $\Gamma$  nor in  $\Delta$  (see our explanation of the operation  $\cup$  above), thus the evaluation of  $e$  in the premise takes place with unbound  $p$ . This is a technique known as *blackholing*: when the value of  $p$  is needed before a WHNF is reached, the program cannot terminate. Thus by removing the binding of  $p$  we can detect some simple cases of non-termination. When the WHNF has been successfully computed, an indirection to this WHNF is added to  $\Delta$  at address  $p$ .

When such an indirection is met, rule Ind shows how the resulting WHNF is found by simply dereferencing the indirection. No further evaluation is necessary, since the indirection is guaranteed to point to a WHNF, since only rule Var allocates indirections, and it does so only after normalizing to WHNF.<sup>2</sup>

Unsaturated function application is described with rule App<sup>(1)</sup>, the function  $e$  is reduced to a  $\lambda$ -abstraction that has more abstracted variables than arguments are given. Thus a new  $\lambda$ -abstraction is allocated at address  $r$  with the supplied arguments substituted in the body.

Saturated function application is posed by rule App<sup>(2)</sup>: all abstracted variables are replaced by the given arguments, and the resulting expression  $e'[p/x]$  is applied to the remaining arguments.

Local bindings are evaluated as shown in rule Let. All  $\lambda$ -forms of the right hand sides are allocated in the heap, and made available by substituting the bound variables  $x$  with the fresh pointers  $p$ .

Rule Case describes the evaluation of case expressions. When the scrutinee  $e$  evaluates to a constructor expression, the corresponding branch of the alternatives is instantiated with the constructor arguments.

The next three rules define the behavior of accumulators in different contexts. Rule Accu defines an accumulator as a new kind of WHNF, no further evaluation is required in this situation.

More interesting is rule App<sup>(3)</sup>. When an expression  $e$  in function context is evaluated to an accumulator  $k$ , the accumulator grabs the first argument, and the accumulated application

<sup>2</sup> Here we assume that the initial heap we started the normalization with does not contain indirections to heap values not in WHNF. This is not a severe restriction, since the most useful initial heaps are either the empty heap or heaps containing only accumulators, as mentioned above.

---


$$\begin{array}{c}
\text{Lam} \frac{}{\Gamma[p \mapsto \lambda x. e] : p \downarrow \Gamma : p} \\
\\
\text{Cons} \frac{}{\Gamma[p \mapsto C p'] : p \downarrow \Gamma : p} \\
\\
\text{Var} \frac{\Gamma : e \downarrow \Delta : q}{\Gamma \cup [p \mapsto e] : p \downarrow \Delta \cup [p \mapsto q] : q} \\
\\
\text{Ind} \frac{}{\Gamma[p \mapsto q] : p \downarrow \Gamma : q} \\
\\
\text{App}^{(1)} \frac{\Gamma : e \downarrow \Delta[q \mapsto \lambda x y. e'] : q \quad r \text{ fresh}}{\Gamma : e p \downarrow \Delta \cup [r \mapsto \lambda y. e'[p/x]] : r} \\
\\
\text{App}^{(2)} \frac{\Gamma : e \downarrow \Delta[q \mapsto \lambda x. e'] : q \quad \Delta : e'[p/x] p' \downarrow \Theta : r}{\Gamma : e p p' \downarrow \Theta : r} \\
\\
\text{Let} \frac{\Gamma \cup [p \mapsto \text{lf}[p/x]] : e[p/x] \downarrow \Delta : q \quad p \text{ fresh}}{\Gamma : \text{let } x = \text{lf in } e \downarrow \Delta : q} \\
\\
\text{Case}^{(1)} \frac{\Gamma : e \downarrow \Delta[p \mapsto C_i p'] : p \quad \Delta : e'_i[p'/x_i] \downarrow \Theta : q}{\Gamma : \text{case } e \text{ of } Cx \rightarrow e' \downarrow \Theta : q} \\
\\
\text{Accu} \frac{}{\Gamma[p \mapsto k] : p \downarrow \Gamma : p} \\
\\
\text{App}^{(3)} \frac{\Gamma : e \downarrow \Delta[q \mapsto k] : q \quad \Delta \cup [r \mapsto \langle q p \rangle] : r p' \downarrow \Theta : s \quad r \text{ fresh}}{\Gamma : e p p' \downarrow \Theta : s} \\
\\
\text{Case}^{(2)} \frac{\Gamma : e \downarrow \Delta[p \mapsto k] : p \quad q \text{ fresh}}{\Gamma : \text{case } e \text{ of } \text{alt} \downarrow \Delta \cup [q \mapsto \langle \text{case } p \text{ of } \text{alt} \rangle] : q}
\end{array}$$


---

Figure 3.5: Semantics S<sub>4</sub>

---


$$\begin{array}{c}
 \text{Lam}_{\uparrow} \frac{\Gamma \cup [q \mapsto \langle y \rangle] : p \ q \ \uparrow \Delta : v \quad q, y \text{ fresh}}{\Gamma[p \mapsto \lambda x. e] : p \ \uparrow \Delta : \lambda y. v} \\
 \\
 \text{Cons}_{\uparrow} \frac{\bigwedge_{i=1}^{|q|} \Gamma_i : q_i \ \uparrow \Gamma_{i+1} : v_i}{\Gamma_1[p \mapsto C \ q] : p \ \uparrow \Gamma_{|q|+1} : C \ v} \\
 \\
 \text{Accu}_{\uparrow} \frac{\Gamma : k \ \uparrow_{\langle \rangle} \Delta : v}{\Gamma[p \mapsto k] : p \ \uparrow \Delta : v} \\
 \\
 \text{Var}_{\uparrow_{\langle \rangle}} \frac{}{\Gamma : \langle x \rangle \ \uparrow_{\langle \rangle} \Gamma : x} \\
 \\
 \text{App}_{\uparrow_{\langle \rangle}} \frac{\Gamma : k \ \uparrow_{\langle \rangle} \Delta : h \quad \Delta : q \ \uparrow \Theta : v}{\Gamma[p \mapsto k] : \langle p \ q \rangle \ \uparrow_{\langle \rangle} \Theta : h \ v} \\
 \\
 \text{Case}_{\uparrow_{\langle \rangle}} \frac{\begin{array}{l} \Gamma : k \ \uparrow_{\langle \rangle} \Delta_1 : h \\ \bigwedge_{i=1}^{|\mathbf{alt}|} \Delta_i \cup \Theta_i : \text{case } p' \text{ of } \mathbf{alt} \ \uparrow \Delta_{i+1} : v_i \\ \text{where } \mathbf{alt} = C \ x \rightarrow e, |q_i| = |y_i| = |x_i| \text{ and } p', q, y \text{ fresh} \\ \text{and } \Theta_i = [p' \mapsto C_i \ q_i] \cup [q_i \mapsto \langle y_i \rangle] \end{array}}{\Gamma[p \mapsto k] : \langle \text{case } p \text{ of } \mathbf{alt} \rangle \ \uparrow_{\langle \rangle} \Delta_{|\mathbf{alt}|+1} : \text{case } h \text{ of } C \ y \rightarrow v}
 \end{array}$$


---

Figure 3.6: Read Back Definition for S4

is allocated at a fresh address  $r$  and applied to the remaining arguments.

The corresponding situation for case discriminations is found in rule  $\text{Case}^{(2)}$ . When the scrutinee is evaluated to an accumulator, no branch of the alternatives can be selected, hence the discrimination is suspended in a newly allocated accumulator, saving all branches for further analysis by the read back phase.

### 3.6 READ BACK

The read back relation is inductively defined by the rules in figure 3.6. It makes use of a four place helper relation  $\cdot : \cdot \uparrow_{\langle \rangle} \cdot : \cdot$  for reading back accumulators:  $\Gamma : k \uparrow_{\langle \rangle} \Delta : h$  means that the accumulator  $k$  can be read back under the initial heap  $\Gamma$  to the unreducible head  $h$ , resulting in the final heap  $\Delta$ .

Rule  $\text{Lam}_{\uparrow}$  defines read back for  $\lambda$ -abstractions. To evaluate the body under the abstraction, we generate a pseudo-argument, namely a fresh variable placed as an accumulator  $\langle y \rangle$  onto the heap. The application of the abstraction to this accumulator is normalized to a value  $v$  that now possibly contains  $y$  as a free variable. To close the resulting expression, we place  $v$  below an abstraction over  $y$  in the conclusion. Note that we supply the pseudo-arguments one at a time, even when more variables are abstracted. Also, we could substitute  $q$  directly into the body. But in both cases, the implementation of the abstract machines defined in the following chapters, especially when we finally deal with compiled code, becomes much simpler when we use  $\text{Lam}_{\uparrow}$  as it is: we do not need to know how many arguments a compiled abstraction expects, nor do we have to substitute pointers into compiled code.

The read back for constructor values is defined by rule  $\text{Cons}_{\uparrow}$ . We simply normalize all constructor arguments one at a time. The notation  $\bigwedge_{i=1}^{|q|}$  is a shorthand to define the individual premises needed for each constructor argument.

Accumulators are delegated to the helper relation by rule  $\text{Accu}_{\uparrow}$ .

Free variables need no further normalization, as stated by rule  $\text{Var}_{\uparrow \emptyset}$ .

Suspended applications are read back by reading back the function part (pointing always to an accumulator, as stated above), and normalizing the argument.

Suspended case discriminations are handled by rule  $\text{Case}_{\uparrow \emptyset}$ . The scrutinee is read back to value  $h$ . Moreover, for each alternative, a new constructor value  $C_i q_i$  is allocated, where the constructor arguments are pointers to fresh free variable accumulators. Then, this constructor expressions are scrutinized, thereby passing the accumulators  $\langle y_i \rangle$  to the corresponding branch of the alternative that now can be evaluated.

### 3.7 EXAMPLE

To show the interplay of the relations defined in the latter sections, we next give an example of a strong normalization. We restrict ourselves to the simple example of the identity function, where only a small amount of weak evaluation is necessary, since our focus is on the relationship between strong normalization, weak evaluation and read back.

The identity function  $\text{id}$  can be given as  $\text{let } f = \lambda x. x \text{ in } f$  using the syntax of  $\text{FUN}$ . Its normalization to  $\lambda y. y$  is shown in figure 3.7.

Since inference trees typically grow very fast for examples of even small complexity, we use a vertical notation here. Each

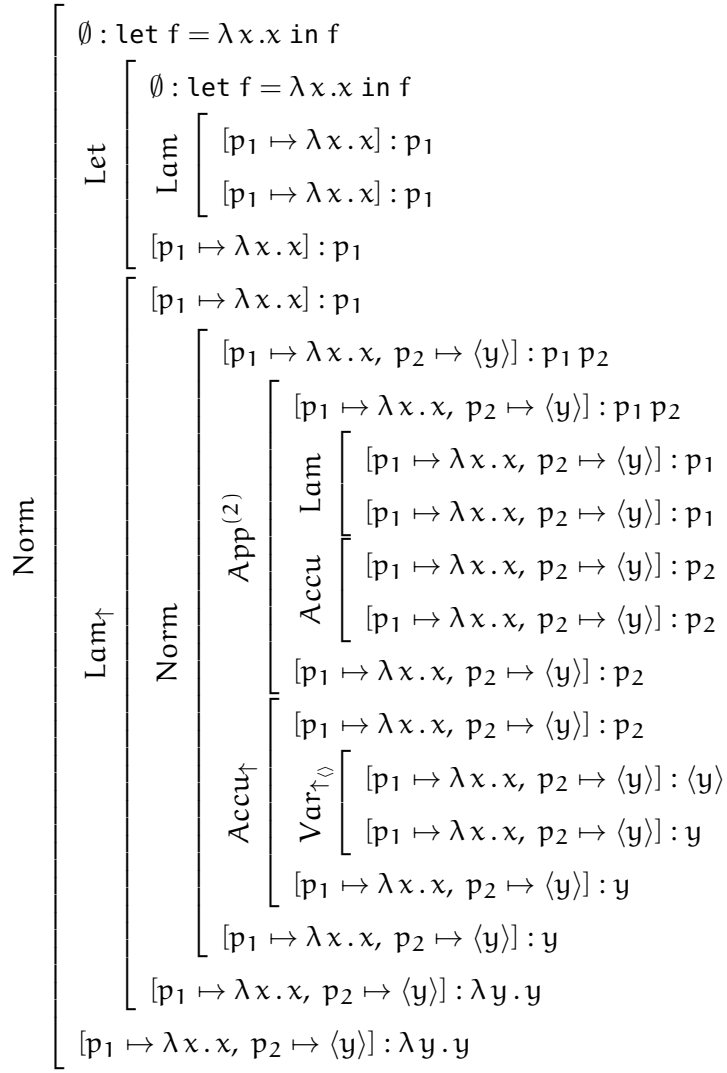


Figure 3.7: Normalization Example

application of an inference rule is depicted with a bracket that is annotated with the name of the applied rule to the left. The first two arguments of the conclusion of the rule are found on the upper right end of the bracket, while the last two arguments are on the lower right end. The trees for the premises of the rule are shown between these pairs of arguments.

To normalize `id`, it first is evaluated under the empty heap to `WHNF`, resulting in  $\lambda x. x$  allocated at address  $p_1$ . This heap value is read back by allocation of a fresh accumulator  $\langle y \rangle$  at address  $p_2$  and normalization of the application  $p_1 p_2$ . This results in  $y$ , not surprisingly, since  $p_1$  points to the identity function. Since we supplied  $y$  as a pseudo-argument, we have to abstract over  $y$  to obtain the normal form  $\lambda y. y$ .

## 3.8 WELL-FORMEDNESS

For the following correctness proofs, it will be convenient to group pointers into two non-overlapping sets  $P_1$  and  $P_2$ . We assume for all rules except  $\text{App}^{(1)}$  that require fresh pointers, that these pointers are members of  $P_1$ , and rule  $\text{App}^{(1)}$  requires fresh pointers to be members of  $P_2$ . This can be considered as dividing the heaps into two segments, where all allocations are performed in segment  $P_1$ , with the single exception of allocations done for partial applications. This segmentation makes the equivalence proofs for the semantics given in the next chapter easier, because it deals differently with partial applications.

We define *well-formed* heaps and judgments using this segmentation. A well-formed heap contains only pointers of  $P_1$  in its codomain, except for the destinations of indirections. A well-formed evaluation judgment requires the initial heap to be well-formed and the expression considered may only contain pointers from  $P_1$ . The function  $\mathcal{P}$  assigns to a heap value or an expression all pointers they contain.

**DEFINITION 3.1** (Well-formed heaps and judgments) *A heap  $\Gamma$  is well-formed, if it satisfies following condition.*

$$\forall p, hv. \Gamma[p \mapsto hv] \implies \mathcal{P}(hv) \subseteq P_1 \vee \exists q. hv = \text{?} q$$

*A judgment  $\Gamma : e \downarrow \Delta : p$  is well-formed, if  $\Gamma$  is well-formed and  $\mathcal{P}(e) \subseteq P_1$ . The same holds for judgments of the normalization and read back relations.*

Note that in the proof tree of a well-formed judgment, all occurring intermediate judgments are well-formed as well, i. e. well-formedness is maintained as an invariant in all our rules. Moreover, the final heap is well-formed, too.

Since the empty heap  $\emptyset$  is well-formed, and all interesting evaluations start with the empty heap or another well-formed heap, we will restrict ourselves for the remainder of this thesis to well-formed heaps and judgments.



We next define SSTG<sub>1</sub>, short for strongly normalizing STG machine. It is similar to the weakly normalizing semantics STG<sub>1</sub> defined by Encina and Peña [15], but enhanced with accumulators enabling read back and strong normalization.

Semantics SSTG<sub>1</sub> is a small-step operational semantics, thus it is a step towards a compiled system: s<sub>4</sub> gave the meaning of expressions by relating them to their weak head normal forms, and the inferences proving this relation had the shape of trees branching whenever a rule had more than one premise. Semantics SSTG<sub>1</sub> is given as a deterministic transition relation on configurations, thus a linear sequence of configurations models the linear character of machine computations and states. We have to adapt the strong normalization and read back relations to the new semantics, paying attention this time not to create new code during the normalization process.

To ensure the suitability of SSTG<sub>1</sub>, we have to show the equivalence to s<sub>4</sub>, asserting that strong normalization gives the same results with both semantics.

#### 4.1 STACKS AND HEAPS

As usual, to keep track of subexpressions needed for later steps, we introduce stacks *S* as sequences of stack values *sv*, as given by figure 4.1. A stack value may be a function argument, a sequence of case alternatives, or an update frame #*p*, marking heap locations for later thunk updates.

Besides of breaking the evaluation down in small steps, SSTG<sub>1</sub> changes the evaluation of partial applications slightly: instead of allocating a new abstraction with the supplied arguments substituted into the body, the arguments are left on the stack. During thunk update, an application is built instead of a substitution using a special heap binding denoted by the symbol  $\Rightarrow$ . This

---


$$\begin{aligned} sv &::= p \mid \mathbf{alt} \mid \#p \\ S &::= sv \end{aligned}$$


---

Figure 4.1: Stacks for SSTG<sub>1</sub>

kind of binding does not place update frames onto the stack, since a potential update would repeatedly overwrite it with the same partial application.

#### 4.2 WEAK EVALUATION

The weak evaluation is performed using the reduction rules shown in figure 4.2. A configuration of this reduction system is given by three components: a heap, a control expression that is to be evaluated and a stack.

When the evaluation of a thunk is started by rule  $\text{var}^{(1)}$ , an update frame is put onto the stack to save the address of the thunk that has to be overwritten by its WHNF after evaluation. Moreover, the evaluated expression is deallocated from the heap, again providing the detection of simple loops by blackholing. The WHNF might be an abstraction, constructor or accumulator, so these heap values check for update frames and perform the update by allocating an indirection at the address saved in the update frame, as stated by rules  $\text{var}^{(2)}$ ,  $\text{var}^{(3)}$  and  $\text{var}^{(4)}$ .

An indirection is evaluated by dereferencing the stored address with rule  $\text{ind}^{(1)}$ . Indirections for partial applications, that is expressions allocated with the spacial heap binding  $\Rightarrow$ , are evaluated by rule  $\text{ind}^{(2)}$  similar as  $\text{var}^{(1)}$ , but without pushing an update frame.

The arguments of applications are pushed onto the stack by rule  $\text{app}^{(1)}$ , and the evaluation continues with the function expression. When the evaluation of the function expression reaches a  $\lambda$ -abstraction, the arguments are popped from the stack and substituted into the function body by rule  $\text{app}^{(2)}$ . Otherwise, when the expression evaluates to an accumulator,  $\text{app}^{(3)}$  grabs the topmost argument from the stack, allocating a new accumulator representing this application. If more than one argument is present on the stack, further applications of the same rule will grab the other arguments in successive steps.

Local bindings are allocated on the heap by rule  $\text{let}$ , substituting the fresh addresses for the bound variables.

The evaluation of case expressions is started by rule  $\text{case}^{(1)}$ , pushing the alternatives onto the stack until the scrutinee is evaluated. When the scrutinee evaluates to a constructor expression, the appropriate branch is selected by rule  $\text{case}^{(2)}$ . And when the scrutinee evaluates to an accumulator, the branches are saved by rule  $\text{case}^{(3)}$  in a freshly allocated case discrimination accumulator for later normalization.

Heap	Control	Stack	Name
$\Gamma \cup [p \mapsto e]$ $\rightarrow \Gamma$	$p$ $e$	$S$ $\#p :: S$	$\text{var}^{(1)}$
$\Gamma[p \mapsto \lambda x. e]$ $\rightarrow \Gamma \cup [q \mapsto p \ p']$ where $ p'  <  x $	$p$ $p$	$p' \# \#q :: S$ $p' \# S$	$\text{var}^{(2)}$
$\Gamma[p \mapsto C \ p']$ $\rightarrow \Gamma \cup [q \mapsto q \ p]$	$p$ $p$	$\#q :: S$ $S$	$\text{var}^{(3)}$
$\Gamma[p \mapsto k]$ $\rightarrow \Gamma \cup [q \mapsto q \ p]$	$p$ $p$	$\#q :: S$ $S$	$\text{var}^{(4)}$
$\Gamma[p \mapsto q \ q]$ $\rightarrow \Gamma$	$p$ $q$	$S$ $S$	$\text{ind}^{(1)}$
$\Gamma[p \mapsto e]$ $\rightarrow \Gamma$	$p$ $e$	$S$ $S$	$\text{ind}^{(2)}$
$\Gamma$ $\rightarrow \Gamma$	$e \ p$ $e$	$S$ $p \# S$	$\text{app}^{(1)}$
$\Gamma[p \mapsto \lambda x. e]$ $\rightarrow \Gamma$	$p$ $e[p'/x]$	$p' \# S$ $S$	$\text{app}^{(2)}$
$\Gamma[p \mapsto k]$ $\rightarrow \Gamma \cup [q \mapsto \langle p \ p' \rangle]$ where $q$ fresh	$p$ $q$	$p' :: S$ $S$	$\text{app}^{(3)}$
$\Gamma$ $\rightarrow \Gamma \cup [p \mapsto \text{lf}[p/x]]$ where $p$ fresh	$\text{let } x = \text{lf in } e$ $e[p/x]$	$S$ $S$	$\text{let}$
$\Gamma$ $\rightarrow \Gamma$	$\text{case } e \text{ of } \text{alt}$ $e$	$S$ $\text{alt} :: S$	$\text{case}^{(1)}$
$\Gamma[p \mapsto C_i \ p']$ $\rightarrow \Gamma$	$p$ $e_i[p'/x_i]$	$C \ x \rightarrow e :: S$ $S$	$\text{case}^{(2)}$
$\Gamma[p \mapsto k]$ $\rightarrow \Gamma \cup [q \mapsto \langle \text{case } p \text{ of } \text{alt} \rangle]$ where $q$ fresh	$p$ $q$	$\text{alt} :: S$ $S$	$\text{case}^{(3)}$

Figure 4.2: Semantics sSTG1

---


$$\text{Norm} \frac{(\Gamma, e, S) \rightarrow^* (\Delta, p, p') \quad \mathcal{W}(\Delta, p, p') \quad \Delta : p : p' \uparrow \Theta : v}{\Gamma : e : S \Downarrow \Theta : v}$$


---

Figure 4.3: Normalization using SSTG1

### 4.3 NORMALIZATION AND READ BACK

We will reuse the symbols  $\Downarrow$ ,  $\uparrow$  and  $\uparrow_{\langle \rangle}$  for strong normalization and read back. It will be clear from the context in most cases whether the variants of  $s_4$  or of SSTG1 are meant. In the remaining cases, we will annotate the symbols with the name of the semantics.

Strong normalization  $\Downarrow$  and read back  $\uparrow$  are now five-place relations, taking an initial stack as an additional argument.

The strong normalization using SSTG1 is defined by rule Norm in figure 4.3. An expression  $e$  is evaluated in several steps to weak head normal form, as usual  $\rightarrow^*$  denotes the transitive and reflexive closure of  $\rightarrow$ . We define a predicate  $\mathcal{W}$  on SSTG1 configurations to describe feasible final states.

**DEFINITION 4.1** (Legal final states of SSTG1)  $\mathcal{W}(\Delta, p, p')$  holds iff

- $\Delta[p \mapsto \lambda x. e] \wedge |p'| < |x|$ , or
- $\Delta[p \mapsto C q] \wedge p' = \diamond$ , or
- $\Delta[p \mapsto k] \wedge p' = \diamond$ .

That is, either  $p$  points to a  $\lambda$ -abstraction with more formal parameters than actual parameters are present in  $p'$ , or  $p$  points to a constructor value or accumulator and the stack is empty.

While other final states are possible, e.g.  $(\Delta[p \mapsto C q], p, r)$ , these do not represent a successful computation and are not reflected by  $s_4$ , hence we rule them out by the above definition.

The read back definition for SSTG1 is given in figure 4.4. In contrast to the corresponding definitions for  $s_4$ , this time we avoid the generation of new code: while figure 3.6 created applications in rule  $\text{Lam}_{\uparrow}$ , the pseudo-argument is placed onto the stack in figure 4.4. Similarly, rule  $\text{Case}_{\uparrow_{\langle \rangle}}$  created case-expressions in figure 3.6, where we now place the case alternatives onto the stack, where they will be found by  $\text{case}^{(2)}$  in the first evaluation step.

---


$$\begin{array}{c}
\text{Lam}_{\uparrow} \frac{\Gamma \cup [q \mapsto \langle y \rangle] : p : p' : q \Downarrow \Delta : v \quad q, y \text{ fresh}}{\Gamma[p \mapsto \lambda x. e] : p : p' \uparrow \Delta : \lambda y. v} \\
\\
\text{Cons}_{\uparrow} \frac{\bigwedge_{i=1}^{|q|} \Gamma_i : q_i : \diamond \Downarrow \Gamma_{i+1} : v_i}{\Gamma_1[p \mapsto C q] : p : \diamond \uparrow \Gamma_{|q|+1} : C v} \\
\\
\text{Accu}_{\uparrow} \frac{\Gamma : k \uparrow_{\langle \rangle} \Delta : v}{\Gamma[p \mapsto k] : p : \diamond \uparrow \Delta : v}
\end{array}$$


---


$$\begin{array}{c}
\text{Var}_{\uparrow_{\langle \rangle}} \frac{}{\Gamma : \langle x \rangle \uparrow_{\langle \rangle} \Gamma : x} \\
\\
\text{App}_{\uparrow_{\langle \rangle}} \frac{\Gamma : k \uparrow_{\langle \rangle} \Delta : h \quad \Delta : q : \diamond \Downarrow \Theta : v}{\Gamma[p \mapsto k] : \langle p q \rangle \uparrow_{\langle \rangle} \Theta : h v} \\
\\
\text{Case}_{\uparrow_{\langle \rangle}} \frac{\begin{array}{l} \Gamma : k \uparrow_{\langle \rangle} \Delta_1 : h \\ \bigwedge_{i=1}^{|\mathbf{alt}|} \Delta_i \cup \Theta_i : p' : \mathbf{alt} : \diamond \Downarrow \Delta_{i+1} : v_i \\ \text{where } \mathbf{alt} = C x \rightarrow e, |q_i| = |y_i| = |x_i| \text{ and } p', q, y \text{ fresh} \\ \text{and } \Theta_i = [p' \mapsto C_i q_i] \cup [q_i \mapsto \langle y_i \rangle] \end{array}}{\Gamma[p \mapsto k] : \langle \text{case } p \text{ of } \mathbf{alt} \rangle \uparrow_{\langle \rangle} \Delta_{|\mathbf{alt}|+1} : \text{case } h \text{ of } C y \rightarrow v}
\end{array}$$


---

Figure 4.4: Read Back Definition for SSTG1

## 4.4 HEAP CORRESPONDENCE

As mentioned before, the semantics  $\text{sSTG1}$  handles partial applications differently than semantics  $\text{s4}$ : in rule  $\text{App}^{(1)}$   $\text{s4}$  allocates a new  $\lambda$ -abstraction for each partial application after substituting the given arguments, where the fresh pointer is a member of  $P_2$ .  $\text{sSTG1}$  does no allocation for partial applications but leaves the arguments on the stack. Moreover, when a thunk is overwritten with a partial application,  $\text{sSTG1}$  allocates an application expression in rule  $\text{var}^{(2)}$ , using the special binding  $\Rightarrow$  that suppresses update frames upon entering.

Thus, the heaps used in  $\text{s4}$  and  $\text{sSTG1}$  are not exactly the same but obey a correspondence relation  $\cdot \sim \cdot$  defined as follows.

**DEFINITION 4.2** (Corresponding heaps for  $\text{s4}$  and  $\text{sSTG1}$ ) *Suppose  $\Gamma$  is a heap used in  $\text{s4}$  and  $\Gamma'$  is a heap used in  $\text{sSTG1}$ . Then  $\Gamma \sim \Gamma'$  iff the following condition<sup>1</sup> is met.*

$$\begin{aligned}
 \forall p . \quad & \Gamma[p \mapsto hv] \wedge \Gamma'[p \mapsto hv] \wedge \neg \exists q. hv = \mapsto q \\
 \vee \quad & \Gamma[p \not\mapsto] \wedge \Gamma'[p \not\mapsto] \\
 \vee \quad & \Gamma[p \mapsto \lambda x. e] \wedge \Gamma'[p \not\mapsto] \wedge p \in P_2 \\
 \vee \quad & \Gamma[p \mapsto \mapsto q] \wedge \Gamma[q \mapsto \lambda y. e[r/x]] \\
 & \quad \wedge \Gamma'[p \Rightarrow q' r] \wedge \Gamma'[q' \mapsto \lambda x y. e] \\
 \vee \quad & \Gamma[p \mapsto \mapsto q] \wedge \Gamma[q \mapsto C p'] \\
 & \quad \wedge \Gamma'[p \mapsto \mapsto q] \wedge \Gamma'[q \mapsto C p'] \\
 \vee \quad & \Gamma[p \mapsto \mapsto q] \wedge \Gamma[q \mapsto k] \\
 & \quad \wedge \Gamma'[p \mapsto \mapsto q] \wedge \Gamma'[q \mapsto k]
 \end{aligned}$$

The individual constituents of this condition can be read as follows. The heaps  $\Gamma$  and  $\Gamma'$  are corresponding when one of these conditions is met:

- in both heaps  $p$  is bound to the same heap value that is no indirection,
- $p$  is unbound in both heaps,
- $p$  points to a  $\lambda$ -abstraction in  $\Gamma$  and is unbound in  $\Gamma'$  (this is the result of the rule  $\text{App}^{(1)}$ , and thus only allowed for pointers from segment  $P_2$ ),
- $p$  points to an indirection to a  $\lambda$ -abstraction in  $\Gamma$  and a matching partial application in  $\Gamma'$ ,
- in both heaps  $p$  is bound to an indirection to a constructor application, or

<sup>1</sup> The free variables in this condition lack existential quantifiers. They are omitted to reduce clutter. The scope of the quantifiers becomes clear from the description of the constituents.

- in both heaps  $p$  is bound to an indirection to an accumulator.

