Safe Strict Evaluation of Redundancy-Free Programs From Proofs

by

Brent Eric Knight
B.Sc., University of Alberta, 1992

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

[redacted] the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]
Dr. M.R. Levy, Supervisor (Department of Computer Science)

[redacted]
Dr. B.M. Kapron, Departmental Member (Department of Computer Science)

[redacted]
Dr. M. van Emden, Departmental Member (Department of Computer Science)

[redacted]
Dr. H. Aït-Kaci, External Examiner (Simon Fraser University)

University of Victoria

Supervisor: Dr. Michael R. Levy

Abstract

Many investigators have considered the possibility of extracting programs in a functional language directly from natural deduction proofs in intuitionistic logic. This approach delivers programs littered with redundant components and has led people to examine various ways of pruning these useless subprograms. Here we give an algorithm for automatically performing this pruning without requiring the user to manually annotate their whole proof. It then becomes possible to allow classical reasoning at points in the proof whose putative computational content is redundant to the extracted program. One particularly good place to take advantage of this new ability is in a rule for well-founded induction which allows us to extract explicitly recursive programs. Unfortunately, all this new flexibility leads to a complication which has not been mentioned in the literature: by pruning supposedly redundant parts of the extracted program we destroy the subject reduction and strong normalization properties that are touted as one of the main advantages of the programs-from-proofs approach. The main original result of this thesis is a propositional type system which inserts a relatively small number of *delay* and *force* operators to restore these properties. We attempt to extend the result to a dependent type system based on first-order predicate logic via the erasure of dependent types.

Examiners:

[redacted]

Dr. M.R. Levy, Supervisor (Department of Computer Science)

[redacted]

Dr. B.M. Kapron, Departmental Member (Department of Computer Science)

[redacted]

Dr. M. van Emden, Departmental Member (Department of Computer Science)

[redacted]

Dr. H. Aït-Kaci, External Examiner (Simon Fraser University)

Table of Contents

List of Figures

Acknowledgements

Thanks are due to my supervisor, Dr. Michael Levy, University of Victoria, who provided support both tangible and intangible. The most obvious examples of the latter are the Sun-3/50 on my desk at school and the Macintosh on my desk at home. This thesis was produced on that Macintosh, which allowed me to perform typographical feats that would have required an additional year of study in order to accomplish in TEX.

Thanks are also due to my co-supervisor, Dr. Bruce Kapron, University of Victoria. We have had many stimulating discussions. Also, he made it possible for me to attend a summer school on proofs and types in Sweden, where I was able to meet and speak with some of the most important people in the field of type theory.

In addition, I would like to thank Dr. Jim Hoover, University of Alberta, who introduced me to the idea of programs-from-proofs during the summer of 1992 as a student research assistant.

Finally, I thank The Natural Science and Engineering Research Council which has supported me financially, in the form of an NSERC 1967 scholarship.

Dedication

For my parents

and

Jody

Chapter 1
Introduction

In this thesis, we will propose a technique for extracting programs from proofs. Our goal is to make this paradigm more practical, by making the extracted programs look more like programs that would have been written by humans and by ensuring that they can be evaluated relatively efficiently – by a strict evaluator. Some familiarity with natural deduction, lambda calculus, and first order predicate logic is assumed, although this chapter includes an introduction to the idea that an inference system can serve both as a logic and as a type system.

The rest of this introduction consists of three parts. In the first, we summarize four perceived shortcomings of the usual formulations of type theory. The second section provides the promised introduction to type theory and the whole proofs-as-programs idea. Finally, drawing on the background material the third section elaborates on the problems introduced in the first section using a small example to illustrate the proposed solutions.

The second chapter develops the standard type system and the strict evaluator. It provides the basic measure against which our subsequent proposals must be judged. The third chapter introduces the extended type system which makes use of annotations to keep track of the computational content – or lack thereof – of various types, thus permitting us to prune computationally irrelevant components from the extracted program. Also in this chapter is an algorithm for taking a derivation in the standard system, together with a little extra information about the desired final type, and transforming the standard derivation

into an extended derivation which extracts the smallest possible term. The fourth chapter proposes an improved type system to addresses the problem introduced by the extended type system of the third chapter: redundancy-free programs may now fail when executed with a strict evaluator. The fifth chapter sketches the possibility of extending the inference rules with a nonconstructive, classical inference rule and with a well-founded induction rule whose use causes the extraction of an explicitly recursive program. The sixth chapter is a survey of related work. The seventh chapter concludes the thesis, and points out directions for future research. There is a Collected Rules Appendix which collects together all the inference rules of our final system. Finally, there is an Examples Appendix in which we show a sample session with our prototype tool for performing automated proof marking and program extraction. It includes some more significant examples than are treated in the body of the thesis.

1.1 Four Common Complaints About Type Theory

The type theory approach [1], [4], [6], [10], [14], [21] to extracting functional programs from proofs differs from the more traditional Floyd/Hoare style [7] of verifying imperative programs in two essential ways. First of all, the Floyd/Hoare style has a much more sharply defined dichotomy between the object language (e.g. Pascal) and the meta language (first order predicate logic). In type theory, the line dividing the object language and the meta language is less clear. In fact, in the pure version of type theory, the extracted programs are isomorphic to the proof from which they were extracted, in the sense that executing the program corresponds to proof normalization. The second difference is that of verification versus synthesis. Traditionally, the Floyd/Hoare style of reasoning has concentrated on program verification. In this methodology, the user presents a program together with a proof. In type theory, the emphasis is on expressing the specification first, and then deriving a program which will meet that specification. In recent years, this

difference has eroded somewhat as so-called "program refinement" methods have been developed to extend the Floyd/Hoare approach to program synthesis.

There are several disadvantages to working within the framework of type theory. It is a relatively new approach, and it suffers from the fact that the extracted programs are purely functional, that they typically contain redundant constructive components, and that they use lazy evaluation. Another limiting factor is that the standard formulations disallow recursion and classical reasoning. Although it is beyond the scope of this thesis to extract some kind of program other than a functional one, we will address the issues involved in getting more realistic functional programs from proofs – programs that look more like the ones a human would have written "by hand" if they had done the correctness proof "in their head".

To address these problems we will propose four solutions. The first is to recognize and prune redundant parts of the extracted program. The second step is to use a strict evaluator. We would expect this to be harmless, given that we typically have results like strong normalization for programs extracted in type theory. Unfortunately, this is not the case, and strict evaluation interacts badly with the pruning operation; some optimized programs will execute safely on a lazy evaluator, but fail with a strict evaluator. Third, as a consequence of the optimization process, we will recognize that certain types/propositions have no significant computational content. As a result, it will be sound to use classical logic to prove them. Finally, we would like to include a more general induction/recursion scheme than just structural recursion. We now elaborate on each of these points.

1.1.1 Redundancy

Redundancy refers to the characteristic of type theory which causes it to clutter up the program extracted from a proof with constructive witnesses that are typically irrelevant to

the human user. Examples of this kind of thing include the "proof" that one term is equal to another. Sometimes all the user cares about is the fact that a proof exists, not what exact form it takes. Other times, there is never more than one possible proof of a statement, as when proving that one term is equal to another. A strict evaluator wastes time evaluating these extra subterms to normal forms. This has led most presentations of type theory to recommend a lazy evaluator for executing their extracted programs. However, even the proponents of lazy functional programming languages admit that the main source of recent advances in this technology are a direct result of program analysis techniques which allow them to use strict evaluation wherever possible [16]. Moreover, even in a lazy evaluation of a program containing redundant constructive subexpressions, there is still extra work to be done building run-time representations of these subexpressions, even if they are never evaluated the way they would be in a strict evaluator.

Various attempts to solve the problem of redundancy have been reported. They vary in expressiveness and in the amount of work that must be done manually by the user to annotate their proof. The simplest proposals are the not-very-expressive and un-automated subset types of [4] and [14]. In [10] there is an automated version of the subset type, but it is still not very expressive. In [15], a more expressive framework is proposed but no automation is present. Finally, [3] and [20] describe automated, expressive methods, including algorithms for marking the proof tree with annotations which capture quite precisely their constructive content. It is from these algorithms that we draw our inspiration.

1.1.2 Lazy Evaluation

As explained above, lazy evaluation is often used as a crutch to avoid having to prune redundant components from extracted programs. To accommodate this, the run-time representation for every type must now include representations for unevaluated

expressions of that type. These go by such colorful names as *suspensions*, *delays*, or *thunks*. Building them at run time requires time and space, even if they are never fully evaluated. Given that type theory is usually strongly normalizing, that is, all execution strategies terminate, one would assume that using a strict evaluator to execute the extracted programs would not pose a problem. This is certainly true for the standard presentations of type theory. However, as we shall see, there is an unexpected interaction between the pruning process and the strict evaluator. The problem is that by removing function abstractions that appear to be redundant, we expose terms to the evaluator which would not otherwise have been exposed.

The solution we propose to this problem is to perform something akin to strictness analysis to determine which parts of a program can be executed safely by a strict evaluator. The richer framework of type theory enables us to do a cleaner, simpler analysis than the standard abstract-interpretation based approach. We can get a simple criterion with which to judge whether a term can cause the strict evaluator to fail. This criterion is directly related to whether the term has some precondition, in the Floyd/Hoare sense. To be precise about the evaluation of terms, we shall make use of a natural semantics which captures the call-by-value nature of the language [9].

1.1.3 Classical Reasoning Is Not Allowed

The standard formulations of type theory have developed from a background in intuitionistic logic and philosophy. As a result, they tend not to include any provision for performing classical reasoning, for example, double-negation elimination, or the law of the excluded middle. At first glance, this makes sense from a program-extraction point of view, since such proof techniques fail to provide a program for us to extract. On the other hand, the process of removing redundant components from the extracted program leaves us with portions of the proof where we do not care what the extracted program would be,

because it has been pruned. In these situations, all we care about is whether or not something is true or false, not the shape of its elements when viewed as a set of proofs. In these cases we will allow classical reasoning.

Thus, the solution to this problem comes practically "for free" from our solution to the more fundamental problem of pruning redundant components from programs. We will allow classical reasoning in the form of a double-negation elimination. In the chapter on related work, we will contrast our proposal to the very different approach taken by Murthy [13] and Griffin [8], who give an operational interpretation to classical inference rules.

1.1.4 General Recursion Is Not Allowed

Standard formulations of type theory eschew general recursion, preferring to stick to the more "intuitive" technique of structural induction/recursion over data structures. It is easy to find examples for which this is not enough: quicksort, Euclid's GCD algorithm, etc. In quicksort, we pick a "pivot" element of a list and use it to partition the remainder of the list into two parts. Since each must be shorter than the original list (we removed the pivot element), this process cannot continue indefinitely. But this does not fit the element-at-a-time structural recursion that would usually be provided in a type theoretic formulation of lists. Euclid's GCD algorithm exhibits a similar behavior with numbers: at each step the arguments shrink by some amount, even though the amount of this decrement is not strictly equal to one each time.

Fortunately, it is not difficult to include a well-founded induction rule from which we can extract an explicitly recursive program. Intuitively, the rule is in the same vein as the loop-invariant rule of Floyd/Hoare style program verification. The inference rule for well-founded recursion is doubly interesting because it also presents an opportunity to discard a purely logical (i.e. redundant) proof obligation: the proof that recursive calls are well-

founded. And as was mentioned above, any such opportunity is an opportunity to use classical reasoning. Stated more directly, this means that we are free to use classical reasoning to prove the termination of well-founded recursion. Combining these two possibilities – classical reasoning and well-founded recursion – makes it possible to extract a program which embodies Markov's Principle: if we know classically that a number exists with a certain property, and it is possible to test for that property, then we know constructively that such a number exists by a simple unbounded search. Such a program cannot be derived in the standard formulation of type theory [21].

1.2 Background

At the beginning of this century some mathematicians and philosophers raised doubts concerning the validity of certain classical proof techniques, like proof by contradiction, and the law of the excluded middle. For background on the mathematical and philosophical roots of this school of thought, consult [2], and [24]. For us, this is relevant because it inspired some investigators to attribute an explicit operational meaning to proofs. In [12], Kleene explains his *realizability* interpretation of intuitionistic logic. This approach is very similar to that of type theory, except that it is phrased in terms of recursion theory, rather than lambda calculus. In [18], Prawitz explains and proves important results concerning proof normalization. In [11], Howard demonstrates the *Curry-Howard isomorphism* between proofs and lambda calculus terms. A common thread through these works is the *Heyting interpretation* of the logical connectives. For our purposes, we embrace these interpretations in order to enable ourselves to extract programs from proofs, without making any particular commitment to intuitionistic logic or philosophy at the meta-level. In the remainder of this section we introduce the important concepts of this technique which sits on the borderline between logic and programming languages.

1.2.1 The Heyting Interpretation

- A proof of "*P* and *Q*" is a proof of "*P*" together with a proof of "*Q*".
- A proof of "if *P* then *Q*" is a function for transforming a proof of "*P*" into a proof of "*Q*".
- A proof of "*P* or *Q*" is a pair consisting of two parts. The first part is a tag of some sort, for instance a natural number, which indicates whether "*P*" or "*Q*" has been proved. The second part is a proof of the indicated disjunct.
- A proof of "for all *x* in *P*, it is the case that *Q(x)*" is a function for transforming an element, *x*, of "*P*" into an proof of "*Q(x)*".
- A proof of "there exists *x* in *P*, such that *Q(x)*" is an element, *x*, of "*P*" together with a proof of "*Q(x)*".

Under this interpretation, a typical specification for a function may be written as "for all *x* in *A* such that *P(x)*, there exists *y* in *B* such that *Q(x,y)*." Here, *P* serves as a precondition and *Q* plays the role of a postcondition, in Floyd/Hoare terminology.

1.2.2 Natural Deduction

Natural deduction is the name of a style of writing formal proofs that closely mimics the informal style that is most commonly used. It bears a strong relationship with the intuitionistic interpretation of the logical connectives. A natural deduction proof is often written as a *proof tree*. It represents a proof of the theorem at the bottom (root) of the tree, possibly under the assumption of some hypotheses at the top (leaves) of the tree. Inference

rules are written as a sequence of zero or more hypotheses, followed by a conclusion, separated by a horizontal line. Here are two examples:

$$\frac{\begin{array}{cc} \vdots & \\ A & B \end{array}}{A \wedge B} \qquad \frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B}$$

The first of these expresses the rule "if A is provable and B is provable, then A-and-B is provable." The second expresses the rule "if, under the assumption that A is provable, we can prove B, then we may *discharge* the assumption, having shown that if-A-then-B is provable."

1.2.3 Sequents

Rather than use raw proof trees, in the body of this thesis we shall make use of sequents that show the *context* of each node in a proof tree, that is, the list of undischarged assumptions. Thus, a sequent will appear like this:

$$x_1{:}A_1 \ , \ \ldots \ , \ x_k{:}A_k \vdash b(x_1,\ldots,x_k) : B(x_1,\ldots,x_k)$$

The list of assumptions will be abbreviated as Γ or written as [] if there are no assumptions. We note here that the order of discharge of the assumptions determines the order in which the extracted function accepts its arguments. Different assumption ordering leads to different functions being extracted. We also note that we will be dealing primarily with type systems without dependent types, in which case our sequents will be written as:

$$x_1{:}A_1 \ , \ \ldots \ , \ x_k{:}A_k \vdash b(x_1,\ldots,x_k) : B$$

1.2.4 Conjunctions are Records

To introduce a pair, we must have derived both its components:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ a{:}A & b{:}B\end{array}}{\langle a,b\rangle{:}(A\wedge B)}\ (\wedge\text{-}intro)$$

Having derived a pair, we may extract one or the other of its components:

$$\frac{\begin{array}{c}\vdots \\ e{:}(A\wedge B)\end{array}}{fst(e){:}A}\ (\wedge\text{-}elim\text{-}left) \qquad \frac{\begin{array}{c}\vdots \\ e{:}(A\wedge B)\end{array}}{snd(e){:}B}\ (\wedge\text{-}elim\text{-}right)$$

In Pascal, we would call this a *record*. In C we would call this a *struct*:

```
typedef struct {int fst, snd;} pair;

main()
{
      pair e;
      e.fst = 0; e.snd = 1;
      printf("<%d,%d>\n",e.fst,e.snd);
}
```

1.2.5 Implications are Functions

Having derived something under a hypothesis, we have demonstrated a function which transforms a proof of the hypothesis into a proof of the conclusion:

$$\frac{\begin{array}{c}[x{:}A] \\ \vdots \\ b{:}B\end{array}}{(\lambda(x)b){:}(A\rightarrow B)}\ (\rightarrow\text{-}intro)$$

Having derived a function, we may supply it with an argument:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ f{:}(A\rightarrow B) & a{:}A\end{array}}{f(a){:}B}\ (\rightarrow\text{-}elim)$$

The λ in the first rule simply indicates function abstraction, for example, in C:

```
B f(A x)
{
        ...;
}
```

1.2.6 Disjunctions are Tagged Records

For the technical results of this thesis we shall stick to a simple inference system whose types are consist of truth, falsity, and the natural numbers, joined by implications and conjunctions. In this system, it is possible to simulate the Heyting interpretation of disjunction. However, a more human-friendly system should include disjunction as a primitive. As with the other connectives, we have introduction rules:

$$\frac{\begin{array}{c}\vdots\\ a{:}A\end{array}}{inl(a){:}(A \vee B)}\ (\vee\text{-}intro\text{-}left) \qquad \frac{\begin{array}{c}\vdots\\ b{:}B\end{array}}{inr(b){:}(A \vee B)}\ (\vee\text{-}intro\text{-}right)$$

And an elimination rule:

$$\frac{\begin{array}{ccc}\vdots & [x{:}A] & [y{:}B]\\ e{:}A \vee B & \vdots & \vdots\\ & f{:}C & g{:}C\end{array}}{(case\ e\ of\ inl(x) \Rightarrow f \mid inr(y) \Rightarrow g){:}C}\ (\vee\text{-}elim)$$

These are available in Pascal as *variant records*, although not in a type-safe way. In C, *union* types can be made to play this role, as long as one is careful to include a tag field. These constructions are more popular in modern, typed functional languages – for example, ML or Haskell. On a more pragmatic note, object-oriented programming languages are typically implemented in a way similar to this. That is, each object is tagged with its type. For example, the boolean classes in Smalltalk have a strong lambda-calculus flavor to them.

1.2.7 Proof Normalization

The rules of the natural deduction proof system are categorized as either *introduction* or *elimination* rules. There is a kind of circuitousness whenever an elimination rule immediately follows an introduction rule. Whenever such a configuration occurs, there is a more direct route that could have been taken. The process of removing these indirect steps from a proof is called *proof normalization*. When performed by a computer, this gives an operational meaning to a natural deduction proof, just like a programming language.

Assembling then immediately disassembling a conjunction is redundant:

$$\dfrac{\dfrac{\begin{array}{cc}\Pi_1 & \Pi_2 \\ \vdots & \vdots \\ A & B\end{array}}{A \wedge B}}{A} \Rightarrow \dfrac{\Pi_1}{\vdots\ A} \quad \text{and, symmetrically,} \quad \dfrac{\dfrac{\begin{array}{cc}\Pi_1 & \Pi_2 \\ \vdots & \vdots \\ A & B\end{array}}{A \wedge B}}{B} \Rightarrow \dfrac{\Pi_2}{\vdots\ B}$$

Arguing hypothetically is redundant when the hypothesis is already proved. This also goes by the name of *cut elimination*:

$$\dfrac{\dfrac{\begin{array}{c}[A] \\ \vdots \\ \Pi_1 \\ \vdots \\ B\end{array} \quad \begin{array}{c}\Pi_2 \\ \vdots \\ A\end{array}}{A \rightarrow B}}{B} \Rightarrow \dfrac{\begin{array}{c}\Pi_2 \\ \vdots \\ \Pi_1 \\ \vdots \\ B\end{array}}{}$$

It is redundant to prove something, then perform a case analysis to decide whether or not it was proven. Thus we have the following conversions:

$$\dfrac{\dfrac{\Pi_1}{\dfrac{A}{A\vee B}}\quad \begin{matrix}[A]\\ \Pi_2\\ C\end{matrix}\quad \begin{matrix}[B]\\ \Pi_3\\ C\end{matrix}}{C} \Rightarrow \begin{matrix}\Pi_1\\ A\\ \Pi_2\\ C\end{matrix} \qquad \text{and, symmetrically,} \qquad \dfrac{\dfrac{\Pi_1}{\dfrac{B}{A\vee B}}\quad \begin{matrix}[A]\\ \Pi_2\\ C\end{matrix}\quad \begin{matrix}[B]\\ \Pi_3\\ C\end{matrix}}{C} \Rightarrow \begin{matrix}\Pi_1\\ B\\ \Pi_3\\ C\end{matrix}$$

1.2.8 Program Evaluation

Corresponding to each of the cases of proof normalization, there is a notion of program evaluation. For conjunctions/pairs we have:

$$\dfrac{\dfrac{\dfrac{\Pi_1}{a:A}\quad \dfrac{\Pi_2}{b:B}}{\langle a,b\rangle:(A\wedge B)}}{fst(\langle a,b\rangle):A} \Rightarrow \dfrac{\Pi_1}{a:A} \qquad \text{and, symmetrically,} \qquad \dfrac{\dfrac{\dfrac{\Pi_1}{a:A}\quad \dfrac{\Pi_2}{b:B}}{\langle a,b\rangle:(A\wedge B)}}{snd(\langle a,b\rangle):B} \Rightarrow \dfrac{\Pi_2}{b:B}$$

For implications/functions:

$$\dfrac{\dfrac{\begin{matrix}[x:A]\\ \Pi_1\\ b:B\end{matrix}\quad \dfrac{\Pi_2}{a:A}}{(\lambda x.b):A\rightarrow B}}{(\lambda x.b)(a):B} \Rightarrow \dfrac{\begin{matrix}\Pi_2\\ \Pi_1\\ b[a/x]:B\end{matrix}}{}$$

Here, the notation *b[a/x]* simply means "replace *x* by *a* in *b*", with the usual stipulations concerning free and bound variables. This is exactly the notion of function application – that is, replacing a formal parameter with an actual argument.

Finally, the two cases for disjunctions/tagged records. First, if the left alternative was proven:

$$\dfrac{\dfrac{\begin{array}{c}\Pi_1\\ \vdots\\ a : A\end{array}}{inl(a) : (A \vee B)} \quad \begin{array}{c}[x{:}A]\\ \vdots\\ \Pi_2\\ \vdots\\ f : C\end{array} \quad \begin{array}{c}[y{:}B]\\ \vdots\\ \Pi_3\\ \vdots\\ g : C\end{array}}{(case\ inl(a)\ of\ inl(x) \Rightarrow f \mid inr(y) \Rightarrow g) : C} \quad \Rightarrow \quad \begin{array}{c}\Pi_1\\ \vdots\\ a{:}A\\ \vdots\\ \Pi_2\\ \vdots\\ f[a/x] : C\end{array}$$

Second, if the right alternative was proven:

$$\dfrac{\dfrac{\begin{array}{c}\Pi_1\\ \vdots\\ b : B\end{array}}{inr(b) : (A \vee B)} \quad \begin{array}{c}[x{:}A]\\ \vdots\\ \Pi_2\\ \vdots\\ f : C\end{array} \quad \begin{array}{c}[y{:}B]\\ \vdots\\ \Pi_3\\ \vdots\\ g : C\end{array}}{(case\ inr(b)\ of\ inl(x) \Rightarrow f \mid inr(y) \Rightarrow g) : C} \quad \Rightarrow \quad \begin{array}{c}\Pi_1\\ \vdots\\ b{:}B\\ \vdots\\ \Pi_3\\ \vdots\\ g[b/y] : C\end{array}$$

1.2.9 Strong Normalization and Structural Recursion

By repeated application of the transformations just described we can completely expand a proof/program to its normal form. This means that our programs are guaranteed to terminate.

Unfortunately, the situation becomes trickier if we add recursion. As long as we restrict ourselves to structural recursion over inductively defined datatypes (e.g. lists, strings, trees, etc.) everything is alright. The simplest such type we can imagine is the natural numbers constructed from zero by application of the successor function:

$$\dfrac{}{0{:}Nat}\,(zero) \qquad \dfrac{n{:}Nat}{succ(n){:}Nat}\,(succ)$$

And just like the other types, there is an elimination rule. This one is different, though, because it is recursive, just like the definition of the type:

$$\frac{n : Nat \qquad z : A(0) \qquad \begin{array}{c}[x{:}Nat;\ y{:}A(x)] \\ \vdots \\ s : A(succ(x))\end{array}}{let\ natrec(n) \equiv (case\ n\ of\ 0 \Rightarrow z \mid succ(x) \Rightarrow s(x, natrec(x)))\ in\ natrec(n) : A(n)}$$

Thus, we see that an inductive proof naturally leads to a recursive program, expressed here in the usual style of sugared lambda-calculus [16]. We save precise discussion of its operational interpretation until we present the strict evaluator in the next chapter. Notice, however, that the manner in which the extracted program recurses is determined once-and-for all: the only value upon which it may make a recursive call is the immediate predecessor of the number in question.

1.2.10 Negation

In classical logic, if a statement is true, then its negation is false, and vice versa. Moreover, every statement is either true or false, *a priori*. This principle goes by the name of the law of the excluded middle:

$$\frac{}{A \vee \neg A}$$

In constructive/intuitionistic logic, a statement is may be proven or unproven, but there is a crucial difference between being unable to prove a statement, and being able to prove its negation. The law of the excluded middle is not taken for granted.

Typically, negation is not even included among the built-in connectives, and is instead defined as follows, where $\bot$ stands for logical *falsity*, or *contradiction*:

$$\neg A \equiv A \rightarrow \bot$$

Here are the derived rules for negation:

$$\frac{\Gamma, x{:}A \vdash \bot}{\Gamma \vdash \neg A}\ (\neg\text{-}intro) \qquad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \bot}\ (\neg\text{-}elim)$$

A number of other classical principles of reasoning are equivalent to the law of the excluded middle, including double-negation elimination:

$$\frac{\neg\neg A}{A}$$

Intuitionism rejects these equivalent notions as well. For a very informal idea of what the problem is, consider the following line of C code (recalling that the exclamation mark is C syntax for negation):

```
printf("%d %d %d\n", !!0, !!1, !!2);
```

It will display:

```
0 1 1
```

Why is this interesting? This use of negation displays an asymmetry which is analogous to that of constructive logic. In constructive logic, if we negate a "false" statement we get one which is simply true, but when we negate something which was "true", we remove all the additional information. In C, when we negate the representation of false (that is, `0`) we get the canonical representation of true (that is, `1`), and if we negate it again, we get back what we started with. On the other hand, if we negate any of the many other representatives of "true", the representation of "false" which is returned has lost all traces of its origin, and double-negation is an essentially information-reducing operation, even though it preserves the two-valued true/false interpretation.

It should not come as much of a surprise that the double negation of the law of the excluded middle is derivable in intuitionistic logic, even though the positive statement is not:

1. $\neg(A \vee \neg A), A \vdash A$		by assumption
2. $\neg(A \vee \neg A), A \vdash A \vee \neg A$		by $\vee$-intro-left from 1
3. $\neg(A \vee \neg A), A \vdash \neg(A \vee \neg A)$		by assumption
4. $\neg(A \vee \neg A), A \vdash \bot$		by $\neg$-elim from 2, 3
5. $\neg(A \vee \neg A) \vdash \neg A$		by $\neg$-intro from 4
6. $\neg(A \vee \neg A) \vdash A \vee \neg A$		by $\vee$-intro-right from 5
7. $\neg(A \vee \neg A) \vdash \neg(A \vee \neg A)$		by assumption
8. $\neg(A \vee \neg A) \vdash \bot$		by $\neg$-elim from 6, 7
9. $[] \vdash \neg\neg(A \vee \neg A)$		by $\neg$-intro from 8

As a consequence of this, and the soundness of intuitionistic logic, it follows that even though the law of the excluded middle is not derivable, neither is its negation, otherwise we could show a contradiction.

1.2.11 Quantifiers: The Dependent Types

The real power of type theory arises from the ability to express the so-called dependent types. These come in two varieties: the dependent function space, which corresponds to a universal quantifier, and the dependent product which corresponds to an existential quantifier. An example of a dependent function type would be a function which takes a number as input, and returns an array of the given length. If *Array(n)* is the type of arrays of length n, then this type could be written: $\forall n{:}Nat.Array(n)$. This is sometimes written in a more set-theoretic notation as: $\Pi n{:}Nat.Array(n)$. Similarly, we could represent an array of length n paired with its length as: $\exists n{:}Nat.Array(n)$ sometimes written as $\Sigma n{:}Nat.Array(n)$. Elements of this type consist of a natural number, n, together with an

array whose length is n. Together with other dependent types, like equality, they allow us to express anything that expressible in first-order predicate logic.

To illustrate the difference between what can be expressed in propositional logic and what can be expressed in predicate logic, consider the following two types which can be assigned to the identity function on naturals: $Nat \rightarrow Nat$ versus $\forall x{:}Nat.\exists y{:}Nat.x{=}y$. Certainly, the second type enforces more of an implementation than the first. For instance, both $\lambda(x)x$ and $\lambda(x)0$ are of type $Nat \rightarrow Nat$, whereas only $\lambda(x)x$ is of type $\forall x{:}Nat.\exists y{:}Nat.x{=}y$. Read as a specification, the first type is that of functions from the naturals to the naturals, and this (weak) specification is met by the constant function – which always returns zero – as well as the identity function. The second type includes an additional constraint: that the output must be equal to the input, thus ruling out the possibility that the constant function could meet this (strong) specification.

Thus we see that a typical function specification is of the form $\forall x{:}A.\exists y{:}B.C(x,y)$. The additional expressivity provided by dependent types is the key to the power of type theory as opposed to the kinds of types that have found their way into programming languages, like the (polymorphic) propositional type system of ML. Fortunately, although the additional expressivity of these types is useful for expressing specifications, it is not always necessary in its full generality. That is, a dependent type can be collapsed down to a propositional type. And since this erasing process maintains derivability, it will provide a means to simplify the proof of certain results. For the most part, we will prove results for the propositional type system, knowing that we can transform the predicate type system into the propositional type system via a type-erasing map.

1.2.12 Proofs versus Types

In logic, usually all we care about is the fact that a proof exists. As far as truth and falsity are concerned, it does not matter if there is more than one way of proving a statement. On the other hand, in a type system it is vital to distinguish between different elements of the same type, since they denote distinct values. Consider again the type of natural numbers:

$$\frac{}{0 : Nat} \qquad \frac{n : Nat}{succ(n) : Nat}$$

Here, the difference between 0 and *succ(0)* is important. In contrast to this, consider a definition of "less than or equal" on the natural numbers:

$$\frac{}{x \leq x} \qquad \frac{x \leq y}{x \leq succ(y)} \qquad \frac{x \leq y \quad y \leq z}{x \leq z}$$

Here, the difference between various proofs of $0 \leq succ(0)$ probably does not matter. Thus even though the following are differ in their construction, what matters is their conclusion:

$$\frac{0 \leq 0}{0 \leq succ(0)} \quad \text{versus} \quad \frac{\dfrac{0 \leq 0}{0 \leq 0 \quad 0 \leq succ(0)}}{0 \leq succ(0)}$$

In a nutshell, this is the essential difference between a type that is being used for its logical content and one that is being used for its computational content. For a logical statement all that one cares to know is whether it is true or false, which can be shown by a single proof in a sound inference system – the difference between alternative proofs of the same thing are irrelevant. On the other hand, a type like the natural numbers does not make much sense as a "set of proofs", and it is unlikely that we would want to collapse all the natural numbers to a single point. From this discussion, it should be apparent that orthodox type theoretic approaches to program synthesis may be too strict in their identification of proofs and programs.

1.3 Running Example: The Predecessor Function

In this section, we give a capsule overview of the running example that we shall use throughout this thesis to illustrate the techniques that we advocate. As a result, it refers to things which have not yet been explained. Nevertheless, it has been included as a kind of map for the remainder of the thesis.

Throughout this thesis we shall use the predecessor function as a running example to demonstrate the things we are attempting to achieve. It was chosen because it appears to be the simplest possible partial function that exhibits the poor behavior described in the first section of this introduction. That is, when redundant components are pruned out of the extracted program it becomes unsafe to execute it using the strict evaluator.

In the standard version of type theory, we are likely to extract the following term from a derivation of the type to the right of the colon which specifies the predecessor function on the naturals:

$$pred \equiv \lambda n.\lambda p.natrec(n,abort,(x)(y)\langle x,empty\rangle) : \forall(n{:}Nat).n{\neq}0{\rightarrow}\exists(m{:}Nat).n{=}succ(m)$$

Unnatural redundancy occurs in the input and in the output. First, notice that the term ends up taking two arguments at run-time. The first argument is the "real" argument, while the second is a proof that the "real" argument is not equal to 0. Given a proof of this precondition, it is impossible that the argument will be 0. It should not matter what expression is executed in that case, so we might as well use *abort*. On the other hand, given the predecessor of the argument (in x) and the result of a recursive call to this function (in y), we can return the predecessor (i.e. x), together with a "proof" that the stated relationship holds between the argument and the result, represented here as the term *empty*.

By fairly straightforward means, we can "prune" the redundant function abstraction and result pairing from this term. The following function computes the predecessor of a natural number. It is a partial function, since the predecessor of 0 is undefined.

$$pred \equiv \lambda n.natrec(n,abort,(x)(y)x) : \forall(n{:}Nat).n{\neq}0{\rightarrow}\exists(m{:}Nat).n{=}succ(m)$$

Applying this to an argument which obviously fails to satisfy the precondition gives:

$$app(pred,0) \ : 0{\neq}0{\rightarrow}\exists(m{:}Nat).0{=}succ(m)$$

Unfortunately, this will cause our strict evaluator to abort, that is $app(pred,0){\Downarrow}abort$.

At the top-level we could have predicted this possibility just by inspecting the type of *app(pred,0)* and seeing that it has a non-computational precondition. But what if this term were buried as a subterm inside some other term? What if we were to pass *app(pred,0)* as an argument to the constant function that ignores one of its arguments, for example, the K combinator? The strict evaluator would fail in that case too, and the possibility of failure would not be obvious from the top-level type.

To prevent the kind of unfortunate behavior demonstrated in this example, we can ensure that instead of extracting a raw *abort* subterm, we extract a term whose shape depends on the desired type of that subterm. In this particular case, that means extracting a dummy term of type *Nat*. What can we use to play the role of this "default" or "error" value? Anything of type *Nat* will do, so we might as well use 0, giving:

$$pred \equiv \lambda n.natrec(n,0,(x)(y)x) : \forall(n{:}Nat).n{\neq}0{\rightarrow}\exists(m{:}Nat).n{=}succ(m)$$

And again we can make an "erroneous" application:

$$app(pred,0) \ : 0{\neq}0{\rightarrow}\exists(m{:}Nat).0{=}succ(m)$$

The vital difference is that now the term can be safely evaluated, to 0: $app(pred,0){\Downarrow}0$.

The practice of extracting dummy "abort" terms which are specialized to the type desired seems to have solved the problem. But what if the failure exhibited by the evaluator is caused by nontermination rather than a mere type mismatch? As long as we stick to the tradition of type theory and only construct structurally recursive functions – avoiding more general forms of recursion – then we are assured that the possibility of non-termination will not arise. But if we extend our program extraction rules with a well-founded induction rule which extracts an explicitly recursive term, then this becomes a possibility. Much later, the example of Markov's Principle will demonstrate that an unbounded search can fail to terminate if we fail to guarantee its precondition. It will turn out that the cause of unsafety due to overeager evaluation can be attributed to the rules associated with "logical" or "content-free" preconditions. For the moment, rather than dragging in the machinery necessary to exhibit Markov's Principle, we will illustrate with the predecessor function, in which potentially unsafe subterms can be made safe. We should keep in mind the fact that the kind of failure exhibited by the predecessor function is strictly less dangerous than the nontermination that is possible with Markov's Principle – an evaluator can be written which will detect the former but not the latter.

First, we extend the evaluation mechanism with *delay* and *force* operators. They behave like degenerate function abstraction and application in that *delay* literally delays the evaluation of its subterm, while *force* causes the evaluation of a delayed term to proceed. The only danger here is that if we are overzealous we may put in so many of these operations that we might as well have used a lazy functional language in the first place! Thus our goal will be to ensure termination under a strict evaluation scheme without adding so many delays as to negate the effect of the pruning that we originally did.

We would like to extract a term whose potentially unsafe subterms were wrapped in delays, at least until their noncomputational preconditions are satisfied. Naively, we could extract this term for the predecessor function:

$$pred \equiv \lambda n.delay(natrec(n,0,(x)(y)x)) : \forall(n{:}Nat).n{\neq}0{\rightarrow}\exists(m{:}Nat).n{=}succ(m)$$

Once again, we can try evaluating this function outside its intended domain, as shown by its type:

$$app(pred,0) \ : 0{\neq}0{\rightarrow}\exists(m{:}Nat).0{=}succ(m)$$

Here's what the evaluator does with it: $app(pred,0)\Downarrow delay(natrec(0,0,(x)(y)x))$. That is, its evaluation has been delayed, pending a proof that the argument satisfies the precondition.

Unfortunately, this naive scheme reverses the optimization we achieved by removing noncomputational lambda abstractions and applications. Everywhere we could have pruned off a seemingly redundant abstraction or application, this scheme will insert delays and forces. Therefore, we propose a more sophisticated version of the rules which will extract:

$$pred \equiv \lambda n.force(delay(natrec(n,0,(x)(y)x))) : \forall(n{:}Nat).n{\neq}0{\rightarrow}\exists(m{:}Nat).n{=}succ(m)$$

This term seems even bigger and more redundant than the previous one, but it now contains an obvious force-delay redex which can be removed at the time when the program is extracted from the proof. This leads us to the final version of the predecessor function:

$$pred \equiv \lambda n.natrec(n,0,(x)(y)x) : \forall(n{:}Nat).n{\neq}0{\rightarrow}\exists(m{:}Nat).n{=}succ(m)$$

Certainly, it is safe to evaluate this term. And, if we apply it to an argument, the final version of the rules will ensure that the result will be delayed:

$$delay(app(pred,0)) : 0\neq 0 \rightarrow \exists(m{:}Nat).0=succ(m)$$

Similarly, if we apply it to *succ(0)*:

$$delay(app(pred,succ(0))) : succ(0)\neq 0 \rightarrow \exists(m{:}Nat).succ(0)=succ(m)$$

Finally, if we are able to prove that the precondition is satisfied, as in the latter case, then we can extract:

$$force(delay(app(pred,succ(0)))) : \exists(m{:}Nat).succ(0)=succ(m)$$

The trivial force-delay will be removed by the peephole optimizer, giving:

$$app(pred,succ(0)) : \exists(m{:}Nat).succ(0)=succ(m)$$

Expanding the definition of *pred* gives us:

$$app((\lambda n.natrec(n,0,(x)(y)x)),succ(0)) : \exists(m{:}Nat).succ(0)=succ(m)$$

Finally, we note a small optimization that is inferred by the system: since this use of *natrec* does not actually require a recursive call, it can be replaced by a degenerate version we shall call *natpred*, for obvious reasons. The resulting term is:

$$app((\lambda n.natpred(n,0,(x)x)),succ(0)) : \exists(m{:}Nat).succ(0)=succ(m)$$

Contrast this with the corresponding term extracted by the naive version of the force and delay rules, which also make use of the dummy abort term:

$$force(app((\lambda n.delay(natrec(n,0,(x)(y)x))),succ(0))) : \exists(m{:}Nat).succ(0)=succ(m)$$

Chapter 2
The Standard System

In this chapter, we will introduce the syntax of terms and types, the mechanics of the strict evaluator, the inference rules for what we shall refer to as the *standard* propositional type system, modelled after System T [5], [6], and their properties.

2.1 Preliminaries

Throughout this thesis, we will be concerned with a relationship between triples of the form $\Gamma \vdash a : A$, called a *sequent*. Γ is the *context*, that is, the list of open assumptions. a is the extracted program. A is the *formula* or *type*.

FIGURE 1. The abstract syntax of terms, values, and types

$$
\begin{aligned}
\mathit{Var}\ x ::= \quad & x \mid y \mid \ldots \\
\mathit{Term}\ a ::= \quad & x \mid \lambda(x)e \mid \mathit{app}(f,a) \\
& \mid \mathit{pair}(a,b) \mid \mathit{fst}(e) \mid \mathit{snd}(e) \\
& \mid 0 \mid \mathit{succ}(n) \mid \mathit{natrec}(n,z,(x)(y)s) \\
& \mid \mathit{empty} \mid \mathit{abort} \\
& (\text{where } x, y \in \mathit{Var} \text{ and } a, b, e, f, n, s, z \in \mathit{Term}) \\
\mathit{Value}\ v ::= \quad & \lambda(x)e \mid \mathit{pair}(u,v) \mid 0 \mid \mathit{succ}(v) \\
& (\text{where } e \in \mathit{Term} \text{ and } u, v \in \mathit{Value}) \\
\mathit{Type}\ A ::= \quad & \mathit{Nat} \mid \top \mid \bot \mid A \wedge B \mid A \rightarrow B \\
& \text{where } A, B \in \mathit{Type}
\end{aligned}
$$

In an attempt to provide a mnemonic hint, the meta-variables in the argument positions of *natrec* are written as n (for "number"), z (for "zero"), and s (for "successor"). Unfortunately, this conflicts with the common usage of z as standing for a variable. Hopefully, not too much confusion will result if we make a note of this now.

As is usual, the function/implication arrow associates to the right, that is:

$$A \rightarrow B \rightarrow C \equiv A \rightarrow (B \rightarrow C)$$

Also worthy of note is that the syntax of values is simply restriction of the syntax of general terms. They correspond to the kinds of things we would expect to be representable inside a strict, i.e. call-by-value, interpreter: closures, pairs, and numerals. We shall sometimes slip into a more equational manner of speech and refer to values as *normal forms*.

Occasionally it will be useful to distinguish between the free and bound variables appearing in a term. Generally, as seen in the abstract syntax above, we follow the example of [14] and indicate the binding of a variable in a subterm by placing the parenthesized variable name in front of the subterm.

Definition 1: (Free Variables) The following clauses define the set of variables that appear *free* in a term. All other variables which occur are *bound.*

$$\begin{aligned}
&FV(x) && \equiv \{x\} \\
&FV(\langle a,b \rangle) \equiv FV(app(a,b)) && \equiv FV(a) \cup FV(b) \\
&FV(fst(a)) \equiv FV(snd(a)) \equiv FV(succ(a)) && \equiv FV(a) \\
&FV(\lambda(x)a) && \equiv FV(a) \setminus \{x\} \\
&FV(0) && \equiv \varnothing \\
&FV(natrec(n,z,(x)(y)s)) && \equiv FV(n) \cup FV(z) \cup (FV(s) \setminus \{x,y\})
\end{aligned}$$

Often, especially when discussing the evaluator, it will be possible to limit ourselves to considering *closed* terms, that is, terms with no free variables. The other way in which we make use of the notion of free versus bound variables is in the definition of substitution, in order to prevent the unintentional capture of free variables.

Definition 2: (Substitution)

$x[a/x]$	$\equiv$	a
y[a/x]	$\equiv$	y, if $y \neq x$
$0[a/x]$	$\equiv$	0
$succ(n)[a/x]$	$\equiv$	$succ(n[a/x])$
$<d,e>[a/x]$	$\equiv$	$<d[a/x],e[a/x]>$
$fst(e)[a/x]$	$\equiv$	$fst(e[a/x])$
$snd(e)[a/x]$	$\equiv$	$snd(e[a/x])$
$app(f,b)[a/x]$	$\equiv$	$app(f[a/x],b[a/x])$
$(\lambda(y)e)[a/x]$	$\equiv$	$(\lambda(y')e[y'/y][a/x])$, where $y' \notin FV(a)$ and $y' \notin FV(e)$
$natrec(n,z,(w)(y)s)[a/x]$	$\equiv$	$natrec(n[a/x],z[a/x],(w')(y')s[w'/w,y'/y,a/x])$, where $w', y' \notin FV(a)$ and $w', y' \notin FV(s)$

We abbreviate $b[d/x][e/y]$ by $b[d/x,e/y]$ or $b[d/x;e/y]$.

Here is the subset of the inference rules for the standard system which are identical to System T. They are expressed in the notation that we shall use throughout this thesis.

FIGURE 2. Standard Propositional Rules - Assumptions, Connectives, and Naturals

$$\frac{}{\Gamma_1 , x{:}A , \Gamma_2 \vdash x : A}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \langle a,b \rangle : A \wedge B} \qquad \frac{\Gamma \vdash e : A \wedge B}{\Gamma \vdash fst(e) : A} \qquad \frac{\Gamma \vdash e : A \wedge B}{\Gamma \vdash snd(e) : B}$$

$$\frac{\Gamma , x{:}A \vdash e : B}{\Gamma \vdash \lambda(x)e : A \rightarrow B} \qquad \frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash app(f,a) : B}$$

$$\frac{}{\Gamma \vdash 0 : Nat} \qquad \frac{\Gamma \vdash n : Nat}{\Gamma \vdash succ(n) : Nat}$$

$$\frac{\Gamma \vdash n : Nat \qquad \Gamma \vdash z : A \qquad \Gamma , x{:}Nat , y{:}A \vdash s : A}{\Gamma \vdash natrec(n,z,(x)(y)s) : A}$$

Definition 3: (Negation) Following the standard convention, we define negation in terms of implication and falsity: $\neg A \equiv A \rightarrow \bot$.

To the bare set of rules we add two for dealing with logical truth and logical falsity. These two rules simply express the fact that truth is always provable, and that if we ever prove falsity, then we can prove anything.

FIGURE 3. Standard Propositional Rules - Truth and Falsity

$$\frac{}{\Gamma \vdash empty : \top} \qquad \frac{\Gamma \vdash e : \bot}{\Gamma \vdash abort : A}$$

Lemma 1: Weakening and permutation of assumptions are allowed. That is, if $\Gamma_1 \vdash a : A$ and $\Gamma_1 \subseteq \Gamma_2$ then $\Gamma_2 \vdash a : A$.

Proof: By induction on the derivation of $\Gamma_1 \vdash a : A$. The important cases occur when a variable/assumption is reiterated from the context and when a variable/assumption is discharged from the context. □

Lemma 2: (Substitution) If Γ_1 , $x{:}A$, $\Gamma_2 \vdash b : B$ and $\Gamma_1 \vdash a : A$ are derivable, then we can construct a derivation of Γ_1 , $\Gamma_2 \vdash b[a/x] : B$.

Proof: By a straightforward induction on the derivation of Γ_1 , $x{:}A$, $\Gamma_2 \vdash b : B$. Notice that the result holds trivially in the base case where b is just a variable associated with a reiterated assumption. Then for the induction, notice that none of the inference rules depends on the *value* of the extracted term in any of its premises – only its *type*, which is preserved by induction. □

Lemma 3: There is no derivation of $[] \vdash a : \bot$ for any term, a.

Proof: Any derivation in the standard type system can easily be transformed into a proof in propositional logic if we map the type *Nat* to truth, and everything else to their logical counterparts. Assume there was a derivation ending in falsity. We could transform it into a proof of falsity in propositional logic. But this contradicts the soundness of propositional logic. Therefore, there can be no such derivation in the propositional type system. □

Lemma 4: There is no derivation of $[] \vdash \mathit{abort} : A$ for any A.

Proof: If there were such a derivation, it would end with:

$$\frac{[] \vdash e : \bot}{[] \vdash \mathit{abort} : A}$$

But the previous lemma tells us that there is no derivation of $[] \vdash e : \bot$, so this cannot be the case. □

2.2 Predicate Logic and Dependent Types

If this section had a subtitle, it would be "Why propositional types are almost enough." We will briefly present the standard rules for extracting programs from proofs in a dependent type system based on first order predicate logic. Then we present an erasure function which maps formulas in this system into formulas in the standard propositional system.