## 4.5 EQUIVALENCE OF S4 AND SSTG1

The equivalence of  $s_4$  and  $sstg1$  is shown in two steps. First, we show the *completeness* of  $sstg1$ , i.e. we show that every normal form obtained with  $s_4$  can be obtained with  $sstg1$ , too. This means, that every inference tree of  $s_4$  can be linearized to an  $sstg1$  reduction sequence. Next, we show the *soundness* of  $sstg1$ , i.e. we show that a normal form obtained with  $sstg1$  can also be obtained with  $s_4$ , and the linearization process can be reversed. Equivalence of  $s_4$  and  $sstg1$  then follows as a corollary.

Note that normalization using  $sstg1$  is as deterministic as it is using  $s_4$ : there are no configurations where more than one rule of the reduction relation matches.

## 4.5.1 Completeness

**LEMMA 4.3** (Completeness of weak evaluation) *Whenever we have  $\Gamma : e \downarrow \Delta : p$ , then for any  $\Gamma'$  with  $\Gamma \sim \Gamma'$  there is a heap  $\Delta'$  with  $\Delta \sim \Delta'$ , a pointer  $p'$  and a sequence of pointers  $\mathbf{q}$  such that for any stack  $S$  semantics  $sstg1$  allows the reduction  $\Gamma' : e : S \rightarrow^* \Delta' : p' : \mathbf{q} \# S$  with*

$$\begin{aligned} & \exists \mathbf{x}, \mathbf{y}, e' \quad . \quad \Delta[p \mapsto \lambda \mathbf{y} . e'[\mathbf{q}/\mathbf{x}]] \wedge \Delta'[p' \mapsto \lambda \mathbf{x} \mathbf{y} . e'] \\ \vee & \exists \mathbf{C}, \mathbf{p}'' \quad . \quad \Delta[p \mapsto \mathbf{C} \mathbf{p}''] \wedge p = p' \wedge \mathbf{q} = \diamond \\ \vee & \exists \mathbf{k} \quad . \quad \Delta[p \mapsto \mathbf{k}] \wedge p = p' \wedge \mathbf{q} = \diamond. \end{aligned}$$

Note that  $\Delta[p \mapsto \mathbf{C} \mathbf{p}'']$  implies  $\Delta'[p \mapsto \mathbf{C} \mathbf{p}'']$  in the conclusion, and  $\Delta[p \mapsto \mathbf{k}]$  implies  $\Delta'[p \mapsto \mathbf{k}]$ , because  $\Delta \sim \Delta'$ . That is, lemma 4.3 states that if  $e$  evaluates to a  $\lambda$ -abstraction using  $s_4$ , it evaluates to a  $\lambda$ -abstraction using  $sstg1$ , too, but possibly located at another address and with some arguments left on the stack instead of being substituted into the body. And when  $e$  evaluates to a constructor expression or accumulator using  $s_4$ , it evaluates to the same constructor expression or accumulator, located at the same address, with an unmodified stack, using  $sstg1$ .

**PROOF** The proof proceeds by induction on the size of the proof of  $\Gamma : e \downarrow \Delta : p$ , i.e. on the number of inference rules used. A case differentiation on the rule used at the root of the derivation shows that the lemma holds for all derivation trees. The detailed proof is given in appendix A.1.  $\square$

The next step is the completeness of the read back and strong normalization definitions of  $sstg1$ .

**LEMMA 4.4** (Completeness of read back and normalization) *The following three propositions hold:*

- Whenever  $\Gamma : p \uparrow^{s4} \Delta : v$ , then for any  $\Gamma'$  with  $\Gamma \sim \Gamma'$  there is a heap  $\Delta'$  with  $\Delta \sim \Delta'$  such that

$$\begin{aligned}
 & \exists \mathbf{y}, e \quad . \quad \Gamma[p \mapsto \lambda \mathbf{y} . e] \wedge \\
 & \quad \forall \mathbf{q}, \mathbf{q}', \mathbf{x}, e' . \Gamma'[q \mapsto \lambda \mathbf{x} \mathbf{y} . e'] \\
 & \quad \wedge e = e'[\mathbf{q}'/\mathbf{x}] \\
 & \quad \Rightarrow \Gamma' : q : \mathbf{q}' \uparrow^{\text{SSTG1}} \Delta' : v \\
 & \vee \exists \mathbf{C}, \mathbf{p}' \quad . \quad \Gamma[p \mapsto \mathbf{C} \mathbf{p}'] \wedge \Gamma' : p : \diamond \uparrow^{\text{SSTG1}} \Delta' : v \\
 & \vee \exists \mathbf{k} \quad . \quad \Gamma[p \mapsto \mathbf{k}] \wedge \Gamma' : p : \diamond \uparrow^{\text{SSTG1}} \Delta' : v.
 \end{aligned}$$

- Whenever  $\Gamma : k \uparrow_{\langle \rangle}^{s4} \Delta : h$ , then for any  $\Gamma'$  with  $\Gamma \sim \Gamma'$  there is a heap  $\Delta'$  with  $\Delta \sim \Delta'$  such that  $\Gamma' : k \uparrow_{\langle \rangle}^{\text{SSTG1}} \Delta' : h$ .
- Whenever  $\Gamma : e \updownarrow^{s4} \Delta : v$ , then for any  $\Gamma'$  with  $\Gamma \sim \Gamma'$  there is a heap  $\Delta'$  with  $\Delta \sim \Delta'$  such that  $\Gamma' : e : \diamond \updownarrow^{\text{SSTG1}} \Delta' : v$ .

PROOF The proof proceeds by simultaneous induction on the number of inference rules used for the proofs of the relations  $\uparrow^{s4}$ ,  $\updownarrow^{s4}$  and  $\up_{\langle \rangle}^{s4}$ , followed by a case differentiation on the rule at the root of the proof tree. The details of the proof can be found in appendix A.2.  $\square$

#### 4.5.2 Soundness

Reduction sequences of SSTG1 are linearizations of deduction trees of s4. However, not every (part of a) reduction sequence directly corresponds to a deduction tree. For example, if we take a deduction and its corresponding reduction sequence, and we arbitrarily cut out a segment in the middle of the reduction sequence, we might miss the boundaries of a single subtree of the deduction. Therefore, we define the concept of a balanced reduction sequence, and show how these sequences correspond to deduction trees.

DEFINITION 4.5 (Balanced reduction sequence) *A sequence of reduction steps  $(\Gamma, e, S) \rightarrow^* (\Gamma', e', S')$  is balanced iff the stacks of all participating configurations end with the initial stack  $S$ , i.e. they have the shape  $S_0 \# S$  with varying  $S_0$ .*

That is, a reduction sequence is balanced if no element of the initial stack gets popped during reduction. Since we introduced the stack to remember subexpressions during the linearization of one subtree of an s4 evaluation for later linearization of another subtree, this definition nicely allows to separate the different fragments of SSTG1 reduction sequences corresponding to different subtrees of s4 evaluation trees.

LEMMA 4.6 (Soundness of weak evaluation) *For any balanced reduction sequence of SSTG1  $(\Gamma, e, S) \rightarrow^* (\Delta, p, q \# S)$  with  $\mathcal{W}(\Delta, p, q)$ ,*

for all heaps  $\Gamma'$  with  $\Gamma' \sim \Gamma$  there is a heap  $\Delta'$  with  $\Delta' \sim \Delta$  and a pointer  $p'$  such that  $s_4$  allows the deduction of  $\Gamma' : e \Downarrow \Delta' : p'$  with

$$\begin{aligned} & \exists \mathbf{x}, \mathbf{y}, e' \quad . \quad \Delta[p \mapsto \lambda \mathbf{x} \mathbf{y} . e'] \wedge \Delta'[p' \mapsto \lambda \mathbf{y} . e'[q/\mathbf{x}]] \\ \vee & \exists \mathbf{C}, \mathbf{p}'' \quad . \quad \Delta[p \mapsto \mathbf{C} \mathbf{p}''] \wedge p = p' \\ \vee & \exists \mathbf{k} \quad . \quad \Delta[p \mapsto \mathbf{k}] \wedge p = p'. \end{aligned}$$

As in lemma 4.3,  $\Delta[p \mapsto \mathbf{C} \mathbf{p}'']$  implies  $\Delta'[p \mapsto \mathbf{C} \mathbf{p}'']$  in the conclusion, and  $\Delta[p \mapsto \mathbf{k}]$  implies  $\Delta'[p \mapsto \mathbf{k}]$ , because  $\Delta' \sim \Delta$ .

Thus, either both semantics give a  $\lambda$ -abstraction, but while SSTG1 leaves arguments on the stack, they are substituted into the body in the case of  $s_4$ , or  $e$  evaluates to the same constructor expression or accumulator using SSTG1 and  $s_4$ .

PROOF The proof proceeds by induction on the length of the reduction sequence of SSTG1, assuming the induction hypothesis for all shorter balanced sequences. The details can be found in appendix A.3.  $\square$

The last missing piece before closing this chapter with the equivalence of  $s_4$  and SSTG1 is the soundness of read back and strong normalization.

LEMMA 4.7 (Soundness of read back and normalization) *The following three propositions hold:*

- Whenever  $\Gamma : p : \mathbf{q}' \uparrow^{\text{SSTG1}} \Delta : v$ , then for any  $\Gamma'$  with  $\Gamma' \sim \Gamma$  there is a heap  $\Delta'$  with  $\Delta' \sim \Delta$  such that

$$\begin{aligned} & \exists \mathbf{x}, \mathbf{y}, e \quad . \quad \Gamma[p \mapsto \lambda \mathbf{x} \mathbf{y} . e] \wedge \\ & \quad \quad \quad \forall q. \Gamma'[q \mapsto \lambda \mathbf{y} . e[q'/\mathbf{x}]] \Rightarrow \Gamma' : q \uparrow^{s_4} \Delta' : v \\ \vee & \exists \mathbf{C}, \mathbf{p}' \quad . \quad \Gamma[p \mapsto \mathbf{C} \mathbf{p}'] \wedge \Gamma' : p \uparrow^{s_4} \Delta' : v \\ \vee & \exists \mathbf{k} \quad . \quad \Gamma[p \mapsto \mathbf{k}] \wedge \Gamma' : p \uparrow^{s_4} \Delta' : v. \end{aligned}$$

- Whenever  $\Gamma : k \uparrow_{\diamond}^{\text{SSTG1}} \Delta : h$ , then for any  $\Gamma'$  with  $\Gamma' \sim \Gamma$  there is a heap  $\Delta'$  with  $\Delta' \sim \Delta$  such that  $\Gamma' : k \uparrow_{\diamond}^{s_4} \Delta' : h$ .
- Whenever  $\Gamma : e : \diamond \downarrow^{\text{SSTG1}} \Delta : v$ , then for any  $\Gamma'$  with  $\Gamma' \sim \Gamma$  there is a heap  $\Delta'$  with  $\Delta' \sim \Delta$  such that  $\Gamma' : e \downarrow^{s_4} \Delta' : v$ .

PROOF We prove all three proposition by simultaneous induction on the number of inference rules used. The details are given in appendix A.4.  $\square$

#### 4.5.3 Correctness

After having shown that normalization by SSTG1 is sound and complete with respect to  $s_4$ , we can sum up our results.

**COROLLARY 4.8** (Correctness of normalization) *For any expression  $e$  and strong normal form  $v$ , the following equivalence holds:*

$$\exists \Delta. \emptyset : e \uparrow^{s4} \Delta : v \quad \Leftrightarrow \quad \exists \Delta'. \emptyset : e : \Diamond \uparrow^{\text{SSTG1}} \Delta' : v$$

**PROOF** Simple consequence of lemmas 4.4 and 4.7. Note that  $\emptyset \sim \emptyset$  holds trivially.  $\square$

In the semantics  $s_4$  and  $sstg_1$  substitutions were used, e. g. when  $\lambda$ -abstractions were applied to their arguments in rules  $App^{(2)}$  and  $app^{(2)}$ , respectively. However, such substitutions are an obstacle for compiling expressions to machine code, since substituting some values for variables changes the expression at hand, while changing the generated machine code for this expression at run time is something that usually is avoided in compiler construction.<sup>1</sup>

The standard approach, that we are following in this chapter when we define the semantics  $sstg_2$ , solves this problem by the introduction of *closures*. A closure is a tuple consisting of an expression that may contain free variables and an *environment*, i. e. a finite map from variables to their values. Instead of performing a substitution, for example  $e[p/x]$ , we now modify the environment of  $e$ , for example  $E$ , by adding a value  $p$  for variable  $x$  as in  $E \cup [x \mapsto p]$ . Thus,  $sstg_2$  brings us a further step closer to a system using compiled code.

Semantics  $sstg_2$  resembles Encina's and Peña's  $stg_2$ , again with addition of accumulators [15] to represent free variables and other weak head normal forms during evaluation.

## 5.1 MODIFICATIONS OF THE SYNTAX

In  $FUN$  we allowed pointers to appear in expressions, easing the presentation of  $s_4$  by using substitutions without giving to much implementation details at this abstraction level. But now we approach an implementation running on an abstract machine, so we want to replace substitutions with environment manipulations. Therefore, expressions do not need to contain pointers any more. Additionally, we do not want to keep environments unnecessarily large by mapping unused variables to otherwise potentially unreachable pointers, since this would prevent proper garbage collection.

Thus we introduce the language  $FUN_2$  that is a variant of  $FUN$ . The definition is given in figure 5.1. Besides restricting all occurrences of references in  $FUN$  to variables,  $FUN_2$  introduces so

<sup>1</sup> Of course, there are exceptions: Prolog implementations allow to add and remove rules during run time, easing for example some applications in artificial intelligence, and Erlang implements run time code updates to satisfy very high uptime requirements. But the effort of run time code generation has to be justified by some serious advantages of the resulting system, and we certainly do not want to depend on it just to implement function applications.

---

$e$	$::=$	$x$	
		$  \quad e \ x$	
		$  \quad \text{let } x = \text{lf} _t \text{ in } e$	
		$  \quad \text{case } e \text{ of } \text{alt} _t$	
$\text{lf}$	$::=$	$e$	
		$  \quad C \ x$	
		$  \quad \lambda x . e$	where $ x  > 0$
$\text{alt}$	$::=$	$C \ x \rightarrow e$	
$\text{hv}$	$::=$	$(\text{lf}, E)$	
		$  \quad \varphi \mapsto p$	
		$  \quad k$	
$k$	$::=$	$\langle x \rangle$	
		$  \quad \langle p \ q \rangle$	
		$  \quad \langle \text{case } p \text{ of } (\text{alt}, E) \rangle$	

---

Figure 5.1: Syntax of FUN2, its heap values and its accumulators

called *trimmers*  $t$ . These variable sequences are annotations of `let` and `case` expressions to indicate, which variables are needed for the evaluation of the bound  $\lambda$ -form or the selected case alternative, respectively. Thus, we make use of trimmers whenever some subexpression is delayed: either when a  $\lambda$ -form is allocated on the heap for later evaluation, or when case alternatives are delayed until the evaluation of the scrutinee is finished. Since we do not want to keep unnecessary large environments during these delays, we strip all unneeded variables from them.

Note that each binding of a  $\lambda$ -form has its own trimmer, while all alternatives of a case analysis share the same trimmer. This is because we do not know which alternative is selected when the evaluation of the scrutinee starts, so we have to keep all variables that occur in any alternative, thus needing only a single trimmer for all alternatives.

In the following, we assume that the trimmers contain at least all free variables of the  $\lambda$ -forms or case alternatives they annotate.

To associate  $\lambda$ -forms allocated on the heap with the values of their free variables, heaps for SSTG2 use tuples of a  $\lambda$ -form of FUN2 and a suitable environment where SSTG1 uses simply  $\lambda$ -forms of FUN. For details on environments see the next section.

Accumulators for blocked case analyses have to contain an environment for the alternatives as well.

The normal forms of FUN2 are the same as for FUN, so figure 3.2 still applies.

## 5.2 ENVIRONMENTS

We denote environments with  $E$ ,  $E'$  and so on. Moreover, we use the same notations for environment bindings as for heap bindings, thus  $E[x \mapsto p]$  means that  $E$  maps  $x$  to  $p$ , while  $E \cup [x \mapsto p]$  means that a new binding is added to  $E$ .

To substitute all variables in an expression  $e$  by their value given in an environment  $E$ , we write  $e[E]$ . We consider this a mapping from FUN2 to FUN, throwing away all trimmer annotations.

To restrict an environment  $E$  to the variables of a trimmer  $t$  we reuse the trimmer notation from the FUN2 syntax, writing  $E|_t$ .

## 5.3 NORMALIZATION

The rules for weak evaluation using SSTG2 are given in figure 5.2. A configuration consists of 4 components: a heap, a control expression, a current environment binding the free variables of the control expressions, and a stack.

The rules of SSTG2 correspond one to one to the rules of SSTG1. Every explicit substitution is replaced by a manipulation of an environment. Moreover, instead of pointers we find variables in expressions that we have to look up in the corresponding environment.

In rule  $\text{var}^{(1)}$  of SSTG1 the control expression is a pointer, while we have a variable with a corresponding binding in the current environment in SSTG2. Moreover, when we switch to the expression  $e$  contained in the thunk, we take the environment  $E'$  of the thunk as our new current environment.

For the allocation of partial applications during updates in rule  $\text{var}^{(2)}$ , we have to create a new environment  $E''$ , since we cannot place the pointers directly into the partial application expression.

The changes to rules  $\text{var}^{(3)}$  through  $\text{app}^{(1)}$  have similar modifications as rules  $\text{var}^{(1)}$  and  $\text{var}^{(2)}$ .

In rule  $\text{app}^{(2)}$  the body of a  $\lambda$ -abstraction is entered with the arguments added to the environment of the body. In SSTG1, the arguments were directly substituted into the body.

Rule  $\text{app}^{(3)}$  makes a lookup of the variable in the control expression in the current environment.

When a local definition is evaluated with rule  $\text{let}$ , we build a new environment  $E'$  by adding the heap locations of the bindings to the current environment  $E$ . While  $E'$  is the new current environment, it is stripped to each  $\lambda$ -form's trimmer in the allocated closures. In SSTG1 substitutions were performed instead.

	Heap	Control	Environment	Stack	Name
$\rightarrow$	$\Gamma \cup [p \mapsto (e, E')]$ $\Gamma$	$x$ $e$	$E[x \mapsto p]$ $E'$	$S$ $\#p :: S$	$\text{var}^{(1)}$
$\rightarrow$	$\Gamma[p \mapsto (\lambda \mathbf{y} . e, E')]$ $\Gamma \cup [q \mapsto (x x', E'')]$ where $ p'  <  y $ and $E'' := [x \mapsto p] \cup [x' \mapsto p']$	$x$ $x$	$E[x \mapsto p]$ $E$	$p' \# \#q :: S$ $p' \# S$	$\text{var}^{(2)}$
$\rightarrow$	$\Gamma[p \mapsto (C \mathbf{y}, E')]$ $\Gamma \cup [q \mapsto \text{q} \mapsto p]$	$x$ $x$	$E[x \mapsto p]$ $E$	$\#q :: S$ $S$	$\text{var}^{(3)}$
$\rightarrow$	$\Gamma[p \mapsto k]$ $\Gamma \cup [q \mapsto \text{q} \mapsto p]$	$x$ $x$	$E[x \mapsto p]$ $E$	$\#q :: S$ $S$	$\text{var}^{(4)}$
$\rightarrow$	$\Gamma[p \mapsto \text{q} \mapsto q]$ $\Gamma$ where $x'$ fresh	$x$ $x'$	$E[x \mapsto p]$ $[x' \mapsto q]$	$S$ $S$	$\text{ind}^{(1)}$
$\rightarrow$	$\Gamma[p \mapsto (e, E')]$ $\Gamma$	$x$ $e$	$E[x \mapsto p]$ $E'$	$S$ $S$	$\text{ind}^{(2)}$
$\rightarrow$	$\Gamma$ $\Gamma$	$e x$ $e$	$E[x \mapsto p]$ $E$	$S$ $p \# S$	$\text{app}^{(1)}$
$\rightarrow$	$\Gamma[p \mapsto (\lambda \mathbf{y} . e, E')]$ $\Gamma$	$x$ $e$	$E[x \mapsto p]$ $E' \cup [y \mapsto p']$	$p' \# S$ $S$	$\text{app}^{(2)}$
$\rightarrow$	$\Gamma[p \mapsto k]$ $\Gamma \cup [q \mapsto \langle p p' \rangle]$ where $x', q$ fresh	$x$ $x'$	$E[x \mapsto p]$ $[x' \mapsto q]$	$p' :: S$ $S$	$\text{app}^{(3)}$
$\rightarrow$	$\Gamma$ $\Gamma \cup [p \mapsto (\text{lf}, E' _t)]$ where $E' := E \cup [x \mapsto p]$ and $p$ fresh	$\text{let } x = \text{lf} _t \text{ in } e$ $e$	$E$ $E'$	$S$ $S$	$\text{let}$
$\rightarrow$	$\Gamma$ $\Gamma$	$\text{case } e \text{ of } \text{alt} _t$ $e$	$E$ $E$	$S$ $(\text{alt}, E _t) :: S$	$\text{case}^{(1)}$
$\rightarrow$	$\Gamma[p \mapsto (C_i \mathbf{y}, E'')]$ $\Gamma$ where $E''[y \mapsto p']$ and $\text{alt} := C x \rightarrow e$	$x$ $e_i$	$E[x \mapsto p]$ $E' \cup [x_i \mapsto p']$	$(\text{alt}, E') :: S$ $S$	$\text{case}^{(2)}$
$\rightarrow$	$\Gamma[p \mapsto k]$ $\Gamma \cup [q \mapsto k']$ where $k' := \langle \text{case } p \text{ of } (\text{alt}, E') \rangle$ and $x', q$ fresh	$x$ $x'$	$E[x \mapsto p]$ $[x' \mapsto q]$	$(\text{alt}, E') :: S$ $S$	$\text{case}^{(3)}$

Figure 5.2: Semantics SSTG2

Rule  $\text{case}^{(1)}$  stores the alternatives on the stack together with the current environment. At this, the environment is trimmed since not all variables bound in  $E$  might be needed.

While rule  $\text{case}^{(2)}$  in SSTG1 substituted the arguments of a constructor into the selected case alternative, SSTG2 adds them to the environment saved on the stack, making it to the new current environment.

Rule  $\text{case}^{(3)}$  takes the saved alternatives with their environment and stores them in a new accumulator.

The changes to the strong normalization and read back relations (figure 5.3 and 5.4, respectively) are simple, too. Strong normalization now is a five place relation, taking an environment for the expression to be normalized into account. Correspondingly, the other rules have to create environments for fresh variables where pointers were normalized before. The definition of  $\mathcal{W}$  for legal final states remains unchanged from SSTG1.

#### 5.4 EQUIVALENCE OF SSTG1 AND SSTG2

Since the modifications in the step from SSTG1 to SSTG2 are simple, it is easy to establish an equivalence result. We start with the definition of corresponding heaps. The symbol  $\sim$  is re-used for this correspondence relation, since no confusion is possible when considering the context.

**DEFINITION 5.1** (Correspondence between SSTG1 and SSTG2) *Two accumulators  $k$  and  $k'$  are corresponding, written  $k \sim k'$ , iff either  $k = k'$  or  $k = \langle \text{case } q \text{ of } \mathbf{alt} \rangle$  and  $k' = \langle \text{case } q \text{ of } (\mathbf{alt}', E) \rangle$  with  $\mathbf{alt} = \mathbf{alt}'[E]$ .*

*Two heaps  $\Gamma$  of SSTG1 and  $\Gamma'$  of SSTG2 are corresponding, written  $\Gamma \sim \Gamma'$ , iff for all pointers  $p$*

- $\Gamma[p \mapsto \text{lf}]$  and  $\Gamma'[p \mapsto (\text{lf}', E)]$  with  $\text{lf} = \text{lf}'[E]$ , or
- $\Gamma[p \mapsto \text{q} \rightarrow q]$  and  $\Gamma'[p \mapsto \text{q} \rightarrow q]$ , or
- $\Gamma[p \mapsto e[E]]$  and  $\Gamma'[p \mapsto (e, E)]$ , or
- $\Gamma[p \mapsto k]$  and  $\Gamma'[p \mapsto k']$  with  $k \sim k'$ .

*Two stacks  $S$  of SSTG1 and  $S'$  of SSTG2 are corresponding, written  $S \sim S'$ , iff the elements at the same positions are either equal or  $S$  contains  $\mathbf{alt}$  where  $S'$  contains  $(\mathbf{alt}', E)$  with  $\mathbf{alt} = \mathbf{alt}'[E]$ .*

*Two configurations  $(\Gamma, e, S)$  of SSTG1 and  $(\Gamma', e', E, S)$  of SSTG2 are corresponding, written  $(\Gamma, e, S) \sim (\Gamma', e', E, S')$ , iff  $\Gamma \sim \Gamma'$ ,  $e = e'[E]$  and  $S \sim S'$ .*

---


$$\text{Norm} \frac{(\Gamma, e, E, S) \rightarrow^* (\Delta, x, E'[x \mapsto p], p') \quad \mathcal{W}(\Delta, p, p') \quad \Delta : p : p' \uparrow \Theta : v}{\Gamma : e : E : S \updownarrow \Theta : v}$$


---

Figure 5.3: Normalization using SSTG2

---


$$\text{Lam}_{\uparrow} \frac{\Gamma \cup [q \mapsto \langle y \rangle] : z : [z \mapsto p] : p' \cdot q \updownarrow \Delta : v \quad q, y, z \text{ fresh}}{\Gamma[p \mapsto \lambda x. e] : p : p' \uparrow \Delta : \lambda y. v}$$

$$\text{Cons}_{\uparrow} \frac{\bigwedge_{i=1}^{|q|} \Gamma_i : x_i : [x_i \mapsto q_i] : \diamond \updownarrow \Gamma_{i+1} : v_i}{\Gamma_1[p \mapsto (C x, [x \mapsto q])] : p : \diamond \uparrow \Gamma_{|q|+1} : C v}$$

$$\text{Accu}_{\uparrow} \frac{\Gamma : k \uparrow_{\langle \rangle} \Delta : v}{\Gamma[p \mapsto k] : p : \diamond \uparrow \Delta : v}$$


---

$$\text{Var}_{\uparrow_{\langle \rangle}} \frac{}{\Gamma : \langle x \rangle \uparrow_{\langle \rangle} \Gamma : x}$$

$$\text{App}_{\uparrow_{\langle \rangle}} \frac{\Gamma : k \uparrow_{\langle \rangle} \Delta : h \quad \Delta : x : [x \mapsto q] : \diamond \updownarrow \Theta : v \quad x \text{ fresh}}{\Gamma[p \mapsto k] : \langle p q \rangle \uparrow_{\langle \rangle} \Theta : h v}$$

$$\begin{aligned} & \Gamma : k \uparrow_{\langle \rangle} \Delta_1 : h \\ & \bigwedge_{i=1}^{|\mathbf{alt}|} \Delta_i \cup \Theta_i : x' : [x' \mapsto p'] : (\mathbf{alt}, E) \cdot \diamond \updownarrow \Delta_{i+1} : v_i \\ & \text{where } \mathbf{alt} = C x \rightarrow e, \\ & |q_i| = |y_i| = |x_i| = |z_i| \text{ with } p', q, x', y, z \text{ fresh,} \\ & \text{and } \Theta_i = [p' \mapsto (C_i z_i, [z_i \mapsto q_i])] \cup [q_i \mapsto \langle y_i \rangle] \\ \text{Case}_{\uparrow_{\langle \rangle}} & \frac{}{\Gamma[p \mapsto k] : \langle \text{case } p \text{ of } (\mathbf{alt}, E) \rangle \uparrow_{\langle \rangle} \Delta_{|\mathbf{alt}|+1} : \text{case } h \text{ of } C y \rightarrow v} \end{aligned}$$


---

Figure 5.4: Read Back Definition for SSTG2

LEMMA 5.2 (Equivalence of weak evaluation) *Assume two configurations  $C$  of SSTG1 and  $C'$  of SSTG2 with  $C \sim C'$ . Then we have*

$$\forall D. C \xrightarrow[\text{SSTG1}]{*} D \Rightarrow \exists D'. C' \xrightarrow[\text{SSTG2}]{*} D' \text{ with } D \sim D'$$

and

$$\forall D'. C' \xrightarrow[\text{SSTG2}]{*} D' \Rightarrow \exists D. C \xrightarrow[\text{SSTG1}]{*} D \text{ with } D \sim D'.$$

PROOF (Sketch) First, prove that this proposition holds for single step reductions, by case analysis of the possible rules. Since the corresponding rules of SSTG1 and SSTG2 are very similar, this is straightforward.

Next, show that this result carries over to the transitive reflexive closure of the reduction relations by induction on the length of the reduction sequence.  $\square$

LEMMA 5.3 (Equivalence of read back and strong normalization) *The following three propositions hold:*

- Whenever  $\Gamma \sim \Gamma'$  then

$$\exists \Delta. \Gamma : p : p' \uparrow^{\text{SSTG1}} \Delta : v \Leftrightarrow \exists \Delta'. \Gamma' : p : p' \uparrow^{\text{SSTG2}} \Delta' : v$$

where  $\Delta \sim \Delta'$ .

- Whenever  $\Gamma \sim \Gamma'$  and  $k \sim k'$  then

$$\exists \Delta. \Gamma : k \uparrow^{\text{SSTG1}}_{\langle \rangle} \Delta : h \Leftrightarrow \exists \Delta'. \Gamma' : k' \uparrow^{\text{SSTG2}}_{\langle \rangle} \Delta' : h$$

where  $\Delta \sim \Delta'$ .

- Whenever  $\Gamma \sim \Gamma'$ ,  $e = e'[E]$  and  $S \sim S'$  then

$$\exists \Delta. \Gamma : e : S \downarrow^{\text{SSTG1}} \Delta : v \Leftrightarrow \exists \Delta'. \Gamma' : e' : E : S' \downarrow^{\text{SSTG2}} \Delta' : v$$

where  $\Delta \sim \Delta'$ .