Moreover, we show that this map almost preserves derivability and results in the same term being extracted. Although this process does not solve all our problems, it is intended to reduce the plausibility gap between the simple propositional type system we use for our technical results and a more interesting dependently typed system we would hope to use in practice. In the same vein, we conclude this section with a brief description of how we might encode disjunction in this system.

2.2.1 Predicate Logic Inference Rules

FIGURE 4. Predicate Logic - Universal Quantifier Rules

$$\frac{\Gamma, x{:}A \vdash^{\bullet} b{:}B}{\Gamma \vdash^{\bullet} \lambda(x)b : \forall x{:}A.B(x)} \quad \forall\text{-}intro$$

$$\frac{\Gamma \vdash^{\bullet} f : \forall x{:}A.B(x) \qquad \Gamma \vdash^{\bullet} a : A}{\Gamma \vdash^{\bullet} app(f,a) : B(a)} \quad \forall\text{-}elim$$

FIGURE 5. Predicate Logic - Existential Quantifier Rules

$$\frac{\Gamma \vdash^{\bullet} a : A \qquad \Gamma \vdash^{\bullet} b : B(a)}{\Gamma \vdash^{\bullet} \langle a,b \rangle : \exists x{:}A.B(x)} \quad \exists\text{-}intro$$

$$\frac{\Gamma \vdash^{\bullet} e : \exists x{:}A.B(x)}{\Gamma \vdash^{\bullet} fst(e) : A} \quad \exists\text{-}elim\text{-}left$$

$$\frac{\Gamma \vdash^{\bullet} e : \exists x{:}A.B(x)}{\Gamma \vdash^{\bullet} snd(e) : B(fst(e))} \quad \exists\text{-}elim\text{-}right$$

The variable x becomes bound in $B(x)$ by these quantifiers, and may be renamed in the usual way as long as no variable that was free in $B(x)$ becomes bound thereby.

Moreover, it is possible to define the propositional types – implication and conjunction – by restrictions on the predicate types:

$$\left.\begin{aligned} A \rightarrow B &\equiv \forall x{:}A.B(x) \\ A \wedge B &\equiv \exists x{:}A.B(x) \end{aligned}\right\}, x \notin FV(B(x))$$

The fact that this encoding gives the desired result is a consequence of the properties of the dependent type erasure technique described of the next section.

2.2.2 Erasing Dependent Types

We can erase the dependent types from a proof in the predicate inference system in such a way that provability and term extraction are almost completely preserved. First, we define an erasure map from dependent types to propositional types.

FIGURE 6. Erasing Dependent Types

$$
\begin{aligned}
(\forall x{:}A.B(x))^{\bullet} &\equiv A^{\bullet}\rightarrow B(x)^{\bullet} \\
(\exists x{:}A.B(x))^{\bullet} &\equiv A^{\bullet}\wedge B(x)^{\bullet} \\
(a{=}b)^{\bullet} &\equiv \top \\
(\top)^{\bullet} &\equiv \top \\
(\bot)^{\bullet} &\equiv \bot \\
(Nat)^{\bullet} &\equiv Nat
\end{aligned}
$$

Universal quantification corresponds to implication, and existential quantification to conjunction. Atomic dependent predicates, such as equality, are simply replaced with truth. If this seems nonintuitive, recall that we only intend to perform this erasure on formulas which have proofs in the predicate system. Notice that since terms only appear in the equality predicate, the translated type will contain no free variables, that is, for all $B(x)$, we have $x \notin FV(B(x)^{\bullet})$. As a consequence, $B(a)^{\bullet} = B(b)^{\bullet}$ for any terms, a and b.

For example, here is the predecessor function:

$$\lambda n.\lambda p.natrec(n,abort,(x)(y)\langle x,empty\rangle) : \forall(n{:}Nat).n{\neq}0\rightarrow\exists(m{:}Nat).n{=}succ(m)$$

And here is how it looks after erasure, recalling that negation is shorthand for an implication of falsity:

$$\lambda n.\lambda p.natrec(n,abort,(x)(y)\langle x,empty\rangle) : Nat\rightarrow(\top\rightarrow\bot)\rightarrow(Nat\wedge\top)$$

Of course, we have given no axioms for equality, but the whole point of the dependent type erasure process is that the precise axiomatization is irrelevant to the form of the extracted term. The only requirement is that all the inference rules which conclude with an equality must extract the *empty* term. A typical starting point for an axiomatization of equality would be the following two rules:

$$\frac{\Gamma\vdash^{\bullet} d : D}{\Gamma\vdash^{\bullet} empty : d = d} \qquad \frac{\Gamma\vdash^{\bullet} b : d = e \quad \Gamma\vdash^{\bullet} a : A(d)}{\Gamma\vdash^{\bullet} a : A(e)}$$

Next, we extend the erasure map in the natural way to contexts.

Definition 4: ($\Gamma^{\bullet}$) If $\Gamma = x_1{:}B_1\ ,\ \ldots\ x_k{:}B_k$ then $\Gamma^{\bullet} = x_1{:}B_1^{\bullet}\ ,\ \ldots\ x_k{:}B_k^{\bullet}$.

Lemma 5: Erasure preserves derivability. Specifically, as long as our axiomatization of equality includes only axioms with positive conclusions, like the first example above, then:

$$\text{if } \Gamma\vdash^{\bullet} a{:}A \text{ then } \Gamma^{\bullet}\vdash a{:}A^{\bullet}$$

Proof: By induction on the type derivation in the predicate type system. Certainly the claim holds in the base case, when one reiterates an open assumption. Next, if the derivation ends with:

$$\frac{\Gamma\vdash^{\bullet} d : D}{\Gamma\vdash^{\bullet} empty : d = d}$$

Then, since $(d = d)^{\bullet} \equiv \top$ we construct:

$$\frac{}{\Gamma\vdash empty : \top}$$

The case of any rule which ends in a positive statement of equality is equally simple, since it is translated to logical truth, which is always provable.

If the derivation ends with:

$$\frac{\Gamma \vdash^{\bullet} b : d = e \quad \Gamma \vdash^{\bullet} a : A(d)}{\Gamma \vdash^{\bullet} a : A(e)}$$

Then the induction hypothesis gives us a derivation of $\Gamma \vdash a : A(d)^{\bullet}$ which gives the desired result by virtue of the fact that $A(d)^{\bullet} = A(e)^{\bullet}$.

If the derivation ends with:

$$\frac{\Gamma, x{:}A \vdash^{\bullet} b{:}B(x)}{\Gamma \vdash^{\bullet} \lambda(x)b : \forall x{:}A.B(x)} \forall\text{-}intro$$

then a simple application of induction gives us:

$$\frac{\Gamma^{\bullet}, x{:}A^{\bullet} \vdash b{:}B(x)^{\bullet}}{\Gamma^{\bullet} \vdash \lambda(x)b : A^{\bullet} \rightarrow B(x)^{\bullet}} \rightarrow\text{-}intro$$

The other cases proceed similarly. If the derivation ends with:

$$\frac{\Gamma \vdash^{\bullet} f : \forall x{:}A.B(x) \qquad \Gamma \vdash^{\bullet} a : A}{\Gamma \vdash^{\bullet} app(f,a) : B(a)} \forall\text{-}elim$$

then, since $B(x)^{\bullet} = B(a)^{\bullet}$, induction allows us to construct:

$$\frac{\Gamma \vdash f : A^{\bullet} \rightarrow B(x)^{\bullet} \qquad \Gamma \vdash a : A^{\bullet}}{\Gamma \vdash app(f,a) : B(a)^{\bullet}} \rightarrow\text{-}elim$$

If the derivation ends with:

$$\frac{\Gamma \vdash^{\bullet} a{:}A \quad \Gamma \vdash^{\bullet} b{:}B(a)}{\Gamma \vdash^{\bullet} \langle a,b \rangle : \exists x{:}A.B(x)} \exists\text{-}intro$$

then, since $B(x)^\bullet = B(a)^\bullet$, we are able to construct:

$$\frac{\Gamma \vdash a : A^\bullet \qquad \Gamma \vdash b : B(a)^\bullet}{\Gamma \vdash \langle a,b \rangle : A^\bullet \wedge B(x)^\bullet} \wedge\text{-intro}$$

If the derivation ends with:

$$\frac{\Gamma \vdash^\bullet e : \exists x{:}A.B(x)}{\Gamma \vdash^\bullet fst(e) : A} \exists\text{-elim-left}$$

then we can construct:

$$\frac{\Gamma \vdash e : A^\bullet \wedge B(x)^\bullet}{\Gamma \vdash fst(e) : A^\bullet} \wedge\text{-elim-left}$$

If the derivation ends with:

$$\frac{\Gamma \vdash^\bullet e : \exists x{:}A.B(x)}{\Gamma \vdash^\bullet snd(e) : B(fst(e))} \exists\text{-elim-right}$$

then the fact that $B(x)^\bullet = B(fst(e))^\bullet$ allows us to construct:

$$\frac{\Gamma \vdash e : A^\bullet \wedge B(x)^\bullet}{\Gamma \vdash snd(e) : B(fst(e))^\bullet} \wedge\text{-elim-right}$$

□

Although the preceding proof holds for any axiomatization of equality which makes only positive statements about equality, it will fail to hold if we add an inference rule that makes a negative conclusion about equality. For example, Peano's fourth axiom, stated as an incomplete inference rule:

$$\frac{\Gamma \vdash^\bullet n : Nat}{\Gamma \vdash^\bullet ? : succ(n) \neq 0}$$

Unfolding the definition of inequality in terms of negation, and then unfolding the definition of negation in terms of implication and falsity, we see that this is just:

$$\frac{\Gamma \vdash^{\bullet} n : Nat}{\Gamma \vdash^{\bullet} ? : (succ(n) = 0) \rightarrow \bot}$$

The most likely way to fill in the question mark is as follows:

$$\frac{\Gamma \vdash^{\bullet} n : Nat}{\Gamma \vdash^{\bullet} \lambda(x)abort : (succ(n) = 0) \rightarrow \bot}$$

And if we apply the erasure map, we see that we will need to be able to ensure that the following is at least a derived rule in the standard propositional inference system in order to make the following incomplete proof go through:

$$\frac{\Gamma \vdash n : Nat}{\Gamma \vdash \lambda(x)abort : \top \rightarrow \bot}$$

But if we do that, then it becomes simple to derive falsity and extract a term which evaluates to *abort* from an empty context. Clearly, more work needs to be done here. Nevertheless, we have demonstrated that the leap from dependent types to non-dependent types is not very far – as long as we consider only the extracted programs and not the meta-theoretic properties.

2.2.3 Simulating Disjunction

It is usual to present disjunction as a defined connective, and that is what we shall do here [2], [24]. However, there is a problem with this representation: it causes us to extract programs with large dummy components wherever disjunctions are used. As a result, we advocate that any implementation of a system for extracting programs from the proofs support disjunction directly, rather than through the encoding we describe.

Definition 5: (Disjunction) With predicate logic at our disposal we can encode disjunction:

$$A \vee B \equiv \exists n{:}Nat.(n{=}0 \rightarrow A) \wedge (n \neq 0 \rightarrow B)$$

The normal form of something of this type will be a three-tuple, that is, a pair whose first component is a natural number – the tag – together with a pair containing an element of A and an element of B. Herein lies the problem: regardless of whether the left or right disjunct is known to be proved, we must come up with an element of the other disjunction in order to construct this representation. This is quite simple to do, since each type in the propositional system is inhabited. But it is undesirable to manipulate these dummy values at run-time.

Here are the desired disjunction rules:

$$\frac{\Gamma \vdash^{\bullet} a : A}{\Gamma \vdash^{\bullet} inl(a) : A \vee B} \quad \vee\text{-}intro\text{-}left$$

$$\frac{\Gamma \vdash^{\bullet} b : B}{\Gamma \vdash^{\bullet} inr(b) : A \vee B} \quad \vee\text{-}intro\text{-}right$$

$$\frac{\Gamma \vdash^{\bullet} e : A \vee B \qquad \Gamma, x{:}A \vdash^{\bullet} d : C \qquad \Gamma, y{:}B \vdash^{\bullet} e : C}{\Gamma \vdash^{\bullet} case(e,(x)d,(y)e) : C} \quad \vee\text{-}elim$$

2.3 The Strict Evaluator

In this section, we describe the strict evaluator. In the first part, we define the evaluation relation, $a \Downarrow v$, read "a evaluates to v". In the second part, we describe some of its important properties which are independent of the type system. The evaluator is specified with a natural semantics [9]. This particular method for specifying the evaluator was chosen because of the ease with which it can express the strict order of evaluation that we desire. Also, it provides some uniformity to the thesis; most of the properties we wish to

properties we wish to demonstrate can be shown by induction on derivations – either type derivations or evaluation derivations.

2.3.1 Definition

FIGURE 7. The Strict Evaluator - Functions

$$(\textit{eval-lam}) \quad \frac{}{\lambda(x)e \Downarrow \lambda(x)e}$$

$$(\textit{eval-app}) \quad \frac{f \Downarrow v_f \quad v_f = \lambda(x)e \quad a \Downarrow v_a \quad e[v_a/x] \Downarrow v_e}{app(f,a) \Downarrow v_e} \quad v_a \neq abort$$

FIGURE 8. The Strict Evaluator - Pairs

$$(\textit{eval-pair}) \quad \frac{a \Downarrow v_a \quad b \Downarrow v_b}{<a,b> \Downarrow <v_a,v_b>} \quad v_a \neq abort,\ v_b \neq abort$$

$$(\textit{eval-fst}) \quad \frac{e \Downarrow v_e \quad v_e = <v_a,v_b>}{fst(e) \Downarrow v_a}$$

$$(\textit{eval-snd}) \quad \frac{e \Downarrow v_e \quad v_e = <v_a,v_b>}{snd(e) \Downarrow v_b}$$

FIGURE 9. The Strict Evaluator - Naturals

$$(\textit{eval-0}) \quad \frac{}{0 \Downarrow 0}$$

$$(\textit{eval-succ}) \quad \frac{n \Downarrow v_n}{succ(n) \Downarrow succ(v_n)} \quad v_n \neq abort$$

$$(\textit{eval-natrec-z}) \quad \frac{n \Downarrow 0 \quad z \Downarrow v_z}{natrec(n,z,(x)(y)s) \Downarrow v_z}$$

$$(\textit{eval-natrec-s}) \quad \frac{n \Downarrow succ(v_m) \quad natrec(v_m,z,(x)(y)s) \Downarrow v_r \quad s[v_m/x;v_r/y] \Downarrow v_s}{natrec(n,z,(x)(y)s) \Downarrow v_s} \quad v_r \neq abort$$

Now we will complete this set of rules to ensure that the result of a failing evaluation is *abort*. These additional rules fall into two categories: those that simply detect and propagate an abortive subevaluation, and those that arise from the projections,

applications, and recursions whose primary argument evaluates to a value of the wrong shape. Note that the *abort* term is serving a different role than that of the other possible results from the evaluator: it represents a literal failure of the evaluation and an implementation is free to interpret this as it pleases. That is, we do not expect *abort* to be detectable. In return for this, we expect that such an implementation of the evaluator will be more efficient.

FIGURE 10. The Strict Evaluator - Errors

(*eval-abort*) $$\frac{}{abort \Downarrow abort}$$

(*eval-app-abort*) $$\frac{f \Downarrow v_f \quad a \Downarrow v_a}{app(f,a) \Downarrow abort}, v_f \neq \lambda(x)e \text{ or } v_a = abort$$

(*eval-pair-abort*) $$\frac{a \Downarrow v_a \quad b \Downarrow v_b}{<a,b> \Downarrow abort}, v_a = abort \text{ or } v_b = abort$$

(*eval-fst-abort*) $$\frac{e \Downarrow v_e}{fst(e) \Downarrow abort}, v_e \neq <v_a,v_b>$$

(*eval-snd-abort*) $$\frac{e \Downarrow v_e}{snd(e) \Downarrow abort}, v_e \neq <v_a,v_b>$$

(*eval-succ-abort*) $$\frac{n \Downarrow v_n}{succ(n) \Downarrow abort}, v_n = abort$$

(*eval-natrec-s-abort*) $$\frac{n \Downarrow succ(v_m) \quad natrec(v_m,z,(x)(y)s) \Downarrow v_r}{natrec(n,z,(x)(y)s) \Downarrow abort}, v_r = abort$$

(*eval-natrec-abort*) $$\frac{n \Downarrow v_n}{natrec(n,z,(x)(y)s) \Downarrow abort}, v_n \neq 0 \text{ and } v_n \neq succ(v_m)$$

Finally, the evaluation relation is the smallest relation which is closed under these rules.

Note the sense in which this evaluator is strict: function arguments are evaluated before being substituted into the function body, and subterms of constructed values (for example, <•,•> and *succ(•)*) are fully evaluated. However, the evaluator does not correspond

directly to a particular implementation. For example, these rules do not indicate whether the operator or the operand in a function application should be evaluated first, nor the order in which the components of pair should be evaluated. Thus, this specification captures the essential, abstract aspect of strict evaluation that we wish to model, without being overly prescriptive.

For the sake of comparison, here is how we could write a call-by-name version of the evaluation rule for application:

$$(\textit{eval-app-by-name}) \quad \frac{f \Downarrow v_f \qquad v_f = \lambda(x)e \quad e[a/x] \Downarrow v_e}{app(f,a) \Downarrow v_e}$$

Even if the evaluation of a were to fail, this non-strict version of application would only fail if the evaluation of a is forced in the process of evaluating the body of the function. This is not the rule used by the strict evaluator. In the strict evaluator the whole application will fail if the evaluation of a fails.

In this thesis, we will be concerned with the ways in which an evaluation can fail – that is, the terms, a, for which there is no term, v, such that $a \Downarrow v$. It is important to distinguish between the two possible causes of such a failure: *type-mismatch* (for lack of a better term), and *non-termination*. Type-mismatch is meant to describe the situation that arises when we try to evaluate a term like $app\ (0, 0)$. Any attempt to evaluate this term "gets stuck" because the thing in the operator position of the application does not evaluate to a function closure. Non-termination happens for the opposite reason, that is, when an attempt to evaluate a term never "gets stuck" or finishes, and thus proceeds indefinitely. The well-known term $app((\lambda(x)app(x,x)),(\lambda(x)app(x,x)))$ provides an example of this phenomenon.

The first kind of failure is easier to deal with, both theoretically and practically. A concrete implementation of an evaluator could tag its data values so that these kinds of mismatches can be detected. This corresponds to our abstract treatment, where we have exhaustively extended the list of rules defining the evaluation relation such that any of the undefined cases would lead to an *abort*. The type system developed in this thesis will prevent this kind of failure. Non-termination, on the other hand, presents a more difficult challenge. It will be shown that our propositional type system prevents it, but we wish to add a rule for well-founded recursion and so far we can only sketch a proof for such an enhanced type system.

2.3.2 Properties

In this section we illuminate a few vital properties of the evaluation relation which are independent of any type system.

Lemma 6: For all terms, a, v, such that $a \Downarrow v$, the derivation thereof is unique. As a corollary, the evaluation relationship is (partial) function: for all terms, a, u, v, such that $a \Downarrow u$ and $a \Downarrow v$, we have $u = v$.

Proof: The proof proceeds by case analysis on the form of a, followed in each case by induction on the derivation of $a \Downarrow v_a$. Thus, our induction hypothesis is that for any subderivation $b \Downarrow v_b$ of $a \Downarrow v_a$, it is the unique derivation thereof. We shall show two illustrative cases. In the case that $a = \lambda(x)e$, then its evaluation derivation must end with:

$$(\textit{eval-lam}) \ \frac{}{\lambda(x)e \Downarrow \lambda(x)e}$$

We can see by inspection of the definition of the evaluator that this is the unique way to evaluate it. On the other hand, if a=*app*(f,b), then the evaluation may end with either:

$$(\textit{eval-app}) \ \frac{f \Downarrow v_f \qquad v_f = \lambda(x)e \qquad b \Downarrow v_b \qquad e[v_b/x] \Downarrow v_e}{app(f,b) \Downarrow v_e}, \ v_b \neq abort$$

or with:

$$(\textit{eval-app-abort})\ \frac{f \Downarrow v_f \quad b \Downarrow v_b}{app(f,b) \Downarrow abort},\ v_f \neq \lambda(x)e \text{ or } v_b = abort$$

In either case, the induction hypothesis assures us that all the subderivations are unique. Therefore the overall derivation is unique, since the conditions on the rules are non-overlapping. The proof proceeds similarly in the rest of the cases. Therefore, evaluation derivations are unique. As a corollary, the evaluation relation is a (partial) function. If we were working within an equational framework, the corresponding property would be called *confluence*, or the Church-Rosser property. □

Definition 6: (Termination)

$$a\downarrow \equiv \text{there exists } v_a \text{ such that } a \Downarrow v_a$$

That is, a term terminates if there is some value to which it is reduced by the evaluator.

Definition 7: (Equivalence)

$$a \approx b \equiv \text{for all } v,\ a \Downarrow v \text{ iff } b \Downarrow v$$

That is, two terms are equivalent if their termination properties are identical, and, furthermore, if they do evaluate to some values, then those values are identical. Certainly, this relation is an equivalence, that is, it is reflexive, symmetric, and transitive. Unfortunately, due to the fact that our evaluator does not evaluate the body of a lambda abstraction, we do not have congruence; even if $a \approx b$, it is not necessarily the case that $e[a/x] \approx e[b/x]$ for arbitrary e. Nevertheless, some examples of this relationship include:

If $a \approx b$ then $fst(a) \approx fst(b)$, as demonstrated by:

$$\frac{a \Downarrow v_a \quad v_a = \langle u,v \rangle}{fst(a) \Downarrow u} \text{ iff } \frac{b \Downarrow v_b \quad v_b = \langle u,v \rangle}{fst(b) \Downarrow u},\ \text{since } v_a = v_b$$

If $f \approx g$ then $app(f,a) \approx app(g,a)$, as demonstrated by:

$$\frac{f \Downarrow v_f \quad v_f = \lambda(x)e \quad e[a/x] \Downarrow v_e}{app(f,a) \Downarrow v_e} \text{ iff } \frac{g \Downarrow v_g \quad v_g = \lambda(x)e \quad e[a/x] \Downarrow v_e}{app(g,a) \Downarrow v_e}, \text{ since } v_f = v_g$$

Later, it will be convenient and significantly more concise to work with this notion of equivalence rather than draw out fragments of evaluation deductions like we have just done.

Lemma 7: For for all terms, a, v, if $a \Downarrow v$ then $v \in Value$ or $v = abort$. In other words, when evaluation is *successful* it reduces terms to values.

Proof: By induction on the derivation of $a \Downarrow v$. Thus, our induction hypothesis is that for any subderivation $b \Downarrow u$ of $a \Downarrow v$, we have $u \in Value$ or $u = abort$. Proceed by cases on the last step in the derivation of $a \Downarrow v$. Certainly the result holds trivially for all the *(eval-...-abort)* rules. Otherwise, if the last step is:

$$(eval\text{-}lam)\ \frac{}{\lambda(x)e \Downarrow \lambda(x)e}$$

then $a = \lambda(x)e$ and the result of evaluating it is a value. If the evaluation derivation ends with:

$$(eval\text{-}app)\ \frac{f \Downarrow v_f \qquad v_f = \lambda(x)e \qquad b \Downarrow v_b \qquad e[v_b/x] \Downarrow v_e}{app(f,b) \Downarrow v_e},\ v_b \neq abort$$

then $a = app(f,b)$. By the induction hypothesis, all the results of the subderivations are values, including the overall result, v_e. Again, the proof proceeds similarly the rest of the evaluation rules. Therefore, the result of any successful evaluation is a value. □

Lemma 8: Evaluation is the identity function on values. That is, for all $v \in Value$, it is the case that $v \Downarrow v$. Together with the fact that $abort \Downarrow abort$, this tells us that if $a \Downarrow v$ then $v \Downarrow v$ – that is, evaluation is idempotent.

Proof: By induction on the term, v. Since $v \in Value$, there are only four possibilities for the shape of this term: a lambda abstraction, a pair of values, the numeral zero, or the successor of some value. In the first case, we have:

$$(\textit{eval-lam}) \quad \frac{}{\lambda(x)e \Downarrow \lambda(x)e}$$

Obviously, $v \Downarrow v$ in this case. The same is true of *(eval-0)*, whose evaluation rule also has no premises. In the case of a pair, the evaluation derivation must end with:

$$(\textit{eval-pair}) \quad \frac{a \Downarrow v_a \quad b \Downarrow v_b}{\langle a,b \rangle \Downarrow \langle v_a,v_b \rangle} \quad v_a \neq abort,\ v_b \neq abort$$

since the induction hypothesis guarantees that the two subderivations evaluate to themselves, which also rules out the possibility of the evaluation resulting in an *abort*. The same holds for *(eval-succ)*, which has a single subterm/subevaluation. □

2.4 Evaluation Properties of Typed Terms

So far we have seen properties of the evaluation relation which hold for any term. Now we turn briefly to the primary result that motivates the use of type systems, known as *subject reduction*. It will follow from the fact that substitution preserves types.

Lemma 9: (Subject Reduction) If we have derivations of $[] \vdash a{:}A$ and $a \Downarrow v_a$ then there is a derivation of $[] \vdash v_a{:}A$. In other words, evaluation preserves typing. Together with the fact that *abort* cannot be extracted in an empty context, this ensures that $v_a \neq abort$.

Proof: By case analysis on the form of the term a, then in each case by induction on the derivation of $a \Downarrow v_a$. For any subderivation thereof, we can assume that the subject

reduction property holds. The property holds trivially of those terms which evaluate to themselves:

$$(\textit{eval-0})\ \frac{}{0 \Downarrow 0} \qquad (\textit{eval-lam})\ \frac{}{\lambda(x)e \Downarrow \lambda(x)e}$$

Next, if $a = succ(n)$, then the type derivation must end with just

$$\frac{[] \vdash n : Nat}{[] \vdash succ(n) : Nat}$$

But the evaluation derivation could potentially end with:

$$(\textit{eval-succ})\ \frac{n \Downarrow v_n}{succ(n) \Downarrow succ(v_n)},\ v_n \neq abort$$

Or with:

$$(\textit{eval-succ-abort})\ \frac{n \Downarrow v_n}{succ(n) \Downarrow abort},\ v_n = abort$$

In either case, we have $n \Downarrow v_n$, and thus the induction hypothesis tells us that $[] \vdash v_n : Nat$ from which we infer $v_n \neq abort$. This rules out the possibility that the evaluation ends with the abortive rule. Thus, we can construct the following type derivation for the result of the evaluation, as desired:

$$\frac{[] \vdash v_n : Nat}{[] \vdash succ(v_n) : Nat}$$

The case for $a = \langle d,e \rangle$ proceeds similarly, except with two subterms instead of just one.

If $a = app(f,b)$ then the type derivation must end with:

$$\frac{[] \vdash f : B \rightarrow A \qquad [] \vdash b : B}{[] \vdash app(f,b) : A}$$

But, again, there are two choices for how the evaluation might end. It might end successfully with:

$$(\textit{eval-app})\quad \frac{f \Downarrow v_f \qquad v_f = \lambda(x)e \qquad b \Downarrow v_b \qquad e[v_b/x] \Downarrow v_e}{\textit{app}(f,b) \Downarrow v_e},\ v_b \neq \textit{abort}$$

Or it might end unsuccessfully with:

$$(\textit{eval-app-abort})\quad \frac{f \Downarrow v_f \quad b \Downarrow v_b}{\textit{app}(f,b) \Downarrow \textit{abort}},\ v_f \neq \lambda(x)e \text{ or } v_b = \textit{abort}$$

In either case, $f \Downarrow v_f$ and $b \Downarrow v_b$ which allows us to conclude by the induction hypothesis that there are type derivations of $[] \vdash v_f : B \rightarrow A$ and $[] \vdash v_b : B$. The former must end with:

$$\frac{x{:}B \vdash e : A}{[] \vdash \lambda(x)e : B \rightarrow A}$$

Thus we have $v_f = \lambda(x)e$ and $v_b \neq \textit{abort}$ which eliminates the possibility that the evaluation derivation ends with the abortive rule. In this case, we can construct a derivation of $[] \vdash e[v_b/x] : A$ by substitution, enabling us to appeal to the induction hypothesis to obtain a derivation of $[] \vdash v_e : A$ as desired.

If $a = \textit{fst}(e)$ then the type derivation must end with:

$$\frac{[] \vdash e : A \wedge B}{[] \vdash \textit{fst}(e) : A}$$

Whereas the evaluation derivation might potentially end either with:

$$(\textit{eval-fst})\quad \frac{e \Downarrow v_e \qquad v_e = \langle v_a, v_b \rangle}{\textit{fst}(e) \Downarrow v_a}$$

Or with:

$$(\textit{eval-fst-abort}) \quad \frac{e \Downarrow v_e}{\textit{fst}(e) \Downarrow \textit{abort}}, \; v_e \neq \langle v_a, v_b \rangle$$

In either case, $e \Downarrow v_e$ and the induction hypothesis ensures that there is a derivation of $[] \vdash v_e : A \wedge B$, which must end with:

$$\frac{[] \vdash v_a : A \quad [] \vdash v_b : B}{[] \vdash \langle v_a, v_b \rangle : A \wedge B}$$

That is, $v_e = \langle v_a, v_b \rangle$ which prevents the abortive evaluation rule from being applicable, and we already have a derivation of $[] \vdash v_a : A$ as desired.

The case for $a = \textit{snd}(e)$ proceeds symmetrically.

The final and most difficult case occurs when $a = \textit{natrec}(n,z,(x)(y)s)$. Here, the type derivation must end with:

$$\frac{[] \vdash n : \textit{Nat} \qquad [] \vdash z : A \qquad x{:}\textit{Nat}\,,\, y{:}A \vdash s : A}{[] \vdash \textit{natrec}(n,z,(x)(y)s) : A}$$

Here, the evaluation derivation may end successfully with *(eval-natrec-0)* or *(eval-natrec-s)*, or unsuccessfully with *(eval-natrec-abort)* or *(eval-natrec-s-abort)*. Regardless of which of these is the case, we must have $n \Downarrow v_n$, from which the induction hypothesis allows us to conclude that there is a derivation of $[] \vdash v_n : \textit{Nat}$. Thus, it is a numeral and either $v_n = 0$ or $v_n = \textit{succ}(v_m)$, for some numeral v_m. This shows that the evaluation derivation cannot end with *(eval-natrec-abort)*. We proceed by induction on the form of v_n.

If $v_n = 0$ then the evaluation derivation ends with:

$$(\textit{eval-natrec-z})\ \frac{n \Downarrow 0 \quad z \Downarrow v_z}{\textit{natrec}(n,z,(x)(y)s) \Downarrow v_z}$$

And the induction hypothesis assures us that $[] \vdash v_z : A$ is derivable, as desired.

If $v_n = \textit{succ}(v_m)$ then we shall assume by induction that for any m such that $m \Downarrow v_m$, for which we can construct derivations of $[] \vdash \textit{natrec}(m,z,(x)(y)s) : A$ and $\textit{natrec}(v_m,z,(x)(y)s) \Downarrow v_r$ that we can then assert that $[] \vdash v_r : A$ is derivable. And since evaluation is the identity on values, this holds for v_m itself. The evaluation derivation could potentially end with either *(eval-natrec-s)* or *(eval-natrec-s-abort)*. In either case, $\textit{natrec}(v_m,z,(x)(y)s) \Downarrow v_r$ and we appeal to our induction hypotheses to assert that $[] \vdash v_r : A$ is derivable. Thus, $v_r \neq \textit{abort}$, so *(eval-natrec-s-abort)* is inapplicable and the evaluation derivation must end with:

$$(\textit{eval-natrec-s})\ \frac{n \Downarrow \textit{succ}(v_m) \quad \textit{natrec}(v_m,z,(x)(y)s) \Downarrow v_r \quad s[v_m/x;v_r/y] \Downarrow v_s}{\textit{natrec}(n,z,(x)(y)s) \Downarrow v_s},\ v_r \neq \textit{abort}$$

Finally, by substitution we are able to construct a derivation of $[] \vdash s[v_m/x,v_r/y] : A$ which allows us to appeal to the induction hypothesis for the last time to show that there is a derivation of $[] \vdash v_s : A$ as desired.

□

In fact, we can even strengthen this result to our evaluator's equivalent of "strong normalization". Specifically, if there is a derivation of $[] \vdash a:A$, then $a\downarrow$. That is, all well-typed terms terminate.

To prove this we could present an argument in the style of *reducibility* which is typically used to prove strong normalization for variants of lambda calculus [6]. We shall be proving such a result for the modified system of the following chapters.

Unfortunately, both these properties – subject reduction and termination – fail to hold for the extended "optimizing" type system described in the next chapter. It will be instructive to examine what "goes wrong". By seeing where things fall apart, it will become clear which inference rules must be modified.

Chapter 3
The Extended System

In this chapter, we develop an extended version of the propositional inference rules which will allow us to prune redundant components from programs extracted from proofs in the system. In the first part, we develop the rules of the extended system, giving a little commentary to motivate the extensions. In the second part, we prove some of the properties of system in order to ensure that the rules can legitimately be called a type system. In the third part we see that although the new inference rules allow us to prune supposedly redundant components of programs, they can lead us to extract failure-prone programs. We develop the example of the predecessor function and discuss our options for trying to retrieve the safe evaluation property, foreshadowing the subject of the following chapter. In the fourth part we show how to hide the complexity of the extended type system behind a simpler type system that looks like the standard system to the user, but which allows automatic generation of a derivation in the extended system, given some small amount of extra information about the desired final type of the derivation.

The basic idea is of the extended type system is to annotate each occurrence of an atomic type, like *Nat*, with a 1 or a 0. These marks indicate whether the type is intended to have computational content or whether it is redundant, respectively. By marking the atomic types we induce a marking on the more complex types – conjunctions and implications – that are built from them. These induced marks capture the simple fact that a pair has

computational content if either of its components has content, and that a function has computational content if its result type has content.

3.1 The Extended Inference Rules

In this section, we describe the extended inference rules from scratch, including a more precise account of type formation and context formation rules. In our description of the standard system of the previous chapter, we expressed the formation rules for types by simply giving a context-free syntax. Here, we do essentially the same thing for the extended inference rules, except that we frame the rules as inference rules. This is simply to follow more closely to the usual presentation of type theory. If we were to give a context-free definition, it would look something like this:

$$\textit{Mark}\ M ::= 0 \mid 1$$
$$\textit{Type}\ A ::= \textit{Nat}^M \mid \top^0 \mid \bot^0 \mid (A^M \wedge B^N)^P \mid (A^M \rightarrow B^N)^P$$

However, not all the types that can be formed by this syntax are allowed. Specifically, the top-level mark, P, is a function of the sub-marks, M and N, as captured in the following, more precise formulation. In each case, an mark of 0 is intended to mean that the type is purely logical, and has no computational content. A mark of 1 means the opposite.

FIGURE 11. Extended Type Formation Rules - Naturals

$$\frac{}{\textit{Nat}^0 : \textit{Type}} \qquad \frac{}{\textit{Nat}^1 : \textit{Type}}$$

FIGURE 12. Extended Type Formation Rules - Truth and Falsity

$$\frac{}{\top^0 : \textit{Type}} \qquad \frac{}{\bot^0 : \textit{Type}}$$

FIGURE 13. Extended Type Formation Rules - Conjunction

$$\frac{A^1:Type \quad B^1:Type}{(A^1 \wedge B^1)^1:Type} \qquad \frac{A^0:Type \quad B^0:Type}{(A^0 \wedge B^0)^0:Type}$$

$$\frac{A^1:Type \quad B^0:Type}{(A^1 \wedge B^0)^1:Type} \qquad \frac{A^0:Type \quad B^1:Type}{(A^0 \wedge B^1)^1:Type}$$

FIGURE 14. Extended Type Formation Rules - Implications

$$\frac{A^1:Type \quad B^1:Type}{(A^1 \rightarrow B^1)^1:Type} \qquad \frac{A^0:Type \quad B^0:Type}{(A^0 \rightarrow B^0)^0:Type}$$

$$\frac{A^1:Type \quad B^0:Type}{(A^1 \rightarrow B^0)^0:Type} \qquad \frac{A^0:Type \quad B^1:Type}{(A^0 \rightarrow B^1)^1:Type}$$

For conjunctions/pairs, we can form the usual fully-constructive version, or degenerate versions where one or both of the components is purely logical. In the case that just the left conjunct is marked as having content we have a type which corresponds to the so-called *subset* type of more orthodox formulations of type theory; having the other degenerate version available as well allows finer control over the extracted program than more traditional presentations. We shall continue to use uppercase roman letters without superscripts as meta-variables ranging over types, but we will also use 1 and 0 as superscripts on such variables when we wish to limit our attention to types with or without computational content, respectively

For implications/functions, we can form the fully logical version, the fully computational version, and a computational version with a purely logical precondition. Notice that there is no type $(A^1 \rightarrow B^0)^1$, as one might expect if one were to simply enumerate the possible markings. The lack of such a type captures the essential *contravariance* of an implication/ function. What could such a type mean? In order to return a computational value, the range type of the function must be computational. Of course, this differs from pairs, where either or both of the components may be non-computational without affecting the other.

The type formation rules describe the types that we are allowed to use. And, as we shall see in the extended inference system below, the types allowed by the formation rules are exactly the types that can be inferred by the introduction rules for each connective. Thus, there is no loss of generality by omitting formation rules for nonsensical types like $(A^{1} \rightarrow B^{0})^{1}$. It simply does not arise.

Finally, we emphasize again that the outermost mark of a type is a straightforward function of the marks of its components. Ultimately, all that matter are the marks on the atomic types. And since the marks on the atomic types of truth and falsity are fixed, the only marks that matter are those on the *Nat* types. We shall sometimes make use of this fact and write only some of the marks on a type if the rest can be inferred. As we shall see, the inference rules will not introduce any types of a shape which could not be formed by the formation rules. For example, if the outermost mark is zero, then all the positively occurring embedded marks must be zero by the formation rules and the fact that the inference rules do not let us form any other types.

Definition 8: (A^{-})We define the following content removal map on types, which we are able to state succinctly be requiring that the result by a valid type:

$$(A^{M})^{-} \equiv A^{0}, A \in \{Nat, \top, \bot\}$$
$$(A \wedge B)^{-} \equiv A^{-} \wedge B^{-}$$
$$(A \rightarrow B)^{-} \equiv A \rightarrow B^{-}$$

Lemma 10: This is a function and the type which results from applying it has 0 as its top level mark.

Proof: The proof is by a straightforward induction over the type formation rules. In fact, it is equivalent to this more verbose presentation:

$$
\begin{aligned}
(A^M)^- &\equiv A^0, A \in \{Nat, \top, \bot\} \\
((A^M \wedge B^N)^P)^- &\equiv ((A^M)^- \wedge (B^N)^-)^0 \\
((A^M \rightarrow B^N)^P)^- &\equiv (A^M \rightarrow (B^N)^-)^0
\end{aligned}
$$

□

For the sake of completeness, we give a precise account of the rules by which we may build *contexts*, that is, the lists of open assumptions that appear in sequents.

FIGURE 15. Extended Context Formation Rules

$$\frac{}{[] : Context}$$

$$\frac{\Gamma : Context \qquad \Gamma \vdash A : Type}{\Gamma, x{:}A : Context}$$

Here, [] denotes the empty context, and a comma is used to indicate an extension of a context with an assignment of a type to a free variable. Throughout this thesis, we shall always assume that there are no duplicate variable names in the context. For the same reason, whenever we find ourselves in a position where we must speak of free and bound variables in a chain of reasoning it will be assumed that they are disjoint. In this, we follow the usual convention. Whenever a context, Γ, appears in an inference rule, we implicitly require that Γ: *Context* has been proven as an additional premise of the rule. To give an example, we shall write this extra premise explicitly in the assumption rules.

FIGURE 16. Extended Propositional Rules - Assumptions

$$\frac{\Gamma_1,x{:}A^{\,0},\Gamma_2 : Context}{\Gamma_1,x{:}A^{\,0},\Gamma_2 \vdash empty{:}A^{\,0}} \quad (assume)$$

$$\frac{\Gamma_1,x{:}A^{\,1},\Gamma_2 : Context}{\Gamma_1,x{:}A^{\,1},\Gamma_2 \vdash x{:}A^{\,1}} \quad (var)$$

$$\frac{\Gamma_1,x{:}A^{\,1},\Gamma_2 : Context}{\Gamma_1,x{:}A^{\,1},\Gamma_2 \vdash empty{:}A^{\,-}} \quad (squash)$$

FIGURE 17. Extended Propositional Rules - Truth

$$\frac{}{\Gamma \vdash empty{:}\top^{0}} \quad (truth)$$

FIGURE 18. Extended Propositional Rules - Conjunction Introduction

$$\frac{\Gamma \vdash a{:}A^{1} \qquad \Gamma \vdash b{:}B^{1}}{\Gamma \vdash \langle a,b\rangle : (A^{1} \wedge B^{1})^{1}} \quad and^{AB}$$

$$\frac{\Gamma \vdash a{:}A^{1} \qquad \Gamma \vdash b{:}B^{0}}{\Gamma \vdash a : (A^{1} \wedge B^{0})^{1}} \quad and^{Ab}$$

$$\frac{\Gamma \vdash a{:}A^{0} \qquad \Gamma \vdash b{:}B^{1}}{\Gamma \vdash b : (A^{0} \wedge B^{1})^{1}} \quad and^{aB}$$

$$\frac{\Gamma \vdash a{:}A^{0} \qquad \Gamma \vdash b{:}B^{0}}{\Gamma \vdash empty : (A^{0} \wedge B^{0})^{0}} \quad and^{ab}$$

FIGURE 19. Extended Propositional Rules - Conjunction Elimination Left

$$\frac{\Gamma \vdash e : (A^{1} \wedge B^{1})^{1}}{\Gamma \vdash fst(e) : A^{1}} fst^{AB}$$

$$\frac{\Gamma \vdash e : (A^{1} \wedge B^{0})^{1}}{\Gamma \vdash e : A^{1}} fst^{Ab}$$

$$\frac{\Gamma \vdash e : (A^{0} \wedge B^{1})^{1}}{\Gamma \vdash empty : A^{0}} fst^{aB}$$

$$\frac{\Gamma \vdash e : (A^{0} \wedge B^{0})^{0}}{\Gamma \vdash empty : A^{0}} fst^{ab}$$

FIGURE 20. Extended Propositional Rules - Conjunction Elimination Right

$$\frac{\Gamma \vdash e : (A^1 \wedge B^1)^1}{\Gamma \vdash snd(e) : B^1}\ snd^{AB}$$

$$\frac{\Gamma \vdash e : (A^1 \wedge B^0)^1}{\Gamma \vdash empty : B^0}\ snd^{Ab}$$

$$\frac{\Gamma \vdash e : (A^0 \wedge B^1)^1}{\Gamma \vdash e : B^1}\ snd^{aB}$$

$$\frac{\Gamma \vdash e : (A^0 \wedge B^0)^0}{\Gamma \vdash empty : B^0}\ snd^{ab}$$

FIGURE 21. Extended Propositional Rules - Implication Introduction

$$\frac{\Gamma, x{:}A^1 \vdash b{:}B^1}{\Gamma \vdash \lambda(x)b : (A^1 \rightarrow B^1)^1}\ imp^{AB}$$

$$\frac{\Gamma, x{:}A^0 \vdash b{:}B^1}{\Gamma \vdash b : (A^0 \rightarrow B^1)^1}\ imp^{aB}$$

$$\frac{\Gamma, x{:}A \vdash b{:}B^0}{\Gamma \vdash empty : (A \rightarrow B^0)^0}\ imp^{ab}$$

FIGURE 22. Extended Propositional Rules - Implication Elimination

$$\frac{\Gamma \vdash f : (A^1 \rightarrow B^1)^1 \qquad \Gamma \vdash a : A^1}{\Gamma \vdash app(f,a) : B^1}\ app^{AB}$$

$$\frac{\Gamma \vdash f : (A^0 \rightarrow B^1)^1 \qquad \Gamma \vdash a : A^0}{\Gamma \vdash f : B^1}\ app^{aB}$$

$$\frac{\Gamma \vdash f : (A \rightarrow B^0)^0 \qquad \Gamma \vdash a : A}{\Gamma \vdash empty : B^0}\ app^{ab}$$

FIGURE 23. Extended Propositional Rules - Natural Introduction

$$\frac{}{\Gamma \vdash 0{:}Nat^1}\ zero$$

$$\frac{\Gamma \vdash n{:}Nat^1}{\Gamma \vdash succ(n){:}Nat^1}\ succ$$

$$\frac{}{\Gamma \vdash empty{:}Nat^0}\ natempty$$

FIGURE 24. Extended Propositional Rules - Natural Elimination

$$\frac{\Gamma \vdash n{:}Nat^1 \qquad \Gamma \vdash z{:}A^1 \qquad \Gamma,x{:}Nat^1,y{:}A^1 \vdash s{:}A^1}{\Gamma \vdash natrec(n,z,(x)(y)s){:}A^1}\ natrec$$

$$\frac{\Gamma \vdash n{:}Nat^1 \qquad \Gamma \vdash z{:}A^1 \qquad \Gamma,x{:}Nat^0,y{:}A^1 \vdash s{:}A^1}{\Gamma \vdash natiter(n,z,(y)s){:}A^1}\ natiter$$

$$\frac{\Gamma \vdash n{:}Nat^1 \qquad \Gamma \vdash z{:}A^1 \qquad \Gamma,x{:}Nat^1,y{:}A^0 \vdash s{:}A^1}{\Gamma \vdash natpred(n,z,(x)s){:}A^1}\ natpred$$

$$\frac{\Gamma \vdash n{:}Nat^1 \qquad \Gamma \vdash z{:}A^1 \qquad \Gamma,x{:}Nat^0,y{:}A^0 \vdash s{:}A^1}{\Gamma \vdash natchoose(n,z,s){:}A^1}\ natchoose$$

$$\frac{\Gamma \vdash n{:}Nat^0 \qquad \Gamma \vdash z{:}A^0 \qquad \Gamma,x{:}Nat^0,y{:}A^0 \vdash s{:}A^0}{\Gamma \vdash empty{:}A^0}\ natind$$

We have tried to choose mnemonic names for the degenerate (but still computationally relevant) versions of *natrec*. Specifically, *natiter* is an iterator that performs primitive recursion but does not allow direct access to the predecessor of its input. On the other hand, *natpred* only allows access to the predecessor and performs no recursive calls. Finally, there is *natchoose* which behaves like an if-then-else.

To be precise, we must extend the syntax of terms to include the new forms of *natrec*: *natpred*, *natiter*, and *natchoose*. For concreteness, we show how to extend the evaluation relation to these new forms:

$$\textit{(eval-natrec-z)} \qquad \frac{n \Downarrow 0 \quad z \Downarrow v_z}{natrec(n,z,(x)(y)s) \Downarrow v_z}$$

(same for *natpred*, *natiter*, *natchoose*)

$$\textit{(eval-natpred-s)} \qquad \frac{n \Downarrow succ(v_m) \quad s[v_m/x] \Downarrow v_s}{natpred(n,z,(x)s) \Downarrow v_s}$$

$$\textit{(eval-natiter-s)} \qquad \frac{n \Downarrow succ(v_m) \quad natiter(v_m,z,(y)s) \Downarrow v_r \quad s[v_r/y] \Downarrow v_s}{natiter(n,z,(y)s) \Downarrow v_s}$$

$$\textit{(eval-natchoose-s)} \qquad \frac{n \Downarrow succ(v_m) \quad s \Downarrow v_s}{natchoose(n,z,s) \Downarrow v_s}$$

Each of these is complemented by a pair of rules corresponding to *(eval-natrec-abort)* and *(eval-natrec-s-abort)*.