PROOF (Sketch) By simultaneous induction on the number of inference rules used, under application of lemma 5.2 in the proposition on the strong normalization relations.  $\square$

COROLLARY 5.4 (Correctness of strong normalization) *For any expression  $e$  of FUN2 and any normal for  $v$ , it holds that*

$$\exists \Delta. \emptyset : e[\emptyset] : \diamond \downarrow^{\text{SSTG1}} \Delta : v \Leftrightarrow \exists \Delta'. \emptyset : e : \emptyset : \diamond \downarrow^{\text{SSTG2}} \Delta' : v.$$

PROOF Simple consequence of lemma 5.3.  $\square$



In this chapter we reach our final goal of this part of the thesis. We define an abstract machine that executes a sequence of simple instructions, we show how FUN expressions can be translated to appropriate instructions, and we give definitions of read back and strong normalization that use this abstract machine.

The instructions can be translated further to machine code for physical processors, enhancing the efficiency of interpreted pseudo instructions. We give some details on this translation in chapter 8.

Our abstract machine is called `ISSTG` machine, short for imperative strong normalizing STG machine, because it is close to the imperative character of physical processors. It resembles the weakly normalizing `ISTG` machine by Encina and Peña [15].

## 6.1 AN EXECUTION ENVIRONMENT FOR ISSTG

A configuration for `ISSTG` is a quadruple, consisting of

- an instruction sequence to be executed, replacing the control expression of `SSTG2`,
- a stack,
- a node pointer, referring to the currently evaluated closure, where parts of the current environment of `SSTG2` can be found, and
- a heap, which is a mapping from pointers to pairs of code pointers and some closure data `cd`. The closure data may be a sequences of pointers, replacing the environments stored in closures of `SSTG2`, or an accumulator.

The order in which corresponding components in `SSTG2` and `ISSTG` are given has been changed to reflect the change in the main component driving rule selection. In `SSTG2` rule selection was mostly determined by the kind of heap value referenced by the variable constituting the control expression, while in `ISSTG` the next instruction in the instruction sequence component is most influential.

Moreover, we make use of a code store `cs` and a branch info table `bi`. Both are created during compilation and not modified during normalization, thus we do not include them in the configurations but treat them as parameters for all definitions in this chapter.

---

$\text{ins} ::=$	ENTER		RETURNCON C
	ARGCHECK n		ALLOC n
	BUILDCLS n p $\alpha$		BUILDENV $\alpha$
	PUSHCONT p		UPDMARK
	SLIDE n m		PAP
	ACCU		
$\alpha ::=$	STACK n		
	NODE n		
$\text{is} ::=$	<b>ins</b>		

---

Figure 6.1: Instructions for ISSTG

- The code store  $\text{cs}$  maps pointers to code sequences or to branch tables, implemented as mappings from constructors to code sequences. Figure 6.1 shows the instructions for ISSTG, they are discussed in more detail below.
- The branch info table  $\text{bi}$  stores some information unnecessary for weak evaluation but needed for strong normalization. It is used by the implementation of accumulators for case discriminations as well as their read back. We give details below, too.

For ISSTG, we use a further restricted variant of FUN: function positions in applications and scrutinees in case expressions are restricted to be mere variables, not arbitrary expressions. This simplifies the stack layout to enable the read back phase of normalization with ISSTG, but would complicate matters for  $\text{s}_4$  where rule  $\text{App}^{(2)}$  relies on complex function positions.

These restrictions can easily be met by introducing additional local bindings for function expressions and scrutinees. Proving the correctness, i. e. the conservation of normal forms, of this transformation is trivial. The resulting syntax of the language FUN<sub>3</sub> is shown in figure 6.2, the syntax of normal forms remains unmodified.

In the same figure, we show the modified structure for the heap values  $\text{hv}$  of our compiled semantics. Each closure consists of a pointer to some code fragment, replacing the control expressions in the closures for  $\text{sstg}_2$ . Moreover, a closure contains a closure data field  $\text{cd}$ , which may be a sequence of pointers, replacing the environments of  $\text{sstg}_2$  closures, or an accumulator  $k$ . The special, non-updating heap binding  $\Rightarrow$  is not used for ISSTG. Indirections

---


$$\begin{aligned}
e &::= x \ y \\
&| \text{let } x = lf|_t \text{ in } e \\
&| \text{case } x \text{ of } \mathbf{alt}|_t \\
\\
hv &::= (p, cd) \\
\\
cd &::= p \mid k \\
\\
k &::= \langle x \rangle \\
&| \langle p \ q \rangle \\
&| \langle \text{case } p \text{ of } (q, r) \rangle
\end{aligned}$$


---

Figure 6.2: Expressions, heap values and accumulators of FUN3

$\curlywedge \rightarrow q$  are represented with an code pointer and a closure data field, as any other closure.

The structure of accumulators is almost identical to their definition for SSTG2. Only the variant for stuck case expressions now captures a code pointer to the continuation code and a sequence of pointers, replacing the list of alternatives and the continuation environment, respectively, of SSTG2.

## 6.2 COMPILATION AND WEAK EVALUATION

Figure 6.3 defines the translation of FUN3 expressions to machine code.

While in SSTG2 all variables were defined in a single environment, ISSTG environments are split into two parts. One part of the environment is stored in the closures. This part is accessed not by a current environment as in SSTG2, but via the node pointer referring to the current closure, where the variables' values can be found in the closure data field. The second part is located on the stack. As usual, values on the stack have only static life time, so this part is only used temporarily. We use the stack environment for function arguments and to make the heap locations of locally bound values available to the body of let expressions. In these cases the stack environment is cleared when another closure is entered. Moreover, we continue to use the stack to save environments for case alternatives. These environments stay on the stack during evaluation of the scrutinee, and are cleared only when the selected alternative finally enters another closure.

The translation functions use compile time environments to keep track where the variables' values can be found. We use a stack environment  $\rho$  that maps variables to stack indices, and

---

$\text{trE } (x \ y) \ \rho \ \eta$	$=$	[ BUILDENV $(\rho, \eta) \ x :: (\rho, \eta) \ y,$ SLIDE $(1 +  y ) \ (\text{snd } \rho),$ ENTER ]
$\text{trE } (\text{let } x = \text{lf} _t \text{ in } e) \ \rho \ \eta$	$=$	[ ALLOC $ t  \mid i \leftarrow  x ..1$ ] $\#$ [ BUILDCLS $i \ p_i \ a_i \mid i \leftarrow 1.. x $ ] $\#$ trE $e \ \rho' \ \eta$ where $\rho' = \rho + [x_i \mapsto i \mid i \leftarrow 1.. x ]$ $p = \text{trB } \text{lf} _t$ $a = (\rho', \eta) \ t$
$\text{trE } (\text{case } x \text{ of } \text{alt} _t) \ \rho \ \eta$	$=$	[ BUILDENV $(\rho, \eta) \ t,$ PUSHCONT $p,$ BUILDENV $(\rho +  t  + 1, \eta) \ x,$ SLIDE $(2 +  t ) \ (\text{snd } \rho),$ ENTER ] & $\text{bi}[p \mapsto ( t , \text{info } \text{alt})]$ where $\rho' = ([t_i \mapsto i \mid i \leftarrow 1.. t ],  t )$ $p = \text{trAs } \text{alt } \rho'$ $\text{info } (C \ x \rightarrow e) = (C,  x )$
$\text{trAs } \text{alt } \rho$	$=$	$p$ & $\text{cs}[p \mapsto [C \mapsto \text{trA } \text{alt } \rho]]$ where $\text{alt} = C \ x \rightarrow e$ and $p$ fresh
$\text{trA } (C \ x \rightarrow e) \ \rho$	$=$	$\text{trE } e \ \rho \ [x_i \mapsto i \mid i \leftarrow 1.. x ]$
$\text{trB } (C \ x \  _t)$	$=$	$p$ & $\text{cs}[p \mapsto [\text{RETURNCON } C]]$ where $p$ fresh
$\text{trB } (\lambda x. e \  _t)$	$=$	$p$ & $\text{cs}[p \mapsto [\text{ARGCHECK }  x ] \# \text{trE } e \ \rho \ \eta]$ where $\rho = ([x_i \mapsto i \mid i \leftarrow 1.. x ],  x )$ $\eta = [t_i \mapsto i \mid i \leftarrow 1.. t ]$ and $p$ fresh
$\text{trB } (e \  _t)$	$=$	$p$ & $\text{cs}[p \mapsto [\text{UPDMARK}] \# \text{trE } e \ \rho_\emptyset \ \eta]$ where $\rho_\emptyset = (\emptyset, 0)$ $\eta = [t_i \mapsto i \mid i \leftarrow 1.. t ]$ and $p$ fresh

---

Figure 6.3: Translation of FUN3 to machine code

a node environment  $\eta$  mapping variables to indices in pointer sequences in closure data fields of the current closure.

The node environment  $\eta$  is just a simple mapping from variable names to indices. Since the stack environment is cleared before entering other closures,  $\rho$  is a tuple of a corresponding mapping from variable names to indices on the stack, and the number of local variables that have to be cleared.

We write variable lookups in these environments as  $(\rho, \eta) x$ , resulting in either `STACK n` or `NODE n`, depending on the location of  $x$  on the stack or the heap. We require the domains of  $\rho$  and  $\eta$  to be disjoint. This lookup definition is complemented by the function `ptr`, used in the definition of the semantics of `ISSTG`, with `ptr (STACK i)` being the  $i$ -th stack element and `ptr (NODE i)` the  $i$ -th element of the closure environment. We use a Haskell-like syntax for list comprehensions:  $[e(i) \mid i \leftarrow n..m]$  binds  $i$  to all values from  $n$  to  $m$  in turn, the result is the list of all values  $e(i)$ .

The code store and the branch info table are modified by operations given after `&` as monadic side effects of the translation functions.

The definitions of the translation functions `trE` for expressions, `trAs` and `trA` for case alternatives and `trB` for locally bound  $\lambda$ -forms in figure 6.3 are best read together with the definition of the corresponding transition rules in figure 6.4 of the generated instructions.

### 6.2.1 *Compilation of Applications*

The translation of applications (first rule of `trE`) first builds a stack environment containing the function  $x$  and all arguments  $y$  using the instruction `BUILDENV` that simply pushes its arguments onto the stack. It then removes all local variables from the stack, using `SLIDE`. The first argument of this instruction gives the number of values that have to be kept on top of the stack, in this case this is one (for the pointer to the function) plus the number of arguments. The second argument is the number of stack values to be cleared, stored in the second component of the stack environment  $\rho$ . Finally, the code resulting from the translation enters the function via a tail call using `ENTER`. It takes the topmost stack element, still our function pointer, as the new pointer to the current closure and continues execution with the instruction sequence referred to by the code pointer of this closure.

### 6.2.2 *Compilation of Local Bindings*

Local bindings (second rule of `trE`) are compiled to a sequence of `ALLOC` instructions to reserve heap space. This does not modify the heap as it is modeled in `ISSTG`, but in an implementation

Code	Stack	Node	Heap
[ENTER]	$q :: S$	$p$	$\Gamma[q \mapsto (r, r')]$
$\rightarrow$ is where $cs[r \mapsto is]$	$S$	$q$	$\Gamma$
[RETURNCON C]	$q :: S$	$p$	$\Gamma$
$\rightarrow$ is where $cs[q \mapsto bt]$ and $bt[C \mapsto is]$	$S$	$p$	$\Gamma$
[RETURNCON C]	$\#q :: S$	$p$	$\Gamma \cup [q \mapsto (p_{bh}, r)]$
$\rightarrow$ [RETURNCON C]	$S$	$p$	$\Gamma \cup [q \mapsto (p_{ind}, p)]$
ARGCHECK $n :: is$	$q \# S$	$p$	$\Gamma$
$\rightarrow$ is where $n =  q $	$q \# S$	$p$	$\Gamma$
ARGCHECK $n :: is$	$q \# \#r :: S$	$p$	$\Gamma \cup [r \mapsto (p_{bh}, q')]$
$\rightarrow$ ARGCHECK $n :: is$ where $n >  q $	$q \# S$	$p$	$\Gamma \cup [r \mapsto (p_{pap}, p :: q)]$
ALLOC $n :: is$	$S$	$p$	$\Gamma$
$\rightarrow$ is where $q$ points to a new closure with space for $n$ variables and $\Gamma'$ is the resulting heap after allocation	$q :: S$	$p$	$\Gamma'$
BUILDCLS $i \ q \ a :: is$	$S$	$p$	$\Gamma$
$\rightarrow$ is	$S$	$p$	$\Gamma \cup [S_i \mapsto (q, ptr \ a)]$
BUILDENV $a :: is$	$S$	$p$	$\Gamma$
$\rightarrow$ is	$ptr \ a \# S$	$p$	$\Gamma$
PUSHCONT $q :: is$	$S$	$p$	$\Gamma$
$\rightarrow$ is	$q :: S$	$p$	$\Gamma$
UPDMARK $:: is$	$S$	$p$	$\Gamma \cup [p \mapsto (q, q')]$
$\rightarrow$ is	$\#p :: S$	$p$	$\Gamma \cup [p \mapsto (p_{bh}, q')]$
SLIDE $n \ m :: is$	$q \# q' \# S$	$p$	$\Gamma$
$\rightarrow$ is where $n =  q $ and $m =  q' $	$q \# S$	$p$	$\Gamma$
PAP $:: is$	$S$	$p$	$\Gamma[p \mapsto (q, q')]$
$\rightarrow$ is	$q' \# S$	$p$	$\Gamma$
[ACCU]	$q :: S$	$p$	$\Gamma[p \mapsto (p_{accu}, k)]$
$\rightarrow$ [ACCU] where $r$ fresh	$S$	$r$	$\Gamma \cup [r \mapsto (p_{accu}, \langle p \ q \rangle)]$
[ACCU]	$q :: q' \# S$	$p$	$\Gamma[p \mapsto (p_{accu}, k)]$
$\rightarrow$ [ACCU] where $bi[q \mapsto ( q' , info)]$ , $k' = \langle \text{case } p \text{ of } (q, q') \rangle$ and $r$ fresh,	$S$	$r$	$\Gamma \cup [r \mapsto (p_{accu}, k')]$
[ACCU]	$\#q :: S$	$p$	$\Gamma \cup [q \mapsto (p_{bh}, q')]$
$\rightarrow$ [ACCU]	$S$	$p$	$\Gamma \cup [q \mapsto (p_{ind}, p)]$

Figure 6.4: Semantics ISSTG

that produces machine code for physical processors, as it is our goal in chapter 8, some memory management is performed with this instruction. The reserved heap space is in turn filled by a sequence of BUILDCLS instructions, before the compiled body is executed. The bound  $\lambda$ -forms are compiled by trB.

- Constructors (first rule in the definition of trB) are compiled to a single instruction, RETURNCON, that expects a case continuation on the top of the stack (see the first rule of the RETURNCON instruction in figure 6.4). It dereferences the pointer found on the stack, finds a map from constructors to instruction sequences, and selects the branch corresponding to the instruction's argument.
- Compiled abstractions (second rule of trB) check whether enough arguments are present on the stack via ARGCHECK. When this check is positive (first rule for ARGCHECK in figure 6.4), the body of the abstraction is executed, otherwise (second rule for ARGCHECK in figure 6.4) an update frame is expected on the stack and the corresponding closure on the heap is overwritten by a partial application (*pap*, see below for an explanation of  $p_{pap}$ ).
- The code of other bound expressions (third rule of trB) starts with UPDMARK that pushes an update frame.

### 6.2.3 Compilation of Case Discriminations

For case expressions (third rule of trE), we first push the variables used in the branches and the pointer to the code of the branches, before entering the scrutinee. We look at the generated instructions one by one:

- All variables needed in the branches are saved on the stack first using BUILDENV.
- Then a pointer to the compiled alternatives is left on the stack using PUSHCONT as a continuation. This pointer is obtained as the result of trAs with an appropriately constructed stack environment  $\rho'$ , describing the locations of the values pushed by the last BUILDENV instruction.
- Next, a pointer to the scrutinee is pushed, using BUILDENV again, but the stack environment  $\rho$  is out of date now: we pushed  $|t|$  pointers constituting the continuation environment and one additional pointer to the alternatives' code. Thus we have to add  $|t| + 1$  to the values of  $\rho$ . The notation  $\rho + n$  used in trE denotes a stack environment where all indices are incremented by  $n$ , thus taking care of the new stack values.

- Afterwards, unneeded arguments are purged from the stack with a SLIDE instruction. We have to keep the environment, the code pointer for the alternatives and the scrutinee, summing up to  $|t| + 2$  stack values. The number of values to clear is again found as the second component of  $\rho$ .
- Finally, the scrutinee, still found topmost on the stack, is entered with ENTER.

The branch info table  $bi$  stores the number of variables saved on the stack for use by case branches, and records for which constructors branches exist and how many arguments these constructors have. These are used by the ACCU instruction to grab the right number of pointers from the stack if we reach a stuck case discrimination (see the second rule of ACCU in figure 6.4), and by the read back of such discriminations to select the cases that have to be normalized.

#### 6.2.4 Preallocated Code Sequences

During execution we expect several code sequences preallocated in the code store.

- For indirections, the second rule for RETURNCON and the third rule for ACCU, respectively, make use of  $p_{ind}$  with  $cs[p_{ind} \mapsto [BUILDENV (NODE\ 1), ENTER]]$ . This code sequence pushes the first (and only) pointer of the closure data field onto the stack and enters it immediately, thus mimicking the behavior of  $\triangleright \triangleright q$  in SSTG2.
- When allocating partial applications, the second ARGCHECK rule uses a pointer  $p_{pap}$  with  $cs[p_{pap} \mapsto [PAP, ENTER]]$ . Upon execution, this code sequence will push all pointers of the closure data field onto the stack and enter the topmost pointer, that points to an abstraction taking the remaining pointers as arguments. This resembles the evaluation of partial applications in SSTG2.
- For blackholing, that is for marking expressions under evaluation, UPDMARK uses a pointer  $p_{bh}$  pointing to an empty code sequence, i. e.  $cs[p_{bh} \mapsto \Diamond]$ , such that evaluation gets stuck when an expression is re-entered before reaching WHNF.
- And finally, for accumulator allocations, the first and second rule for ACCU use  $p_{accu}$  with  $cs[p_{accu} \mapsto [ACCU]]$ , i. e.  $p_{accu}$  points to the code that grabs either function arguments or case alternatives together with their environments from the stack, using the branch information table where necessary.

## 6.3 STRONG NORMALIZATION AND READ BACK

The normalization rule, given in figure 6.5, now normalizes a pointer to some heap closure and a stack. The heap closure is entered, taking the remaining stack values as function arguments or pointers to case branches. The current closure pointer  $r$  in the initial configuration in the first premise of Norm can be any pointer, since in the next transition it is replaced by  $q$  anyway. We might even start with  $q$  instead of an arbitrary  $r$ , but had to enter  $q$  nevertheless to load the right code sequence into our configuration.

A legal final configuration is defined by  $\mathcal{W}$  as a state representing an unsaturated application of an abstraction, a constructor expression or an accumulator. In the latter two cases, the stack must be empty after evaluation.

**DEFINITION 6.1** (Legal final states of ISSTG)  $\mathcal{W}(\Delta, p, p')$  holds if and only if  $\Delta[p \mapsto (p_c, cd)]$  and

- $cs[p_c \mapsto [\text{ARGCHECK } n, \dots]] \wedge |p'| < n$ , or
- $cs[p_c \mapsto [\text{RETURNCON } C]] \wedge p' = \diamond$ , or
- $cs[p_c \mapsto [\text{ACCU } C]] \wedge p' = \diamond$ .

This definition translates definition 4.1 of legal SSTG1 states to ISSTG, without conceptual change in its meaning.

The necessary adoptions for read back are shown in figure 6.6.

To implement the read back rules, it seems necessary to analyze the first instruction of the sequence  $r$  points to, to decide between rules  $\text{Lam}_\uparrow$ ,  $\text{Cons}_\uparrow$  and  $\text{Accu}_\uparrow$ . However, this is inconvenient when machine code for physical processors is generated and analyzed. This inconvenience is easily avoided by an additional table created during compilation which records the addresses of compiled  $\lambda$ -abstractions and constructors.

Rule  $\text{Case}_{\uparrow\emptyset}$  uses code pointers  $p_{C_i}$  that are assumed to point to instruction sequences  $[\text{RETURNCON } C_i]$ . These can be pre-located before normalization, after collecting all constructors occurring in a program by a simple static analysis. This avoids the need to generate new code during normalization. Moreover, this rule uses the branch info table to look up for the pointer of the continuation for the case alternatives, for which constructors alternatives are defined, and how many arguments each of these constructors takes.

## 6.4 CORRESPONDENCE BETWEEN SSTG2 AND ISSTG

Again, we define a correspondence relation between accumulators, heaps, stacks and whole configurations. Using this cor-

---


$$\text{Norm} \frac{([\text{ENTER}], q \vdash S, r, \Gamma) \rightarrow^* (\text{is}, \mathbf{p}', p, \Delta) \quad \mathcal{W}(\Delta, p, \mathbf{p}') \quad \Delta : p : \mathbf{p}' \uparrow \Theta : v}{\Gamma : q : S \updownarrow \Theta : v}$$


---

Figure 6.5: Normalization using ISSTG

---


$$\begin{array}{c} \text{Lam}_{\uparrow} \frac{\Gamma \cup [q \mapsto (p_{\text{accu}}, \langle y \rangle)] : p : \mathbf{p}' \vdash q \updownarrow \Delta : v \quad q, y \text{ fresh} \quad \text{cs}[r \mapsto \text{ARGCHECK } n \vdash \text{is}]}{\Gamma[p \mapsto (r, \mathbf{r}')] : p : \mathbf{p}' \uparrow \Delta : \lambda y. v} \\[10pt] \text{Cons}_{\uparrow} \frac{\bigwedge_{i=1}^{|q|} \Gamma_i : q_i : \diamond \updownarrow \Gamma_{i+1} : v_i \quad \text{cs}[r \mapsto [\text{RETURNCON } C]]}{\Gamma_1[p \mapsto (r, \mathbf{q})] : p : \diamond \uparrow \Gamma_{|q|+1} : C v} \\[10pt] \text{Accu}_{\uparrow} \frac{\Gamma : k \uparrow_{\langle \rangle} \Delta : v \quad \text{cs}[r \mapsto [\text{ACCU}]]}{\Gamma[p \mapsto (r, k)] : p : \diamond \uparrow \Delta : v} \\[10pt] \text{Var}_{\uparrow_{\langle \rangle}} \frac{}{\Gamma : \langle x \rangle \uparrow_{\langle \rangle} \Gamma : x} \\[10pt] \text{App}_{\uparrow_{\langle \rangle}} \frac{\Gamma : k \uparrow_{\langle \rangle} \Delta : h \quad \Delta : q : \diamond \updownarrow \Theta : v}{\Gamma[p \mapsto (r, k)] : \langle p q \rangle \uparrow_{\langle \rangle} \Theta : h v} \\[10pt] \text{Case}_{\uparrow_{\langle \rangle}} \frac{\begin{array}{l} \Gamma : k \uparrow_{\langle \rangle} \Delta_1 : h \\ \bigwedge_{i=1}^{|C|} \Delta_i \cup \Delta'_i : p' : q \vdash q' \updownarrow \Delta_{i+1} : v_i \\ \text{where } \text{bi}[q \mapsto (n, (C, m))], |\mathbf{p}_i''| = |\mathbf{y}_i| = m_i \text{ and } p', p'', y \text{ fresh} \\ \text{and } \Delta'_i = [p' \mapsto (p_{C_i} \mathbf{p}_i'')] \cup [\mathbf{p}_i'' \mapsto (p_{\text{accu}}, \langle \mathbf{y}_i \rangle)] \end{array}}{\Gamma[p \mapsto (r, k)] : \langle \text{case } p \text{ of } (q, q') \rangle \uparrow_{\langle \rangle} \Delta_{|alt|+1} : \text{case } h \text{ of } C y \rightarrow v}$$


---

Figure 6.6: Read back definition for ISSTG

respondence, it can be shown that for each SSTG2 transition sequence reaching a legal final state there is an equivalent ISSTG transition sequence for each corresponding initial state, reaching a corresponding final state, and vice versa. This equivalence of weak evaluation sequences leads to the equivalence of the strong normalization relations for SSTG2 and ISSTG.

**DEFINITION 6.2** (Accumulator correspondence) *An accumulator  $k$  of SSTG2 and an accumulator  $k'$  of ISSTG are corresponding, written  $k \sim k'$  iff*

- $k = k' = \langle x \rangle$ , or
- $k = k' = \langle p \ q \rangle$ , or
- $k = \langle \text{case } p \text{ of } (\mathbf{alt}, E) \rangle$  and  $k' = \langle \text{case } p \text{ of } (p_{\mathbf{alt}}, q) \rangle$  with  $p_{\mathbf{alt}} = \text{trAs } \mathbf{alt} \ \rho$  and  $\text{dom } E = \text{dom } \rho = X$  where  $\forall x \in X. E[x \mapsto q_{\rho(x)}]$ .

That is, two accumulators are corresponding if they represent the same free variable, the application of the same function to the same argument pointers, or stuck case expressions of the same scrutinee pointer where the code pointer in the ISSTG variant refers to the compiled version of the alternatives, and the environment  $E$  is connected to the pointers  $q$  via  $\rho$ : Every variable is mapped by  $E$  to a pointer present in  $q$ , with an index given by  $\rho$ . Here and in the following,  $\text{dom}$  is a function to obtain the domain of a mapping.

**DEFINITION 6.3** (Stack correspondence) *Two stacks  $S$  of SSTG2 and  $S'$  of ISSTG are corresponding, written  $S \sim S'$ , iff*

- $S = p \cdot S_R$  and  $S' = p \cdot S'_R$  with  $S_R \sim S'_R$ , or
- $S = \#p \cdot S_R$  and  $S' = \#p \cdot S'_R$  with  $S_R \sim S'_R$ , or
- $S = (\mathbf{alt}, E) \cdot S_R$  and  $S' = p_{\mathbf{alt}} \cdot q \# S'_R$  with  $S_R \sim S'_R$  and  $p_{\mathbf{alt}} = \text{trAs } \mathbf{alt} \ \rho$  and  $\text{dom } E = \text{dom } \rho = X$  where  $\forall x \in X. E[x \mapsto q_{\rho(x)}]$ .

Thus, corresponding stacks contain the same argument pointers and update frames, and case continuations obey the same relationship as in the accumulator correspondence of stuck case expressions.

**DEFINITION 6.4** (Heap correspondence) *A heap  $\Gamma$  of SSTG2 and a heap  $\Gamma'$  of ISSTG are corresponding, written  $\Gamma \sim \Gamma'$ , if and only if for all pointers  $p$*

- $\Gamma[p \nrightarrow] \text{ and } \Gamma'[p \nrightarrow]$ , or
- $\Gamma[p \nrightarrow] \text{ and } \Gamma'[p \mapsto (p_{\mathbf{bh}}, q)]$ , or

- $\Gamma[p \mapsto (lf, E)]$  and  $\Gamma'[p \mapsto (p_c, q)]$  with  $p_c = \text{trB } lf|_t$  and  $E = [t \mapsto q]$ , or
- $\Gamma[p \mapsto k]$  and  $\Gamma'[p \mapsto (p_{\text{accu}}, k')]$  with  $k \sim k'$ , or
- $\Gamma[p \mapsto \text{ind} \rightarrow q]$  and  $\Gamma'[p \mapsto (p_{\text{ind}}, q)]$ , or
- $\Gamma[p \mapsto (x \mathbf{x}', E)]$  and  $\Gamma'[p \mapsto (p_{\text{pap}}, q :: q')]$  with  $E = [x \mapsto q] \cup [x' \mapsto q']$ .

We look at the definition item by item. The relation  $\Gamma \sim \Gamma'$  holds in the following cases:

- $p$  is unbound in both heaps.
- $p$  is unbound in  $\Gamma$  and bound to a black hole in  $\Gamma'$ . This happens when a thunk is entered for evaluation: `sstg2` removes the binding completely from the heap, while `isstg` overwrites the code pointer of the respective closure with  $p_{\text{bh}}$ .
- $p$  is bound to a regular closure in  $\Gamma$  with  $lf$  as its code part and environment  $E$ , and to a regular closure in  $\Gamma'$  with a code pointer to the compiled instructions of  $lf$  and a sequence of pointers  $q$  resembling the environment  $E$ : each variable of the trimmer  $t$  is mapped to the corresponding pointer in  $q$ .
- $p$  is bound to an accumulator in  $\Gamma$  and to a closure with code pointer  $p_{\text{accu}}$  in  $\Gamma'$  that contains a closure data field with a corresponding accumulator.
- $p$  is bound to an indirection  $\text{ind} \rightarrow q$  in  $\Gamma$  and a closure  $(p_{\text{ind}}, q)$  in  $\Gamma'$ .
- $p$  is bound with a non-updating binding to an application on  $\Gamma$  and a closure with code pointer  $p_{\text{pap}}$  in  $\Gamma'$ . The environment  $E$  gives the same values as the pointer sequence  $q :: q'$ .

**DEFINITION 6.5 (Correspondence of Configurations)** *Two configurations  $(\Gamma, e, E, S)$  of `sstg2` and  $(is, S', p, \Gamma')$  of `isstg` are corresponding, written  $(\Gamma, e, E, S) \sim (is, S', p, \Gamma')$ , iff  $\Gamma \sim \Gamma'$  and*

- $is = \text{trE } e \ \rho \ \eta$  and  $S' = q \# S''$  and  $\Gamma'[p \mapsto (p_c, q')]$  with  $\forall x \in \text{dom } \rho. E[x \mapsto q_{\rho(x)}]$  and  $\forall x \in \text{dom } \eta. E[x \mapsto q'_{\eta(x)}]$  where  $\text{dom } E = \text{dom } \rho \cup \text{dom } \eta$  and  $S \sim S''$ , or
- $e = x$  with  $E[x \mapsto p]$  and  $\Gamma[p \mapsto (lf, E')]$  while  $\Gamma'[p \mapsto (p_c, q)]$  and  $cs[p_c \mapsto is]$ , and moreover  $S \sim S'$ , or
- $e = x$  with  $E[x \mapsto p]$  and  $\Gamma[p \mapsto k]$ , and  $is = [\text{ACC}U]$ , and moreover  $S \sim S'$ , or

- $e = x$  with  $E[x \mapsto p]$  and  $\Gamma[p \mapsto q]$ , for the instruction sequence  $is = [\text{BUILDENV}(\text{NODE } 1), \text{ENTER}]$  holds, and moreover  $S \sim S'$ , or
- $e = x$  with  $E[x \mapsto p]$  and  $\Gamma[p \mapsto y \ y']$ , and  $is = [\text{PAP}, \text{ENTER}]$ , and moreover  $S \sim S'$ .