FIGURE 25. Extended Propositional Rules - Contradiction

$$\frac{\Gamma \vdash e : \bot^0}{\Gamma \vdash empty : A^0} \qquad \frac{\Gamma \vdash e : \bot^0}{\Gamma \vdash abort : A^1} \quad \textit{(abort)}$$

Notice that the new rules have been divided into *families*. Each family corresponds directly to a single inference rule in the standard system. For example, the *and* family includes the rules and^{AB}, and^{Ab}, and^{aB}, and^{ab}; it allows finer distinctions than the single conjunction-introduction rule of the standard system. This foreshadows our intent to take a derivation in the standard, unannotated system, plus a little extra information about the content desired in the overall conclusion, and automatically generate a proof in this more complex system.

In what follows, we shall refer to this as the *extended* propositional type system, as opposed to the *standard* propositional type system. There are now multiple versions of the rule for each of the connectives and later we shall give an algorithm for automatically inferring the best one to use. As was mentioned when we described the type formation rules, computational content arises from the *Nat* types The content of implications and conjunctions is a function of the content of the types from which they are composed. Truth and falsity have no content. Moreover, as we saw with the formation rules, a top-level zero-content mark indicates that all the positively-occurring subtypes must be marked as zero-content as well.

3.2 Aside – Extended Predicate Logic

In the remainder of this thesis, we shall consider propositional inference systems exclusively, until we add well-founded induction. Earlier, we showed that we could define an erasure map from dependent types to non-dependent types that almost preserved derivability. We can extend the mapping in an analogous way to map a marked predicate formula to a marked propositional formula. This induces an extended inference system for predicate logic. We shall not dwell this here, but we mention in passing that the important part of a marking on a predicate formula is the same as that on a propositional formula: the marks on the *Nat* types.

As an example, here are the induced introduction rules for the universal quantifier in an extended inference system for predicate logic:

$$\frac{\Gamma, x{:}A^{1} \vdash^{\bullet} b{:}B^{1}}{\Gamma \vdash^{\bullet} \lambda(x)b : (\forall x{:}A^{1}.B(x)^{1})^{1}}\ \forall^{AB}$$

$$\frac{\Gamma, x{:}A^{0} \vdash^{\bullet} b{:}B^{1}}{\Gamma \vdash^{\bullet} b : (\forall x{:}A^{0}.B(x)^{1})^{1}}\ \forall^{aB}$$

$$\frac{\Gamma, x{:}A \vdash^{\bullet} b{:}B^{0}}{\Gamma \vdash^{\bullet} empty : (\forall x{:}A.B(x)^{0})^{0}}\ \forall^{ab}$$

Similarly, here are the extended inference rules for introducing an existential quantifier in the induced inference system:

$$\frac{\Gamma \vdash^{\bullet} a{:}A^{1} \qquad \Gamma \vdash^{\bullet} b{:}B(a)^{1}}{\Gamma \vdash^{\bullet} \langle a,b \rangle : (\exists x{:}A^{1}.B(x)^{1})^{1}}\ \exists^{AB}$$

$$\frac{\Gamma \vdash^{\bullet} a{:}A^{1} \qquad \Gamma \vdash^{\bullet} b{:}B(a)^{0}}{\Gamma \vdash^{\bullet} a : (\exists x{:}A^{1}.B(x)^{0})^{1}}\ \exists^{Ab}$$

$$\frac{\Gamma \vdash^{\bullet} a{:}A^{0} \qquad \Gamma \vdash^{\bullet} b{:}B(a)^{1}}{\Gamma \vdash^{\bullet} b : (\exists x{:}A^{0}.B(x)^{1})^{1}}\ \exists^{aB}$$

$$\frac{\Gamma \vdash^{\bullet} a{:}A^{0} \qquad \Gamma \vdash^{\bullet} b{:}B(a)^{0}}{\Gamma \vdash^{\bullet} empty : (\exists x{:}A^{0}.B(x)^{0})^{0}}\ \exists^{ab}$$

3.3 Properties

Given a proof of a marked formula in the extended system, we can erase all the marks from the whole proof, and the result is a proof in propositional logic, as can be seen by inspection of the rules. Conversely, given a proof in propositional logic, we can mark everything as zero-content, thus giving a proof in the extended system. Given this direct correspondence between propositional logic and this extended propositional type system, we can think of zero-marked types as their logical equivalents.

A weaker result holds with respect to the standard propositional type system: given a proof therein, we can translate it into the extended system by marking everything as having content. By inspection of the inference rules, we see that the rules of the standard propositional type system are embedded herein. The converse does not necessarily hold – as we shall see momentarily, it is possible to extract *abort* in the empty context.

Lemma 11: If $\Gamma \vdash a : A^0$ then $a = empty$.

Proof: Proof by inspection of inference rules with zero-content conclusions. □

Lemma 12: Empty never appears as a subterm of a term extracted from a non-zero-content type.

Proof: Proof by inspection of the inference rules with zero-content premises. □

Thus, there appears to be no point in forming a lambda-abstraction when discharging a zero-content hypothesis. However, as we shall see, this is exactly the source of our difficulties once we introduce a strict (i.e. call-by-value) evaluator for executing our extracted programs.

The usual structural rules, like weakening and permutation of the assumptions in the context are derivable.

Lemma 13: (Substitution Preserves Types) If we have derivations of $\Gamma_1 , x{:}A , \Gamma_2 \vdash b : B$ and $\Gamma_1 \vdash a : A$ then there is a derivation of $\Gamma_1,\Gamma_2 \vdash b[a/x] : B$. In other words, substitution preserves typing.

Proof: By induction on the derivation of $\Gamma_1 , x{:}A , \Gamma_2 \vdash b : B$. The fundamental observation is that none of the inference rules is conditional on the shape of the extracted term. As a result, any derivation ending in the same type as the derivation being replaced

will allow the same inference rule to be applied, and will give a derivation of the same type. □

3.4 Pruning Causes Failure

Unfortunately, it is possible to extract *abort* from a non-zero-content formula in the empty context. For example:

$$\begin{array}{lll} x{:}A^0\,,\ y{:}\neg A^0 & \vdash & \mathit{empty} : \bot^0 \\ x{:}A^0\,,\ y{:}\neg A^0 & \vdash & \mathit{abort} : \mathit{Nat}^1 \\ x{:}A^0 & \vdash & \mathit{abort} : \neg A^0 \to \mathit{Nat}^1 \\ [] & \vdash & \mathit{abort} : A^0 \to \neg A^0 \to \mathit{Nat}^1 \end{array}$$

In this way, the extended propositional type system differs seriously from the standard system. This does not seem to have been treated in the literature.

3.4.1 Example: Predecessor

We now present a concrete example of a well-typed term which goes awry when derived in the extended type system. Recall the predecessor function:

$$\lambda n.\lambda p.\mathit{natrec}(n,\mathit{abort},(x)(y)\langle x,\mathit{empty}\rangle) : \forall(n{:}\mathit{Nat}).n{\neq}0 \to \exists(m{:}\mathit{Nat}).n{=}\mathit{succ}(m)$$

After erasing dependent types and re-deriving it in the extended system, it looks like this, recalling that negation is shorthand for an implication of falsity:

$$\lambda n.\mathit{natrec}(n,\mathit{abort},(x)(y)x) : \mathit{Nat}^1 \to (\top^0 \to \bot^0)^0 \to (\mathit{Nat}^1 \wedge \top^0)^1$$

In fact, we shall just write it as follows, leaving the dependent type erasure and inference of the marks implicit:

$$\mathit{pred} \equiv \lambda n.\mathit{natrec}(n,\mathit{abort},(x)(y)x) : \forall(n{:}\mathit{Nat}).n{\neq}0 \to \exists(m{:}\mathit{Nat}).n{=}\mathit{succ}(m)$$

From this we can derive:

$$app(pred,0) \ : 0\neq 0 \rightarrow \exists(m{:}Nat).0=succ(m)$$

But look what our strict evaluator does with this term:

$$\dfrac{pred\Downarrow\lambda n.natrec(n,abort,(x)(y)x) \quad 0\Downarrow 0 \quad \dfrac{0\Downarrow 0 \quad abort\Downarrow abort}{natrec(0,abort,(x)(y)x)\Downarrow abort}}{app(pred,0)\Downarrow abort}$$

Certainly this behavior could have been anticipated in this example by inspection of the overall type of the term. But what if we were to pass *app(pred,0)* as an argument to another function? For example, the constant function that ignores or discards its argument? The strict evaluator would fail in that case too. A lazy evaluator, on the other hand, would successfully avoid this pitfall. We must determine:

- When such behavior can occur, and
- How to prevent it, without totally giving up our pruning optimizations.

3.4.2 Discussion

In the next chapter, we will propose a two-part solution to "the abort problem". But first, it is worthwhile to mention an alternative approach: why not just make *abort* into a value, like any other? The disadvantage of this is that the representation scheme for every type now has to include a distinguished value. This is unacceptable for a similar reason to lazy evaluation: an evaluator would basically need tags to distinguish between the representation of *abort* and a legitimate value of the type in question.

Instead, the approach we shall take is to generate dummy values at the required type. This is possible because we can generate dummy values of any base type – for example, the

naturals – by simply choosing a value from the type. Moreover, we can extend this to any conjunction type by pairing together two dummy values of the constituent types, and to any functional type by lambda-abstraction over a dummy term value of the range type of the function. One disadvantage of this scheme is that we have to generate these dummy values, and then the evaluator has to evaluate them – a waste of time, but probably comparable to what a lazy evaluator would spend on building and manipulating *thunks*. The other, more serious, disadvantage is that this treatment does not immediately generalize to the more insidious cause of failure in an evaluator: nontermination.

Some authors do not even mention the possibility of rule like the *(abort)* rule in our inference system – for example, if they are working in pure System T or minimal logic. For them, the problem of *abort* does not arise. Others treat it as a value. For the terms we have presented thus far, the fact that the approach of extracting dummy values does not cope with the possibility of nonterminating subexpressions is not a problem since we can only perform structural recursion on finite data values. But later we will want to add an explicit recursion operator in a way that preserves this strong normalization property. We will find ourselves in a similar position: by removing "redundant" lambda abstractions – which actually represent preconditions – we might expose a term which causes the evaluator to fail to terminate. Then, if we pass such a term as an argument to a function, which may even try to ignore this argument, the strict evaluation order will cause non-termination of the application itself. The type system presented in the following chapter is an attempt to solve this problem without giving up all of the pruning that was achieved in this chapter.

Before attacking this problem in the next chapter, we shall finish our presentation of the extended type system by showing how to automatically generate a derivation therein from

a derivation in the standard system coupled with a little extra information: the *Nat* types in the final type that the user wishes to be marked with 1.

3.5 Automatic Annotation

Recall that the original intent of this whole proofs-as-programs paradigm is that users could write in what appears to be the traditional language of mathematical logic. Crucial to this is that the bare-bones system of inference rules mimics a natural deduction presentation of logic. But in our quest to enable the removal of redundant junk from the extracted programs, we have had to make the logic more expressive. This expressivity translates into complexity for the user, and counteracts the original advantages gained by working in a familiar language.

The good news is that the markings on types can be inferred mechanically if the user specifies what final type is desired. It is then the responsibility of the computer to make the complexity of our more subtle formulation of type theory accessible to the user. However, it is important that the system not make any judgements of its own about whether or not types ought to be marked as having content, especially when this may affect the extracted term. How much information will be needed from the user in order to determine what the extracted term will be? Exactly enough to choose the correct extended inference rule from the family which corresponds to the simple inference rule that was used in each case. How can this information be supplied? The user could explicitly provide the term they desire to be extracted at a particular step, thus removing any ambiguity about which inference rule they had in mind. Or they could provide marks on the types. The most obvious place to perform this marking is at the end of a proof. Then it becomes the system's job to back-propagate these marks throughout the proof. In the next sections we set up the problem then simplify it to the point where we can prove that there is a unique best solution.

3.5.1 Posing the Problem

Lemma 14: If $\Gamma \vdash a : A$ then $\Gamma \vdash \mathit{empty} : A^-$.

Proof: By induction on the derivation of $\Gamma \vdash a : A$. In the base case, one of the assumption reiteration rules must have been used, and it can be replaced by a use of *(squash)*. The rest of the cases follow by straightforward application of the induction hypothesis, plus the fact that each family of inference rules includes one member which accepts all zero-content premises and returns a zero-content conclusion. □

Definition 9: ($A \leq B$) We define the following simple ordering on types:

$$A \leq B \equiv A = A^- \text{ or } A = B$$

Lemma 15: If $\Gamma \vdash a : A$, for some term a, and $B \leq A$ then $\Gamma \vdash b : B$, for some term b.

Proof: A simple consequence of the preceding definition and lemma. □

Distributing this lemma over the premises of all the inference rules allows us to reformulate the rules as follows, such that if we can derive a formula/type in this system, we can derive the same formula in the extended propositional system and extract a term which is no larger. The basis of our method for automatic redundancy removal will be a transformation from a set of high-level constraints between types into an equivalent set of low-level constraints between individual marks; a proof that there is a minimum solution to these constraints; and, a simple algorithm for finding it. First, the rules.

FIGURE 26. Constrained Propositional Inference Rules

$$\frac{}{\Gamma_1\ ,\ x{:}A_1\ ,\ \Gamma_2 \vdash^* x : A_2},\ A_2 \leq A_1$$

$$\frac{\Gamma \vdash^* a : A_1 \quad \Gamma \vdash^* b : B_1}{\Gamma \vdash^* \langle a,b \rangle : A_2 \wedge B_2},\ A_2 \leq A_1,\ B_2 \leq B_1$$

$$\frac{\Gamma \vdash^* e : A_1 \wedge B}{\Gamma \vdash^* fst(e) : A_2},\ A_2 \leq A_1 \qquad \frac{\Gamma \vdash^* e : A \wedge B_1}{\Gamma \vdash^* snd(e) : B_2},\ B_2 \leq B_1$$

$$\frac{\Gamma,\ x{:}A \vdash^* b : B_1}{\Gamma \vdash^* \lambda(x)b : A \rightarrow B_2},\ B_2 \leq B_1$$

$$\frac{\Gamma \vdash^* f : A_1 \rightarrow B_1 \qquad \Gamma \vdash^* a : A_2}{\Gamma \vdash^* app(f,a) : B_2},\ B_2 \leq B_1,\ A_1 \leq A_2$$

$$\frac{}{\Gamma \vdash^* 0 : Nat^1} \qquad \frac{\Gamma \vdash^* n : Nat^M}{\Gamma \vdash^* succ(n) : Nat^N},\ Nat^N \leq Nat^M$$

$$\frac{\Gamma \vdash^* n : Nat^M \qquad \Gamma \vdash^* z : A_1 \qquad \Gamma,\ x{:}Nat^N\ ,\ y{:}A_2 \vdash^* a : A_3}{\Gamma \vdash^* natrec(n,z,(x)(y)s) : A_4^P}$$
$$P \Rightarrow M,\ Nat^N \leq Nat^M,\ A_4^P \leq A_1,\ A_4^P \leq A_3,\ A_2 \leq A_1,\ A_2 \leq A_3$$

$$\frac{}{\Gamma \vdash^* empty : \top^0} \qquad \frac{\Gamma \vdash^* e : \bot^0}{\Gamma \vdash^* abort : A}$$

The rule for induction/recursion over the naturals has one extra boolean constraint between the mark on the type of its primary premise/argument and the mark on its conclusion. Its meaning will be described in the next section. It is generated here to ensure that enough constructive content will be present in the primary input of the *natrec* to enable it to choose between two alternative sub-computations, in the case that the content of the result is desired.

Definition 10: ($a \leq b$) We define an ordering on terms:

$empty \leq a$, for any a

$x \leq x$, for any variable x

$abort \leq abort$, $0 \leq 0$

$\lambda(x)a \leq \lambda(x)b$, if $a \leq b$

$app(f,a) \leq app(g,b)$, if $f \leq g$ and $a \leq b$

$\langle a,b \rangle \leq \langle d,e \rangle$ if $a \leq d$ and $b \leq e$

$succ(a) \leq succ(b)$, $fst(a) \leq fst(b)$, $snd(a) \leq snd(b)$, if $a \leq b$

$natrec(n_1,z_1,(x)(y)s_1) \leq natrec(n_2,z_2,(x)(y)s_2)$, if $n_1 \leq n_2$, $z_1 \leq z_2$, $s_1 \leq s_2$

$a \leq \lambda(x)b$, if $a \leq b$

$f \leq app(g,a)$, if $f \leq g$

$a \leq \langle d,e \rangle$, if $a \leq d$ or $a \leq e$

Lemma 16: If $\Gamma \vdash^* a : A$ then $\Gamma \vdash b : A$ and $b \leq a$.

Proof: By induction on the derivation of $\Gamma \vdash^* a : A$, making use of the previous lemma.

□

3.5.2 Simplifying the Problem

In this section we shall re-cast the problem of solving the type constraints made by the constrained propositional inference rules. We shall demonstrate that given a derivation in the constrained system and a set of marks on the *Nat* types that the user wishes to be set to 1 there is a unique minimum solution to the transformed version of the problem.

Initially, our input is a derivation in the constrained propositional system and a set of marks which the user desires to be set to 1 – some subset of the *Nat* types appearing in the final formula/type. We now show that determining which marks in the rest of the

derivation should be set to 1 and which can be set to 0 is equivalent to the problem of solving a set of simple boolean constraints between individual marks. These constraints will take three forms: $\overline{\mathrm{M}}$, $M \Rightarrow N$, and $M \,\&\, P \Rightarrow Q$, where boolean negation, implication, and conjunction have their usual truth tables.

The first set of constraints arises simply from the type formation rules which determine the ways in which types can be legally marked. That is, for each (sub)type of the form $(A^M \wedge B^N)^P$ appearing in the derivation, add the constraints $\{M \Rightarrow P, N \Rightarrow P\}$ and for each (sub)type of the form $(A^M \rightarrow B^N)^P$ add the constraint $\{N \Rightarrow P\}$. In addition, constrain the marks of occurrences of truth and falsity to be 0. The total set of constraints is obtained by applying the following definition to every type appearing in the derivation:

$$
\begin{aligned}
\mathit{Consistency}(\mathit{Nat}^M) &\equiv \varnothing \\
\mathit{Consistency}(A^M) &\equiv \{\overline{M}\},\ A \in \{\top, \bot\} \\
\mathit{Consistency}((A^M \wedge B^N)^P) &\equiv \{M \Rightarrow P,\ N \Rightarrow P\} \cup \mathit{Consistency}(A^M) \cup \mathit{Consistency}(B^N) \\
\mathit{Consistency}((A^M \rightarrow B^N)^P) &\equiv \{N \Rightarrow P\} \cup \mathit{Consistency}(A^M) \cup \mathit{Consistency}(B^N)
\end{aligned}
$$

The second set of constraints arises from the constraints of the form $A^M \leq B^N$ generated in the process of creating a derivation from the constrained propositional inference rules. For each constraint, we have either $A^M = A^-$ or $A^M = B^N$, by definition. In the former case, we must have $M = 0$, thus if $M = 1$ then the latter case must hold and we can decompose this into the set of constraints given by the following definition:

$$
\begin{aligned}
\mathit{EQS}(A^M, A^N) &\equiv \{M{=}N\},\ A \in \{\mathit{Nat}, \top, \bot\} \\
\mathit{EQS}((A^M \wedge B^N)^P, (D^Q \wedge E^R)^S) &\equiv \{P{=}S\} \cup \mathit{EQS}(A^M, D^Q) \cup \mathit{EQS}(B^N, E^R) \\
\mathit{EQS}((A^M \rightarrow B^N)^P, (D^Q \rightarrow E^R)^S) &\equiv \{P{=}S\} \cup \mathit{EQS}(A^M, D^Q) \cup \mathit{EQS}(B^N, E^R)
\end{aligned}
$$

Which can be rewritten, equivalently, as:

$$\begin{aligned}
EQS(A^M,A^N) &\equiv \{M\Rightarrow N,\ N\Rightarrow M\},\ A \in \{Nat,\top,\bot\}\\
EQS((A^M\wedge B^N)^P,(D^Q\wedge E^R)^S) &\equiv \{P\Rightarrow S,\ S\Rightarrow P\} \cup EQS(A^M,D^Q) \cup EQS(B^N,E^R)\\
EQS((A^M\rightarrow B^N)^P,(D^Q\rightarrow E^R)^S) &\equiv \{P\Rightarrow S,\ S\Rightarrow P\} \cup EQS(A^M,D^Q) \cup EQS(B^N,E^R)
\end{aligned}$$

And since each of these holds under the assumption that $M = 1$, we make the following definition:

$$Eqs(A^M\leq B^N) \equiv \{\ M\Rightarrow(P\Rightarrow Q) \mid P\Rightarrow Q \in EQS(A^M,B^N)\ \}$$

Which can be rewritten, equivalently, as:

$$Eqs(A^M\leq B^N) \equiv \{\ (M\&P)\Rightarrow Q \mid P\Rightarrow Q \in EQS(A^M,B^N)\ \}$$

Thus, we have transformed the high-level problem of finding a solution setting the marks so that all the $A^M \leq B^N$ constraints will be satisfied to the equivalent low-level problem of finding a solution so that the union of the *Consistency* and *Eqs* constraints are satisfied. It should be clear how simple this new version of the problem is. We shall demonstrate that if a solution exists, starting from a set of Nat types in the final type which must be marked with 1, then there is a unique minimal solution.

3.5.3 Solving the Problem

To be precise, we define $Constraints \equiv Consistency \cup Eqs$. Then a set of marks set to 1, call it *Ones*, is a solution if for all $\overline{M} \in Constraints$ it is the case that $M \notin Ones$; for all $M\Rightarrow N \in Constraints$, if $M \in Ones$ then $N \in Ones$; and for all $M\&P\Rightarrow Q \in Constraints$, if $M \in Ones$ and $P \in Ones$ then $Q \in Ones$.

There is an obvious iterative algorithm for solving this problem: starting from some initial set of marks set to 1, repeatedly enlarge it by means of the preceding rules. The algorithm terminates when a fixed point is reached or a contradiction arises. Termination is

guaranteed because there are a finite number of marks in a derivation and the set of marks forced to 1 is only allowed to grow, not shrink. Certainly this algorithm finds *a* solution, but is this enough? Are there other solutions? Are there *better* solutions? No.

Lemma 17: If there are two solutions, $Ones_1$ and $Ones_2$, then there is a third, smaller, solution: $Ones_3 = Ones_1 \cap Ones_2$.
Proof: Consider arbitrary $\overline{M} \in Constraints$. Since $Ones_1$ is a solution, $M \notin Ones_1$. Thus $M \notin Ones_3 = Ones_1 \cap Ones_2$, as desired.

Consider arbitrary $M \Rightarrow N \in Constraints$. Assume $M \in Ones_3 = Ones_1 \cap Ones_2$. That is, $M \in Ones_1$ and $M \in Ones_2$. Since $Ones_1$ and $Ones_2$ are solutions, we have $N \in Ones_1$ and $N \in Ones_2$. Thus $N \in Ones_3 = Ones_1 \cap Ones_2$, as desired.

Consider arbitrary $M \& P \Rightarrow Q \in Constraints$. Assume $M, P \in Ones_3 = Ones_1 \cap Ones_2$. That is, $M, P \in Ones_1$ and $M, P \in Ones_2$. Since $Ones_1$ and $Ones_2$ are solutions, we have $Q \in Ones_1$ and $Q \in Ones_2$. Thus $Q \in Ones_3 = Ones_1 \cap Ones_2$, as desired. □

As a consequence of this fact, we can take the subset relation as the order on solutions. Thus, if we take the intersection of *all* possible solutions, we have the unique minimum solution. Furthermore, this is exactly the solution that will be found by the obvious iterative algorithm, if it exists.

Lemma 18: If $\Gamma \vdash^* a : A$ and there are two solutions, $Ones_1 \subseteq Ones_2$, to the generated constraints, then these solutions induce derivations of $\Gamma \vdash a_1 : A_1$ and $\Gamma \vdash a_2 : A_2$, respectively. Moreover, $a_1 \leq a_2$.
Proof: By induction on the derivation of $\Gamma \vdash^* a : A$. □

This concludes our demonstration of an automatic method for transforming a derivation in the standard type system into a derivation in our extended type system, given only the

desired marks on the final type – specifically, the marks on the occurrences of the *Nat* type in the final type.

Chapter 4
Delay and Force

In this chapter we update the term language and the evaluator to include two new forms: *delay* and *force*. These are degenerate versions of abstraction and application. Their purpose is to give us back some control over the order of evaluation of subterms which have purely logical – that is, zero-marked – preconditions. As we shall see, these are the types from which the extended type system of the previous chapter can extract programs which are prone to failure. The first kind of failure, where the evaluator aborts due to a type-mismatch can actually be avoided without all the extra machinery we shall introduce. In fact, if this were the only source of failure, we could be happy with just a part of the solution we shall propose – specifically, the extraction of correctly-shaped dummy terms in place of *abort* in the inference rule of the same name.

But failure also comes in the form of nontermination – the possibility of which will not arise until the next chapter where we add a well-founded induction rule from which we extract an explicitly recursive program. This is where the *delay*/*force* machinery serves a real purpose. In this chapter we show that within the improved propositional system which we shall introduce here that we have subject reduction and safety, that is, evaluation preserves types and every term evaluates to a value. This is the main original contribution of this thesis. Unfortunately, the proofs are not directly applicable in the case of the well-founded induction rule of the next chapter, because proving the termination of the program extracted from that rule appears to require that we prove subject reduction

directly for a predicate, as opposed to propositional, type system. Nevertheless, we hope that the proofs given for the propositional logic system in this chapter could serve as a very detailed guide in the construction of proofs for a system based on full predicate logic.

4.1 Updating The Evaluator

First, we enrich the syntax of terms and values to include two new forms.

FIGURE 27. Syntax - Delay and Force

$$\begin{aligned} \text{Term} ::= \quad & \cdots \mid \mathit{delay}(e) \mid \mathit{force}(e) \\ & (\mathit{where}\ \ e \in \mathit{Term}\ ) \\ \text{Value} ::= \quad & \cdots \mid \mathit{delay}(e) \\ & (\mathit{where}\ \ e \in \mathit{Term}\ ) \end{aligned}$$

Neither of these constructs binds any variables. More precisely, we extend the definition of the set of free variables as follows:

$$\begin{aligned} \mathit{FV}(\mathit{delay}(d)) &\equiv \mathit{FV}(d) \\ \mathit{FV}(\mathit{force}(e)) &\equiv \mathit{FV}(e) \end{aligned}$$

Similarly, we extend the notion of substitution:

$$\begin{aligned} \mathit{delay}(d)[a/x] &\equiv \mathit{delay}(d[a/x]) \\ \mathit{force}(e)[a/x] &\equiv \mathit{force}(e[a/x]) \end{aligned}$$

And the size ordering on terms:

$$\begin{aligned} &\mathit{delay}(a) \le \mathit{delay}(b), \mathit{force}(a) \le \mathit{force}(b), \text{ if } a \le b \\ &a \le \mathit{delay}(b),\ a \le \mathit{force}(b), \text{ if } a \le b \end{aligned}$$

FIGURE 28. The Strict Evaluator - Delay and Force

$$(\textit{eval-delay}) \qquad \frac{}{\textit{delay}(a)\Downarrow\textit{delay}(a)}$$

$$(\textit{eval-force}) \qquad \frac{a\Downarrow v_a \qquad v_a=\textit{delay}(b) \qquad b\Downarrow v_b}{\textit{force}(a)\Downarrow v_b}$$

$$(\textit{eval-force-abort}) \qquad \frac{a\Downarrow v_a \qquad v_a\neq\textit{delay}(b)}{\textit{force}(a)\Downarrow \textit{abort}}$$

Certainly, $\textit{force}(\textit{delay}(e))\approx e$ as demonstrated by:

$$\frac{\textit{delay}(e)\Downarrow\textit{delay}(e) \quad \textit{delay}(e)=\textit{delay}(e) \quad e\Downarrow v_e}{\textit{force}(\textit{delay}(e))\Downarrow v_e} \text{ iff } \frac{}{e\Downarrow v_e}$$

Also, if $a\approx b$ then $\textit{force}(a)\approx\textit{force}(b)$, as demonstrated by:

$$\frac{a\Downarrow v_a \quad v_a=\textit{delay}(d) \quad d\Downarrow v_d}{\textit{force}(a)\Downarrow v_d} \text{ iff } \frac{b\Downarrow v_b \quad v_b=\textit{delay}(d) \quad d\Downarrow v_d}{\textit{force}(b)\Downarrow v_d}, \text{ since } v_a = v_b$$

4.2 Delay and Force – Naive Version

In this section, we demonstrate how the extended rules cause safe evaluation to go astray, and introduce an obvious – naive – solution. Notice how some of the extended rules do not leave any indication of the fact that they were invoked, unlike the rules of the standard type system, whose use can always be detected by the telltale construction of the extracted program. For example, we have:

$$\frac{\Gamma,x{:}A^{\,0}\vdash b{:}B^{\,1}}{\Gamma\vdash b:(A^{\,0}\rightarrow B^{\,1})^{\,1}}\ \textit{imp}^{aB}$$

Breaking away from the pure Curry-Howard isomorphism has helped us by extracting smaller programs, but this very fact hinders us by wrecking various nice properties of the

system – for example, the fact that extracted programs would never abort. In the extended system, the following rules are *ambiguous* in the sense we have just described:

$$and^{Ab},\ and^{aB},\ fst^{Ab},\ snd^{aB},\ imp^{aB},\ app^{aB}$$

Normally, since the evaluator preserves types, it should be possible to infer that falsity was provable if a term evaluates to *abort*. But because of the ambiguous rules, we are unable to infer that the last rule applied was *(abort)*. Thus, we fail to prove falsity in the empty context, and thereby fail to contradict the soundness of propositional logic.

These ambiguous rules simply take a term from their premises and repeat it in their conclusion. Therefore, none of them can be blamed directly for putting the *abort* term into the extracted program. Once again, the only inference rule that can do that is *(abort)* itself. And, even though it may be "buried" under several applications of the ambiguous rules, it still must be there. Notice that each of the ambiguous rules (except imp^{aB}) has the same context in its premises as in its conclusion. In each of these other cases, if the conclusion is proved in the empty context then the premises must have been proved in the empty context as well. As a result, none of the ambiguous rules (except imp^{aB}) can prove falsity, without contradicting the soundness of propositional logic. So, as long as we refrain from using imp^{aB}, we get the same result as we had for the standard propositional type system: that *abort* cannot be extracted from an empty context. Therefore we conclude that because imp^{aB} allows us to add assumptions to the context without leaving a mark on the extracted program, it is this one rule that is at the heart of our difficulties.

In principle, the solution is simple: ensure that the term extracted by this rule differs in such a way that the evaluator can distinguish it and postpone its evaluation. *Delay* and *force* will do this for us. They are the degenerate versions of abstraction and application. Why not just use abstraction and application? Well, by distinguishing *delay* and *force*, we

get some simpler rules than if we were to "overload" the existing abstraction and application mechanism to serve two masters. Moreover, we will be able to identify a vital kind of peephole optimization in a more obvious way. Here is a naive version of such a set of inference rules, incorporating *delay* and *force* to ensure that the extracted terms can be evaluated safely.

FIGURE 29. Naive force/delay Rules

$$\frac{\Gamma,x{:}A \vdash b{:}B^1}{\Gamma \vdash delay(b) : (A \to B^1)^1}\ imp^{aB}$$

$$\frac{\Gamma \vdash f : (A \to B^1)^1 \qquad \Gamma \vdash a : A}{\Gamma \vdash force(f) : B^1}\ app^{aB}$$

Does this accomplish what we set out to? Certainly it gets us past the obstacle explained above. We can prove that the strict evaluator does not abort on terms extracted from well-typed derivations using these rules. Let's see how the predecessor function looks when extracted with these new rules:

$$pred \equiv \lambda n.delay(natrec(n,abort,(x)(y)x)) : \forall(n{:}Nat).n{\neq}0 \to \exists(m{:}Nat).n{=}succ(m)$$

Once again, we can try evaluating this function outside its intended domain, as shown by its type:

$$app(pred,0) \ : 0{\neq}0 \to \exists(m{:}Nat).0{=}succ(m)$$

Here's what the evaluator does with it:

$$app(pred,0) \Downarrow delay(natrec(0,abort,(x)(y)x))$$

And if we apply it to an argument for which we can also prove that the precondition holds, for example, *succ(0)*, then we can extract:

$$force(app(pred,succ(0))) \ : \exists(m{:}Nat).succ(0){=}succ(m)$$

Which evaluates, as expected:

$$force(app(pred,succ(0)))\Downarrow 0$$

In this example we see that failure has been avoided. But at what cost?

4.3 Delay and Force – Improved Version

The previous rules accomplished what they set out to: they solved the problem of unsafe evaluations. Unfortunately, they are wasteful – just like the original, unoptimized inference rules of the standard system. For example, a function abstracted over several preconditions will be wrapped under several *delay*'s, and when it is used it will be subjected to several *force*'s. In order to prevent these chains from building up, it suffices to have the inference rules keep track of what has already been delayed. The following property, defined on types, will facilitate this.

Definition 11:

$$\begin{aligned}
&\text{HasPrecondition}((D^0 \rightarrow E^1)^1) \equiv \text{true}\\
&\text{HasPrecondition}((D^1 \wedge E^0)^1) \equiv \text{HasPrecondition}(D^1)\\
&\text{HasPrecondition}((D^0 \wedge E^1)^1) \equiv \text{HasPrecondition}(E^1)\\
&\text{HasPrecondition}(A^1) \equiv \text{false, otherwise}
\end{aligned}$$

Unsuprisingly, this property is true when the extracted program has a precondition which must be true before it should be expected to operate safely. In the following rules, HasPrecondition(A) has been abbreviated to HP(A). The vital invariant that these rules will maintain (as will be demonstrated) is that if $\Gamma \vdash a : A$ is derivable, and HasPrecondition(A), and correctly-typed values have been substituted for its free variables, then a can safely be evaluated (to a *delay*, in fact).

Before proving that, however, we make a definition which will allow us to extract dummy terms of the correct shape for any application of the *(abort)* rule, without introducing explicit *abort* terms into the extracted program. This too is vital for ensuring safe evaluation under the strict evaluation regime. We make the definition:

FIGURE 30. Dummy Abort Terms

$$abort_{Nat^1} = 0$$

$$abort_{A^1 \wedge B^1} = \langle abort_A, abort_B \rangle$$

$$abort_{A^1 \wedge B^0} = abort_A$$

$$abort_{A^0 \wedge B^1} = abort_B$$

$$abort_{A^1 \to B^1} = \lambda(x) abort_B \quad , if \neg HP(B^1)$$

$$abort_{A^1 \to B^1} = \lambda(x) force(abort_B), \; if \; HP(B^1)$$

$$abort_{A^0 \to B^1} = delay(abort_B) \quad , if \neg HP(B^1)$$

$$abort_{A^0 \to B^1} = abort_B \quad , if \; HP(B^1)$$

And then we replace the existing *(abort)* rules with a version which takes advantage of this definition:

FIGURE 31. Improved Abort Rule

$$\frac{\Gamma \vdash e : \bot}{\Gamma \vdash empty : A^0} \quad \frac{\Gamma \vdash e : \bot}{\Gamma \vdash abort_{A^1} : A^1} \; abort$$

A simple inspection of the definition of $abort_A$ reveals each such term is a value, and therefore will evaluate to itself. Thus, we always have: $abort_A \Downarrow abort_A$. There is some potential for confusion in this notation; to avoid it, keep in mind that $abort_A \neq abort$.

Here now are the improved rules for dealing with the introduction and elimination of implications, that is, functions. It is here that we really make use of the definition of HasPrecondition.

FIGURE 32. Improved Implication Introduction Rules

$$\frac{\Gamma,x{:}A^{1} \vdash b{:}B^{1}}{\Gamma \vdash \lambda(x)b : (A^{1}\rightarrow B^{1})^{1}} \; imp^{AB}_{\neg HP(B^{1})}$$

$$\frac{\Gamma,x{:}A^{1} \vdash b{:}B^{1}}{\Gamma \vdash \lambda(x)(force(b)) : (A^{1}\rightarrow B^{1})^{1}} \; imp^{AB}_{HP(B^{1})}$$

$$\frac{\Gamma,x{:}A^{0} \vdash b{:}B^{1}}{\Gamma \vdash delay(b) : (A^{0}\rightarrow B^{1})^{1}} \; imp^{aB}_{\neg HP(B^{1})}$$

$$\frac{\Gamma,x{:}A^{0} \vdash b{:}B^{1}}{\Gamma \vdash b : (A^{0}\rightarrow B^{1})^{1}} \; imp^{aB}_{HP(B^{1})}$$

$$\frac{\Gamma,x{:}A \vdash b{:}B^{0}}{\Gamma \vdash empty : (A\rightarrow B)^{0}} \; imp^{ab}$$

FIGURE 33. Improved Implication Elimination Rules

$$\frac{\Gamma \vdash f : (A^{1}\rightarrow B^{1})^{1} \qquad \Gamma \vdash a : A^{1}}{\Gamma \vdash app(f,a) : B^{1}} \; app^{AB}_{\neg HP(B^{1})}$$

$$\frac{\Gamma \vdash f : (A^{1}\rightarrow B^{1})^{1} \qquad \Gamma \vdash a : A^{1}}{\Gamma \vdash delay(app(f,a)) : B^{1}} \; app^{AB}_{HP(B^{1})}$$

$$\frac{\Gamma \vdash f : (A^{0}\rightarrow B^{1})^{1} \qquad \Gamma \vdash a : A^{0}}{\Gamma \vdash force(f) : B^{1}} \; app^{aB}_{\neg HP(B^{1})}$$

$$\frac{\Gamma \vdash f : (A^{0}\rightarrow B^{1})^{1} \qquad \Gamma \vdash a : A^{0}}{\Gamma \vdash f : B^{1}} \; app^{aB}_{HP(B^{1})}$$

$$\frac{\Gamma \vdash f : (A\rightarrow B)^{0} \qquad \Gamma \vdash a : A}{\Gamma \vdash empty : B^{0}} \; app^{ab}$$

In what follows, we shall refer to this as the *improved* propositional type system, as opposed to the *standard* or *extended* systems. These rules maintain the claimed invariant,

with respect to HasPrecondition. That is, if a well-typed term has such a type, then it will evaluate to a *delay*. Moreover, these rules will create trivial redexes of the form *force(delay(e))*. We will optimize these away. Certainly we have $force(delay(e) \approx e$. Unfortunately, it is not the case that if $a \approx b$ then $e[a/x] \approx e[b/x]$, due to the fact that our evaluator does not perform any evaluation in the body of an abstraction or a *delay*. Nevertheless, for any reasonable notion of *semantic* equality between terms, we would require that two terms with the same termination properties and the same normal form (if it exists) to be equal under this notion [9].

At this point, we note that if it were not for the pair of rules imp_{HP}^{AB} and app_{HP}^{AB}, we would have the same result as for the extended system of the previous chapter: if a derivation can be remarked with a smaller set of marks, then the extracted term will be smaller as well. Why do we have these rules if they wreck this property? As demonstrated in the next section, we feel that they address an important special case which occurs when a function is abstracted over a concrete input plus one or more logical preconditions on that input. Even if this case did not turn out to be as common as we expect, we could simply eliminate the distinction between the with-precondition and without-precondition version of the rules for imp^{AB} and app^{AB}, using current the rules for $imp_{\neg HP}^{AB}$ and $app_{\neg HP}^{AB}$ in both cases. In this way we could retrieve the above-mentioned property – that the smaller the marking is, the smaller the extracted term will be.

4.4 Example: Predecessor

Now that the final version of the program extraction rules has been given, we can see what happens to the predecessor function, along with some other examples.As we saw before, the naive scheme for delaying and forcing terms negates some of the optimization

originally achieved by removing noncomputational lambda abstractions and applications. The improved version of the rules will extract:

$$pred \equiv \lambda n.force(delay(natrec(n,abort_{Nat},(x)(y)x))) : \forall(n{:}Nat).n{\neq}0{\rightarrow}\exists(m{:}Nat).n{=}succ(m)$$

where we have $abort_{Nat} = 0$.

This term seems bigger and more redundant than necessary, but it now contains an obvious force-delay redex which can be removed by our peephole optimizer. This leads us to the final version of the predecessor function:

$$pred \equiv \lambda n.natrec(n,abort_{Nat},(x)(y)x) : \forall(n{:}Nat).n{\neq}0{\rightarrow}\exists(m{:}Nat).n{=}succ(m)$$

Certainly, it is safe to evaluate this term. And, if we apply it to an argument without satisfying the precondition, the final version of the rules will ensure that the result will be delayed:

$$delay(app(pred,0)) : 0{\neq}0{\rightarrow}\exists(m{:}Nat).0{=}succ(m)$$

Similarly, if we apply it to *succ(0)*:

$$delay(app(pred,succ(0))) : succ(0){\neq}0{\rightarrow}\exists(m{:}Nat).succ(0){=}succ(m)$$

Then, if we are able to prove that the precondition is satisfied, which we can do for *succ(0)*, then we can extract:

$$force(delay(app(pred,succ(0)))) : \exists(m{:}Nat).succ(0){=}succ(m)$$

The trivial force-delay will be removed by the peephole optimizer, giving:

$$app(pred,succ(0)) : \exists(m{:}Nat).succ(0){=}succ(m)$$

Expanding the definition of *pred* gives us:

$$app((\lambda n.natrec(n,0,(x)(y)x)),succ(0)) : \exists(m{:}Nat).succ(0){=}succ(m)$$

Finally, we note that this use of *natrec* is replaced by *natpred*. The resulting term is:

$$app((\lambda n.natpred(n,0,(x)x)),succ(0)) : \exists(m{:}Nat).succ(0){=}succ(m)$$

We conclude our treatment of the predecessor function by contrasting this with the corresponding term extracted by the naive version of the *force* and *delay* rules, also making use of a dummy abort term:

$$force(app((\lambda n.delay(natrec(n,0,(x)(y)x))),succ(0))) : \exists(m{:}Nat).succ(0){=}succ(m)$$

As demonstrated in the following examples, the improved rules prevent chains of delays, chains of forces, and they can "reach inside" to remove delays from a function body.

In the improved system we have:

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1 \;,\; x{:}A^0 \;,\; y{:}B^0 \;,\; z{:}C^0 \vdash f : A^0{\to}B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1 \;,\; x{:}A^0 \;,\; y{:}B^0 \;,\; z{:}C^0 \vdash f : B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1 \;,\; x{:}A^0 \;,\; y{:}B^0 \;,\; z{:}C^0 \vdash f : C^0{\to}Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1 \;,\; x{:}A^0 \;,\; y{:}B^0 \;,\; z{:}C^0 \vdash force(f) : Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1 \;,\; x{:}A^0 \;,\; y{:}B^0 \vdash delay(force(f)) : C^0{\to}Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1 \;,\; x{:}A^0 \vdash delay(force(f)) : B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1 \vdash \begin{array}{l} delay(force(f)) \\ : A^0{\to}B^0{\to}C^0{\to}Nat^1 \end{array}$$

$$[] \vdash \begin{array}{l} \lambda(f)force(delay(force(f))) \\ : (A^0{\to}B^0{\to}C^0{\to}Nat^1) \\ {\to}A^0{\to}B^0{\to}C^0{\to}Nat^1 \end{array}$$

From which an application of the peephole optimizer gives: $\lambda(f)force(f)$.

Versus the naive system:

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^0\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash f : A^0{\to}B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^0\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash force(f) : B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^0\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash force(force(f)) : C^0{\to}Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^0\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash force(force(force(f))) : Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^0\ ,\ y{:}B^0 \vdash delay(force(force(force(f)))) : C^0{\to}Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^0 \vdash delay(delay(force(force(force(f))))) : B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^0{\to}B^0{\to}C^0{\to}Nat^1 \vdash delay(delay(delay(force(force(force(f)))))) : A^0{\to}B^0{\to}C^0{\to}Nat^1$$

$$[] \vdash \lambda(f)delay(delay(delay(force(force(force(f)))))) : (A^0{\to}B^0{\to}C^0{\to}Nat^1){\to}A^0{\to}B^0{\to}C^0{\to}Nat^1$$

Also in the improved system, we can construct:

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash delay(app(f,x)) : B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash delay(app(f,x)) : C^0{\to}Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash force(delay(app(f,x))) : Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1\ ,\ y{:}B^0 \vdash delay(force(delay(app(f,x)))) : C^0{\to}Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1 \vdash delay(force(delay(app(f,x)))) : B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1 \vdash \lambda(x)force(delay(force(delay(app(f,x))))) : A^1{\to}B^0{\to}C^0{\to}Nat^1$$

$$[] \vdash \lambda(f)\lambda(x)force(delay(force(delay(app(f,x))))) : (A^1{\to}B^0{\to}C^0{\to}Nat^1){\to}A^1{\to}B^0{\to}C^0{\to}Nat^1$$

And then we can apply to peephole optimizer to arrive at *app(f,x)*.

Here, on the other hand, is how the derivation would proceed in the naive scheme:

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash app(f,x) : B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash force(app(f,x)) : C^0{\to}Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash force(force(app(f,x))) : Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1\ ,\ y{:}B^0 \vdash delay(\ force(force(app(f,x)))\ ) : C^0{\to}Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1\ ,\ x{:}A^1 \vdash delay(delay(\ force(force(app(f,x)))))\ : B^0{\to}C^0{\to}Nat^1$$

$$f{:}A^1{\to}B^0{\to}C^0{\to}Nat^1 \vdash \lambda(x)delay(delay(\ force(force(app(f,x)))))\ : A^1{\to}B^0{\to}C^0{\to}Nat^1$$

$$[] \vdash \lambda(f)\lambda(x)delay(delay(\ force(force(app(f,x)))))\ : (A^1{\to}B^0{\to}C^0{\to}Nat^1){\to}A^1{\to}B^0{\to}C^0{\to}Nat^1$$

In this example, the improved rules will remove a chain of *delay*'s inside a function body:

$$x{:}A^1\ ,\ y{:}B^0\ ,\ z{:}C^0 \vdash 0 : Nat^1$$

$$x{:}A^1\ ,\ y{:}B^0 \vdash delay(0) : C^0{\to}Nat^1$$

$$x{:}A^1 \vdash delay(0) : B^0{\to}C^0{\to}Nat^1$$

$$[] \vdash \lambda(x)force(delay(0)) : A^1{\to}B^0{\to}C^0{\to}Nat^1$$

Again, as the final step, we apply the peephole optimizer to arrive at $\lambda(x)0$.

Contrast this with the term extracted with the naive rules:

$$
\begin{array}{ll}
x{:}A^1\ ,\ y{:}B^0\ ,\ z{:}C^0 & \vdash\ 0 : Nat^1 \\
x{:}A^1\ ,\ y{:}B^0 & \vdash\ delay(0) : C^0 \rightarrow Nat^1 \\
x{:}A^1 & \vdash\ delay(delay(0)) : B^0 \rightarrow C^0 \rightarrow Nat^1 \\
[\,] & \vdash\ \lambda(x)delay(delay(0)) : A^1 \rightarrow B^0 \rightarrow C^0 \rightarrow Nat^1
\end{array}
$$

In general, if we discharge several hypotheses in a row, the extracted term will have one lambda abstraction for each computational hypothesis extracted, plus one outermost *delay* in the case that the last hypothesis discharged was non-computational. Furthermore, if we supply such a function with its arguments and preconditions, then the result will be as many function applications as there were computational arguments, plus one outermost *delay* in the case that the resulting term still has a precondition.