Thus, two configurations are corresponding in the following cases.

- The control expression of  $\text{sstg2}$  is reflected by the code sequence of  $\text{isstg}$ , where the translation was performed by some  $\rho$  and  $\eta$  that allow finding the variables' values either in the topmost stack frame or via the current node pointer.
- A  $\lambda$ -form  $lf$  is evaluated, where  $is$  of  $\text{isstg}$  is the result of  $\text{trB}$  for  $lf$ . To see why this follows from the definition, note the third case in the definition of  $\Gamma \sim \Gamma'$ .
- An accumulator is evaluated, and  $is = [\text{ACCU}]$ . Note that  $\Gamma \sim \Gamma'$  implies  $\Gamma'[p \mapsto k']$  with  $k \sim k'$ .
- An indirection is evaluated,  $is = [\text{BUILDENV}(\text{NODE } 1), \text{ENTER}]$ , our code for indirections in  $\text{isstg}$  located in the code store at  $p_{\text{ind}}$
- A partial application is evaluated, and  $is = [\text{PAP}, \text{ENTER}]$ , our  $\text{isstg}$  partial applications at  $p_{\text{pap}}$ .

## 6.5 EQUIVALENCE OF SSTG2 AND ISSTG

An equivalence proof of  $\text{sstg2}$  and  $\text{isstg}$  is divided into a proof of the equivalence of weak evaluation and a proof of equivalence of the read back and strong normalization relations. Each of these proofs is in turn divided into a completeness and a soundness part.

**LEMMA 6.6** (Equivalence of weak evaluation) *Assume two corresponding configurations  $(\Gamma, e, E, S)$  of  $\text{sstg2}$  and  $(is, S', p, \Gamma')$  of  $\text{isstg}$ . Then we have*

$$(\Gamma, e, E, S) \xrightarrow[\text{SSTG2}]{}^* (\Delta, x, E[x \mapsto p], q) \quad \text{with } \mathcal{W}(\Delta, p, q)$$

is equivalent to

$$(is, S', p, \Gamma') \xrightarrow[\text{ISSTG}]{}^* (is', q, p, \Delta') \quad \text{with } \mathcal{W}(\Delta', p, q)$$

where  $(\Delta, x, E, q) \sim (is', q, p, \Delta')$ .

**PROOF** (Sketch) If we start with a reduction sequence of either  $\text{sstg2}$  or  $\text{isstg}$ , we can partition this sequence into a finite set

of segments  $C \rightarrow^* D$ , such that we can find for each segment a reduction sequence in the other semantics  $C' \rightarrow^* D'$ , where the configurations  $C$  and  $C'$  correspond as well as  $D$  and  $D'$ . Taking all segments in the other semantics together, we can form a reduction sequence reaching the desired legal final state.

Since the partitioning of sequences and the translation into the other semantics is the most crucial part, we show details in appendix B.  $\square$

**LEMMA 6.7** (Equivalence of strong evaluation) *The following three propositions hold.*

- *For the strong normalization relation:*

$$\Gamma : x : E[x \mapsto p] : S \updownarrow^{\text{SSTG2}} \Theta : v \Leftrightarrow \Gamma' : p : S' \updownarrow^{\text{ISSTG}} \Theta' : v$$

*whenever  $\Gamma \sim \Gamma'$  and  $S \sim S'$ . For the resulting heaps,  $\Theta \sim \Theta'$  holds.*

- *For read back:*

$$\Gamma : p : p' \updownarrow^{\text{SSTG2}} \Theta : v \Leftrightarrow \Gamma' : p : p' \updownarrow^{\text{ISSTG}} \Theta' : v$$

*whenever  $\Gamma \sim \Gamma'$ . For the resulting heaps,  $\Theta \sim \Theta'$  holds.*

- *For read back of accumulators:*

$$\Gamma : k \updownarrow^{\text{SSTG2}} \Theta : v \Leftrightarrow \Gamma' : k' \updownarrow^{\text{ISSTG}} \Theta' : v$$

*whenever  $\Gamma \sim \Gamma'$  and  $k \sim k'$ . For the resulting heaps,  $\Theta \sim \Theta'$  holds.*

**PROOF (Sketch)** For each direction, i.e.  $\text{SSTG2} \Rightarrow \text{ISSTG}$  and  $\text{ISSTG} \Rightarrow \text{SSTG2}$ , show an implication by induction on the deduction tree of the premise. For the equivalence of strong normalization, lemma 6.6 is used.  $\square$

Thus, we get following corollary for strong normalization.

**COROLLARY 6.8** (Correctness of strong normalization) *For any expression  $e$  that is closed, any pointer  $p$ , and any normal form  $v$ , it holds that*

$$\begin{aligned} & \exists \Delta. \emptyset : e : \emptyset : \Diamond \updownarrow^{\text{SSTG2}} \Delta : v \\ \Leftrightarrow & \exists \Delta'. [p \mapsto (\text{trB } e|_{\Diamond}, \Diamond)] : p : \Diamond \updownarrow^{\text{ISSTG}} \Delta' : v. \end{aligned}$$

That is, to normalize an expression  $e$ , we may either use  $\text{SSTG2}$  for normalization, starting with an empty heap, or we may start with a heap containing a single closure with a pointer to the compiled code of  $e$ , and normalize this closure with an empty stack using  $\text{ISSTG}$ .

**PROOF** Consequence of lemma 6.7. Technically, we need an exception in definition 6.4 of the heap correspondence relation dealing with the single preallocated closure bound to  $p$ .  $\square$

## 6.6 CORRECTNESS OF COMPILED NORMALIZATION

We are now ready to prove that the compilation scheme from figure 6.3 faithfully adheres to the semantics  $s_4$  from chapter 3, which is our main corollary.

**COROLLARY 6.9** (Correctness of compilation) *A normalization using semantics  $s_4$  leads to the same normal form as the normalization of the compiled code of this expression using semantics  $ISSTG$ . Formally, for all closed expressions  $e$ , all pointers  $p$  and all normal forms  $v$  the following equivalence holds:*

$$\begin{aligned} & \exists \Delta. \emptyset : e \Downarrow^{s_4} \Delta : v \\ \Leftrightarrow & \exists \Delta'. [p \mapsto (\text{trB } e|_{\diamond}, \diamond)] : p : \diamond \Downarrow^{ISSTG} \Delta' : v. \end{aligned}$$

**PROOF** Simple consequence of the correctness corollaries 4.8, 5.4 and 6.8 from the previous chapters.  $\square$

That is, whenever we can normalize an expression  $e$  to a normal form  $v$  using  $s_4$ , we can obtain the same normal form using  $ISSTG$ . Therefore, the transformation of  $s_4$  via  $SSTG1$  and  $SSTG2$  to our abstract machine  $ISSTG$  is correct. Now, we may rely on this correctness result and replace interpreted normalization with the normalization using compiled code. But this is just the first step of our way to the goal of this thesis, namely a dependent type checker using compiled code: showing that the code obtained by our compilation can indeed be used during dependent type checking with reasonable performance is subject of the next and final part of this work.



### Part III

## DEPENDENT TYPE CHECKING



In part II we defined a strong normalization system using compiled code that implements lazy evaluation. We did this to find a suitable reduction mechanism for dependent type checkers.

With this chapter we start the final part of our presentation, and show how our normalization system can be used in this context, and how it applies to one instance of a dependently typed system: as an example of a dependent type system, we elected the pure type systems (PTS for short), as presented by Barendregt [6].

In this chapter, we give a short introduction into their definition. The description of pure type systems gives a framework for many concrete type systems, and can be instantiated by choosing a set of sorts, a set of axioms and a set of so-called rules. We detail these degrees of freedom and give an example of a concrete system in the following.

The choices of the sorts, axioms and rules have crucial consequences for the system. Not only is the set of all well-typed expressions determined by their specification, but normalizability is influenced as well: in certain systems, every well-typed expression is strongly normalizable, while with other choices expressions without normal form may exist.

Next, we show how a PTS can be extended by inductive data types, case analyses and fixed points, as proposed by Paulin-Mohring [30] and adapted for the use in the proof assistant Coq [28].

This chapter is meant only as an overview to make it easier for the reader to follow the implementation description in chapter 8. A more thorough treatment can be found in the cited literature.

## 7.1 BASIC TYPING RULES

The specification of a PTS consists of three sets  $S$ ,  $A$ , and  $R$ .

- $S$  is the set of all *sorts*.
- $A$  is the set of *axioms* of the form  $s_1 : s_2$  where  $s_1$  and  $s_2$  are elements of  $S$ .
- $R$  is the set of *rules* of the form  $(s_1, s_2, s_3)$  where again the  $s_i$  are elements from  $S$ . As a shorthand notation, we often use  $(s_1, s_2)$  to stand for  $(s_1, s_2, s_2)$  for the common case that the second and third component of a rule are equal.

---


$$\begin{array}{lcl}
e & ::= & x \\
& | & s \\
& | & e\ e \\
& | & \lambda x : e. e \\
& | & \Pi x : e. e
\end{array}$$


---

Figure 7.1: Syntax of PTS expressions

The meaning of these sets will get clearer when we cover the typing rules and an example of a PTS below. For a first intuition, note that a typical choice is  $S = \{*, \square\}$ , where  $*$  is the type of all types, and  $\square$  is the type of  $*$ , thus  $A = \{* : \square\}$ .

### 7.1.1 Syntax

Before looking at the typing rules for pure type systems, we introduce the syntax of PTS expressions in figure 7.1.

In the simplest case, an expression is just a variable  $x$ .

Next, a sort  $s$  is an expression, where  $s$  is a member of the set of all sorts  $S$ .

An application is denoted by juxtaposition, and not restricted to variables in function or argument positions, as it was necessary in FUN expressions.

Abstractions are another possible case of expressions, where the type of the abstracted variable has to be annotated.

The last variant of PTS expressions are so-called dependent products, where  $\Pi x : e_1. e_2$  is the type of functions taking an argument  $x$  of type  $e_1$  to a result of type  $e_2$ , as for instance  $\lambda$ -abstractions like  $\lambda x : e_1. e_3$  where  $e_3$  has type  $e_2$ . Note however that  $x$  may occur in  $e_2$ , so the type of the resulting value may depend on the value of the argument. We will use a short-hand notation  $e_1 \rightarrow e_2$  when  $x$  is not free in  $e_2$  to make the connection to the ordinary function space clear.

In comparison to other type systems for functional languages it is remarkable that types and terms are on the same language level: an expression  $e$  might be e.g. an abstraction or a (dependent product) type. In the following, we use the meta-variable  $e$  for arbitrary expressions, while we use  $T$  when an expression is used as a type. This cannot be a strict dichotomy, since expressions and types are not syntactically separated. So we just want to aid intuition into the most prominent role an expression plays in a given context when we chose between  $e$  and  $T$  as a meta-variable name.

---


$$\begin{array}{c}
\text{Ax} \frac{s_1 : s_2 \in A}{\emptyset \vdash s_1 : s_2} \\
\\
\text{Env} \frac{\Gamma \vdash T : s \quad x \notin \Gamma}{\Gamma \cup \{x : T\} \vdash x : T} \\
\\
\text{Weak} \frac{\Gamma \vdash e : T \quad \Gamma \vdash T' : s \quad x \notin \Gamma}{\Gamma \cup \{x : T'\} \vdash e : T} \\
\\
\text{Prod} \frac{\Gamma \vdash T : s_1 \quad \Gamma \cup \{x : T\} \vdash T' : s_2 \quad (s_1, s_2, s_3) \in R}{\Gamma \vdash (\Pi x : T. T') : s_3} \\
\\
\text{App} \frac{\Gamma \vdash e_1 : (\Pi x : T. T') \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 e_2 : T'[e_2/x]} \\
\\
\text{Abs} \frac{\Gamma \cup \{x : T\} \vdash e : T' \quad \Gamma \vdash (\Pi x : T. T') : s}{\Gamma \vdash (\lambda x : T. e) : (\Pi x : T. T')} \\
\\
\text{Conv} \frac{\Gamma \vdash e : T \quad \Gamma \vdash T' : s \quad T =_\beta T'}{\Gamma \vdash e : T'}
\end{array}$$


---

Figure 7.2: Typing rules for pure type systems

### 7.1.2 Typing

The typing rules for pure type systems are given in figure 7.2, taken from [6] with only slight changes to the typography. Typing is a three place relation  $\cdot \vdash \cdot : \cdot$  where  $\Gamma \vdash e : T$  means that expression  $e$  has type  $T$  in environment  $\Gamma$ .<sup>1</sup> A type environment is a set of type assignments  $x : T$  that gives a type  $T$  to variable  $x$ .

Rule Ax states that the typing of sorts is directly derived from the set of axioms  $A$  specifying a concrete PTS instance: whenever a matching axiom is contained in  $A$ , we can conclude that the corresponding typing of sorts holds in an empty environment.

<sup>1</sup> Type environments are unrelated to the heaps of our semantics  $s_4$  through  $\text{ISSTG}$ , we just use the same Greek letter to stick to the conventions in existing literature. It will be clear from the context the letter  $\Gamma$  appears in whether a heap or a type environment is meant.

With rule Env we can conclude that a variable has the type  $T$  given by the environment, provided that  $T$  is a legal type, i.e. the type of  $T$  is a sort.

The weakening rule Weak allows us to add unneeded type assignments to the environment: if we are able to derive the typing judgment  $\Gamma \vdash e : T$ , we can add an additional type assignment  $x : T'$  to the environment, provided that  $T'$  is a legal type and  $x$  does not occur in  $\Gamma$ .

The typing of product types uses rule Prod. A dependent product  $\Pi x : T. T'$  is well-typed when  $T$  and  $T'$  are legal types with sorts  $s_1$  and  $s_2$ , respectively. Since  $x$  may occur in  $T'$ , we add an appropriate type assignment to the environment here. The product then has sort  $s_3$  as a type, if the triple  $(s_1, s_2, s_3)$  is in  $R$ . Only in this case the product is a legal type. Note that the specific typings derivable with this rule are dependent on the specification of the PTS instance. Products are the types of abstractions (see below), so we can influence via a choice of  $R$  which abstractions are possible. We will make this point clearer in an example given in section 7.1.3.

Rule App describes the typing of applications. Expression  $e_1$  in function positions has to have a product type. The type of argument  $e_2$  must be the type of the formal argument in this product type. In the result type of the product, i.e.  $T'$ , the formal argument  $x$  may occur, so we replace it with the actual argument  $e_2$  in the conclusion. We will justify this substitution in our example in section 7.1.3.

According to rule Abs, an abstraction  $\lambda x : T. e$  has product type  $\Pi x : T. T'$ , if the body of the abstraction  $e$  has type  $T'$ , where we add an type assignment for  $x$  to the environment, and the product in fact is a legal type of some sort  $s$ . To derive this, rule Prod has to be used, so here the choice of  $R$  influences the abstractions possible.

The last rule Conv is the reason we need a normalization system for type checking PTS expressions. It makes use of  $\beta$  equivalence, that is equivalence up to  $\beta$ -reductions and  $\beta$ -expansions. A  $\beta$ -reduction is defined as usual by the transition

$$(\lambda x : T. e_1) e_2 \rightarrow e_1[e_2/x].$$

We further subsume  $\alpha$ -conversion, so that names of bound variables do not influence the typing judgments, without formally reflecting this in the typing rules.

If we can derive type  $T$  for an expression  $e$ , and  $T$  is  $\beta$ -equivalent to some legal type  $T'$ , then we can conclude that  $e$  has type  $T'$ , too. That means, we treat all  $\beta$ -convertible types as identical. To implement this rule, that clearly is undecidable in general, we may try to reduce  $T$  and  $T'$  to strong normal forms and compare these. As already mentioned, some PTS instances

are strongly normalizing, so for these this would be a complete strategy. For other instances, this is an incomplete strategy, sometimes failing to accept feasible typings. However, using a lazy normalization system as developed in part II of the present work maximizes the number of normalizable types.

### 7.1.3 Example: The Calculus of Constructions

We now consider a specific instance of a PTS to aid the intuition of the typing rules.

The calculus of constructions, introduced by Coquand and Huet [10], is the PTS defined by following specification.

$$\begin{aligned} S &= \{*, \square\} \\ A &= \{* : \square\} \\ R &= \{(*, *), (\square, *), (*, \square), (\square, \square)\} \end{aligned}$$

So  $*$  is the type of all types, with sort  $\square$  as its own type.

Let us consider how this choice of  $R$  influences the abstractions possible in this PTS. Recall that  $(s_1, s_2)$  is a shorthand notation for  $(s_1, s_2, s_2)$ .

The first rule  $(*, *)$  lets us build ordinary abstractions. If we take  $\Gamma = \{x : *\}$ , we may derive

$$\Gamma \vdash (\lambda y : x. y) : (x \rightarrow x).$$

That is, in  $\Gamma$  we capture the assumption that  $x$  is a type, and derive under this assumption that  $\lambda y : x. y$  is a function mapping elements of  $x$  to elements of  $x$ . In the derivation we have to check whether  $\Pi y : x. x$  (that is,  $x \rightarrow x$  without our short-hand notation) is a legal type using rule Prod. Since  $\Gamma \vdash x : *$  and  $\Gamma \cup \{y : x\} \vdash x : *$  can be derived, and  $(*, *) \in R$ , we can conclude  $\Gamma \vdash (\Pi y : x. x) : *$ .

The second rule  $(\square, *)$  allows us to build polymorphic abstractions. For example, we can derive

$$\emptyset \vdash (\lambda x : *. \lambda y : x. y) : (\Pi x : *. x \rightarrow x).$$

For this derivation, we have to check whether  $\Pi x : *. x \rightarrow x$  is a valid type. Since  $\emptyset \vdash * : \square$  and  $\Gamma \vdash x \rightarrow x : *$  as shown above, we can conclude using rule Prod that

$$\emptyset \vdash (\Pi x : *. x \rightarrow x) : *$$

since  $(\square, *) \in R$ . In this example,  $\lambda x : *. \lambda y : x. y$  is the polymorphic identity function taking a type  $x$  as an argument and returning the identity function for this type as a result. Taking  $\Gamma'$  to be  $\{t : *\}$  we can derive that monomorphic instances gained

by applying the polymorphic function to a type argument are well-typed:

$$\Gamma' \vdash (\lambda x : *. \lambda y : x. y) t : (t \rightarrow t).$$

Here, we have to apply rule App, substituting the actual argument  $t$  for the formal argument  $x$  in the result type of the product  $\Pi x : *. x \rightarrow x$ .

Next we look at an example using the fourth rule  $(\square, \square)$ , postponing the third rule for a moment. With this rule, we can build types for polymorphic type constructors, that is for functions mapping types to types. For example, we can derive

$$\emptyset \vdash (* \rightarrow *) : \square$$

by rule Prod since  $\emptyset \vdash * : \square$  and  $\{t : *\} \vdash * : \square$  (where  $t$  is the unused variable hidden in our short-hand notation) and we have  $(\square, \square) \in R$ . So  $* \rightarrow *$  is the legal type of all type constructors in this PTS.

The third rule in  $R$ , namely  $(*, \square)$ , allows dependent types, i. e. abstractions over values that return types. We can derive a type for e. g. vectors of fixed length when we assume a type  $n$  that shall play the role of the natural numbers:

$$\{n : *\} \vdash (n \rightarrow * \rightarrow *) : \square.$$

Here,  $n \rightarrow * \rightarrow *$  is the type of functions taking a natural number as an argument and returning a type constructor. This typing is possible because of rule Prod where both typing hypotheses  $\{n : *\} \vdash n : *$  and  $\{n : *, x : n\} \vdash * \rightarrow * : \square$  (where again  $x$  is a hidden variable) are derivable, and  $(*, \square) \in R$ .

Our last two examples, for the third and fourth rule in  $R$ , have shown that it is possible to define the types of type constructors and dependent types, but not how to define values of these types. We will return to this question in section 7.2.3 where we give examples for inductive data type definitions.

## 7.2 INDUCTIVE DATA TYPES

Pure type systems can be extended by the definition of inductive data types. Such a definition introduces a new type together with a set of constructors to define values of this type. The type of constructors can not only use parameters as in the parametric polymorphism implemented in functional languages, but can make use of so-called *indices*, too. The index value can be different for different constructors of the same inductive type, so it might be possible to build values with a certain index only with a single constructor, allowing to conclude that the other constructors cannot match. We will show an example for this in section 7.2.3.

---


$$\begin{array}{l}
\text{data } T (\mathbf{x} : \mathbf{T_P}) : T_a \text{ where} \\
C_1 : T_{C_1} \\
\vdots \\
C_n : T_{C_n}
\end{array}$$


---

Figure 7.3: Syntax of inductive definitions

The specification of PTS instances is extended with a further set  $I$ , listing the possible sorts of the defined types. A typical choice when starting with a PTS specified with  $S = \{*, \square\}$  is  $I = \{*\}$ , that is, all inductively defined types have to have type  $*$ .

We need several restrictions for inductive types to avoid the possibility of non-termination in otherwise strongly normalizing type systems, and will list them in section 7.2.2, without going into detail how their violation contributes to possible non-terminations.

### 7.2.1 Syntax

An inductive definition of a new data type has the shape shown in figure 7.3.

This defines a type  $T$ , parametrized by  $\mathbf{T_P}$ , of so-called arity  $T_a$ . Moreover, it defines constructors  $C_1$  through  $C_n$  having types  $T_{C_1}$  through  $T_{C_n}$ .

The type parameters  $\mathbf{T_P}$  are used for parametric polymorphism, they define type arguments that have to occur unchanged in the types of the constructors. The following sections will make this clear.

The arity  $T_a$  is in the simplest case a sort  $s$ , which is the type of  $T$ . In more complex situations, the arity can be a product type, adding type indices to the type parameters that may change in the types of subsequent constructors. Again, the following sections will bring clarification.

### 7.2.2 Typing

For a type definition to be legal, several restrictions must be met. First, the parameters  $\mathbf{T_P}$  must be legal types, making only type correct usage of the variables  $\mathbf{x}$ . Thus if we have parameters

$$(\mathbf{x}_1 : T_{P_1}) \dots (\mathbf{x}_n : T_{P_n})$$

then we must be able to derive for each index  $i$

$$\{\mathbf{x}_1 : T_{P_1}, \dots, \mathbf{x}_{i-1} : T_{P_{i-1}}\} \vdash T_{P_i} : s_i.$$

Next, the arity must be a product type with a sort as a final result type, i. e.  $T_a$  must have the shape  $\prod \mathbf{y} : \mathbf{T}_y . s_T$  with possibly empty sequences  $\mathbf{y}$  and  $\mathbf{T}_y$ . The sort  $s_T$  must be contained in the set  $I$  from the PTS specification. The complete type of the type constructor  $T$  is then

$$T_T = \prod \mathbf{x} : \mathbf{T}_P . T_a$$

where  $T_T$  must be a legal type, i. e.  $\emptyset \vdash T_T : s$  must be derivable for some sort  $s$ .

Moreover, each of the constructors' types has to be a legal type that may make reference to  $T$  and the parameters  $\mathbf{x}$ . The sort of the constructors' type must be the sort  $s_T$ , the final sort of the arity. Thus we must be able to derive

$$\{\mathbf{x} : \mathbf{T}_P, T : T_T\} \vdash T_{C_i} : s_T.$$

Another syntactic criterion on the types of the constructors is that they have to be product types with the final result being  $T$  or an application of  $T$  to some arguments. Otherwise it would not make sense to call  $C_i$  a constructor of  $T$ . If arguments are present, the first arguments must be exactly the parameters  $\mathbf{x}$ ; this is called the parametricity constraint.

Finally, all occurrences of  $T$  in  $T_{C_i}$  must be strictly positive. This means that if  $T_{C_i}$  has the shape  $\prod \mathbf{y} : \mathbf{T}_y . T e$ , and any of the  $\mathbf{T}_y$  is a product, then  $T$  may occur as the product's result, but not in its argument.

If all these conditions and restrictions are met, the complete type of constructor  $C_i$  then is  $\prod \mathbf{x} : \mathbf{T}_P . T_{C_i}$ .

A formal definition of the parametricity constraint and strict positivity can be found in the literature [28, 30].

### 7.2.3 Examples

All of the following examples of inductive definitions have to be seen in the context of the calculus of constructions, as specified in section 7.1.3. We assume  $I = \{*\}$ .

Our very first inductive data type is trivial, defining a type of Boolean values.

```
data bool : * where
  true : bool
  false : bool
```

Thus `bool` is a type of sort `*`, without any parameters, and having the two obvious constructors for our purposes.

Of course, inductive definitions may be recursive, so our next example shows the natural numbers in Peano style.

```
data nat : * where
  zero : nat
  succ : nat → nat
```

The type `nat` has two constructors: `zero` is a number, and `succ` gives for any number another number: its successor.

For an example using parametric polymorphism, we next consider a type `seq` for sequences.

```
data seq (a : *) : * where
  nil : seq a
  cons : a → seq a → seq a
```

Here, `nil` is the empty list and `cons` takes a new first element and a tail list to build a resulting list of type `seq a`. Note that the parameter `a` occurs as a first (and only) argument to every occurrence of `seq` in the constructors' types, fulfilling the parametricity constraint. The type of `seq` resulting from this definition is  $* \rightarrow *$ , i.e. the type of polymorphic type constructors mentioned in section 7.1.3. In a PTS without rule  $(\square, \square)$ , it would be impossible to define `seq`. The constructor `nil` has as its complete type  $\prod a : *. \text{seq } a$ , the complete type of `cons` is  $\prod a : *. a \rightarrow \text{seq } a \rightarrow \text{seq } a$ .

We consider next the type of vectors, that is of lists of fixed length, where the length is part of the type. This is accomplished by the use of an index as follows.

```
data vector (a : *) : nat → * where
  nil : vector a zero
  cons : a →  $\prod n : \text{nat}. \text{vector } a \ n \rightarrow \text{vector } a \ (\text{succ } n)$ 
```

We overload constructors `nil` and `cons` from the sequences of the last examples, since they play very similar roles. The type of `vector` resulting from this definition is  $* \rightarrow \text{nat} \rightarrow *$ , i.e. a function taking a type and a number to a new type. Compare this with our example of dependent types in section 7.1.3: only the order of the type and length arguments are switched. This is a legal type because we have  $(*, \square) \in R$ , otherwise the definition of `vector` would be impossible.

The constructor `nil` now specifies `zero` as the length of the vector, since no elements are present. More interesting is `cons`, prepending a first element in front of a vector of length `n`, resulting in a vector of length `succ n`.

Note that while the first argument to `vector` is always `a` in the constructors' types, fulfilling again the parametricity constraint, the second argument has different values at each occurrence. This is only allowed because the second argument is not introduced as a parameter but as an index part of the arity.

Now suppose we are given a value of type `vector a zero`: then we know that it has to be built with constructor `nil`, since `cons` can only construct values of type `vector a (succ n)` for some number `n`. Thus the index value gives some information about the constructor used to obtain a value.

Accordingly, when we are given a vector  $a$  ( $\text{succ} (\text{succ zero})$ ), we know that it must be built with  $\text{cons}$  with a tail argument of type vector  $a$  ( $\text{succ zero}$ ). So this tail again has to be built using  $\text{cons}$ , this time with a tail of type vector  $a$  zero, i.e. the empty vector. Summing up, we can conclude from the type vector  $a$  ( $\text{succ} (\text{succ zero})$ ) that the vector has length two.

As a final example, we show a type that captures the equality of two values of the same type.

```
data eq (a : *) (x : a) : a → * where
  refl : eq a x x
```

With this definition,  $\text{eq}$  has type  $\Pi a : *. a \rightarrow a \rightarrow *$ . In the propositions-as-types interpretation, dating back to the work of Curry [13], the type  $*$  can be seen as the type of propositions. Thus the type of  $\text{eq}$  can be read as the polymorphic type parametrized over  $a$  of binary functions returning a proposition on their arguments.

So  $\text{eq bool true true}$  has type  $*$  and can be seen as a proposition on the Boolean arguments  $\text{true}$  and  $\text{true}$ , while the application  $\text{eq bool true false}$  is also well-typed and represents another proposition (a false one, incidentally).

The constructor  $\text{refl}$  has the full type  $\Pi a : *. \Pi x : a. \text{eq } a \ x \ x$ , so it constructs values of the  $\text{eq}$  type where both non-type arguments are the same. This corresponds to the reflexivity of equality: we can obtain a value of type  $\text{eq bool true true}$  with  $\text{refl bool true}$ .

But we cannot construct a value of type  $\text{eq bool true false}$  using  $\text{refl}$ , since the Boolean arguments differ, and  $\text{refl}$  only builds values where they coincide. In fact, since the constructors are the only way to build values of inductive types, it is entirely impossible to give a value of type  $\text{eq bool true false}$ . This is a crucial property in the propositions-as-types interpretation: while it is possible to give values for types representing true propositions, it is impossible to give values for false ones.

### 7.3 CASE ANALYSES

Reasoning about which constructor has been used to build a given value, as we did in the last section with the length indices of vectors, is a nice property of indexed data types. However, we also need a way to reason about constructors programmatically. Therefore, we add the possibility of case analyses of inductive data types to a PTS. At this, the type of the result of an analysis may change dependent on the alternative chosen.