We have accomplished what we desired: the extracted functions and uses thereof appear substantially similar to what would have been written by a human. Credit for this accomplishment rests in two places: making the distinction between the precondition and non-precondition types, and in forcing the body of a function before forming a lambda abstraction. Although both seem obvious in hindsight, the second is certainly much less intuitive than the first. But, by adding *force* operations to the extracted term, we create opportunities for extraction-time reductions which save us from performing run-time reductions.

4.5 Properties

Do our earlier results from the standard and extended propositional type system carry over to the improved propositional system? Yes. Not only do we have the usual properties, but we also have the advantages of the standard system – subject reduction and safety – together with the advantage of the extended system – redundancy-free terms. On the other

hand, we avoid the disadvantage of the standard system – redundant terms – and the glaring disadvantage of the extended system – failure-prone terms.

As far as the usual properties of type systems go, it is certainly the case that weakening and permutation of the assumptions are still permissible. Furthermore, substitution preserves types. That is, if Γ_1 , $x{:}A$, $\Gamma_2 \vdash b : B$ and $\Gamma_1 \vdash a : A$ then $\Gamma_1,\Gamma_2 \vdash b[a/x] : B$. It is this property that allows us to legitimately call these inference rules a type system. As for the previous systems, the proof is accomplished by a straightforward induction on the form of the derivation of Γ_1 , $x{:}A$, $\Gamma_2 \vdash b : B$.

We shall briefly reiterate the properties inherited from the extended system. *Empty* is still extracted from all zero-content types. *Empty* never appears as a subterm of a term extracted from a non-zero-content type. And, as explained earlier, the smaller the marking on a derivation, the smaller the extracted term – with the sole exception of the two rules imp_{HP}^{AB} and app_{HP}^{AB} which we argued would create redexes easily removed by a peephole optimizer.

The remainder of this chapter will be devoted to proofs that the improved system we have presented has the subject reduction and safety properties.

4.6 Subject Reduction

Definition 12: (Γ^0) A context that can contain only zero-marked assumptions will be denoted by Γ^0, as in the sequent $\Gamma^0 \vdash a : A$. That is, $\Gamma^0 \equiv x_1{:}A_1^0, \ldots , x_k{:}A_k^0$, for $k{\geq}0$.

Theorem 19: (Subject Reduction) If $\Gamma^0 \vdash a{:}A^1$ and $a{\Downarrow}v_a$ then $\Gamma^0 \vdash v_a{:}A^1$ and $v_a \neq abort$.

Proof: By case analysis on the form of a, where each case is handled by nested inductions. The outer induction is on the derivation of $a{\Downarrow}v_a$, and the inner induction is on the

derivation of $\Gamma^0 \vdash a{:}A^1$. To begin with, the following constructors evaluate to themselves and therefore subject reduction holds trivially:

$$0,\ delay(d),\ \lambda(x)e,\ abort_A$$

The simplest case is when $a = succ(n)$. Even in this simple case, the evaluation derivation could potentially end with either:

$$(eval\text{-}succ)\ \frac{n \Downarrow v_n}{succ(n) \Downarrow succ(v_n)},\ v_n \neq abort$$

Or with:

$$(eval\text{-}succ\text{-}abort)\ \frac{n \Downarrow v_n}{succ(n) \Downarrow abort},\ v_n = abort$$

What must the last line of the $\Gamma^0 \vdash succ(n){:}A^1$ derivation have been? There is one rule that fits exactly, but each of the ambiguous rules could also have ended the derivation. As a result, we cannot even conclude that $A^1{=}Nat^1$. We are forced to reason by induction on the derivation of $\Gamma^0 \vdash succ(n){:}A^1$. In the base case, the unambiguous rule was applied:

$$\frac{\Gamma^0 \vdash n : Nat^1}{\Gamma^0 \vdash succ(n) : Nat^1}$$

By the outer induction hypothesis, we get a derivation of $\Gamma^0 \vdash v_n : Nat^1$ and $v_n \neq abort$ – refuting the possibility that the evaluation ended with failure. We can construct

$$\frac{\Gamma^0 \vdash v_n : Nat^1}{\Gamma^0 \vdash succ(v_n) : Nat^1}$$

as desired. That concludes the base case of the inner induction for this case of the outer induction. Now we must inductively consider the possibility that one of the ambiguous rules was the last in the type derivation. The ambiguous rules are:

$$and^{Ab},\ and^{aB},\ fst^{Ab},\ snd^{aB},\ imp_{HP}^{aB},\ app_{HP}^{aB}$$

It is important to notice that the inner induction will not depend on the particular shape of the term under consideration. This means we can make use of the proof of the inner induction verbatim in each of the other cases of the outer induction.

The first case of the inner induction:

$$\frac{\Gamma^0 \vdash a{:}D^1 \quad \Gamma^0 \vdash e{:}E^0}{\Gamma^0 \vdash a : (D^1 \wedge E^0)^1}\ and^{Ab}$$

That is, $A^1 = D^1 \wedge E^0$. By induction we can construct

$$\frac{\Gamma^0 \vdash v_a{:}D^1 \quad \Gamma^0 \vdash e{:}E^0}{\Gamma^0 \vdash v_a : (D^1 \wedge E^0)^1}\ and^{Ab}$$

as desired. The case for the other conjunction introduction rule is symmetric. The two ambiguous conjunction elimination cases are also symmetric to one another. We consider the case where the type derivation ends with:

$$\frac{\Gamma^0 \vdash a : (A^1 \wedge B^0)^1}{\Gamma^0 \vdash a : A^1}\ fst^{Ab}$$

From which the inner induction allows us to construct:

$$\frac{\Gamma^0 \vdash v_a : (A^1 \wedge B^0)^1}{\Gamma^0 \vdash v_a : A^1}\ fst^{Ab}$$

In the case of the ambiguous implication elimination:

$$\frac{\Gamma^0 \vdash a : (B^0 \to A^1)^1 \qquad \Gamma^0 \vdash b : B^0}{\Gamma^0 \vdash a : A^1}\ app_{HP}^{aB}$$

we can apply the inner induction hypothesis, to arrive at:

$$\frac{\Gamma^0 \vdash v_a : (B^0 \to A^1)^1 \qquad \Gamma^0 \vdash b : B^0}{\Gamma^0 \vdash v_a : A^1}\ app_{HP}^{aB}$$

If the type derivation ends with an ambiguous implication introduction:

$$\frac{\Gamma^0, x{:}D^0 \vdash a{:}E^1}{\Gamma^0 \vdash a : (D^0 \to E^1)^1}\ imp_{HP}^{aB}$$

from which we infer that $A^1 = D^0 \to E^1$ and induction gives:

$$\frac{\Gamma^0, x{:}D^0 \vdash v_a{:}E^1}{\Gamma^0 \vdash v_a : (D^0 \to E^1)^1}\ imp_{HP}^{aB}$$

In this final case, it becomes clear why we did not limit the claim to completely closed terms, but instead permitted a context of zero-marked assumptions. Without this, the inner induction would not have gone through. Also, it is important that we are doing induction on the shape of the derivation, and not the shape of the sequent that is proved.

This concludes the inner induction for the case $a = succ(n)$. We shall refer to it several times in the other cases of the outer induction. In each case, we will supply the "base" case of the unambiguous rule which could end the derivation of $\Gamma^0 \vdash a{:}A^1$, and will simply refer to the "inner induction" to recall the above proof for each of the possible ambiguous rules which could end the derivation.

PAIRING: If $a = \langle d,e \rangle$ then the evaluation might end successfully with:

$$(\textit{eval-pair}) \quad \frac{d \Downarrow v_d \quad e \Downarrow v_e}{\langle d,e \rangle \Downarrow \langle v_d,v_e \rangle}, \; v_d \neq \textit{abort}, \; v_e \neq \textit{abort}$$

Or unsuccessfully with:

$$(\textit{eval-pair-abort}) \quad \frac{d \Downarrow v_d \quad e \Downarrow v_e}{\langle d,e \rangle \Downarrow \textit{abort}}, \; v_d = \textit{abort} \text{ or } v_e = \textit{abort}$$

Regardless, we have $a \Downarrow v_a$ and $b \Downarrow v_b$. The base case of the inner induction is:

$$\frac{\Gamma^0 \vdash d:D^1 \quad \Gamma^0 \vdash e:E^1}{\Gamma^0 \vdash \langle d,e \rangle : (D^1 \wedge E^1)^1} \; \textit{and}^{AB}$$

in which case we infer $A^1 = D^1 \wedge E^1$. Thus the outer induction gives us the premises necessary to construct the desired derivation:

$$\frac{\Gamma^0 \vdash v_d:D^1 \quad \Gamma^0 \vdash v_e:E^1}{\Gamma^0 \vdash \langle v_d,v_e \rangle : (D^1 \wedge E^1)^1} \; \textit{and}^{AB}$$

At the same time, it tells us that $v_d \neq \textit{abort}$ and $v_e \neq \textit{abort}$, ensuring that the evaluation ends successfully. The inner induction takes care of the ambiguous cases.

PROJECTION: The cases where the evaluation ends with a projection are symmetric. If $a = \textit{fst}(e)$ then there are two possibilities for the way the evaluation may end. Either the evaluation succeeds:

$$(\textit{eval-fst}) \quad \frac{e \Downarrow v_e \qquad v_e = \langle v_a,v_b \rangle}{\textit{fst}(e) \Downarrow v_a}$$

Or the evaluation fails:

$$(\textit{eval-fst-abort}) \quad \frac{e \Downarrow v_e \qquad v_e \neq \langle v_a, v_b \rangle}{\textit{fst}(e) \Downarrow \textit{abort}}$$

To avoid this second case, we must prove that $v_e = \langle v_a, v_b \rangle$. The base case for the inner induction is:

$$\frac{\Gamma^0 \vdash e : (A^1 \wedge B^1)^1}{\Gamma^0 \vdash \textit{fst}(e) : A^1} \; \textit{fst}^{AB}$$

From which the outer induction hypothesis tells us that there is a derivation of $\Gamma^0 \vdash v_e : (A^1 \wedge B^1)^1$. If it was inferred from contradiction, then it ends with:

$$\frac{\Gamma^0 \vdash e : \bot}{\Gamma^0 \vdash \textit{abort}_{A^1 \wedge B^1} : A^1 \wedge B^1}$$

and since $\textit{abort}_{A^1 \wedge B^1} \equiv \langle \textit{abort}_A, \textit{abort}_B \rangle$ we deduce that $v_a = \textit{abort}_A$ and we can construct

$$\frac{\Gamma^0 \vdash e : \bot}{\Gamma^0 \vdash \textit{abort}_A : A^1}$$

as desired. Otherwise, the type derivation must have ended with:

$$\frac{\Gamma^0 \vdash v_a : A^1 \qquad \Gamma^0 \vdash v_b : B^1}{\Gamma^0 \vdash \langle v_a, v_b \rangle : (A^1 \wedge B^1)^1} \; \textit{and}^{AB}$$

Thus, $v_e = \langle v_a, v_b \rangle$ and $\Gamma^0 \vdash v_a : A^1$ is derivable, as desired, and the inner induction covers the ambiguous cases.

FORCE: If $a = \textit{force}(f)$ then the evaluation may end successfully with:

$$(\textit{eval-force}) \quad \frac{f \Downarrow v_f \qquad v_f = \textit{delay}(d) \qquad d \Downarrow v_a}{\textit{force}(f) \Downarrow v_a}$$

Or unsuccessfully with:

$$(\textit{eval-force-abort})\ \frac{f \Downarrow v_f \qquad v_f \neq \textit{delay}(d)}{\textit{force}(af) \Downarrow \textit{abort}}$$

To ensure that the evaluation succeeds we must show that $v_f = \textit{delay}(d)$. In the base case of the inner induction, the type derivation must end with:

$$\frac{\Gamma^0 \vdash f : (B^0 \rightarrow A^1)^1 \qquad \Gamma^0 \vdash b : B^0}{\Gamma^0 \vdash \textit{force}(f) : A^1}\ \textit{app}^{aB}_{\neg HP}$$

from which the outer induction tells us that there is a derivation of $\Gamma^0 \vdash v_f : (B^0 \rightarrow A^1)^1$. If this derivation comes from a contradiction, we have:

$$\frac{\Gamma^0 \vdash e : \bot}{\Gamma^0 \vdash \textit{abort}_{B^0 \rightarrow A^1} : B^0 \rightarrow A^1}$$

and $\textit{abort}_{B^0 \rightarrow A^1} = \textit{delay}(\textit{abort}_A)$ (since $\neg\mathrm{HP}(\mathrm{A}^1)$) thus we deduce that $v_a = \textit{abort}_A$ and of course we can construct:

$$\frac{\Gamma^0 \vdash e : \bot}{\Gamma^0 \vdash \textit{abort}_A : A^1}$$

as desired. Otherwise, the type derivation must have ended with:

$$\frac{\Gamma^0, x{:}B^0 \vdash d{:}A^1}{\Gamma^0 \vdash \textit{delay}(d) : (B^0 \rightarrow A^1)^1}\ \textit{imp}^{aB}_{\neg HP}$$

Note: here we have to make use of the side condition concerning HasPrecondition in order to rule out the other possibility for an inference rule which extracts a *delay*.

By substitution, we construct a derivation of $\Gamma^0 \vdash d[b/x]{:}A^1$, which of course is a derivation of $\Gamma^0 \vdash d{:}A^1$ since the variable corresponding to a zero-marked assumption

does not appear free in the extracted term. Finally, the outer induction takes this and $d \Downarrow v_d$ and gives us a derivation of $\Gamma^0 \vdash v_d : A^1$. And the ambiguous cases are handled as before.

APPLICATION: If $a=app(f,b)$ the evaluation either ends successfully with:

$$(eval\text{-}app)\ \frac{f \Downarrow v_f \qquad v_f = \lambda(x)e \qquad b \Downarrow v_b \qquad e[v_b/x] \Downarrow v_a}{app(f,b) \Downarrow v_a},\ v_b \neq abort$$

Or unsuccessfully with:

$$(eval\text{-}app\text{-}abort)\ \frac{f \Downarrow v_f \quad b \Downarrow v_b}{app(f,b) \Downarrow abort},\ v_f \neq \lambda(x)e \text{ or } v_b = abort$$

But in either case, $f \Downarrow v_f$ and $b \Downarrow v_b$. To ensure that the evaluation ends successfully we must show that $v_f = \lambda(x)e$ and $v_b \neq abort$. The base case for the inner induction occurs when the type derivation ends with:

$$\frac{\Gamma^0 \vdash f : (B^1 \rightarrow A^1)^1 \qquad \Gamma^0 \vdash b : B^1}{\Gamma^0 \vdash app(f,b) : A^1}\ app^{AB}_{\rightarrow HP}$$

In this case, the outer induction tells us that there is a derivation of $\Gamma^0 \vdash v_f : (B^1 \rightarrow A^1)^1$. As in the cases for the projections and *force*, this may have been derived from a contradiction or the normal introduction rule. If it was derived from a contradiction, the derivation ends with:

$$\frac{\Gamma^0 \vdash c : \bot}{\Gamma^0 \vdash abort_{B^1 \rightarrow A^1} : B^1 \rightarrow A^1}$$

and since $abort_{B^1 \rightarrow A^1} = \lambda(x) abort_A$ we deduce that $v_a = abort_A$ and of course we can construct

$$\frac{\Gamma^0 \vdash c : \bot}{\Gamma^0 \vdash abort_A : A^1}$$

as desired. Otherwise the type derivation must have ended with:

$$\frac{\Gamma^0, x{:}B^1 \vdash e{:}A^1}{\Gamma^0 \vdash \lambda(x)e : (B^1 \rightarrow A^1)^1}\ imp^{AB}_{\neg HP}$$

Note: again in this case we have had to make use of the side condition on HasPrecondition in order to rule out another possibility. Regardless of whether the function was derived from a contradiction or not, we have $v_f = \lambda(x)e$.

The outer induction also gives a derivation of $\Gamma^0 \vdash v_b : B^1$ and $v_b \neq abort$. Thus the evaluation must have ended successfully. By substitution we can construct a derivation of $\Gamma^0 \vdash e\ [v_b/x]: A^1$. One last application of the outer induction hypothesis gives $\Gamma^0 \vdash v_a: A^1$, as desired, concluding the base case of the inner induction, which proceeds as before for the ambiguous cases.

NATREC (similarly *natpred*, *natchoose*, *natiter*): If $a=natrec(n,z,(x)(y)s)$ then we are dealing with the most complex of the cases, since it requires an additional induction on the shape of the numeral upon which the recursion is being performed. The evaluation may end successfully with either:

$$(eval\text{-}natrec\text{-}z)\ \frac{n \Downarrow 0 \quad z \Downarrow v_z}{natrec(n,z,(x)(y)s) \Downarrow v_z}$$

or with:

$$(eval\text{-}natrec\text{-}s)\ \frac{n \Downarrow succ(v_m) \quad natrec(v_m,z,(x)(y)s) \Downarrow v_r \quad s[v_m/x;v_r/y] \Downarrow v_s}{natrec(n,z,(x)(y)s) \Downarrow v_s}$$

Or unsuccessfully with either:

$$(eval\text{-}natrec\text{-}abort)\ \frac{n \Downarrow v_n}{natrec(n,z,(x)(y)s) \Downarrow abort},\ v_n \neq 0 \text{ and } v_n \neq succ(v_m)$$

or with:

$$(\textit{eval-natrec-s-abort})\ \frac{n \Downarrow succ(v_m) \quad natrec(v_m,z,(x)(y)s) \Downarrow v_r}{natrec(n,z,(x)(y)s) \Downarrow abort},\ v_r = abort$$

Nevertheless, in each of these four cases we have $n \Downarrow v_n$. To begin with, we must show that v_n=0 or v_n=$succ(v_m)$ if we are to have a hope of ensuring successful evaluation. The base case for the inner induction occurs when the type derivation ends with:

$$\frac{\Gamma^0 \vdash n:Nat^1 \qquad \Gamma^0 \vdash z:A^1 \qquad \Gamma^0,x:Nat^1,y:A^1 \vdash s:A^1}{\Gamma^0 \vdash natrec(n,z,(x)(y)s):A^1}\ natrec$$

The outer induction assures us that $v_n \neq abort$ and that there is a derivation of $\Gamma^0 \vdash v_n : Nat^1$. In the case that this derivation ends with contradiction, we have:

$$\frac{\Gamma^0 \vdash e : \bot}{\Gamma^0 \vdash abort_{Nat} : Nat^1}$$

and since $v_n = abort_{Nat} = 0$ and the only other rules which introduce naturals – *(zero)* and *(succ)* – introduce numerals, it is legitimate to perform induction on the form of the numeral v_n. At the same time, this shows that the evaluation derivation cannot end with *(eval-natrec-abort)*.

In the base case, v_n=0. Thus the evaluation ends with the *(eval-natrec-z)*. In this case, the outer induction hypothesis allows us to assert that there is a derivation of $\Gamma^0 \vdash v_z:A^1$ as desired. If $v_n = succ(v_m)$ then we must have a derivation of $\Gamma^0 \vdash v_m : Nat^1$ and we shall assume by induction that for any m such that $m \Downarrow v_m$, for which we can construct a derivations of $\Gamma^0 \vdash natrec(m,z,(x)(y)s) : A^1$ and $natrec(m,z,(x)(y)s) \Downarrow v_r$ that we can derive $\Gamma^0 \vdash v_r : A^1$ and $v_r \neq abort$. Since evaluation is the identity on values, this holds for v_m itself. The evaluation derivation could potentially end with either *(eval-natrec-s)* or *(eval-natrec-s-abort)*. In either case, $natrec(v_m,z,(x)(y)s) \Downarrow v_r$ and certainly we can derive

$\Gamma^0 \vdash natrec(v_m,z,(x)(y)s) : A^1$ which permits us to appeal to our induction hypothesis to assert that $\Gamma^0 \vdash v_r : A^1$ is derivable and $v_r \neq abort$. Thus *(eval-natrec-s-abort)* is inapplicable and the evaluation derivation must end with:

$$(eval\text{-}natrec\text{-}s)\ \frac{n \Downarrow succ(v_m) \quad natrec(v_m,z,(x)(y)s) \Downarrow v_r \quad s[v_m/x;v_r/y] \Downarrow v_s}{natrec(n,z,(x)(y)s) \Downarrow v_s},\ v_r \neq abort$$

Finally, by substitution we are able to construct a derivation of $\Gamma^0 \vdash s[v_m/x,v_r/y] : A^1$ which allows us to appeal to the outer induction hypothesis for the last time to show that there is a derivation of $\Gamma^0 \vdash v_s : A^1$ as desired. This completes the induction on v_n for the base case of the inner induction, and the ambiguous cases are taken care of.

This completes the proof of subject reduction. □

4.7 Safety

Definition 13: (Safe terms in the extended type system) Safe terms are a subset of the closed, well-typed terms which terminate and adhere to the following inductive definition on types. We define safety on zero-marked and one-marked types separately. To begin with, $safe(a{:}A^0) \equiv [] \vdash a{:}A^0$. Then, $safe(a{:}A^1) \equiv [] \vdash a{:}A^1$ and $a{\downarrow}$ and by induction on A^1:

$A^1=Nat^1$	$\equiv$	true
$A^1=D^1 \wedge E^1$	$\equiv$	safe(fst(a):D^1) and safe(snd(a):E^1)
$A^1=D^1 \wedge E^0$	$\equiv$	safe(a:D^1)
$A^1=D^0 \wedge E^1$	$\equiv$	safe(a:E^1)
$A^1=D^1 \rightarrow E^1$, $HP(E^1)$	$\equiv$	for all d, if safe(d:D^1) then safe(delay(app(a,d)):E^1)
$A^1=D^1 \rightarrow E^1$, $\neg HP(E^1)$	$\equiv$	for all d, if safe(d:D^1) then safe(app(a,d):E^1)
$A^1=D^0 \rightarrow E^1$, $HP(E^1)$	$\equiv$	for all d, if safe(d:D^0) then safe(a:E^1)
$A^1=D^0 \rightarrow E^1$, $\neg HP(E^1)$	$\equiv$	for all d, if safe(d:D^0) then safe(force(a):E^1)

Lemma 20: (Safety Preserved By Equivalence) If $safe(a_1{:}A^1)$ and $\Gamma \vdash a_2{:}A^1$ and $a_1 \approx a_2$ then $safe(a_2{:}A^1)$.

Proof: The proof is by induction on the type. To begin with, we note that $a_2\downarrow$.

If $A^1 = Nat^1$ then the fact that the second term is well-typed and terminates is sufficient.

If $A^1 = D^1 \wedge E^1$ then $safe(fst(a_1){:}D^1)$ and $safe(snd(a_1){:}E^1)$. And since $fst(a_1) \approx fst(a_2)$ and $snd(a_1) \approx snd(a_2)$, induction gives us that $safe(fst(a_2){:}D^1)$ and $safe(snd(a_2){:}E^1)$. Thus, by definition, $safe(a_2{:}D^1 \wedge E^1)$.

If $A^1 = D^1 \wedge E^0$ then $safe(a_1{:}D^1)$. By induction, $safe(a_2{:}D^1)$. Thus, $safe(a_2{:}D^1 \wedge E^0)$.

If $A^1 = D^0 \wedge E^1$ then the proof proceeds symmetrically to the previous case.

If $A^1 = D^0 \rightarrow E^1$, $HP(E^1)$ then we assume there is some d such that $safe(d{:}D^0)$. Thus, $safe(a_1{:}E^1)$ and by induction $safe(a_2{:}E^1)$. Thus $safe(a_2{:}D^0 \rightarrow E^1)$, $HP(E^1)$.

If $A^1 = D^0 \rightarrow E^1$, $\neg HP(E^1)$ then we assume there is some d such that $safe(d{:}D^0)$. Thus, $safe(force(a_1){:}E^1)$ and since $force(a_1) \approx force(a_2)$, induction gives us $safe(force(a_2){:}E^1)$. Thus $safe(a_2){:}D^0 \rightarrow E^1)$, $\neg HP(E^1)$.