The specification of PTS instances is further extended by a set  $C$  of tuples of two sorts each. To scrutinize a value of inductive type  $T_I$  with sort  $s_1$ , resulting in values of type  $T_R$  with sort

---

```

case e as x return T of
  C1 y1 -> e1
  ⋮
  Cn yn -> en

```

---

Figure 7.4: Case analysis for pure type systems

$s_2$ , we need the pair  $(s_1, s_2)$  in  $C$ . A possible choice for a PTS with  $S = \{*, \square\}$  and  $I = \{*\}$  is  $C = \{(*, *)\}$ , so the result of a case analysis must be a value having a type that in turn has sort  $*$ , i.e. the result must be a regular value. Another possible choice would be  $C = \{(*, *), (*, \square)\}$ , allowing types as the result values of analyses, too.

### 7.3.1 Syntax

The syntax of a case analysis is given in figure 7.4. It looks very similar to case discriminations in **FUN**, the only difference is the phrase *as x return T*. Here,  $T$  is the return type of the case analysis, i.e. the type of the  $e_i$ . However, as mentioned above, the resulting type may vary depending on the chosen alternative. Therefore,  $x$  is bound in  $T$ , representing the scrutinee  $e$  and the chosen constructor pattern. The following sections will give details on this. If  $x$  does not occur in  $T$ , we will omit the phrase *as x*.

### 7.3.2 Typing

For a case discrimination to be well-typed, the scrutinee  $e$  must be well-typed, of course. We assume that the type of  $e$  is  $T_e$ . Then,  $T_e$  must be an inductively defined type, since it is not possible to discriminate on e.g. functions.

Moreover,  $T$  must be a legal type, and since it may refer to  $x$ , we check this in a type environment augmented with  $x : T_e$ . The resulting type of the whole case expression then is  $T[e/x]$ , that is we replace  $x$  with  $e$  in the resulting type.

When we assume that the sort of  $T_e$  is  $s_1$  and the sort of  $T$  is  $s_2$ , then the pair  $(s_1, s_2)$  must be an element of the set  $C$  from the PTS specification.

In the alternatives, the variables  $y_i$  may be used in expression  $e_i$ , with types according to the argument types declared in the inductive definition of  $T_e$  for the constructor  $C_i$ . The type of  $e_i$  then must adhere to  $T$ , but as we substituted  $e$  for  $x$  to determine

the resulting type of the whole expression, we now use  $C_i y_i$  to allow the type of the alternative to depend on the constructor pattern. Thus,  $e_i$  must have type  $T[C_i y_i/x]$ .

For a formal exposition of the typing rules for case expressions, also known as eliminations, we once again refer to the literature [28, 30]. Here an additional possibility for dependent types can be found, too, where the type of an alternative not only depends on the constructor pattern, but also on the index values of the scrutinee's type. We implemented such dependencies in our prototype described in chapter 8, but do not give a description here since it is not necessary for our presentation.

### 7.3.3 Examples

In this examples we again use the setting of the calculus of constructions as described in 7.1.3, and make use of the inductive types described in section 7.2.3. We assume  $C = \{(*, *)\}$ .

It is easy to define a function negating Boolean values.

```
not = λ b : bool . case b return bool of
      true -> false
      false -> true
```

We now want to show that every Boolean value is equal to its double negation using the propositions-as-types interpretation. Under this interpretation, the universal quantifier corresponds to the PTS product operator. Thus, we need to define a value of type

$$\prod b : \text{bool} . \text{eq bool } b \text{ (not (not } b \text{))}.$$

Note that  $\text{not (not } b \text{)}$  does not normalize to  $b$ : the case discrimination of  $\text{not}$  is on variable  $b$  here, so no specific branch can be selected.

The following expression fulfills our needs.

```
λ b : bool . case b as x return eq bool x (not (not x)) of
      true -> refl bool true
      false -> refl bool false
```

To see why this expression has the desired type, consider the first alternative. According to the typing rules, it has to have type  $\text{eq bool } x \text{ (not (not } x \text{))}$  as specified after return, but with  $x$  replaced by the constructor pattern  $\text{true}$ . This gives  $\text{eq bool true (not (not true))}$  that can be normalized to the type  $\text{eq bool true true}$  and we can define a value of this type using  $\text{refl}$ .

The second alternative is similar. So the type of the whole case expression is  $\text{eq bool } x \text{ (not (not } x \text{))}$  with  $x$  replaced by the scrutinee  $b$ , so we get  $\text{eq bool } b \text{ (not (not } b \text{))}$  as desired.

---


$$\text{fix } f (x : T) : T_e = e$$


---

Figure 7.5: Fixed points in pure type systems

#### 7.4 DEFINITIONS BY FIXED POINTS

It is well known that recursive functions can be defined via fixed points of higher-order functions. We will now show a language construct that allows such definitions in our PTS setting.

##### 7.4.1 Syntax

We extend the syntax of PTS expressions as shown in figure 7.5. The meaning of this expression is the solution of the recursive equation

$$f = \lambda x : T. e$$

where  $f$  occurs free in  $e$ . To be precise, it is the least solution in the sense of Scott's domain theory [33], as it is used in the denotational semantics of  $\lambda$ -calculi.

We require the sequence of variables  $x$  to contain at least one element (the sequence of types  $T$ , too, of course).

##### 7.4.2 Typing

The type of the fixed point, and accordingly the type of  $f$  in  $e$  is  $T_f$  defined as  $\Pi (x : T). T_e$ . Of course,  $T_f$  must be a legal type, so  $\Gamma \vdash T_f : s$  must hold for some sort  $s$ , where  $\Gamma$  gives types for the free variables occurring in the fixed point.

The type of the body  $e$  has to be  $T_e$ . Since the body may make use of the parameters  $x$  and the recursion variable  $f$ , we have to add them to the environment when checking this:

$$\Gamma \cup \{f : T_f, x : T\} \vdash e : T_e.$$

These two conditions ensure type correctness, so we know that no run time errors can occur. But we have to add a further restriction if we want to avoid terms without normal forms in an otherwise strongly normalizing PTS. We have to assure that at least one argument gets structurally smaller in every recursion. In our implementation, we require this to be the last argument of the argument sequence  $x$ . So in every occurrence of  $f$  in  $e$ , the last argument must be a value that has to be obtained from

one or more pattern matchings using a case analysis of this last element of  $x$ . We make this clear in the next section, giving two examples.

While it is possible to gain a little bit efficiency by allowing an arbitrary but fixed argument to structurally decrease, it is always possible to fulfill our restriction by introducing additional arguments after the decreasing argument not in the head of the fixed point, but to start the body of the fixed point with a  $\lambda$ -abstraction instead.

### 7.4.3 Examples

Using fixed points, we can define the addition of natural numbers as follows.

```
add = fix f (x : nat) (y : nat) : nat =
  case y of
    zero -> x
    succ y' -> succ (f x y')
```

This definition is quite unsurprising. Note that the last argument is indeed structurally decreasing. We perform a case analysis on the last argument variable  $y$ , obtain its predecessor  $y'$  from the pattern matching, and use  $y'$  as a last argument in the recursive call to  $f$ .

Moreover, we can define a predicate testing for even numbers.

```
even = fix f (x : nat) : bool =
  case x of
    zero -> true
    succ x' -> case x' of
      zero -> false
      succ x'' -> f x''
```

This example shows that an argument may get structurally smaller by more than one case analysis. On the path to the recursive call we perform first a pattern matching on  $x$ , obtaining its predecessor  $x'$ , and then a second matching on  $x'$ , obtaining  $x''$ , thereby decreasing the argument of  $f$  by two on each recursive call.

## NOTES ON THE IMPLEMENTATION

To prove the applicability of our approach to dependent type checking, we implemented a prototype checker for pure type systems in Haskell. This chapter describes the implemented features and the techniques used for the implementation.

## 8.1 FEATURES OF OUR IMPLEMENTATION

Our PTS checker implements the basic PTS typing rules, inductive definitions, case analyses and fixed points as described in the previous chapter. At this, it is not restricted to a single specific PTS instance. Instead, it makes use of a configuration file that lists the sets of

- available sorts,
- axioms,
- rules,
- allowed sorts of inductive definitions, and
- elimination possibilities.

A specification of the calculus of constructions might look in the syntax of our implementation as in listing 8.1. This configuration file defines exactly the system used for our examples in chapter 7.

Listing 8.2 shows a file with some definitions using this configuration file. It consists of multiple declarations. We first define the Peano numbers (lines 1-3), and have the inductively defined type `nat` and its constructors `zero` and `succ` available in the remainder of the file. Next, addition and multiplication are defined.

---

```

1 sorts:      *, #
2
3 axioms:     * : #
4
5 rules:      (*,*), (#,*), (*, #), (#, #)
6
7 idefs:      *
8
9 cases:      (*,*)

```

---

Listing 8.1: Specification file for the calculus of constructions

---

```

1 data nat : * where
2   zero : nat
3   succ : nat -> nat
4
5 add
6   = fix f (x : nat) (y : nat) : nat =
7     case y return nat of
8       zero -> x
9       succ y' -> succ (f x y')
10
11 mul : nat -> nat -> nat
12   = fix f (x : nat) (y : nat) : nat =
13     case y return nat of
14       zero -> zero
15       succ y' -> add x (f x y')

```

---

Listing 8.2: Basic arithmetic in the calculus of constructions

While the definition of `mul` makes reference to `add`, it must not use `mul` in its own body: recursion must be expressed using explicit fixed points. It is possible, as done in line 11, to give a type for a definition, but this is not mandatory, as exemplified by line 5.

Not shown in this example is the possibility to omit not the type information but the defining right hand side of a declaration. This introduces a free variable of the given type for the remainder of the file.

When our implementation is given these two files, it checks whether all definitions are well-typed according to the PTS specification file. The type checker is a straightforward implementation of the typing rules from chapter 7. At its heart lies a function for the basic rules of figure 7.2, taking the environment and the expression to check as parameters, and returning either the type of the expression or an appropriate error message. The only rule that cannot be directly converted to this functional style is rule *Conv* since it would have to guess a type. This rule is not *algorithmic* in the sense of Pierce [32]. Thus, instead of checking  $\beta$ -convertibility in a separate rule, we strongly normalize each type before adding it to the environment and return only types in strong normal form.

Before a type is normalized, it is always checked that it is indeed a legal type, so we don't need to deal with run time type errors. Moreover, if the PTS specified in the configuration file is strongly normalizing, that is, if every expression of the PTS has a strong normal form, we do not have to fear infinite computations and non-terminating type checks. It would be easy, but has not been done, to implement a reduction counter to deal with other PTS specifications without too long delays for the user of our system. Instead, normalization continues until either memory is

---

```

1 add:    (nat -> (nat -> nat))
2 mul:    (nat -> (nat -> nat))
3 nat:    *
4 succ:   (nat -> nat)
5 zero:   nat

```

---

Listing 8.3: Result of type checking arithmetic functions

exhausted or the process is terminated by the user or operating system.

The normalization during type checking can be done in two different ways, controlled by command line flags: either by a byte code interpreter for `ISSTG` instructions, or by a translation to machine code for x86 processors. The machine code generation is carried out using Harpy [18].

When type checking has finished successfully, the types of all declared identifiers are printed in alphabetical order. The result of our running example can be found in listing 8.3.

Some aspects are important for real implementations but unnecessary for our prototype. So our implementation has neither a module system nor a garbage collector. Both could be added to our system.

## 8.2 FIXED POINTS

Our exposition of pure type systems included fixed points as a special language construct. This is in contrast to `FUN`, where we allowed local bindings to be recursive, but did not have special syntax to indicate such usage. The consequences are unpleasant: while it is no problem if the expression to be normalized uses a recursive definition, we are in trouble when the result of the expression is itself recursively defined. For example, it is possible to normalize `FUN` expressions using a definition of the higher-order list function `map` that applies a function pointwise to the elements of a list. But it is not possible to compute a normal form for `map` itself, since the recursive definition would be unfolded infinitely in this case.

The `PTS` fixed points have a property that helps us in this situation: there is a designated argument of the fixed point – the last one – that must get structurally smaller with each recursive step. Thus, if this last argument is an accumulator, further unrolling of the definition does not make sense: the necessary case analysis on this accumulator would get stuck anyway. So we unfold fixed points only as long as the last argument evaluates to a constructor expression.

---

$lf$	$::=$	$\dots$	
	$ $	$\text{fix } f \, \mathbf{x} = e$	
$k$	$::=$	$\dots$	
	$ $	$\langle \text{fix } p \, q \, \mathbf{r} \rangle$	
$h$	$::=$	$\dots$	
	$ $	$(\text{fix } f \, \mathbf{x} = v) (\mathbf{v}' : \cdot h)$	$ \mathbf{v}' : \cdot h  =  \mathbf{x} $
$v$	$::=$	$\dots$	
	$ $	$(\text{fix } f \, \mathbf{x} = v) \mathbf{v}'$	$ \mathbf{v}'  <  \mathbf{x} $

---

Figure 8.1: Syntactic changes for fixed points in FUN

We will now show how fixed points can be integrated into FUN. We introduce new syntax, show how this syntax can be compiled to ISSTG code, how the ISSTG evaluation semantics has to be extended and how fixed points are read back.

### 8.2.1 New Syntax for FUN

We add fixed points as new  $\lambda$ -forms to FUN expressions as shown in figure 8.1. As all  $\lambda$ -forms, these are allocated as closures at run time.

Moreover, we add fixed points as new ISSTG accumulators and new normal forms.

Accumulators for fixed points are built whenever the last argument supplied to a fixed point is an accumulator. The accumulator built in this situation captures a pointer  $p$  to the compiled code of the fixed point body  $e$ , the pointer to the closure of the fixed point  $q$ , and a sequence of argument pointers  $\mathbf{r}$ . We will see shortly how these pointers are obtained and used.

A fixed point is an unreducible head when all its arguments are supplied, the last argument being a head itself. As for all heads, this forms a normal form, too. But there is a second kind of normal form for fixed points: an unsaturated fixed point where not all arguments are supplied. These two kinds of normal forms are differentiated in syntax, since the former can be used in function position of applicative normal forms, while the latter can not.

---


$$\begin{aligned}
\text{trB } (\text{fix } f \, x = e \mid t) &= p \\
&\& \text{cs}[p \mapsto [ \text{ARGCHECK } |x|, \\
&\quad \text{PUSHNODE}, \\
&\quad \text{PUSHNODE}, \\
&\quad \text{PUSHCONT } q \\
&\quad \text{BUILDENV } (\text{STACK } (3 + |x|)) \\
&\quad \text{ENTER} \quad \quad \quad ] ] \\
&\& \text{cs}[q \mapsto [\text{POPNODE}] \# \text{trE } e \, \rho \, \eta] \\
&\& \text{fi}[q \mapsto |x|]
\end{aligned}$$

where  $\rho = ([f \mapsto 1] \cup [x_i \mapsto i + 1 \mid i \leftarrow 1..|x|, 1 + |x|])$   
 $\eta = [t_i \mapsto i \mid i \leftarrow 1..|t|]$   
 and  $p, q$  fresh

---

Figure 8.2: Translation of fixed points to ISSTG

### 8.2.2 Compilation to ISSTG

The compilation of fixed points to ISSTG code is done with the new clause for  $\text{trB}$  shown in figure 8.2, extending the definition of figure 6.3. For the compiled code to work, we add the rules from figure 8.3 to the transition relation of figure 6.4.

The compiled code for the fixed point first checks whether all arguments are present with  $\text{ARGCHECK}$ , just as it is done for  $\lambda$ -abstractions. We then push the current node pointer value onto the stack for later use in the compiled body  $e$ . This saved pointer has two usages: on the one hand, it is necessary to access the closure variables  $t$ , on the other hand, it is the value of the recursion variable  $f$ . During ordinary evaluation, we can use the same pointer for both purposes. But during read back, we will bind  $f$  to an accumulator to intercept recursion, while still needing the fixed point closure pointer to access the closure's variables. Thus, we push the node pointer twice using the new  $\text{PUSHNODE}$  instruction to be able to exchange one copy during read back.

The body  $e$  gets compiled to a separate code sequence located at  $q$  in the code store, and  $q$  is pushed as a continuation. We then place the last argument of our fixed point, that is found after three pushes at stack index  $3 + |x|$ , at the top of the stack, and enter it.

When the last argument evaluates to a constructor, it jumps to the compiled code of the body  $e$ . Since in the original ISSTG definition constructors expected pointers to branch tables, and now find a pointer to a simple code sequence instead, we add a variant of the  $\text{RETURNCON}$  transition rule. The code of  $e$  is preceded

	Code	Stack	Node	Heap
	PUSHNODE $\vdash$ is	S	p	$\Gamma$
$\rightarrow$	is	$p \vdash S$	p	$\Gamma$
	POPNODE $\vdash$ is	$q \vdash S$	p	$\Gamma$
$\rightarrow$	is	S	q	$\Gamma$
	[RETURNCON C]	$q \vdash S$	p	$\Gamma$
$\rightarrow$	is	S	p	$\Gamma$
	where $cs[q \mapsto is]$			
	[ACCU]	$q \vdash q' \vdash q' \vdash q'' \vdash S$	p	$\Gamma$
$\rightarrow$	[ACCU]	S	r	$\Gamma'$
	where $\Gamma' = \Gamma \cup [r \mapsto (p_{\text{accu}}, \langle \text{fix } q' q'' \rangle)]$ with r fresh and $fi[q \mapsto  q'']$			

Figure 8.3: Extension of semantics ISSTG

with a POPNODE instruction, removing the first of the two copies of the fixed point closure pointer from the stack and restoring it as the node pointer for later access to the variables in  $t$  via  $\eta$ . The second copy stays at the stack and is accessed via  $\rho$ , where our recursion variable is bound to the topmost stack element, followed by the arguments  $x$ .

When the last argument evaluates not to a constructor, but to an accumulator, a new rule for the ACCU instruction is used. It makes use of a new fixed point info table  $fi$ , where the number of the arguments gets stored during compilation. The new rule then allocates a fresh accumulator, capturing the continuation address, the fixed point's node pointer and the arguments for later read back.

### 8.2.3 Read Back

For the read back of fixed point accumulators we use rule  $\text{Fix}_{\uparrow \emptyset}$  from figure 8.4 that extends the other read back rules for ISSTG from figure 6.6.

We allocate a closure containing only the continuation pointer to the compiled body, and several fresh variable accumulators for the recursion variable and the arguments. When the continuation code at  $p$  is executed, it pops the fixed point closure pointer  $p'$  from the stack, thus gaining access to the closure variables. Pointer  $r$  is taken for the recursion variable, thus intercepting all recursive calls. Here, we have two different pointers, namely  $p'$  and  $r$ , where we had two times the same pointer before, left by the double PUSHNODE instruction from figure 8.2. The accumulators located at  $r'$  serve as arguments for the fixed point's body.

Moreover, we normalize all original arguments  $p''$  as usual.

$$\text{Fix}_{\uparrow\langle\rangle} \frac{\Gamma' : q : p' \cdot r \cdot r' \Downarrow \Delta_1 : v \quad \bigwedge_{i=1}^{|\mathbf{p}''|} \Delta_i : p_i'' \Downarrow \Delta_{i+1} : v'_i}{\Gamma : \langle \text{fix } p \, p' \, p'' \rangle \uparrow_{\langle\rangle} \Delta_{|\mathbf{p}''|+1} : (\text{fix } f \, x = v) \, v'}$$

where  $\Gamma' = \Gamma \cup [q \mapsto (p, \diamond)][r \mapsto (p_{\text{accu}}, \langle f \rangle)][r' \mapsto (p_{\text{accu}}, \langle x \rangle)]$   
 and  $q, r, r', f, x$  fresh with  $|x| = |\mathbf{p}''|$

Figure 8.4: Read back of fixed points

Another rule, not shown here, deals with the read back of unsaturated fixed points. This is done by supplying accumulators as pseudo-arguments, as it was done for  $\lambda$ -abstractions by rule  $\text{Lam}_{\uparrow}$  in figure 6.6.

### 8.3 TRANSLATION FROM PTS TO FUN

The syntax of FUN expressions differs from the PTS syntax in several aspects. Most superficially, FUN expressions are flat, i. e. they do not have complex expressions in argument positions, and constructors and  $\lambda$ -abstractions may only occur in the right hand side of local bindings. However, as already mentioned, it is trivial to introduce additional local bindings to avoid this limitation.

Another simply solvable problem are the new kinds of names introduced in PTS definitions. We may find sorts, names of inductively defined types and variables declared to have certain types, but given no definition. Fortunately, we are well prepared for this situation: we allocate an accumulator for each of these names.

More intricate are the differences regarding two other aspects: FUN expressions do not have type annotations, while PTS expressions do. And in FUN expressions, constructors have to be saturated, while in PTS expressions they do not.

#### 8.3.1 Type Annotations

The first issue is that type annotations may contain an additional binding construct besides  $\lambda$ -expressions, namely dependent product types. For the translation of a product  $\prod x : T. T'$ , we use a special constructor P with a name that cannot occur in our PTS files. We then build the constructor expression  $P (\lambda x : T. T')$  and translate it as usual. When we face the special constructor P during read back, we first normalize its argument, yielding the

$\lambda$ -abstraction  $(\lambda y : T_n . T'_n)$  in strong normal form, and translate this abstraction back to  $\Pi y : T_n . T'_n$ .

Using this trick we can represent types as FUN expressions. But we still have to find a way to store them: while it is easy to add e.g. type annotated abstractions to SSTG2, it is not obvious how to do this for ISSTG.

Type annotations occur in the return phrase of case analyses, at  $\lambda$ -abstractions and at fixed points. Each of these corresponds to a certain kind of WHNF. So we need

- a way to find the compiled type annotation for a given WHNF, and
- a way to provide an appropriate environment for the type annotation that may contain free variables as well as variables bound in the annotation's context.

For case discriminations, consider as an example the expression

case  $e$  as  $x$  return  $T$  of ...

where  $T$  may contain the free variable  $x$  as well as any other variable bound in the context of the expression, that might be used in the case alternatives as well. When compiling this expression, we compile  $T$  using  $\text{trE}$  from figure 6.3 with the same stack environment  $\rho'$  as the alternatives, and a node environment containing only  $x$ . We then keep a map associating the location of the code of the translated alternatives with the code of the type annotation. When we face a stuck case discrimination during read back, we are confronted with an accumulator  $\langle \text{case } p \text{ of } (q, q') \rangle$ . We then use a modified version of rule  $\text{Case}_{\uparrow\emptyset}$ : besides normalizing  $p$  and the alternatives, it looks up the location of the code sequence stored for  $q$  in our map, builds a closure with a code pointer to this sequence and a closure data field containing a pointer to a fresh accumulator  $\langle y \rangle$ , and normalizes this closure with a stack containing  $q'$  to a normal form  $v$ . Then the normal form of the stuck case gets the type annotation as  $y$  return  $v$ .

For type annotations of abstractions and fixed points we proceed similarly. Since the bodies of these constructs expect values for variables in their context in the node environment, we re-use the node environment of the abstraction's closure for the type annotation, providing accumulator values for free variables via the stack this time.

### 8.3.2 Constructor Saturation

Regarding constructor saturation, the simplest solution is to  $\eta$ -expand unsaturated constructors according to their declared type. However, done without taking some care, this would violate an

Register	Usage
eax	heap pointer
ecx	stack pointer
edx	stack base
ebx	node pointer
esi	scratch A and tag return
edi	scratch B

Figure 8.5: X86 register usage

important goal of this thesis as described in chapter 1, namely not to introduce  $\eta$ -expansions during normalization.

Therefore, while we perform an  $\eta$ -expansion, for example from  $C$  to  $\lambda x. C x$ , we keep track of it by remembering a pointer to the compiled code for the generated abstraction in a further info table. When an abstraction is read back, and its code is found in this table, an  $\eta$ -reduction is performed. Using this table, we can guarantee that  $\eta$ -redices introduced by a user in his PTS expressions will be preserved, but no additional  $\eta$ -redices will be introduced.

#### 8.4 TRANSLATION FROM ISSTG TO x86 MACHINE CODE

As mentioned, our PTS implementation not only can run ISSTG instructions using an abstract machine, it also is able to generate and execute x86 machine code for physical processors. This section gives some details on the machine code generated from ISSTG instructions.

##### 8.4.1 *Memory Model*

During normalization we use one large chunk of main memory. It consists of 32 bit words aligned at word boundaries. The part with the lower addresses is used as heap space, while the upper part contains the stack.

We use six x86 registers as shown in the overview in figure 8.5. The heap pointer references the next free word of heap storage, and is incremented at each heap allocation. Closures on the heap consist of a code pointer for the compiled control expression, followed by a number of data pointers. Each closure contains at least one data pointer, using the dummy value zero if none is needed. Therefore, closures are large enough to be overwritten by the update routines in the run time system, since indirection closures need one data pointer to the closure they indirect to.

On the stack, we store pointers to arguments, update frames and continuations for case discriminations and fixed points. The

topmost element on the stack is referenced via the stack pointer, decremented at each pushing instruction. This layout allows to detect memory exhaustion by comparison of the heap and stack pointer registers.

The stack base register points to the topmost update frame on the stack. This separates arguments needed for an update from further arguments to a function.

As in *ISSTG*, the node register references the currently active closure, allowing access to the free variables' values.

The x86 processor architecture allows no direct memory-to-memory data transfers. So to push e.g. a value from the current closure to the stack, we cannot directly copy a pointer within main memory, but have to load it temporarily into a processor register. For this, we use two scratch registers. The first of these serves as a tag return register, too. We will describe in the next section, what this means.

#### 8.4.2 *Translation of Instructions*

The *ISSTG* instructions map quite nicely to the x86 architecture using our memory model. All instructions incrementing the heap pointer or decrementing the stack pointer have to perform a check whether this leads to a memory overflow, in addition to the behavior described below.

**ENTER:** Three x86 instructions load the topmost stack value into the node register, increment the stack pointer, and jump to the code of the entered closure with an indirect jump via the node register. This is possible because the first pointer in every closure is the appropriate code pointer.

**RETURNCON C:** In *ISSTG* this instruction has three possible consequences. Either an update has to be performed, or the right branch of a branch table has to be selected, or a code pointer to a fixed point's body has to be dereferenced. In our generated machine code, we collapse these three alternatives. In update frames, we place a code pointer to an update routine topmost on the stack. Furthermore, we implement branch tables with code scrutinizing a tag value, i.e. a small integer. A pointer to this code is pushed as a case continuation. So in any case, we find a suitable code pointer on the stack that we can simply jump to. Before we do this, we only have to load the tag value of the constructor into the tag return register, where it can be found by the code implementing the branch tables if necessary.

**ARGCHECK n:** The difference between the stack pointer and the stack base register is computed. When negative, not enough

arguments are present on the stack, and an update is performed.

**ALLOC  $n$ :** The current value of the heap pointer is pushed onto the stack, and the heap pointer is incremented by  $n$  words.

**BUILDCLS  $i$   $p$   $\alpha$ :** The pointer at index  $i$  on the stack points to a freshly allocated closure on the heap. It gets filled with the code pointer  $p$  followed by data pointers found on the stack or in the current closure as specified by  $\alpha$ .

**BUILDENV  $\alpha$ :** The data pointers specified by  $\alpha$  are pushed onto the stack.

**PUSHCONT  $p$ :** Code pointer  $p$  is placed onto the stack.

**UPDMARK:** An update frame is pushed, consisting of three pointers. As explained above, the first one is a code pointer to an update routine for constructors. Moreover, the current node pointer is placed into the update frame, since this references the closure that has to be overwritten when an update is performed. Finally, the stack base register is saved in the frame, and loaded with the now current stack pointer, thus referencing the topmost update frame on the stack. The stack base is later restored from its saved value during update.

**SLIDE  $n$   $m$ :** We have to keep  $n$  stack values that lie atop of  $m$  values we want to get rid of. This can be done by a simple loop moving the kept values by  $m$  positions, and a final x86 instruction incrementing the stack pointer.

**PAP:** The update routine of the run time system saves partially applied functions together with the supplied arguments in a closure. The machine code for PAP contains a loop to push all these pointers onto the stack. To determine how often the loop has to be executed, we save a last pointer with a numerical value of zero in closures for partial applications.

**ACCU:** Accumulator closures on the heap contain a code pointer  $p_{\text{accu}}$  pointing to the code sequence [ACCU]. This is the only place where this instruction is used. When an accumulator is entered, the ACCU instruction stays in control until a final state is reached: all ISSTG transition rules keep this instruction sequence unmodified in the code component of the configuration. Thus, this instruction builds a kind of a loop that unwinds the stack and collects the unreducible expression parts into new accumulators. We implemented this loop as a recursive Haskell function, and placed a code sequence at  $p_{\text{accu}}$  that ends the execution of the generated

x86 code to start the evaluation of this loop. We chose a representation for the closure data fields containing accumulators that makes a Haskell implementation as easy as possible. We use the so-called stable pointers of the Haskell foreign function interface. We can produce a stable pointer for any Haskell value, which then can be stored as a 32 bit word, and which allows to get the original value back. In our implementation, we represent accumulators as a usual Haskell data structure, storing their stable pointers after `paccu` in accumulator closures.

**PUSHNODE and POPNODE:** The current value of the node pointer register is saved to and restored from the stack, respectively.

## 8.5 DESIGN ALTERNATIVES

In an earlier project using our ideas of strongly normalizing compilation systems, we implemented Ulysses, a dependently typed programming language [23]. Most aspects of the PTS system presented in this chapter can be found in Ulysses, too. However, some design decisions were made differently.

In Ulysses we normalized types not to strong normal forms before adding them to type environments, but to weak head normal form only. The remaining redices are then normalized when needed. Such a strategy was used in former literature, too, see for example Coquand's description of a dependent type checking algorithm [9], and allows some failing type checks to be done faster: when e. g. an inductively defined type is expected, but a function type is given, the normalization of the function's range and domain does not have to be performed. We see no hindrances to implement this scheme for our PTS checker, too.

Ulysses has no designated fixed point syntax, but uses recursive defining equations instead. Such recursive equations were also used for the definition of recursive data types. Therefore, these definitions cannot be strongly normalized. To check recursive definitions for  $\beta$ -equivalence nevertheless, we implemented a pointer comparison algorithm for closure graphs based on Park's bisimilarity [29]. Closures containing equivalent pointers at corresponding positions are guaranteed to evaluate to equivalent values. However, it was not easily possible to see as a Ulysses user, whether this algorithm could be employed successfully on a given type checking problem: while it could detect several equivalent closure graphs, some equivalent pairs have to remain undetected in such an approach. Therefore, we judge our explicit fixed point implementation described here superior.

A third difference between Ulysses and our PTS implementation is the number of stacks: Ulysses uses two separate stacks, one

for function arguments and one for continuations and update frames. This eases the implementation of a garbage collector, but complicates matters for accumulators considerably. In `ISSTG` the `ACCU` instruction decides what kind of new accumulator has to be constructed based on the topmost stack element. With separate stacks, it is not distinguishable whether a function argument or a case continuation was pushed last, since they lie on different stacks. Therefore, the accumulators of *Ulysses* were annotated with type information: a function accumulator would grab an argument from the argument stack while an accumulator for an inductive data type would look for case continuations. The freshly constructed accumulator values then had to be given a appropriate new type annotation.



## PERFORMANCE EVALUATION

---

In the preceding chapters we have shown that it is indeed possible to use our compilation system during the checking of dependent types for the necessary strong normalizations. In this chapter we give some performance numbers as an evidence for the efficiency of our approach. However, our interest here is not a detailed analysis of performance characteristics, but only an overview about the order of magnitude of different implementations.

### 9.1 BENCHMARK SETTING

We measured the running times of three different implementations of reduction systems.

As our reference for existing checkers for dependent types we chose Coq. It is a very mature proof assistant with an active community. Two distinct reduction mechanisms are available to the user. The default mechanism is an interpreter, in constant development since the earliest versions of Coq. On user request, Coq is able to use compiled code for normalizations since version 8.1. This is an implementation based on the ideas of Leroy and Grégoire [19, 20] which were the inspiration for our work, too. Here, Coq terms are compiled to instructions for a virtual machine, which are then interpreted by a routine written in C. This routine is heavily hand optimized, to the extent that e.g. annotations assign specific registers for local C variables. The normalization is done strictly. In the figures of this chapter, we refer to Coq’s interpreter with the short hand “coq” and to the virtual machine implementation with “coq-vm”. The version used for our benchmarks is 8.2pl1.

To give a comparison with a widely used compiler for a lazy language, we measured run times of Haskell code compiled with the Glasgow Haskell compiler. This is an optimizing compiler that uses STG code as intermediate language, so the generated machine code is similar to ours, but is not suitable for strong normalization. We took our timing values using version 6.10.4 and refer to it with “ghc” in the following figures.

The third implementation under test is our own PTS checker. We measured running times using an interpreter of ISSTG instructions as well as x86 machine code generated from these instructions as detailed before. Both are prototype implementations lacking two features influencing the benchmarks: as mentioned before, a garbage collector is missing, that would add

an overhead to the running times. And we implemented no optimizations, thus losing the opportunity for faster execution. It is not clear whether the sum of these two effects is positive or negative. Moreover, the implementation of the `ISSTG` code interpreter was not tuned for performance, so we expect that it could benefit from a more careful implementation that pays more attention to efficiency. We refer to our implementations as “pts-vm” and “pts-x86”, respectively, in the figures.

All timings were done on an AMD Athlon processor, model 4850e, with a clock speed of 2.5 GHz. This is a dual core processor, but none of the benchmark candidates makes use of more than one core. The machine was equipped with four Gigabytes of main memory, running Ubuntu Linux 9.10.

Each implementation was used to run each benchmark ten times, the times reported in the following sections are the averages of these runs. This was done to cushion the effects of non-deterministic timings due to e.g. operating system activity. However, the standard deviations of our measurements show that these non-deterministic influences were negligible.

Since the measured timings differ drastically between implementations, we use logarithmic scales in this chapter for graphical representations.

This chapter gives only short descriptions of the problems used for benchmarking. For reproducibility the complete source codes can be found in appendix C.

## 9.2 PEANO NUMBERS

The first benchmark does some calculations using Peano style numbers, defined as inductive data types. The source code can be found in appendix C.1. We define addition and multiplication, a predecessor function and a higher-order function for iterated function application. We then build up the number 100 000 and compute the 100 000th predecessor of it. To force the evaluation in the type checkers of Coq and our `pts` implementation, we give a proof that the result is zero. To verify this, the type checkers have to normalize the expression for the iterated predecessor function until they reach zero as a normal form.

The average running times and their standard deviations in minutes, seconds and hundredths of a second are given in figure 9.1, using the short hand names of the systems as described in the preceding section. Only the times for the interpreter of Coq are given in hours, minutes and seconds. Moreover, the figure shows a visual representation of the measured times on a logarithmic scale. Our implementations are marked with a shaded bar.

The interpretative approach of Coq is almost prohibitively slow, taking over four hours. All other implementations are



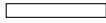

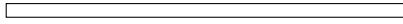
Name	Time	0:00.01 0:00.10 0:01.00 0:10.00 1:40.00 16:40.00 2:46:40 →	Deviation
ghc	0:00.11		0:00.01
pts-x86	0:00.11		0:00.01
coq-vm	0:00.36		0:00.04
pts-vm	1:01.62		0:00.18
coq	4:16:36		0:00:48

Figure 9.1: Arithmetic with Peano numbers

very fast in comparison. On the one hand, this is clearly a weakness of an interpreted implementation. But another design decision contributes to this long running time: the expressions are not normalized in one go, but the interpreter of Coq applies a heuristic to delay the unfolding of definitions, checking for  $\alpha$ -convertibility, and normalizing further if this check fails. Since this check cannot succeed before the normal form is reached in this example, this work is in vain.

Equally fast are the machine code implementations of the Haskell compiler and our PTS checker, taking about a tenth of a second each. This is unsurprising, since the generated code is similar, and the special provisions for strong normalizations are not needed in this benchmark.

The Coq checker using the compiled normalization is in the middle field, clearly outperforming the interpreter.

This test shows that it is indeed beneficial to switch to compiled code for normalization in dependent type checkers, and that our approach leads to a competitive candidate for fast implementations.

### 9.3 CHURCH NUMBERS

While the previous benchmark focused on the efficiency of inductive data structures, fixed points and case analyses, the next one focuses on function applications. We again use the iteration of the predecessor function as a test, but this time numbers are represented by Church numerals. The source code of the this benchmarks can be found in appendix C.2.

Since this involves the normalization of abstractions, something not possible with native Haskell, we implemented untyped normalization by evaluation for this benchmark. Since the simple implementations from the literature [2, 16] only deal with the very restricted syntax of the untyped  $\lambda$ -calculus, we mechanically generated an expression using abstractions as substitutes for lo-


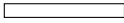
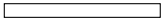

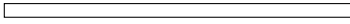
Name	Time	0:00.01 0:00.10 0:01.00 0:10.00 1:40.00 16:40.00 2:46:40	Deviation
pts-x86	0:00.35		0:00.04
coq-vm	0:00.74		0:00.06
ghc	0:02.67		0:00.02
pts-vm	1:34.15		0:00.14
coq	40:46.33		0:06.55

Figure 9.2: Arithmetic with Church numbers

cal definitions of the arithmetic functions:  $\text{let } x = e_1 \text{ in } e_2$  is expressed as  $(\lambda x. e_2) e_1$ .

The average running times and their standard deviation of the computation of the 1 000th predecessor of 1 000 are shown in figure 9.2.

The interpretative approach of Coq is again slowest, taking about forty minutes. All other implementations are much faster.

At least in this benchmark, approaches using accumulators and adapted compilers, namely the virtual machine of Coq and our x86 machine code generating PTS checker, beat untyped normalization by evaluation. We suppose that this is due to the administrative layers needed to discriminate between what is called syntactic and semantic terms in this approach.

So this test affirms that our approach is viable. For the normalization of church numerals, our implementation of accumulators was used to be provided as pseudo arguments. Apparently, there is no large performance penalty in computations employing accumulators, that could prevent their application.

#### 9.4 INTERPRETATION OF THE RESULTS

These benchmarks show that, while the timings of the interpreter for ISSTG instructions could benefit from an implementation taking more care for efficiency, the machine code generated from ISSTG instructions is very competitive. At least in the two scenarios presented, our approach is significantly faster than the type checker of Coq.

Of course, these two simple benchmarks cannot give an exhaustive comparison of the considered implementations. But the main focus in these tests were also the main ingredients of functional languages: with Peano numbers, we tested the efficiency of constructors, case analyses and fixed points, and with Church numbers, we looked at the performance of function call implementation. So in the core features of functional languages, our implementation shows very good first results.

Moreover, the `ISSTG` compilation had to sacrifice some clever implementation opportunities, as e. g. complex expressions in function or scrutinee position to allow a strong normalization. The Glasgow Haskell compiler can of course exploit them freely. Nevertheless, the impact is at least not large enough to show up in our benchmarks.

The heuristics employed by `Coq` seem to hinder efficient results in these two arithmetic scenarios. However, in certain situations, they prove beneficial as well. It is possible to include such heuristics in our approach, too, in the same way as they were included by Grégoire [19]: some defined variables can be replaced by accumulators to avoid their unfolding. After read back, an  $\alpha$ -conversion check can be performed, and if it fails, the original definition can be put back into place, performing further reductions. However, we did not implement any such heuristics.



## CONCLUSION

---

It's been a long journey, but finally we reached our goal. We have shown that it is possible to construct a compiler that produces code suitable for strong normalization. The resulting system is based on the STG machine, a standard approach for lazy functional languages. Since the STG machine in its original presentation only performs weak normalization, several problems had to be solved. Weak normalization does not proceed under  $\lambda$ -abstractions and case analyses, and thus avoids the problem of free variables in redices altogether. We adopted accumulators to have a run time representation of free variables and unreducible head values to solve this problem. Moreover, the results of a weak normalization may contain further redices. Therefore, we adapted a read back relation that extracts such redices and normalizes them.

We obtained the compilation system by a derivation involving several steps. We defined a high level semantics  $s_4$  that is easy to understand, and transformed it via  $s_{STG1}$  and  $s_{STG2}$  to  $is_{STG}$ , a small-step operational semantics using an instruction set for an abstract machine resembling actual processor hardware. This derivation allowed us to judge the correctness of our approach.

That our approach is not only correct, but indeed usable for dependent type checking, has been exemplified by our implementation of a checker for pure type systems. At this we could show that compilation does not have to stop at abstract machine code, but that code for physical processors may be used as well. A first glance at some performance figures shows that our normalization system not only fulfills its intended purpose, but it does so with quite good efficiency.

To our knowledge, we presented the first system using compiled code for lazy strong normalization used in a dependent type checker.

### 10.1 FUTURE WORK

The foundation of a compiling normalization system has been set. Now the construction of a whole infrastructure can begin. We excluded a module system and a garbage collector from our considerations. Many features of modern functional languages were ignored: primitive values that provide efficient implementations of numerical types, type classes that allow more flexible

definitions, and co-inductive definitions together with co-fixed points, to name just a few.

Moreover, we were content with a very simple machine code generation back end. There is a plethora of published optimizations for STG code. This extensive literature has to be screened for optimizations applicable for our strongly normalizing system to further enhance performance. Another approach to optimizations can be found in the work of Brady [8], who uses information from dependent types to produce better code. Our system regards types merely as annotations, without considering them for code generation decisions.

So it is not enough to be satisfied with the performance of the machine code we generated, but it is time now to further connect the research of dependent type checking with the results of the field of compiler construction and programming languages.

**Part IV**  
**APPENDIX**



EQUIVALENCE OF  $S_4$  AND  $SSTG_1$ 

In this appendix, we give the detailed proofs of the equivalence of  $S_4$  and  $SSTG_1$ .

## A.1 COMPLETENESS OF WEAK EVALUATION

We show that  $SSTG_1$  is complete with respect to  $S_4$ , i.e. that whenever some expression  $e$  is weakly evaluated to some  $WHNF$  located on the final heap at some address  $p$ , then  $e$  can be weakly normalized using  $SSTG_1$  to an equivalent  $WHNF$ , possibly located at a different location on the heap in the case of  $\lambda$ -abstractions. For easier reference, we repeat lemma 4.3 (page 51) here.

**LEMMA (Completeness of weak evaluation)** *Whenever  $\Gamma : e \downarrow \Delta : p$  holds, then for any  $\Gamma'$  with  $\Gamma \sim \Gamma'$  there is a heap  $\Delta'$  with  $\Delta \sim \Delta'$ , a pointer  $p'$  and a sequence of pointers  $\mathbf{q}$  such that for any stack  $S$  semantics  $SSTG_1$  allows the reduction  $\Gamma' : e : S \rightarrow^* \Delta' : p' : \mathbf{q} \# S$  with*

$$\begin{aligned} & \exists \mathbf{x}, \mathbf{y}, e' \quad . \quad \Delta[p \mapsto \lambda \mathbf{y} . e'[\mathbf{q}/\mathbf{x}]] \wedge \Delta'[p' \mapsto \lambda \mathbf{x} \mathbf{y} . e'] \\ \vee & \exists \mathbf{C}, \mathbf{p}'' \quad . \quad \Delta[p \mapsto \mathbf{C} \mathbf{p}''] \wedge p = p' \wedge \mathbf{q} = \diamond \\ \vee & \exists \mathbf{k} \quad . \quad \Delta[p \mapsto \mathbf{k}] \wedge p = p' \wedge \mathbf{q} = \diamond. \end{aligned}$$

**PROOF** The proof proceeds by induction on the size of the proof of  $\Gamma : e \downarrow \Delta : p$ , i.e. on the number of inference rules used. A case differentiation on the rule used at the root of the derivation tree gives the following cases.

**Lam:** We have  $\Gamma : p \downarrow \Gamma : p$  with  $\Gamma[p \mapsto \lambda \mathbf{x} . e]$ . Because the judgment is well-formed, we have  $p \in P_1$ , and thus also  $\Gamma'[p \mapsto \lambda \mathbf{x} . e]$  because  $\Gamma \sim \Gamma'$ . Then,  $\Gamma' : p : S \rightarrow^* \Gamma' : p : S$  in zero steps.

**Cons:** We have  $\Gamma : p \downarrow \Gamma : p$  with  $\Gamma[p \mapsto \mathbf{C} \mathbf{p}']$  and thus also  $\Gamma'[p \mapsto \mathbf{C} \mathbf{p}']$  because  $\Gamma \sim \Gamma'$ . Again,  $\Gamma' : p : S \rightarrow^* \Gamma' : p : S$ .

**Var:** We have  $\Gamma \cup [p \mapsto e] : p \downarrow \Delta \cup [p \mapsto \mathbf{q}] : \mathbf{q}$ . Rule  $\text{var}^{(1)}$  gives  $\Gamma' \cup [p \mapsto e] : p : S \rightarrow \Gamma' : e : \#p : S$ . From the induction hypothesis of the premise  $\Gamma : e \downarrow \Delta : \mathbf{q}$  we have  $\Gamma' : e : \#p : S \rightarrow^* \Delta' : \mathbf{q}' : \mathbf{r} \# \#p : S$  with one of the following three cases.

1.  $\Delta[\mathbf{q} \mapsto \lambda \mathbf{y} . e[\mathbf{r}/\mathbf{x}]] \wedge \Delta'[\mathbf{q}' \mapsto \lambda \mathbf{x} \mathbf{y} . e]$ : then we have  $\Delta' : \mathbf{q}' : \mathbf{r} \# \#p : S \rightarrow \Delta' \cup [p \mapsto \mathbf{q}' \mathbf{r}] : \mathbf{q}' : \mathbf{r} \# S$  by rule  $\text{var}^{(2)}$ .

2.  $\Delta'[q' \mapsto C p'] \wedge q' = q \wedge r = \diamond$ : then the reduction  $\Delta' : q : \#p : S \rightarrow \Delta' \cup [p \mapsto \hookrightarrow q] : q : S$  is possible by rule  $\text{var}^{(3)}$ .
3.  $\Delta'[q' \mapsto k] \wedge q' = q \wedge r = \diamond$ : then  $\Delta' : q : \#p : S \rightarrow \Delta' \cup [p \mapsto \hookrightarrow q] : q : S$  by rule  $\text{var}^{(4)}$ .

Ind: We have  $\Gamma : p \downarrow \Gamma : q$  with  $\Gamma[p \mapsto \hookrightarrow q]$ . From  $\Gamma \sim \Gamma'$  we get three cases.

1.  $\Gamma[q \mapsto \lambda y. e[r/x]] \wedge \Gamma'[p \mapsto q' r] \wedge \Gamma'[q' \mapsto \lambda x y. e]$ : then  $\Gamma' : p : S \rightarrow \Gamma' : q' r : S \rightarrow \Gamma' : q' : r \# S$  by rules  $\text{var}^{(5)}$  and  $\text{app}^{(1)}$ .
2.  $\Gamma[q \mapsto C p'] \wedge \Gamma'[p \mapsto \hookrightarrow q] \wedge \Gamma'[q \mapsto C p']$ : then we get  $\Gamma' : p : S \rightarrow \Gamma' : q : S$  from rule ind.
3.  $\Gamma[q \mapsto k] \wedge \Gamma'[p \mapsto \hookrightarrow q] \wedge \Gamma'[q \mapsto k]$ : then also  $\Gamma' : p : S \rightarrow \Gamma' : q : S$  by rule ind.

App<sup>(1)</sup>: We have  $\Gamma : e p \downarrow \Delta \cup [r \mapsto \lambda y. e'[p/x]] : r$ . Rule  $\text{app}^{(1)}$  gives  $\Gamma' : e p : S \rightarrow \Gamma' : e : p \# S$ . From the induction hypothesis for the premise  $\Gamma : e \downarrow \Delta : q$  we get the reduction sequence  $\Gamma' : e : p \# S \rightarrow^* \Delta' : r' : p' \# p \# S$  with  $\Delta'[r' \mapsto \lambda x' x y. e'']$  and  $e''[p'/x'] = e'$ . Since  $\Delta \sim \Delta'$  and  $r \in P_2$  we have  $\Delta \cup [r \mapsto \lambda y. e'[p/x]] \sim \Delta'$ .

App<sup>(2)</sup>: We have  $\Gamma : e p p' \downarrow \Theta : r$ . From rule  $\text{app}^{(1)}$  we have  $\Gamma' : e p p' : S \rightarrow \Gamma' : e : p \# p' \# S$ . The induction hypothesis for  $\Gamma : e \downarrow \Delta[q \mapsto \lambda x. e'] : q$  gives the reduction sequence  $\Gamma' : e : p \# p' \# S \rightarrow^* \Delta' : q' : p'' \# p \# p' \# S$  with  $\Delta'[q' \mapsto \lambda x' x. e'']$  and  $e''[p''/x'] = e'$ . We next get  $\Delta' : q' : p'' \# p \# p' \# S \rightarrow \Delta' : e''[p''/x'] [p/x] : p' \# S$  from rule  $\text{app}^{(2)}$ . The induction hypothesis for the second premise  $\Delta : e'[p/x] \downarrow \Theta : r$  gives a further SSTG1 sequence  $\Delta' : e''[p''/x'] [p/x] p' : S \rightarrow^* \Theta' : r' : r'' \# S$ . The first step in this reduction sequence can only be performed by rule  $\text{app}^{(1)}$ , and we obtain by stripping this first step the final reduction  $\Delta' : e''[p''/x'] [p/x] : p' \# S \rightarrow^* \Theta' : r' : r'' \# S$ .

Let: We have  $\Gamma : \text{let } x = \text{lf in } e \downarrow \Delta : q$ . Rule  $\text{let}$  gives us  $\Gamma' : \text{let } x = \text{lf in } e : S \rightarrow \Gamma' \cup [p \mapsto \text{lf}[p/x]] : e[p/x] : S$ . The induction hypothesis for  $\Gamma \cup [p \mapsto \text{lf}[p/x]] : e[p/x] \downarrow \Delta : q$  then gives  $\Gamma' \cup [p \mapsto \text{lf}[p/x]] : e[p/x] : S \rightarrow^* \Delta' : q' : r \# S$  as needed.

Case<sup>(1)</sup>: We have  $\Gamma : \text{case } e \text{ of } Cx \rightarrow e \downarrow \Theta : q$ . Rule  $\text{case}^{(1)}$  gives  $\Gamma' : \text{case } e \text{ of } Cx \rightarrow e : S \rightarrow \Gamma' : e : Cx \rightarrow e : S$ . Moreover, the induction hypothesis for the first premise  $\Gamma : e \downarrow \Delta[p \mapsto C_i p'] : p$  gives us  $\Gamma' : e : Cx \rightarrow e : S \rightarrow^* \Delta' : p : Cx \rightarrow e : S$  with  $\Delta'[p \mapsto C_i p']$ . Then by rule  $\text{case}^{(2)}$   $\Delta' : p : Cx \rightarrow e : S \rightarrow \Delta' : e_i[p'/x_i] : S$ . The

induction hypothesis for  $\Delta : e_i[p'/x_i] \downarrow \Theta : q$  finally gives  $\Delta' : e_i[p'/x_i] : S \rightarrow^* \Theta' : q' : r \# S$ .

Accu: We have  $\Gamma : p \downarrow \Gamma : p$  with  $\Gamma[p \mapsto k]$ . Since  $\Gamma \sim \Gamma'$  also  $\Gamma'[p \mapsto k]$  holds and  $\Gamma' : p : S \rightarrow^* \Gamma' : p : S$  in zero steps.

App<sup>(3)</sup>: We have  $\Gamma : e p p' \downarrow \Theta : s$ . From rule app<sup>(1)</sup> we obtain  $\Gamma' : e p p' : S \rightarrow \Gamma' : e : p : p' \# S$ . The first premise of App<sup>(3)</sup>  $\Gamma : e \downarrow \Delta[q \mapsto k] : q$  gives with the induction hypothesis  $\Gamma' : e : p : p' \# S \rightarrow^* \Delta' : q : p : p' \# S$  with  $\Delta'[q \mapsto k]$ . Then we have a next SSTG1 reduction step  $\Delta' : q : p : p' \# S \rightarrow \Delta' \cup [r \mapsto \langle q p \rangle] : r : p' \# S$  by rule app<sup>(3)</sup>. From premise  $\Delta \cup [r \mapsto \langle q p \rangle] : r p' \downarrow \Theta : S$  we get with the induction hypothesis a further reduction sequence  $\Delta' \cup [r \mapsto \langle q p \rangle] : r p' : S \rightarrow^* \Theta' : s' : p'' \# S$ . Since the first step in this reduction sequence has to be an application of rule app<sup>(1)</sup>, we can strip this step and get the remaining sequence  $\Delta' \cup [r \mapsto \langle q p \rangle] : r : p' \# S \rightarrow^* \Theta' : s' : p'' \# S$ .

Case<sup>(2)</sup>: In this final case of our discrimination, we consider  $\Gamma : \text{case } e \text{ of } \mathbf{alt} \downarrow \Delta \cup [q \mapsto \langle \text{case } p \text{ of } \mathbf{alt} \rangle] : q$ . Rule case<sup>(1)</sup> gives  $\Gamma' : \text{case } e \text{ of } \mathbf{alt} : S \rightarrow \Gamma' : e : \mathbf{alt} : S$ . The induction hypothesis for the premise  $\Gamma : e \downarrow \Delta[p \mapsto k] : p$  gives  $\Gamma' : e : \mathbf{alt} : S \rightarrow^* \Delta' : p : \mathbf{alt} : S$  with  $\Delta'[p \mapsto k]$ . Thus  $\Delta' : p : \mathbf{alt} : S \rightarrow \Delta' \cup [q \mapsto \langle \text{case } p \text{ of } \mathbf{alt} \rangle] : q : S$  by rule case<sup>(3)</sup>.  $\square$

## A.2 COMPLETENESS OF READ BACK AND NORMALIZATION

Before proving the completeness of read back and strong normalization, we need to state a helping lemma about the read back of  $\lambda$ -abstractions.

LEMMA A.1 *Whenever  $\Gamma[p \mapsto \lambda x. e]$  and  $\Gamma : p \uparrow^{s_4} \Delta : v$ , then there are  $\Gamma', q, y$  and  $v'$  such that  $v = \lambda y. v'$  and*

$$\Gamma' \cup [q \mapsto \langle y \rangle] : e[q/x] \downarrow^{s_4} \Delta : v',$$

*and for all  $\Gamma'', \Gamma \sim \Gamma'' \Leftrightarrow \Gamma' \sim \Gamma''$ . At this, the normalization in the conclusion can be done in strictly fewer steps than the read back in the premise.*

PROOF (Sketch) By induction on the length of  $x$ . The heaps  $\Gamma$  and  $\Gamma'$  differ only by allocations of pointers from  $P_2$  for partial applications when not enough accumulators are present as pseudo-arguments.  $\square$

Next we show the completeness of read back and strong normalization of SSTG1 with respect to  $s_4$ , that is, we show that every strong normal form obtained using  $s_4$  can be obtained using

SSTG1 as well. We repeat lemma 4.4 (page 51) here, before giving the detailed proof.

LEMMA (Completeness of read back and normalization)

The following three propositions hold:

- Whenever  $\Gamma : p \uparrow^{S4} \Delta : v$ , then for any  $\Gamma'$  with  $\Gamma \sim \Gamma'$  there is a heap  $\Delta'$  with  $\Delta \sim \Delta'$  such that

$$\begin{aligned} \exists \mathbf{y}, e \quad & \Gamma[p \mapsto \lambda \mathbf{y}. e] \wedge \\ & \forall \mathbf{q}, \mathbf{q}', \mathbf{x}, e'. \Gamma'[q \mapsto \lambda \mathbf{x} \mathbf{y}. e'] \\ & \wedge e = e'[\mathbf{q}'/\mathbf{x}] \\ & \Rightarrow \Gamma' : \mathbf{q} : \mathbf{q}' \uparrow^{\text{SSTG1}} \Delta' : v \\ \vee \exists \mathbf{C}, \mathbf{p}' \quad & \Gamma[p \mapsto \mathbf{C} \mathbf{p}'] \wedge \Gamma' : \mathbf{p} : \diamond \uparrow^{\text{SSTG1}} \Delta' : v \\ \vee \exists \mathbf{k} \quad & \Gamma[p \mapsto \mathbf{k}] \wedge \Gamma' : \mathbf{p} : \diamond \uparrow^{\text{SSTG1}} \Delta' : v. \end{aligned}$$

- Whenever  $\Gamma : k \uparrow_{\langle \rangle}^{S4} \Delta : h$ , then for any  $\Gamma'$  with  $\Gamma \sim \Gamma'$  there is a heap  $\Delta'$  with  $\Delta \sim \Delta'$  such that  $\Gamma' : k \uparrow_{\langle \rangle}^{\text{SSTG1}} \Delta' : h$ .
- Whenever  $\Gamma : e \downarrow^{S4} \Delta : v$ , then for any  $\Gamma'$  with  $\Gamma \sim \Gamma'$  there is a heap  $\Delta'$  with  $\Delta \sim \Delta'$  such that  $\Gamma' : e : \diamond \downarrow^{\text{SSTG1}} \Delta' : v$ .

PROOF The proof proceeds by simultaneous induction on the number of inference rules used for the proofs of the relations  $\uparrow^{S4}$ ,  $\downarrow^{S4}$  and  $\uparrow_{\langle \rangle}^{S4}$ . We focus on  $\uparrow^{S4}$  first. A case differentiation on the rule at the root of the proof tree gives following cases.

$\text{Lam}_{\uparrow^{S4}}$ : We have  $\Gamma : p \uparrow^{S4} \Delta : \lambda \mathbf{y}. v$  with  $\Gamma[p \mapsto \lambda \mathbf{x}. e]$ . Assume  $\Gamma'$  with  $\Gamma \sim \Gamma'$ . From lemma A.1 we get  $\Gamma'', \mathbf{r}, \mathbf{y}'$  and  $v'$  with  $\Gamma'' \cup [\mathbf{r} \mapsto \langle \mathbf{y}' \rangle] : e[\mathbf{r}/\mathbf{x}] \downarrow^{S4} \Delta : v'$  and  $\Gamma'' \sim \Gamma'$ . We now get  $\Gamma' \cup [\mathbf{r} \mapsto \langle \mathbf{y}' \rangle] : e[\mathbf{r}/\mathbf{x}] : \diamond \downarrow^{\text{SSTG1}} \Delta' : v'$  from the induction hypothesis. This can only hold because of rule Norm, thus we also have  $\Gamma' \cup [\mathbf{r} \mapsto \langle \mathbf{y}' \rangle] : e[\mathbf{r}/\mathbf{x}] : \diamond \rightarrow^* \Theta : \mathbf{r}' : \mathbf{r}''$  with  $\mathcal{W}(\Theta : \mathbf{r}' : \mathbf{r}'')$  and  $\Theta : \mathbf{r}' : \mathbf{r}'' \uparrow^{\text{SSTG1}} \Delta' : v'$ .

Next assume  $q, q', z$  and  $e''$  with  $\Gamma'[q \mapsto \lambda z \mathbf{x}. e'']$  and  $e = e''[q'/z]$ . Then we get  $\Gamma' \cup [\mathbf{r} \mapsto \langle \mathbf{y}' \rangle] : q : q' \# \mathbf{r} \rightarrow \Gamma' \cup [\mathbf{r} \mapsto \langle \mathbf{y}' \rangle] : e''[q'/z][\mathbf{r}/\mathbf{x}] : \diamond$  by rule app<sup>(2)</sup>. Thus we have  $\Gamma' \cup [\mathbf{r} \mapsto \langle \mathbf{y}' \rangle] : q : q' \# \mathbf{r} \downarrow^{\text{SSTG1}} \Delta' : v'$ .

Finally, applying rules  $\text{Lam}_{\uparrow}$  and Norm  $|\mathbf{x}|$  times, we get  $\Gamma' : q : q' \downarrow^{\text{SSTG1}} \Delta' : \lambda \mathbf{y}'. v'$  where  $\lambda \mathbf{y}'. v' = \lambda \mathbf{y}. v$  due to the results of lemma A.1.

$\text{Cons}_{\uparrow^{S4}}$ : We have  $\Gamma_1 : p \uparrow^{S4} \Gamma_{|q|+1} : C v$  with  $\Gamma_1[p \mapsto C q]$ . Applying the induction hypothesis on all premises, and then rule  $\text{Cons}_{\uparrow^{\text{SSTG1}}}$  gives  $\Gamma'_1 : p : \diamond \uparrow^{\text{SSTG1}} \Gamma'_{|q|+1} : C v$ .

$\text{Accu}_{\uparrow^{S4}}$ : We have  $\Gamma : p \uparrow^{S4} \Delta : v$  with  $\Gamma[p \mapsto k]$ . Applying the induction hypothesis on the premises  $\Gamma : k \uparrow_{\langle \rangle}^{S4} \Delta : v$ , and then rule  $\text{Accu}_{\uparrow^{\text{SSTG1}}}$  gives  $\Gamma' : p : \diamond \uparrow^{\text{SSTG1}} \Delta' : v$ .

Focusing on  $\uparrow_{\diamond}^{s4}$  next, we have to consider following three cases.

$\text{Var}_{\uparrow_{\diamond}^{s4}}$ : trivial.

$\text{App}_{\uparrow_{\diamond}^{s4}}$ : A simple application of the induction hypotheses gives the desired result.

$\text{Case}_{\uparrow_{\diamond}^{s4}}$ : The read back of a stuck case discrimination gives  $\Gamma[p \mapsto k] : \langle \text{case } p \text{ of } \mathbf{alt} \rangle \uparrow_{\diamond}^{s4} \Delta_{|\mathbf{alt}|+1} : \text{case } h \text{ of } \mathbf{C} \mathbf{y} \rightarrow v$ . The first premise  $\Gamma : k \uparrow_{\diamond}^{s4} \Delta_1 : h$  together with the induction hypothesis gives  $\Gamma' : k \uparrow_{\diamond}^{\text{SSTG1}} \Delta'_1 : h$ .

Then we apply the induction hypothesis on the premises for the normalization of the case branches, thus obtaining  $\Delta'_i \cup \Theta'_i : \text{case } p \text{ of } \mathbf{alt} : \diamond \downarrow^{\text{SSTG1}} \Delta'_{i+1} : v_i$ . Inversion of Norm gives  $\Delta'_i \cup \Theta'_i : \text{case } p \text{ of } \mathbf{alt} : \diamond \rightarrow^* \Delta'' : r : r'$ . The first step of this reduction sequence must be an application of rule  $\text{case}^{(1)}$ , if we drop it, we get  $\Delta'_i \cup \Theta'_i : p : \mathbf{alt} : \diamond \rightarrow^* \Delta'' : r : r'$  and from this  $\Delta'_i \cup \Theta'_i : p : \mathbf{alt} : \diamond \downarrow^{\text{SSTG1}} \Delta'_{i+1} : v_i$ .

Thus  $\Gamma' : \langle \text{case } p \text{ of } \mathbf{alt} \rangle \uparrow_{\diamond}^{\text{SSTG1}} \Delta'_{|\mathbf{alt}|+1} : \text{case } h \text{ of } \mathbf{C} \mathbf{y} \rightarrow v$  by rule  $\text{Case}_{\uparrow_{\diamond}^{\text{SSTG1}}}$ .

Finally, we look at normalization  $\uparrow_{\diamond}^{s4}$ , where only one rule has to be considered.

Norm: We have  $\Gamma : e \downarrow^{s4} \Theta : v$  and the premises  $\Gamma : e \downarrow^{s4} \Delta : p$  and  $\Delta : p \uparrow^{s4} \Theta : v$ . From the first premise and lemma 4.3 we get  $\Gamma' : e : \diamond \rightarrow^* \Delta' : p' : q$  and one of three cases.

- $\Delta[p \mapsto \lambda \mathbf{y} . e[q/x]]$  and  $\Delta'[p' \mapsto \lambda . x \mathbf{y} . e]$ , or
- $\Delta[p \mapsto \mathbf{C} p'']$  and  $p = p'$  and  $q = \diamond$ , or
- $\Delta[p \mapsto k]$  and  $p = p'$  and  $q = \diamond$ .

In each of these cases,  $\mathcal{W}(\Delta' : p' : q)$  holds, and we can conclude  $\Delta' : p' : q \uparrow^{s4} \Theta' : v$  from the induction hypothesis and finally  $\Gamma' : e : \diamond \downarrow^{\text{SSTG1}} \Theta' : v$  by rule Norm for SSTG1.  $\square$

### A.3 SOUNDNESS OF WEAK EVALUATION

In this section we give the proof of soundness of weak evaluation using SSTG1 with respect to  $s4$ , i.e. we show that for each WHNF reached by weak evaluation with SSTG1, an equivalent WHNF can be obtained using  $s4$ . We repeat lemma 4.6 (page 52) here for easier reference.

**LEMMA (Soundness of weak evaluation)** *For any balanced reduction sequence of SSTG1  $(\Gamma, e, S) \rightarrow^* (\Delta, p, q \# S)$  with  $\mathcal{W}(\Delta, p, q)$ , for*

all heaps  $\Gamma'$  with  $\Gamma' \sim \Gamma$  there is a heap  $\Delta'$  with  $\Delta' \sim \Delta$  and a pointer  $p'$  such that s4 allows the deduction of  $\Gamma' : e \downarrow \Delta' : p'$  with

$$\begin{aligned} & \exists \mathbf{x}, \mathbf{y}, e' . \Delta[p \mapsto \lambda \mathbf{x} \mathbf{y} . e'] \wedge \Delta'[p' \mapsto \lambda \mathbf{y} . e'[q/\mathbf{x}]] \\ \vee & \exists C, p'' . \Delta[p \mapsto C p''] \wedge p = p' \\ \vee & \exists k . \Delta[p \mapsto k] \wedge p = p'. \end{aligned}$$

**PROOF** The proof proceeds by induction on the length of the reduction sequence of SSTG1, assuming the induction hypothesis for all shorter balanced sequences.

We first consider the case of zero step reductions, where we have  $(\Gamma, p, S) \rightarrow^* (\Gamma, p, S)$ . Here,  $\mathcal{W}(\Gamma, p, \diamond)$  gives three subcases.

$\Gamma[p \mapsto \lambda \mathbf{x} . e]$ : We have  $\Gamma'[p \mapsto \lambda \mathbf{x} . e]$  from  $\Gamma' \sim \Gamma$ , thus we get  $\Gamma' : p \downarrow \Gamma' : p$  from rule Lam.

$\Gamma[p \mapsto C p'']$ : same argument using rule Cons.

$\Gamma[p \mapsto k]$ : same argument using rule Accu.

For reduction sequences that have at least one step, the first step must use one of the rules  $\text{var}^{(1)}$ ,  $\text{ind}^{(1)}$ ,  $\text{ind}^{(2)}$ ,  $\text{app}^{(1)}$ ,  $\text{let}$  or  $\text{case}^{(1)}$ . All other rules remove elements from the stack, breaking the balancing of the reduction sequence.

$\text{var}^{(1)}$ : This rule puts an update frame  $\#r$  onto the stack in the first step. Since this update frame is not present on the final stack, there must be an application of one of the rules  $\text{var}^{(2)}$ ,  $\text{var}^{(3)}$ ,  $\text{var}^{(4)}$  in the remaining sequence, because only these rules remove update frames from the stack. The part of the reduction sequence after pushing and before popping the update frame is balanced, with a common stack suffix of  $\#r :: S$ . We consider the three cases of the possible rules responsible for popping the update frame.

$\text{var}^{(2)}$ : The reduction sequence starts with the following steps, ending with rule  $\text{var}^{(2)}$ :

$$\begin{aligned} & ( \Gamma \cup [r \mapsto e'], \quad r, \quad S ) \\ \rightarrow & ( \Gamma, \quad e', \quad \#r :: S ) \\ \rightarrow^* & ( \Theta[r' \mapsto \lambda \mathbf{x} \mathbf{y} . e''], \quad r', \quad r'' \# \#r :: S ) \\ \rightarrow & ( \Theta \cup [r \mapsto r' r''], \quad r', \quad r'' \# S ). \end{aligned}$$

No further steps may follow, since any applicable rule in the last configuration above would remove elements from  $S$ , making the sequence unbalanced.

The induction hypothesis for the part of the sequence where  $\#r$  is present gives  $\Gamma' : e' \downarrow \Theta' : r'''$  with  $\Theta'[r''' \mapsto \lambda \mathbf{y} . e'[r''/\mathbf{x}]]$ . Using rule Var we finally get  $\Gamma' \cup [r \mapsto e'] : r \downarrow \Theta' \cup [r \mapsto r'' r'''] : r'''$ .

$\text{var}^{(3)}$ : The reduction sequence starts with the following steps, ending with rule  $\text{var}^{(3)}$ :

$$\begin{aligned} & ( \Gamma \cup [r \mapsto e'], \quad r, \quad S \quad ) \\ \rightarrow & ( \Gamma, \quad e', \quad \#r :: S \quad ) \\ \rightarrow^* & ( \Theta[r' \mapsto C r''], \quad r', \quad \#r :: S \quad ) \\ \rightarrow & ( \Theta \cup [r \mapsto \mathbf{q} r'], \quad r', \quad S \quad ). \end{aligned}$$

No further steps may follow, since any applicable rule in the last configuration above would remove elements from  $S$ , making the sequence unbalanced.

The induction hypothesis for the part of the sequence where  $\#r$  is present gives  $\Gamma' : e' \downarrow \Theta' : r'$ , and from rule  $\text{Var}$  we get  $\Gamma' \cup [r \mapsto e'] : r \downarrow \Theta' \cup [r \mapsto \mathbf{q} r'] : r'$ .

$\text{var}^{(4)}$ : same argument as in the case of  $\text{var}^{(3)}$ , with an accumulator in place of the constructor expression.

$\text{ind}^{(1)}$ : The reduction sequence starts with

$$\begin{aligned} & ( \Gamma[p \mapsto \mathbf{q} r], \quad p, \quad S \quad ) \\ \rightarrow & ( \Gamma, \quad r, \quad S \quad ). \end{aligned}$$

From  $\Gamma' \sim \Gamma$  we get two cases.

$\Gamma[r \mapsto C p'']$  and  $\Gamma'[r \mapsto C p'']$ : No further reductions are possible without making the sequence unbalanced. We have  $\Gamma' : p \downarrow \Gamma' : r$  by rule  $\text{Ind}$ .

$\Gamma[r \mapsto k]$  and  $\Gamma'[r \mapsto k]$ : same argument as in the last case, with an accumulator in place of the constructor expression.

$\text{ind}^{(2)}$ : The reduction sequence starts with

$$\begin{aligned} & ( \Gamma[p \mapsto e'], \quad p, \quad S \quad ) \\ \rightarrow & ( \Gamma, \quad e', \quad S \quad ). \end{aligned}$$

From  $\Gamma' \sim \Gamma$  we get  $e' = r r'$  and  $\Gamma[r \mapsto \lambda \mathbf{y} \mathbf{x}. e'']$ , moreover  $\Gamma'[p \mapsto \mathbf{q} r'']$  and  $\Gamma'[r'' \mapsto \lambda \mathbf{x}. e''[r'/\mathbf{y}]]$ . Thus, the reduction sequence continues with

$$\begin{aligned} & ( \Gamma, \quad r r', \quad S \quad ) \\ \rightarrow & ( \Gamma, \quad r, \quad r' \# S \quad ). \end{aligned}$$

No further reductions are possible without making the sequence unbalanced. We have  $\Gamma' : p \downarrow \Gamma' : r''$  by rule  $\text{Ind}$ .

$\text{app}^{(1)}$ : The reduction sequence has the shape

$$\begin{aligned} & ( \Gamma, \quad e' r, \quad S \quad ) \\ \rightarrow & ( \Gamma, \quad e', \quad r \# S \quad ) \\ \rightarrow^* & ( \Delta, \quad p, \quad \mathbf{q} \# S \quad ). \end{aligned}$$

The function arguments  $\mathbf{r}$  might stay on the stack during the whole rest of the reduction, or (parts of)  $\mathbf{r}$  could be removed from the stack before reaching the final configuration. The removal can be done in several chunks, taking the first arguments from the stack several steps before grabbing additional arguments. So we can consider  $\mathbf{r}$  composed of chunks  $\mathbf{r}^1 \# \mathbf{r}^2 \# \dots \# \mathbf{r}^n$  such that the last chunk stays on the stack and the other ones, if any, are removed by one application of rule  $\text{app}^{(2)}$  or  $\text{app}^{(3)}$  per chunk. No other rule is able to remove function arguments from the stack. So the reduction sequence after pushing  $\mathbf{r}$  has the shape

$$\begin{array}{l}
( \Gamma^1, \quad e^1, \quad \mathbf{r}^1 \# \mathbf{r}^1 \# \dots \# \mathbf{r}^n \# S \quad ) \\
\rightarrow^* ( \Theta^1, \quad p^1, \quad \mathbf{r}'^1 \# \mathbf{r}^1 \# \mathbf{r}^2 \# \dots \# \mathbf{r}^n \# S \quad ) \\
\rightarrow ( \Gamma^2, \quad e^2, \quad \mathbf{r}^2 \# \dots \# \mathbf{r}^n \# S \quad ) \\
\vdots \qquad \qquad \qquad \vdots \\
\rightarrow ( \Gamma^i, \quad e^i, \quad \mathbf{r}^i \# \mathbf{r}^{i+1} \# \dots \# \mathbf{r}^n \# S \quad ) \\
\rightarrow^* ( \Theta^i, \quad p^i, \quad \mathbf{r}'^i \# \mathbf{r}^i \# \mathbf{r}^{i+1} \# \dots \# \mathbf{r}^n \# S \quad ) \\
\rightarrow ( \Gamma^{i+1}, \quad e^{i+1}, \quad \mathbf{r}^{i+1} \# \dots \# \mathbf{r}^n \# S \quad ) \\
\vdots \qquad \qquad \qquad \vdots \\
\rightarrow^* ( \Theta^n, \quad p^n, \quad \mathbf{r}'^n \# \mathbf{r}^n \# S \quad )
\end{array}$$

where the first configuration is equal to  $(\Gamma, e', \mathbf{r} \# S)$  and the last configuration is equal to  $(\Delta, p, \mathbf{q} \# S)$ .

So in a first multi-step sequence  $e'$  is evaluated in a balanced reduction sequence during which some additional arguments  $\mathbf{r}'^1$  might get pushed onto the stack. In the next step, using  $\text{app}^{(2)}$  or  $\text{app}^{(3)}$ , the arguments  $\mathbf{r}'^1 \# \mathbf{r}^1$  are consumed. The evaluation of  $e^2$  might in turn push again some additional arguments, again consumed by  $\text{app}^{(2)}$  or  $\text{app}^{(3)}$  together with the next chunk, and so on.

We prove by induction on this structure that when

$$\begin{array}{l}
( \Gamma^i, \quad e^i, \quad \mathbf{r}^i \# \mathbf{r}^{i+1} \# \dots \# \mathbf{r}^n \# S \quad ) \\
\rightarrow^* ( \Theta^n, \quad p^n, \quad \mathbf{r}'^n \# \mathbf{r}^n \# S \quad )
\end{array}$$

we have for all  $\Gamma^i$  with  $\Gamma^i \sim \Gamma^i$  some  $\Theta'^n$  and  $p'^n$  such that

$$\Gamma^i : e^i \mathbf{r}^i \mathbf{r}^{i+1} \dots \mathbf{r}^n \downarrow \Theta'^n : p'^n$$

and the conclusion of the soundness lemma holds for  $\Theta'^n$  and  $p'^n$ . Note that from the result of this inner induction we directly get  $\Gamma : e' \mathbf{r} \downarrow \Delta' : p'$  as required.

In the base case, we consider the last part of above partitioning of the reduction sequence, i. e.

$$\begin{array}{l}
( \Gamma^n, \quad e^n, \quad \mathbf{r}^n \# S \quad ) \\
\rightarrow^* ( \Theta^n, \quad p^n, \quad \mathbf{r}'^n \# \mathbf{r}^n \# S \quad ).
\end{array}$$

This reduction sequence is balanced, thus the induction hypothesis of the soundness lemma applies and gives the desired result.

In the inductive step, the first part of the reduction sequence from  $\Gamma^i$  to  $\Theta^i$  is balanced. The stack suffix of all intermediate configurations is  $r^i \# \dots \# r^n \# S$ . Thus, the induction hypothesis of the soundness lemma gives  $\Gamma'^i : e^i \downarrow \Theta'^i : p'^i$ .

The last part of the reduction sequence from  $\Gamma^{i+1}$  to  $\Theta^n$  then gives  $\Gamma'^{i+1} : e^{i+1} r^{i+1} \dots r^n \downarrow \Theta'^n : p'^n$  by the inner induction hypothesis.

These two evaluations of  $s_4$  can be combined to a bigger deduction. We consider two cases according to the rule responsible for the removal of the arguments from the stack.

$\text{app}^{(2)}$ : We get  $\Gamma^{i+1} = \Theta^i$ , moreover  $\Theta^i[p^i \mapsto \lambda x y. e'^{i+1}]$  and  $e^{i+1} = e'^{i+1}[r'^i/x][r^i/y]$ . For the evaluation with  $s_4$  this implies that  $\Theta'^i[p'^i \mapsto \lambda y. e'^{i+1}[r^i/x]]$  holds. Thus, rule  $\text{App}^{(2)}$  is applicable and gives the desired deduction.

$\text{app}^{(3)}$ : We have  $\Theta^i[p^i \mapsto k^i]$ . Since  $r'^i \# r^i$  is removed completely from the stack, it contains only a single element  $r''^i$ . Thus,  $\Gamma^{i+1} = \Theta^i \cup [q^i \mapsto \langle p^i r''^i \rangle]$  holds such that rule  $\text{App}^{(3)}$  applies and gives the desired deduction.

**let**: The reduction sequence has the shape

$$\begin{aligned} & ( \Gamma, \quad \text{let } x = \text{lf in } e, \quad S \quad ) \\ \rightarrow & ( \Gamma \cup [r \mapsto \text{lf}[r/x]], \quad e[r/x], \quad S \quad ) \\ \rightarrow^* & ( \Delta, \quad p, \quad q \# S \quad ). \end{aligned}$$

We get  $\Gamma' \cup [r \mapsto \text{lf}[r/x]] : e[r/x] \downarrow \Delta' : p'$  from the induction hypothesis and then  $\Gamma' : \text{let } x = \text{lf in } e \downarrow \Delta' : p'$  from rule **Let**.

$\text{case}^{(1)}$ : The reduction sequence has the shape

$$\begin{aligned} & ( \Gamma, \quad \text{case } e \text{ of } \text{alt}, \quad S \quad ) \\ \rightarrow & ( \Gamma, \quad e, \quad \text{alt} : S \quad ) \\ \rightarrow^* & ( \Delta, \quad p, \quad q \# S \quad ). \end{aligned}$$

Since the alternative branches **alt** are not present on the final stack, there must be some intermediate step where they are popped. This can be done only by rule  $\text{case}^{(2)}$  or  $\text{case}^{(3)}$ .

case<sup>(2)</sup>: The last part of the reduction sequence has the shape

$$\begin{aligned}
& ( \Gamma, \quad e, \quad \mathbf{alt} :: S ) \\
\rightarrow^* & ( \Theta[r \mapsto C_i \mathbf{r}'], \quad r, \quad \mathbf{alt} :: S ) \\
\rightarrow & ( \Theta, \quad e'_i[r'/x_i], \quad S ) \\
\rightarrow^* & ( \Delta, \quad p, \quad q \# S )
\end{aligned}$$

where  $\mathbf{alt}_i = C_i x_i \rightarrow e'_i$ . The induction hypothesis gives  $\Gamma' : e \downarrow \Theta' : r$  and  $\Theta' : e'_i[r'/x_i] \downarrow \Delta' : p'$ , thus we get  $\Gamma' : \text{case } e \text{ of } \mathbf{alt} \downarrow \Delta' : p'$  by rule Case<sup>(1)</sup>.

case<sup>(3)</sup>: The last part of the reduction sequence has the shape

$$\begin{aligned}
& ( \Gamma, \quad e, \quad \mathbf{alt} :: S ) \\
\rightarrow^* & ( \Theta[r \mapsto k], \quad r, \quad \mathbf{alt} :: S ) \\
\rightarrow & ( \Theta \cup [r' \mapsto \langle \text{case } r \text{ of } \mathbf{alt} \rangle], \quad r', \quad S )
\end{aligned}$$

where the last configuration is equal to  $(\Delta, p, q \# S)$  since no further reductions are possible without shortening  $S$ . The induction hypothesis gives  $\Gamma' : e \downarrow \Theta' : r$ , so we can conclude  $\Gamma' : \text{case } e \text{ of } \mathbf{alt} \downarrow \Delta' : r'$  by rule Case<sup>(2)</sup>.  $\square$

#### A.4 SOUNDNESS OF READ BACK AND NORMALIZATION

Before we can prove the soundness of the read back and strong normalization relations for SSTG1, we need to state a helping lemma on the read back and normalization of  $\lambda$ -abstractions.

LEMMA A.2 *Whenever  $\Gamma[p \mapsto \lambda x y . e]$  and  $\Gamma : p : \mathbf{p}' \uparrow^{\text{SSTG1}} \Delta : v$ , then there are  $q, z$  and  $v'$  such that  $v = \lambda z . v'$  and*

$$\Gamma \cup [q \mapsto \langle z \rangle] : e[p'/x][q/y] : \Diamond \uparrow^{\text{SSTG1}} \Delta : v',$$

*At this, the normalization in the conclusion can be done in strictly fewer steps than the read back in the premise.*

PROOF (Sketch) The proof proceeds by induction on  $|y|$ , i.e. the number of missing arguments for the  $\lambda$ -abstraction.  $\square$

Now we can show the soundness of read back and strong normalization using SSTG1 with respect to S4, that is, that every strong normal form obtained using SSTG1 can be obtained with S4 as well. We state lemma 4.7 (page 53) again for easier reference.

LEMMA (Soundness of read back and normalization) *The following three propositions hold:*

- Whenever  $\Gamma : p : q' \uparrow^{\text{SSTG1}} \Delta : v$ , then for any  $\Gamma'$  with  $\Gamma' \sim \Gamma$  there is a heap  $\Delta'$  with  $\Delta' \sim \Delta$  such that

$$\begin{aligned} & \exists \mathbf{x}, \mathbf{y}, e \quad . \quad \Gamma[p \mapsto \lambda \mathbf{x} \mathbf{y} . e] \wedge \\ & \quad \quad \quad \forall q. \Gamma'[q \mapsto \lambda \mathbf{y} . e[q'/\mathbf{x}]] \Rightarrow \Gamma' : q \uparrow^{s4} \Delta' : v \\ \vee & \exists \mathbf{C}, \mathbf{p}' \quad . \quad \Gamma[p \mapsto \mathbf{C} \mathbf{p}'] \wedge \Gamma' : \mathbf{p} \uparrow^{s4} \Delta' : v \\ \vee & \exists \mathbf{k} \quad . \quad \Gamma[p \mapsto \mathbf{k}] \wedge \Gamma' : \mathbf{p} \uparrow^{s4} \Delta' : v. \end{aligned}$$

- Whenever  $\Gamma : k \uparrow_{\langle \rangle}^{\text{SSTG1}} \Delta : h$ , then for any  $\Gamma'$  with  $\Gamma' \sim \Gamma$  there is a heap  $\Delta'$  with  $\Delta' \sim \Delta$  such that  $\Gamma' : k \uparrow_{\langle \rangle}^{s4} \Delta' : h$ .
- Whenever  $\Gamma : e : \diamond \downarrow^{\text{SSTG1}} \Delta : v$ , then for any  $\Gamma'$  with  $\Gamma' \sim \Gamma$  there is a heap  $\Delta'$  with  $\Delta' \sim \Delta$  such that  $\Gamma' : e \downarrow^{s4} \Delta' : v$ .

PROOF We prove all three proposition by simultaneous induction on the number of inference rules used.

Starting with the read back relation  $\uparrow^{\text{SSTG1}}$ , we consider three cases for the possible deduction rules.

$\text{Lam}_{\uparrow^{\text{SSTG1}}}$ : We have  $\Gamma[p \mapsto \lambda \mathbf{x} \mathbf{y} . e] : p : q' \uparrow^{\text{SSTG1}} \Delta : \lambda z.v$ , and lemma A.2 gives  $\Gamma \cup [\mathbf{r} \mapsto \langle \mathbf{z}' \rangle] : e[q'/\mathbf{x}][\mathbf{r}/\mathbf{y}] : \diamond \downarrow^{\text{SSTG1}} \Delta : v'$  with  $\lambda z.v = \lambda z'.v'$ .

Now assume  $\Gamma'$  with  $\Gamma' \sim \Gamma$  and  $\Gamma'[q \mapsto \lambda \mathbf{y} . e[q'/\mathbf{x}]]$ . We define  $\Gamma''$  as

$$\begin{aligned} \Gamma' & \cup [\mathbf{r} \mapsto \langle \mathbf{z}' \rangle] \\ & \cup [\mathbf{r}'_1 \mapsto \lambda y_2 \dots y_n . e[q'/\mathbf{x}][\mathbf{r}_1/y_1]] \\ & \quad \vdots \\ & \cup [\mathbf{r}'_{n-1} \mapsto \lambda y_n . e[q'/\mathbf{x}][\mathbf{r}_1/y_1] \dots [\mathbf{r}_{n-1}/y_{n-1}]] \end{aligned}$$

where  $n = |\mathbf{y}|$  and  $\mathbf{r}'$  fresh from segment  $P_2$ . That is,  $\Gamma''$  binds the results of partial applications at addresses  $\mathbf{r}'$ , as it is done in  $s_4$  evaluation. Now from the induction hypothesis we get  $\Gamma'' : e[q'/\mathbf{x}][\mathbf{r}/\mathbf{y}] \downarrow^{s4} \Delta' : v'$ . By applying rule  $\text{Lam}_{\uparrow^{s4}}$   $n$  times and  $\text{App}^{(1)}$   $n-1$  times, we finally get  $\Gamma' : q \uparrow_{s4} \Delta' : \lambda z'.v'$ .

$\text{Cons}_{\uparrow^{\text{SSTG1}}}$ : Applying the induction hypothesis to the rule's premises and finally rule  $\text{Cons}_{\uparrow^{s4}}$  gives the desired result.

$\text{Accu}_{\uparrow^{\text{SSTG1}}}$ : Apply the induction hypothesis to the rule's premise and finally rule  $\text{Accu}_{\uparrow^{s4}}$ .

Next, we focus on the read back helper relation for accumulators  $\uparrow^{\text{SSTG1}}$ , considering three cases for three possible deduction rules.

$\text{Var}_{\uparrow_{\langle \rangle}^{\text{SSTG1}}}$ : trivial.

$\text{App}_{\uparrow_{\langle \rangle}^{\text{SSTG1}}}$ : A simple application of the induction hypotheses to the premises gives the desired result.

Case $_{\uparrow\langle\rangle}^{\text{SSTG1}}$ : For the premises, note that

$$\Delta_i \cup \Theta_i : p : \mathbf{alt} : \diamond \updownarrow^{\text{SSTG1}} \Delta_{i+1} : v_i$$

implies

$$\Delta_i \cup \Theta_i : \text{case } p \text{ of } \mathbf{alt} : \diamond \updownarrow^{\text{SSTG1}} \Delta_{i+1} : v_i,$$

since the beginning of the SSTG1 reduction sequence involved can be extended by a further step using rule case $^{(1)}$ :

$$\Delta_i \cup \Theta_i : \text{case } p \text{ of } \mathbf{alt} : \diamond \rightarrow \Delta_i \cup \Theta_i : p : \mathbf{alt} : \diamond.$$

Applying the induction hypothesis to this modified premise and the premise for the normalization of the scrutinee gives, together with rule Case $_{\uparrow\langle\rangle}^{\text{S4}}$ , the conclusion of the proposition.

For the proposition on strong normalization, we use inversion on the single deduction rule of  $\updownarrow^{\text{SSTG1}}$ .

Norm: We have  $\Gamma : e : \diamond \updownarrow^{\text{SSTG1}} \Delta : v$  and from the premises the reduction sequence  $(\Gamma, e, \diamond) \rightarrow^* (\Theta, p, q)$  with  $\mathcal{W}(\Theta, p, q)$  and the read back  $\Theta : p : q \uparrow^{\text{SSTG1}} \Delta : v$ .

Lemma 4.6 gives  $\Gamma' : e \downarrow \Theta' : p'$  and in each of the three cases for the possible weak normal forms, i.e.  $\lambda$ -abstractions, constructor expressions or accumulators, the induction hypothesis for the read back applies such that rule Norm for s4 can be applied.  $\square$

For the proof of equivalence of the weak evaluation using SSTG<sub>2</sub> and ISSTG we need to assign to each reduction sequence reaching a legal final state of one of the semantics a corresponding sequence using the other semantics. In this appendix we give for each possible corresponding pair of SSTG<sub>2</sub> and ISSTG configurations a sequence for each semantics, leading to a new corresponding pair of configurations.

All segments shown are either only one step long or pass through intermediate configurations that are clearly non-final. Additionally, all our rules are deterministic. Thus, given a sequence reaching a legal final state in one semantics and a corresponding initial configuration in the other semantics, we can divide the sequence along the segments, and build up a sequence in the other semantics.

We give the segments in the order of the SSTG<sub>2</sub> rules from figure 5.2, listing the SSTG<sub>2</sub> segment first, followed by the corresponding ISSTG segment. Since the ISSTG configurations consume much space due to their instruction sequence, we only show the first instruction of each sequence. The whole instruction sequence can be reconstructed by considering all configurations shown in a segment.

It is easy to check that, under the assumption that the initial configurations in the following segments are corresponding, the final configurations are corresponding, too. However, one has to take all side conditions and equations resulting from the translation from figure 6.3 and the semantics SSTG<sub>2</sub> and ISSTG, given in figures 5.2 and 6.4, respectively, into account.

### B.1 LIST OF SEGMENTS

- When entering a thunk, an update frame is pushed, and the entered closure is deallocated by SSTG<sub>2</sub>.

$$\begin{array}{c} (\Gamma \cup [p \mapsto (e, E')], x, E[x \mapsto p], S) \\ \xrightarrow{\text{var}^{(1)}} (\Gamma, e, E', \#p \cdot S) \end{array}$$

Correspondingly, the code pointer of the current closure is overwritten with  $p_{bh}$  by ISSTG, while the same update frame is pushed.

$$\begin{array}{c} ([\text{UPDMARK}, \dots], S', p, \Gamma' \cup [p \mapsto (p_c, q)]) \\ \rightarrow (\text{trE } e \ \rho_\emptyset \ \eta, \#p \cdot S', p, \Gamma' \cup [p \mapsto (p_{bh}, q)]). \end{array}$$

- When execution reaches an abstraction, a check is performed whether enough arguments are present on the stack. If this check fails, a non-updating application closure is constructed in SSTG2.

We use the abbreviation  $E'' := [x \mapsto p] \cup [x' \mapsto p']$ .

$$\begin{aligned} & (\Gamma[p \mapsto (\lambda x. e, E')], x, E[x \mapsto p], p' \# \#q :: S) \\ \xrightarrow{\text{var}(2)} & (\Gamma \cup [q \mapsto (x x', E'')], x, E, p' \# S) \end{aligned}$$

In ISSTG the failing check results in overwriting a closure with code pointer  $p_{bh}$ , that must have been created when pushing the update frame, with a partial application with code pointer  $p_{p\alpha p}$ . Due to the configuration correspondence, we have an instruction sequence

$$is = [ARGCHECK \ |x|] \# \text{trE } e \ \rho \ \eta$$

in this case.

$$\begin{aligned} & (is, p' \# \#q :: S', p, \Gamma' \cup [q \mapsto (p_{bh}, r)]) \\ \rightarrow & (is, p' \# S', p, \Gamma' \cup [q \mapsto (p_{p\alpha p}, p :: p')]) \end{aligned}$$

- A constructor finding an update frame instead of a sequence of alternatives on the stack in SSTG2 creates an indirection  $\mapsto p$ .

$$\begin{aligned} & (\Gamma[p \mapsto (C \ y, E)], x, E[x \mapsto p], \#q :: S) \\ \xrightarrow{\text{var}(3)} & (\Gamma \cup [q \mapsto \mapsto p], x, E, S) \end{aligned}$$

Compiled constructors in ISSTG create an indirection using  $p_{ind}$ , again overwriting a closure containing  $p_{bh}$ . Due to the definition of configuration correspondence, we have

$$is = [RETURNCON \ C]$$

in this case.

$$\begin{aligned} & (is, \#q :: S', p, \Gamma' \cup [q \mapsto (p_{bh}, r)]) \\ \rightarrow & (is, S', p, \Gamma' \cup [q \mapsto (p_{ind}, p)]) \end{aligned}$$

- Accumulators create indirections in SSTG2 using  $\mapsto p$  when facing an update frame.

$$\begin{aligned} & (\Gamma[p \mapsto k], x, E[x \mapsto p], \#q :: S) \\ \xrightarrow{\text{var}(4)} & (\Gamma \cup [q \mapsto \mapsto p], x, E, S) \end{aligned}$$

In ISSTG  $p_{ind}$  is used instead, overwriting a closure containing  $p_{bh}$ .

$$\begin{aligned} & ([ACCU], \#q :: S', p, \Gamma' \cup [q \mapsto (p_{bh}, r)]) \\ \rightarrow & ([ACCU], S', p, \Gamma' \cup [q \mapsto (p_{ind}, p)]) \end{aligned}$$

- When execution reaches an indirection, the target of the indirection is entered.

$$\begin{aligned}
& (\Gamma[p \mapsto q], x, E[x \mapsto p], S) \\
& \xrightarrow{\text{ind}^{(1)}} (\Gamma, x', [x' \mapsto q], S)
\end{aligned}$$

The code sequence allocated at  $p_{\text{ind}}$  leads to the same effect, but needs two steps. The instruction sequence is of the final configuration corresponds to the control expression of the sstg2 closure at  $q$  due to the heap correspondence  $\Gamma \sim \Gamma'$ .

$$\begin{aligned}
& ([\text{BUILDENV}(\text{NODE } 1), \dots], \\
& \quad S', p, \Gamma'[p \mapsto (p_{\text{ind}}, q)]) \\
& \rightarrow ([\text{ENTER}], q : S', p, \Gamma') \\
& \rightarrow (\text{is}, S', q, \Gamma')
\end{aligned}$$

- A non-updatable binding of a (partial) application is evaluated in two steps in sstg2.

$$\begin{aligned}
& (\Gamma[p \mapsto (y \ y', E')], x, E[x \mapsto p], S) \\
& \xrightarrow{\text{ind}^{(2)}} (\Gamma, y \ y', E'[y' \mapsto q'], S) \\
& \xrightarrow{\text{app}^{(1)}} (\Gamma, y, E'[y \mapsto q], q' : S)
\end{aligned}$$

Semantics *ISSTG* needs two steps, too, entering the partially applied function by pushing it temporarily onto the stack.

$$\begin{aligned}
& ([\text{PAP}, \dots], S', p, \Gamma'[p \mapsto (p_{\text{pap}}, q : q')]) \\
& \rightarrow ([\text{ENTER}], q : q' \# S', p, \Gamma') \\
& \rightarrow (\text{is}, q' \# S', q, \Gamma')
\end{aligned}$$

- Applications are evaluated in a single step in sstg2, pushing the arguments onto the stack.

$$\begin{aligned}
& (\Gamma, x \ y, E[y \mapsto q], S) \\
& \xrightarrow{\text{app}^{(1)}} (\Gamma, x, E[x \mapsto p'], q \# S)
\end{aligned}$$

Evaluation of compiled applications in *ISSTG* is more involved, since local variables  $r$  might be present on the stack as run time equivalent to the compile time environment  $\rho$ , and these local variables have to be slid out before entering the applied function. Take the definition of *trE* for applications in figure 6.3 for reference.

$$\begin{aligned}
& ([\text{BUILDENV}(\rho, \eta)x : (\rho, \eta)y, \dots], \\
& \quad r \# S', p, \Gamma') \\
& \rightarrow ([\text{SLIDE}(1 + |y|)(\text{snd } \rho), \dots], \\
& \quad p' : q \# r \# S', p, \Gamma') \\
& \rightarrow ([\text{ENTER}], p' : q \# S', p, \Gamma') \\
& \rightarrow (\text{is}, q \# S', p', \Gamma')
\end{aligned}$$

- An abstraction in sstg2 that finds all its arguments present on the stack grabs them, removing them from the stack and

adding them to the environment. Execution is continued with the body of the abstraction.

$$\begin{aligned} & (\Gamma[p \mapsto (\lambda \mathbf{y}. e, E')], \mathbf{x}, E[x \mapsto p], \mathbf{p}' \# S) \\ \xrightarrow{\text{app}^{(2)}} & (\Gamma, e, E' \cup [\mathbf{y} \mapsto \mathbf{p}'], S) \end{aligned}$$

In ISSTG the compiled abstraction starts with an ARGCHECK instruction, followed by the compiled body. The body was compiled with the abstraction's arguments added to  $\rho$ , thus the arguments remain on the stack until another closure is entered, when they are slid out just before the ENTER instruction.

$$\begin{aligned} & ([\text{ARGCHECK } |\mathbf{p}'|, \dots], \mathbf{p}' \# S', p, \Gamma') \\ \rightarrow & (\text{trE } e \rho \eta, \mathbf{p}' \# S', p, \Gamma') \end{aligned}$$

- An accumulator that finds an argument on top of the stack allocates a new application accumulator.

$$\begin{aligned} & (\Gamma[p \mapsto k], \mathbf{x}, E[x \mapsto p], \mathbf{p}' \# S) \\ \xrightarrow{\text{app}^{(3)}} & (\Gamma \cup [q \mapsto \langle p \mathbf{p}' \rangle], \mathbf{x}', [x' \mapsto q], S) \end{aligned}$$

The ACCU instruction of ISSTG behaves exactly in the same way.

$$\begin{aligned} & ([\text{ACCU}], \mathbf{p}' \# S', p, \Gamma'[p \mapsto (p_{\text{accu}}, k')]) \\ \rightarrow & ([\text{ACCU}], S', q, \Gamma' \cup [q \mapsto (p_{\text{accu}}, \langle p \mathbf{p}' \rangle)]) \end{aligned}$$

- Upon execution of an expression with local bindings, semantics SSTG2 allocates new closures for the  $\lambda$ -forms and adds the pointers to these to the environment.

We use the abbreviation  $E' = E \cup [x \mapsto q]$ .

$$\begin{aligned} & (\Gamma, \text{let } x = \text{lf}_t \text{ in } e, E, S) \\ \xrightarrow{\text{let}} & (\Gamma \cup [q \mapsto (\text{lf}, E'_t)], e, E', S) \end{aligned}$$

The allocation of new closures is done in several steps in semantics ISSTG. First, one ALLOC instruction per closure is executed and reserves heap space. Next, this space is filled by one BUILDCLS instruction per closure, creating the closures. Each closure contains a code pointer to the translated  $\lambda$ -form as well as pointers matching  $E'_t$ . The remainder of the instruction sequence is the translation of the body, where the fresh pointers are added to  $\rho$  and thus can be found on the stack. Take the definition of trE for local bindings in figure 6.3 for reference.

$$\begin{aligned}
& ([\text{ALLOC } |t_n|, \dots, \text{ALLOC } |t_1|, \dots], \\
& \quad r \# S', p, \Gamma') \\
\rightarrow^* & ([\text{BUILDCLS } n \ p'_n \ a_n, \dots, \text{BUILDCLS } 1 \ p'_1 \ a_1, \dots], \\
& \quad q \# r \# S', p, \Gamma'') \\
\rightarrow^* & (\text{trE } e \ \rho' \ \eta, q \# r \# S', p, \Gamma' \cup [q \mapsto (p', p'')])
\end{aligned}$$

- Case discriminations simply push the alternatives together with the trimmed environment.

$$\begin{aligned}
& (\Gamma, \text{case } x \text{ of } \mathbf{alt}_{|t|}, E, S) \\
\rightarrow_{\text{case}^{(1)}} & (\Gamma, x, E[x \mapsto q], (\mathbf{alt}, E|_t) \cdot S)
\end{aligned}$$

The compiled `ISSTG` code for case discriminations pushes the values used in the case alternatives first, then the pointer to the compiled alternatives' code, and finally the pointer to the scrutinee. Before entering the scrutinee, values for local variables are slid out. Take the definition of `trE` for case discriminations in figure 6.3 for reference.

$$\begin{aligned}
& ([\text{BUILDENV } (\rho, \eta)t, \dots], r \# S', p, \Gamma') \\
\rightarrow & ([\text{PUSHCONT } p', \dots], p_t \# r \# S', p, \Gamma') \\
\rightarrow & ([\text{BUILDENV } (\rho + |t| + 1, \eta)x, \dots], \\
& \quad p' \cdot p_t \# r \# S', p, \Gamma') \\
\rightarrow & ([\text{SLIDE } (2 + |t|) (\text{snd } \rho), \dots], \\
& \quad q \cdot p' \cdot p_t \# r \# S', p, \Gamma') \\
\rightarrow & ([\text{ENTER}], q \cdot p' \cdot p_t \# S', p, \Gamma') \\
\rightarrow & (\text{is}, p' \cdot p_t \# S', q, \Gamma')
\end{aligned}$$

- When the evaluation of a scrutinee reaches a constructor closure, it finds the case alternatives on the stack and selects the appropriate branch. The constructor arguments are added to the saved environment for the alternatives.

$$\begin{aligned}
& (\Gamma[p \mapsto (C_i \ y, E''[y \mapsto q])], \\
& \quad x, E[x \mapsto p], (C \ x \rightarrow e, E') \cdot S) \\
\rightarrow_{\text{case}^{(2)}} & (\Gamma, e_i, E' \cup [x_i \mapsto q], S)
\end{aligned}$$

In `ISSTG` a constructor finds a pointer to the compiled alternatives on the stack and selects the appropriate branch, too. The environment for the selected branch is split into a stack part, i.e. the values for variables contained in the original trimmer for the alternatives, and a node part, where the constructor arguments are reachable via the node pointer, still pointing to the constructor closure. The stack part is later slid out before another closure is entered.

$$\begin{aligned}
& ([\text{RETURNCON } C], p' \cdot r \# S', p, \Gamma') \\
\rightarrow & (\text{trE } e_i \ \rho \ \eta, r \# S', p, \Gamma')
\end{aligned}$$

- When case alternatives are found on top of the stack by an accumulator, it allocates a new accumulator representing a stuck case discrimination.

$$\begin{array}{c} (\Gamma[p \mapsto k], x, E[x \mapsto p], (\mathbf{alt}, E') : S) \\ \xrightarrow{\text{case}^{(3)}} (\Gamma \cup [r \mapsto \langle \text{case } p \text{ of } (\mathbf{alt}, E') \rangle], y, [y \mapsto r], S) \end{array}$$

The ACCU instruction of ISSTG does the same.

$$\begin{array}{c} ([\text{ACCU}], p' : q \# S', p, \Gamma') \\ \rightarrow ([\text{ACCU}], S', r, \Gamma' \cup [r \mapsto \langle \text{case } p \text{ of } (p', q) \rangle]) \end{array}$$

## SOURCE CODE OF BENCHMARKS

---

This appendix gives the full source code of our benchmark problems of chapter 9.

### C.1 PEANO NUMBERS

#### C.1.1 PTS

The PTS implementation is straight forward, following the examples of chapter 7.

---

```

1 data eq (A : *) (a : A) : A -> * where
2   refl : eq A a a
3
4 data nat : * where
5   z : nat
6   s : nat -> nat
7
8 pred = λ n : nat .
9   case n return nat of
10     z -> z
11     s p -> p
12
13 add = λ x : nat . fix rec (y : nat) : nat =
14   case y return nat of
15     z -> x
16     s y' -> s (rec y')
17
18 mul = λ x : nat . fix rec (y : nat) : nat =
19   case y return nat of
20     z -> z
21     s y' -> add x (rec y')
22
23 ten = s (s (s (s (s (s (s (s (s z))))))))
24 hundred = mul ten ten
25 tenthousand = mul hundred hundred
26 hundredthousand = mul tenthousand ten
27
28 nTimes = λ f : (nat -> nat) . λ x : nat .
29   fix rec (n : nat) : nat =
30     case n return nat of
31       z -> x
32       s n' -> f (rec n')
33
34 test : eq nat z (nTimes pred hundredthousand hundredthousand)
35   = refl nat z

```

---

C.1.2 *Coq*

This code can easily be translated to Coq's syntax. The following code uses an interpreter for normalization.

---

```

1 Inductive eq (A : Set) (a : A) : A -> Set :=
2   | refl : eq A a a.
3
4 Inductive nat : Set :=
5   | z : nat
6   | s : nat -> nat.
7
8 Definition pred := λ n : nat =>
9   match n return nat with
10    | z => z
11    | s p => p
12  end.
13
14 Definition add := λ x : nat => fix rec (y : nat) : nat :=
15   match y return nat with
16    | z => x
17    | s y' => s (rec y')
18  end.
19
20 Definition mul := λ x : nat => fix rec (y : nat) : nat :=
21   match y return nat with
22    | z => z
23    | s y' => add x (rec y')
24  end.
25
26 Definition ten := s (s (s (s (s (s (s (s (s z)))))))).
27 Definition hundred := mul ten ten.
28 Definition tenthousand := mul hundred hundred.
29 Definition hundredthousand := mul tenthousand ten.
30
31 Definition nTimes := λ f : (nat -> nat) => λ x : nat =>
32   fix rec (n : nat) : nat :=
33     match n return nat with
34      | z => x
35      | s n' => f (rec n')
36    end.
37
38 Definition test : eq nat z (nTimes pred hundredthousand
39   hundredthousand)
40   := refl nat z.

```

---

To use the virtual machine implementation of Coq, only the last few lines have to be changed.

---

```

38 Theorem test : eq nat z (nTimes pred hundredthousand
39   hundredthousand).
40 vm_compute.
41 apply (refl nat z).
42 Qed.

```

---

C.1.3 *Haskell*

The Haskell code translated by the Glasgow Haskell compiler is as simple as one would expect.

---

```

1 import Prelude hiding (pred)
2
3 data Nat = S Nat | Z
4     deriving Show
5
6 add x Z = x
7 add x (S y) = S (add x y)
8
9 mul x Z = Z
10 mul x (S y) = add x (mul x y)
11
12 ten = S (S (S (S (S (S (S (S (S Z))))))))
13 hundred = mul ten ten
14 tenthousand = mul hundred hundred
15 hundredthousand = mul tenthousand ten
16
17 pred Z = Z
18 pred (S x) = x
19
20 nTimes f x Z = x
21 nTimes f x (S y) = f (nTimes f x y)
22
23 test = nTimes pred hundredthousand hundredthousand
24
25 main = print test

```

---

## C.2 CHURCH NUMBERS

## C.2.1 PTS

The structure of the second benchmark is similar to the first one, using Church numbers instead of inductively defined Peano numbers.

---

```

1 data eq (A : *) (a : A) : A -> * where
2   refl : eq A a a
3
4 nat : *
5   =  $\prod A : * . (A \rightarrow A) \rightarrow A \rightarrow A$ 
6
7 zero : nat
8   =  $\lambda A : * . \lambda s : A \rightarrow A . \lambda z : A . z$ 
9
10 succ : nat -> nat
11   =  $\lambda x : nat . \lambda A : * .$ 
12      $\lambda s : A \rightarrow A . \lambda z : A . s (x A s z)$ 
13

```

```

14 add : nat -> nat -> nat
15   = λ x : nat . λ y : nat . λ A : * .
16       λ s : A -> A . λ z : A . x A s (y A s z)
17
18 mul : nat -> nat -> nat
19   = λ x : nat . λ y : nat . x nat (add y) zero
20
21 ten = λ A : * . λ s : A -> A . λ z : A .
22       s (s (s (s (s (s (s (s (s (s z))))))))))
23 hundred = mul ten ten
24 thousand = mul ten hundred
25
26 pair : * -> * -> *
27   = λ A : * . λ B : * . Π C : * . (A -> B -> C) -> C
28
29 mkPair : Π A : * . Π B : * . A -> B -> pair A B
30   = λ A : * . λ B : * . λ a : A . λ b : B . λ C : * .
31       λ f : A -> B -> C . f a b
32
33 fst : Π A : * . Π B : * . pair A B -> A
34   = λ A : * . λ B : * . λ p : pair A B .
35       p A (λ a : A . λ b : B . a)
36
37 snd : Π A : * . Π B : * . pair A B -> B
38   = λ A : * . λ B : * . λ p : pair A B .
39       p B (λ a : A . λ b : B . b)
40
41 predInit : pair nat nat
42   = mkPair nat nat zero zero
43
44 predUpdate : pair nat nat -> pair nat nat
45   = λ p : pair nat nat .
46       (λ x : nat . mkPair nat nat (succ x) x) (fst nat nat
47           p)
48
49 pred : nat -> nat
50   = λ x : nat . snd nat nat
51       (x (pair nat nat) predUpdate predInit)
52
53 test : eq nat zero (thousand nat pred thousand)
54   = refl nat zero

```

---

### C.2.2 Coq

The Coq code closely resembles the PTS listing above.

```

1 Inductive eq (A : Set) (a : A) : A -> Set :=
2   refl : eq A a a.
3
4 Definition nat : Set
5   := forall A : Set, (A -> A) -> A -> A.
6

```

```

1 Definition zero : nat
2   := λ A : Set => λ s : A -> A => λ z : A => z.
3
4 Definition succ : nat -> nat
5   := λ x : nat => λ A : Set => λ s : A -> A => λ z : A =>
6     s (x A s z).
7
8 Definition add : nat -> nat -> nat
9   := λ x : nat => λ y : nat => λ A : Set =>
10     λ s : A -> A => λ z : A => x A s (y A s z).
11
12 Definition mul : nat -> nat -> nat
13   := λ x : nat => λ y : nat => x nat (add y) zero.
14
15 Definition ten := λ A : Set => λ s : A -> A => λ z : A =>
16   s (s (s (s (s (s (s (s (s z)))))))).
17 Definition hundred := mul ten ten.
18 Definition thousand := mul ten hundred.
19
20 Definition pair : Set -> Set -> Set
21   := λ A : Set => λ B : Set => forall C : Set,
22     (A -> B -> C) -> C.
23
24 Definition mkPair : forall A : Set, forall B : Set,
25   A -> B -> pair A B
26   := λ A : Set => λ B : Set =>
27     λ a : A => λ b : B => λ C : Set =>
28     λ f : A -> B -> C => f a b.
29
30 Definition fst : forall A : Set, forall B : Set,
31   pair A B -> A
32   := λ A : Set => λ B : Set => λ p : pair A B =>
33     p A (λ a : A => λ b : B => a).
34
35 Definition snd : forall A : Set, forall B : Set,
36   pair A B -> B
37   := λ A : Set => λ B : Set => λ p : pair A B =>
38     p B (λ a : A => λ b : B => b).
39
40 Definition predInit : pair nat nat
41   := mkPair nat nat zero zero.
42
43 Definition predUpdate : pair nat nat -> pair nat nat
44   := λ p : pair nat nat =>
45     (λ x : nat => mkPair nat nat (succ x) x)
46     (fst nat nat p).
47
48 Definition pred : nat -> nat
49   := λ x : nat => snd nat nat
50     (x (pair nat nat) predUpdate predInit).
51
52 Definition test : eq nat zero (thousand nat pred thousand)
53   := refl nat zero.

```

---

To use the virtual machine implementation of Coq, we again need to change only the last few lines.

---

```

58 Theorem test : eq nat zero (thousand nat pred thousand).
59 vm_compute.
60 apply (refl nat zero).
61 Qed.

```

---

### C.2.3 *Haskell*

The first part of the Haskell source code for this benchmark consists of the normalization by evaluation machinery. Details can be found in the literature [2, 16].

---

```

1 module Main where
2
3 data Term = Var Int
4           | Lam Int Term
5           | App Term Term
6           deriving Show
7
8 data Sem = Tm (Int -> Term)
9           | Fun (Sem -> Sem)
10
11 down :: Sem -> Int -> Term
12 down = \ s -> \ n -> case s of
13   Tm l -> l n
14   Fun f -> Lam n (down (f (Tm (\ n' -> Var n))) (n+1))
15
16 eval :: Term -> (Int -> Sem) -> Sem
17 eval = \ m -> case m of
18   Var x -> \ p -> p x
19   Lam x m0 -> \ p -> Fun (\ d -> eval m0
20     (\ x' -> if x == x' then d else p x'))
21   App m1 m2 -> \ p -> case eval m1 p of
22     Tm l -> Tm (\ n -> App (l n) (down (eval m2 p) n))
23     Fun f -> f (eval m2 p)
24
25 norm :: Term -> Term
26 norm = \ m -> down (eval m (\ x -> Tm (\ _ -> Var x))) 0
27
28 main = print $ norm benchmark

```

---

The second part is the mechanically generated translation of our benchmark to a data structure of type `Term`. We use negative integers for bound variable names. Since proper layout does not enhance readability of the generated code, we kept the original version.

---

```

29 benchmark = (App (Lam (-12) (App (Lam (-9) (App (Lam (-21)
30   (App (Lam (-16) (App (Lam (-17) (App (Lam (-18) (App (Lam
31     (-2) (App (Lam (-8) (App (Lam (-10) (App (Lam (-4) (App
32       (Lam (-7) (App (Lam (-6) (App (Lam (-3) (App (Lam (-1)
33         (Var (-1))) (App (App (Var (-2)) (Var (-3))) (Var (-2))))))
34       (Lam (-5) (App (Var (-4)) (App (App (Var (-5)) (Var (-6)))
35         (Var (-7))))))) (Lam (-11) (App (Lam (-5) (App (App (Var
36           (-8)) (App (Var (-9)) (Var (-5))) (Var (-5))) (App (Var
37             (-10)) (Var (-11)))))) (App (App (Var (-8)) (Var (-12)))
38       (Var (-12)))) (Lam (-11) (App (Var (-11)) (Lam (-14) (Lam
39         (-13) (Var (-13)))))) (Lam (-11) (App (Var (-11)) (Lam
40         (-14) (Lam (-13) (Var (-14)))))) (Lam (-14) (Lam (-13)
41       (Lam (-15) (App (App (Var (-15)) (Var (-14))) (Var
42         (-13)))))) (App (App (Var (-16)) (Var (-17))) (Var
43       (-18))) (App (App (Var (-16)) (Var (-17))) (Var
44       (-17))) (Lam (-19) (Lam (-20) (App (Var (-19)) (App (Var
45         (-19)) (App (Var (-19)) (App (Var (-19)) (App (Var (-19))
46         (App (Var (-19)) (App (Var (-19)) (App (Var (-19)) (App
47           (Var (-19)) (App (Var (-19)) (Var (-20))))))))))))))
48       (Lam (-5) (Lam (-22) (App (App (Var (-5)) (App (Var (-21))
49         (Var (-22))) (Var (-12)))))) (Lam (-5) (Lam (-22) (Lam
50         (-19) (Lam (-20) (App (App (Var (-5)) (Var (-19))) (App
51           (App (Var (-22)) (Var (-19))) (Var (-20)))))) (Lam
52         (-5) (Lam (-19) (Lam (-20) (App (Var (-19)) (App (App (Var
53           (-5)) (Var (-19)) (Var (-20)))))) (Lam (-19) (Lam
54         (-20) (Var (-20))))))

```

---



## BIBLIOGRAPHY

---

- [1] Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. *Electronic Notes in Theoretical Computer Science*, 173: 17–39, 2007. (Cited on page 25.)
- [2] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalisation by evaluation. *Mathematical Structures in Computer Science*, 14(04):587–611, 2004. (Cited on pages 26, 111, and 142.)
- [3] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalization by evaluation. In Ait Mohamed, Munoz, and Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of LNCS, pages 39–54. Springer, 2008. (Cited on page 26.)
- [4] Lennart Augustsson. Cayenne – a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250. ACM, 1998. (Cited on page 15.)
- [5] Henk Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. (Cited on page 18.)
- [6] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures)*, Abramsky & Gabbay & Maibaum (Eds.), Clarendon. Oxford University Press, 1992. (Cited on pages 16, 19, 81, and 83.)
- [7] Ullrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed  $\lambda$ -calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, July 1991. (Cited on page 25.)
- [8] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005. (Cited on page 116.)
- [9] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26:167–177, 1996. (Cited on page 106.)
- [10] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988. (Cited on pages 16 and 85.)

- [11] Pierre Crégut. An abstract machine for the normalization of  $\lambda$ -terms. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 333–340. ACM Press, 1990. (Cited on pages 23 and 24.)
- [12] Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, September 2007. (Cited on page 24.)
- [13] Haskell Curry. Functionality in combinatory logic. In *Proceedings of the National Academy of Sciences*, volume 20, pages 584–590, 1934. (Cited on page 90.)
- [14] Olivier Danvy. Type-directed partial evaluation. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–257. ACM Press, 1996. (Cited on page 25.)
- [15] Alberto de la Encina and Ricardo Peña. Formally deriving an STG machine. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 102–112. ACM, 2003. (Cited on pages 20, 31, 37, 45, 55, and 63.)
- [16] Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004*, volume 2987 of LNCS, pages 167–181. Springer, 2004. (Cited on pages 26, 111, and 142.)
- [17] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. (Cited on page 12.)
- [18] Martin Grabmüller and Dirk Kleeblatt. Harpy: Run-time code generation in Haskell. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell*, page 94. ACM, 2007. (Cited on pages 11 and 97.)
- [19] Benjamin Grégoire. *Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml*. PhD thesis, Université Paris 7, 2003. (Cited on pages 20, 23, 31, 109, and 113.)
- [20] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 235–246. ACM Press, 2002. (Cited on pages 20, 23, 31, and 109.)
- [21] Dietmar Gärtner and Werner Kluge.  $\pi$ -Red+: An interactive compiling graph reduction system for an applied lambda-

- calculus. *Journal of Functional Programming*, 6(5):723–756, 1996. (Cited on page 24.)
- [22] Dirk Kleeblatt. Checking dependent types efficiently. In W. Dosch et al., editor, *Programmiersprachen und Grundlagen der Programmierung*, volume A-07-07 of *Schriftenreihe A der Institute für Informatik und Mathematik*. Universität zu Lübeck, 2007. (Cited on page 11.)
- [23] Dirk Kleeblatt. Checking dependent types using compiled code. In *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007. Revised Selected Papers*, volume 5083 of *LNCS*, pages 165–182. Springer, 2008. (Cited on pages 11 and 106.)
- [24] Dirk Kleeblatt. Deriving a strongly normalizing STG machine. In Stefan Fischer, Erik Maehle, and Rüdiger Reischuk, editors, *INFORMATIK 2009, Im Focus das Leben*, volume 154 of *Lecture Notes in Informatics*. Gesellschaft für Informatik, 2009. (Cited on page 11.)
- [25] Werner Kluge. *Abstract computing machines*. Springer, 2004. (Cited on page 24.)
- [26] Jean-Louis Krivine. Un interpréteur du lambda-calcul. unpublished, 1985. URL <http://www.pps.jussieu.fr/~krivine/articles/interpert.pdf>. (Cited on page 23.)
- [27] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990. (Cited on page 23.)
- [28] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL <http://coq.inria.fr>. Version 8.0. (Cited on pages 16, 81, 88, and 92.)
- [29] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer, 1981. (Cited on page 106.)
- [30] Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in *LNCS*, 1993. (Cited on pages 81, 88, and 92.)
- [31] Simon Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 184–201. ACM, 1989. (Cited on page 31.)

## Bibliography

- [32] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. (Cited on page 96.)
- [33] Dana Scott. Continuous lattices. In *Toposes, Algebraic Geometry and Logic*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136. Springer, 1972. (Cited on page 93.)
- [34] Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, May 1971. (Cited on page 12.)