If $A^1 = D^1 \rightarrow E^1$, $\neg HP(E^1)$ then we assume there is some d such that $safe(d{:}D^1)$. Thus, $safe(app(a_1,d){:}E^1)$, and since $app(a_1,d) \approx app(a_2,d)$, induction gives us $safe(app(a_2,d){:}E^1)$. Thus $safe(a_2){:}D^1 \rightarrow E^1)$, $\neg HP(E^1)$.

If $A^1 = D^1 \rightarrow E^1$, $HP(E^1)$ then the situation is more complex. As before, we begin by assuming that there is some d such that $safe(d{:}D^1)$. Thus, $safe(delay(app(a_1,d)){:}E^1)$. This is where our difficulty lies: we cannot immediately claim the desired result by induction, since it is not the case that $delay(app(a_1,d)) \approx delay(app(a_2,d))$. Therefore, we must reason by a nested induction on the shape of the type E^1.

The base case occurs when $E^1=B^0\rightarrow C^1$, $\neg HP(C^1)$. In this case, we assume there is some b such that $safe(b{:}B^0)$. Thus, $safe(force(delay(app(a_1,d))){:}C^1)$ and we are in a position to make use of the fact that $force(delay(app(a_1,d)))\approx force(delay(app(a_2,d)))$ to show that $safe(force(delay(app(a_2,d))){:}C^1)$ by appeal to the outer induction hypothesis. Thus $safe(delay(app(a_2,d)){:}B^0\rightarrow C^1)$, $\neg HP(C^1)$, as desired. This completes the base case for the inner induction.

Otherwise, if $E^1=B^0\rightarrow C^1$, $HP(C^1)$ then we assume there is some b such that $safe(b{:}B^0)$. Thus, $safe(delay(app(a_1,d)){:}C^1)$ and we appeal to the inner induction to give $safe(delay(app(a_2,d)){:}C^1)$. Thus $safe(delay(app(a_2,d)){:}B^0\rightarrow C^1)$.

If $E^1=B^1\wedge C^0$ then $safe(delay(app(a_1,d)){:}B^1)$ and the inner induction hypothesis tells us that $safe(delay(app(a_2,d)){:}B^1)$. Thus $safe(delay(app(a_2,d)){:}B^1\wedge C^0)$.

Finally, the case where $E^1=B^0\wedge C^1$ proceeds symmetrically to the previous case.

This completes the inner induction. Thus, $safe(delay(app(a_2,d)){:}E^1)$. And thus $safe(a_2{:}D^1\rightarrow E^1)$, $HP(E^1)$, as desired.

This completes the outer induction as well. □

Theorem 21: (Termination with Delay and Force) As a simply corollary to the following theorem, if $[]\vdash a{:}A^1$ then $a{\downarrow}$. □

Theorem 22: (Safety with Delay and Force) Given any well-typed term, the result of substituting (appropriately typed) safe terms for its free variables results in a safe term. That is, if $x_1{:}B_1^{M_1}, \ldots, x_k{:}B_k^{M_k}\vdash a{:}A^1$, for some k≥0 and we also have

$safe(b_i{:}B_i^{M_i})$, for all $1 \le i \le k$, then $safe(a[b_1/x_1,...,b_k/x_k]{:}A^{\,1})$, which we will abbreviate as $safe(a[\vec{b}/\vec{x}]{:}A^{\,1})$.

Proof: Begin by assuming $x_1{:}B_1^{M_1}, \ldots, x_k{:}B_k^{M_k} \vdash a{:}A^{\,1}$, for some $k \ge 0$ and $safe(b_i{:}B_i^{M_i})$, for all $1 \le i \le k$ then proceed by induction on the derivation of the type of the term, *a*. By induction, we will be able to assume that the claim holds for subderivations. Because there are so many cases to consider, we shall try to impose some structure by grouping the inference rules as follows.

First, there are the ambiguous rules, which are quite straightforward:

$$and^{Ab},\ and^{aB},\ fst^{Ab},\ snd^{aB},\ imp_{HP}^{aB},\ app_{HP}^{aB}$$

Second, there are the elimination rules for which the proof will follow directly from the definition of *safe*:

$$fst^{AB},\ snd^{AB},\ app_{HP}^{AB},\ app_{\neg HP}^{AB},\ app_{\neg HP}^{aB}$$

Third, and finally, there are the introduction rules:

$$zero,\ succ,\ and^{AB},\ imp_{\neg HP}^{aB},\ imp_{\neg HP}^{AB},\ imp_{HP}^{AB}$$

The rules in this last group vary from almost trivial to quite difficult – in the order shown.

Finally, there are three rules that do not fit very well into of the preceding categories. They are: *var*, *abort*, *natrec*.

VAR:

If the type derivation ends with:

$$\frac{}{\Gamma,\ x{:}A^{\,1},\ \Gamma \vdash x{:}A^{\,1}}\ var$$

then simple substitution assures us that $safe(x[\vec{b}/\vec{x}]{:}A^{\,1})$.

ABORT:

$$\frac{\Gamma \vdash e : \bot}{\Gamma \vdash abort_A : A^1}\ abort$$

To begin with, $abort_A[\vec{b}/\vec{x}]=abort_A$. We proceed by induction on the type.

$abort_{Nat^1}=0$ and trivially, $safe(0:Nat^1)$.

$abort_{D^1 \wedge E^1}=\langle abort_D, abort_E \rangle$ and by induction $safe(abort_D:D^1)$ and $safe(abort_E:E^1)$. Together with $fst(\langle abort_D, abort_E \rangle) \approx abort_D$ and $snd(\langle abort_D, abort_E \rangle) \approx abort_E$, this gives $safe(fst(\langle abort_D, abort_E \rangle):D^1)$ and $safe(snd(\langle abort_D, abort_E \rangle):E^1)$. Thus $safe(\langle abort_D, abort_E \rangle:D^1 \wedge E^1)$ as desired.

$abort_{D^1 \wedge E^0}=abort_D$ and by induction, $safe(abort_D:D^1)$. Thus, $safe(abort_D:D^1 \wedge E^0)$.

$abort_{D^0 \wedge E^1}=abort_E$ and the proof in this case is symmetric to the previous case.

$abort_{D^0 \to E^1}=abort_E$, *if* $HP(E^1)$ and we assume there is a d such that $safe(d:D^0)$ and by induction, $safe(abort_E:E^1)$. Thus, $safe(abort_E:D^0 \to E^1)$.

$abort_{D^0 \to E^1}=delay(abort_E)$, *if* $\neg HP(E^1)$ and we assume there is a d such that $safe(d:D^0)$ and by induction $safe(abort_E:E^1)$. Therefore, because $force(delay(abort_E)) \approx abort_E$, we have $safe(force(delay(abort_E)):E^1)$, and thus, $safe(delay(abort_E):D^0 \to E^1)$.

$abort_{D^1 \to E^1}=\lambda(x)abort_E$, *if* $\neg HP(E^1)$ and we assume there is a d such that $safe(d:D^1)$ and by induction $safe(abort_E:E^1)$. Therefore, because $app(\lambda(x)abort_E,d) \approx abort_E$, we have $safe(app(\lambda(x)abort_E,d):E^1)$, and thus, $safe(\lambda(x)abort_E:D^1 \to E^1)$.

$abort_{D^1 \to E^1} = \lambda(x) force(abort_E)$, *if* $HP(E^1)$ and this case has the same complexity as the corresponding case of the previous lemmas, requiring a nested induction on the type E^1. See that proof for details.

NATREC:

$$\frac{\Gamma \vdash n:Nat^1 \qquad \Gamma \vdash z:A^1 \qquad \Gamma,x:Nat^1,y:A^1 \vdash s:A^1}{\Gamma \vdash natrec(n,z,(x)(y)s):A^1}\ natrec$$

By the induction hypothesis, $safe(n[\vec{b}/\vec{x}]:Nat^1)$ which guarantees that there exists v_n such that $n[\vec{b}/\vec{x}] \Downarrow v_n$. Proceed by induction on the shape of this numeric value. In the base case $v_n = 0$, thus:

$$\frac{n[\vec{b}/\vec{x}] \Downarrow 0 \qquad z[\vec{b}/\vec{x}] \Downarrow v_z}{natrec(n[\vec{b}/\vec{x}],z[\vec{b}/\vec{x}],(x)(y)s[\vec{b}/\vec{x}]) \Downarrow v_z}$$

And the (outer) induction hypothesis tells us that $safe(z[\vec{b}/\vec{x}]:A^1)$, which permits us to invoke subject reduction, giving $safe(natrec(n[\vec{b}/\vec{x}],z[\vec{b}/\vec{x}],(x)(y)s[\vec{b}/\vec{x}]):A^1)$. By the definition of substitution, $natrec(n,z,(x)(y)s)[\vec{b}/\vec{x}] = natrec(n[\vec{b}/\vec{x}],z[\vec{b}/\vec{x}],(x)(y)s[\vec{b}/\vec{x}])$. Thus we have $safe(natrec(n,z,(x)(y)s)[\vec{b}/\vec{x}]:A^1)$, which completes the base case.

Otherwise, $v_n = succ(v_m)$, for some v_m, and certainly $v_m \Downarrow v_m$, and by the definition of safety, $safe(v_m:Nat^1)$. We can assume by induction that for all m such that $m \Downarrow v_m$ we have $safe(natrec(m,z,(x)(y)s)[\vec{b}/\vec{x}]:A^1)$. As we have noted, v_m is such a value, thus we may assume that $safe(natrec(v_m,z,(x)(y)s)[\vec{b}/\vec{x}]:A^1)$. As a consequence, there exists v_r such that $natrec(v_m,z,(x)(y)s)[\vec{b}/\vec{x}] \Downarrow v_r$ and thus $safe(v_r:A^1)$. Thus, by the (outer) induction hypothesis, $safe(s[\vec{b}/\vec{x},v_m/x,v_r/y]:A^1)$. Thus, there exists v_s such that $s[\vec{b}/\vec{x},v_m/x,v_r/y] \Downarrow v_s$.

Thus $safe(v_s{:}A^1)$. Now, recalling that $v_n{=}succ(v_m)$, we can construct the evaluation derivation:

$$\frac{n[\vec{b}/\vec{x}]\Downarrow succ(v_m) \quad natrec(v_m,z[\vec{b}/\vec{x}],(x)(y)s[\vec{b}/\vec{x}])\Downarrow v_r \quad s[\vec{b}/\vec{x},v_m/x,v_r/y]\Downarrow v_s}{natrec(n[\vec{b}/\vec{x}],z[\vec{b}/\vec{x}],(x)(y)s[\vec{b}/\vec{x}])\Downarrow v_s}$$

That is, $s[\vec{b}/\vec{x},v_m/x,v_r/y] \approx natrec(n[\vec{b}/\vec{x}],z[\vec{b}/\vec{x}],(x)(y)s[\vec{b}/\vec{x}])$, and as a result, $safe(natrec(n[\vec{b}/\vec{x}],z[\vec{b}/\vec{x}],(x)(y)s[\vec{b}/\vec{x}]){:}A^1)$. Finally, by the substitution $natrec(n,z,(x)(y)s)[\vec{b}/\vec{x}] = natrec(n[\vec{b}/\vec{x}],z[\vec{b}/\vec{x}],(x)(y)s[\vec{b}/\vec{x}])$, we can conclude $safe(natrec(n,z,(x)(y)s)[\vec{b}/\vec{x}]{:}A^1)$. This completes the induction on v_n.

AMBIGUOUS RULES:

If the type derivation ends with:

$$\frac{\Gamma \vdash a{:}A^1 \quad \Gamma \vdash b{:}B^0}{\Gamma \vdash a : (A^1 \wedge B^0)^1}\ and^{Ab}$$

then $safe(a[\vec{b}/\vec{x}]{:}D^1)$ by induction, and thus $safe(a[\vec{b}/\vec{x}]{:}D^1 \wedge E^0)$, as desired.

The case for the other ambiguous pairing rule is symmetric.

If the type derivation ends with:

$$\frac{\Gamma \vdash a : (A^1 \wedge B^0)^1}{\Gamma \vdash a : A^1}\ fst^{Ab}$$

then $safe(a[\vec{b}/\vec{x}]{:}A^1 \wedge B^0)$ by induction, and thus $safe(a[\vec{b}/\vec{x}]{:}A^1)$ as desired.

The case for the second ambiguous projection rule is symmetric.

If the type derivation ends with:

$$\frac{\Gamma \vdash a : (B^0 \rightarrow A^1)^1 \qquad \Gamma \vdash b : B^0}{\Gamma \vdash a : A^1} \; app_{HP}^{aB}$$

then by induction $safe(a[\vec{b}/\vec{x}]:B^0 \rightarrow A^1)$ and by the definition of safety, $safe(b:B^0)$. Putting then together gives the desired result: $safe(a[\vec{b}/\vec{x}]:A^1)$.

If the type derivation ends with:

$$\frac{\Gamma, y:D^0 \vdash a : E^1}{\Gamma \vdash a : (D^0 \rightarrow E^1)^1} \; imp_{HP}^{aB}$$

then we assume there is a d such that $safe(d:D^0)$. And induction gives us $safe(a[\vec{b}/\vec{x},d/y]:E^1)$, that is, $safe(a[\vec{b}/\vec{x}]:E^1)$. Since this holds for any such d, we have $safe(a[\vec{b}/\vec{x}]:D^0 \rightarrow E^1)$, as desired.

ELIMINATION RULES:

If the type derivation ends with:

$$\frac{\Gamma \vdash e : (A^1 \wedge B^1)^1}{\Gamma \vdash fst(e) : A^1} fst^{AB}$$

then by induction, $safe(e[\vec{b}/\vec{x}]:A^1 \wedge B^1)$, and by the definition of safety, $safe(fst(e[\vec{b}/\vec{x}]):A^1)$. Thus, a simple manipulation of substitutions, $fst(e)[\vec{b}/\vec{x}] = fst(e[\vec{b}/\vec{x}])$, gives the desired result: $safe(fst(e)[\vec{b}/\vec{x}]:A^1)$. The second projection has a symmetric proof.

If the type derivation ends with:

$$\frac{\Gamma \vdash f : (D^1 \rightarrow A^1)^1 \qquad \Gamma \vdash d : D^1}{\Gamma \vdash app(f,d) : A^1} \; app_{\neg HP}^{AB}$$

then we appeal to the induction hypothesis which tells us that $safe(f{:}D^1{\rightarrow}A^1)$, $\neg HP(A^1)$ and $safe(d[\vec{b}/\vec{x}]{:}B^1)$. By the definition of safety we know $safe(app(f[\vec{b}/\vec{x}],d[\vec{b}/\vec{x}]){:}A^1)$. By the definition of substitution, $app(f,d)[\vec{b}/\vec{x}] = app(f[\vec{b}/\vec{x}],d[\vec{b}/\vec{x}])$. Thus, $safe(app(f,d)[\vec{b}/\vec{x}]{:}A^1)$, as desired.

If the type derivation ends with:

$$\frac{\Gamma \vdash f : (D^1{\rightarrow}A^1)^1 \qquad \Gamma \vdash d : D^1}{\Gamma \vdash delay(app(f,d)) : A^1}\ app^{AB}_{HP}$$

then by induction, $safe(f[\vec{b}/\vec{x}]{:}D^1{\rightarrow}A^1)$, $HP(A^1)$ and $safe(d{:}D^1)$, thus by the definition of safety, $safe(delay(app(f[\vec{b}/\vec{x}],d[\vec{b}/\vec{x}])){:}A^1)$. By substitution $delay(app(f[\vec{b}/\vec{x}],d[\vec{b}/\vec{x}]) = delay(app(f,d))[\vec{b}/\vec{x}]$ and thus $safe(delay(app(f,d))[\vec{b}/\vec{x}]{:}A^1)$, as desired.

If the type derivation ends with:

$$\frac{\Gamma \vdash f : (D^0{\rightarrow}A^1)^1 \qquad \Gamma \vdash d : D^0}{\Gamma \vdash force(f) : A^1}\ app^{aB}_{\neg HP}$$

then the induction hypothesis gives us $safe(f[\vec{b}/\vec{x}]{:}D^0{\rightarrow}A^1)$, $\neg HP(A^1)$ and by the definition of safety, $safe(d{:}D^0)$. Putting them together gives $safe(force(f[\vec{b}/\vec{x}]){:}A^1)$ and from substitution, $force(f[\vec{b}/\vec{x}]) = force(f)[\vec{b}/\vec{x}]$, we conclude that $safe(force(f)[\vec{b}/\vec{x}]{:}A^1)$.

INTRODUCTION RULES:

We begin with trivial cases and work our way towards the more difficult cases. If the type derivation ends with:

$$\frac{}{\Gamma \vdash 0{:}Nat^1}\ zero$$

then certainly $safe(0{:}Nat^1)$ and by substitution, $0[\vec{b}/\vec{x}] = 0$, thus $safe(0[\vec{b}/\vec{x}]{:}Nat^1)$ as desired.

If the type derivation ends with:

$$\frac{\Gamma \vdash n{:}Nat^1}{\Gamma \vdash succ(n){:}Nat^1}\ succ$$

then by induction $safe(n[\vec{b}/\vec{x}]{:}Nat^1)$ which is enough to guarantee $safe(succ(n[\vec{b}/\vec{x}]){:}Nat^1)$. Finally, substitution, $succ(n[\vec{b}/\vec{x}]) = succ(n)[\vec{b}/\vec{x}]$, gives $safe(succ(n)[\vec{b}/\vec{x}]{:}Nat^1)$.

If the type derivation ends with:

$$\frac{\Gamma \vdash d{:}D^1 \quad \Gamma \vdash e{:}E^1}{\Gamma \vdash \langle d,e\rangle : (D^1 \wedge E^1)^1}\ and^{AB}$$

then by induction, $safe(d[\vec{b}/\vec{x}]{:}D^1)$ and $safe(e[\vec{b}/\vec{x}]{:}E^1)$. Also, we have $fst(\langle d[\vec{b}/\vec{x}],e[\vec{b}/\vec{x}]\rangle) \approx d[\vec{b}/\vec{x}]$ and $snd(\langle d[\vec{b}/\vec{x}],e[\vec{b}/\vec{x}]\rangle) \approx e[\vec{b}/\vec{x}]$. Thus, $safe(\langle d[\vec{b}/\vec{x}],e[\vec{b}/\vec{x}]\rangle{:}D^1 \wedge E^1)$. And by substitution, $\langle d[\vec{b}/\vec{x}],e[\vec{b}/\vec{x}]\rangle = \langle d,e\rangle[\vec{b}/\vec{x}]$, thus $safe(\langle d,e\rangle[\vec{b}/\vec{x}]{:}D^1 \wedge E^1)$.

If the type derivation ends with:

$$\frac{\Gamma,y{:}D^0 \vdash e{:}E^1}{\Gamma \vdash delay(e) : (D^0 \rightarrow E^1)^1}\ imp^{aB}_{\neg HP}$$

then we assume there is a d such that $safe(d{:}D^0)$ and thus by induction $safe(e[\vec{b}/\vec{x}]){:}E^1)$ (recall that the variable of zero-marked type does not appear free to the right of the turnstile). And since $force(delay(e[\vec{b}/\vec{x}])) \approx e[\vec{b}/\vec{x}]$ we may conclude that $safe(force(delay(e[\vec{b}/\vec{x}])){:}E^1)$. Since this is true for any such d, we have proven $safe(delay(e[\vec{b}/\vec{x}]){:}D^0 \rightarrow E^1)$. And by substitution, $delay(e[\vec{b}/\vec{x}]) = delay(e)[\vec{b}/\vec{x}]$, thus $safe(delay(e)[\vec{b}/\vec{x}]{:}D^0 \rightarrow E^1)$, as desired.

If the type derivation ends with:

$$\frac{\Gamma,y{:}D^1 \vdash e{:}E^1}{\Gamma \vdash \lambda(y)e : (D^1 \to E^1)^1}\ imp^{AB}_{\neg HP}$$

consider an arbitrary term d such that $safe(d{:}D^1)$. Therefore there exists v_d such that $d \Downarrow v_d$ and since $d \approx v_d$ we have $safe(v_d{:}D^1)$. By the induction hypothesis, $safe(e[\vec{b}/\vec{x},v_d/y]{:}E^1)$. Thus, since $e[\vec{b}/\vec{x},v_d/y] \approx app((\lambda(y)e[\vec{b}/\vec{x}]),d)$ we conclude $safe(app((\lambda(y)e[\vec{b}/\vec{x}]),d){:}E^1)$. And since this holds for any such term, d, we have shown $safe(\lambda(y)e[\vec{b}/\vec{x}]{:}D^1 \to E^1)$, from which the substitution, $\lambda(y)e[\vec{b}/\vec{x}] = (\lambda(y)e)[\vec{b}/\vec{x}]$, allows us to assert $safe((\lambda(y)e)[\vec{b}/\vec{x}]{:}D^1 \to E^1)$ as desired.

By far the most complex case occurs when the type derivation ends with:

$$\frac{\Gamma,y{:}D^1 \vdash e{:}E^1}{\Gamma \vdash \lambda(y)(force(e)) : (D^1 \to E^1)^1}\ imp^{AB}_{HP}$$

and in this case we begin by considering an arbitrary term d such that $safe(d{:}D^1)$. Therefore there exists v_d such that $d \Downarrow v_d$ and since $d \approx v_d$ we have $safe(v_d{:}D^1)$. By the induction hypothesis, $safe(e[\vec{b}/\vec{x},v_d/y]{:}E^1)$. But we get stuck here and have to perform a nested induction on the form of the type, E^1, keeping in mind the fact that $HP(E^1)$.

In the base case of the inner induction, $E^1 = B^0 \to C^1$, $\neg HP(C^1)$, thus the outer induction hypothesis tells us that $safe(e[\vec{b}/\vec{x},v_d/y]){:}B^0 \to C^1)$, $\neg HP(C^1)$. So we assume there is a term b such that $safe(b{:}B^0)$ and putting the two together gives us $safe(force(e[\vec{b}/\vec{x},v_d/y]){:}C^1)$. This prepares us to make use of the (nontrivial) equivalence, $force(delay(app(\lambda(y)(force(e[\vec{b}/\vec{x}])),d))) \approx force(e[\vec{b}/\vec{x},v_d/y]))$, from which we can conclude $safe(force(delay(app(\lambda(y)(force(e[\vec{b}/\vec{x}])),d))){:}C^1)$. And since this holds for any such b, we have $safe(delay(app(\lambda(y)(force(e[\vec{b}/\vec{x}])),d)){:}B^0 \to C^1)$ as desired. This concludes the base case of the inner induction.

If $E^1=B^0\rightarrow C^1$, $HP(C^1)$ then we assume there is a term b such that $safe(b{:}B^0)$. Thus, we can use the outer induction hypothesis to assert $safe(e[\vec{b}/\vec{x},v_d/y]{:}C^1)$ and then the inner induction gives us $safe(delay(app(\lambda(y)(force(e[\vec{b}/\vec{x}])),d)){:}C^1)$. Of course, since it holds for any such b, we have $safe(delay(app(\lambda(y)(force(e[\vec{b}/\vec{x}])),d)){:}B^0\rightarrow C^1)$.

If $E^1=B^1\wedge C^0$, $HP(B^1)$ then from the outer induction hypothesis we can conclude $safe(e[\vec{b}/\vec{x},v_d/y]{:}B^1)$. And from the inner induction hypothesis, $safe(delay(app(\lambda(y)(force(e[\vec{b}/\vec{x}])),d)){:}B^1)$. Thus, $safe(delay(app(\lambda(y)(force(e[\vec{b}/\vec{x}])),d)){:}B^1\wedge C^0)$.

If $E^1=B^0\wedge C^1$, $HP(C^1)$ then the proof is symmetric to the previous case.

This concludes the inner induction, and thus $safe(delay(app(\lambda(y)(force(e[\vec{b}/\vec{x}])),d)){:}E^1)$. Moreover, since it holds for any such d, we conclude that $safe(\lambda(y)(force(e[\vec{b}/\vec{x}])){:}D^1\rightarrow E^1)$, $HP(E^1)$. Thus, by the substitution, $\lambda(y)(force(e[\vec{b}/\vec{x}])) = (\lambda(y)(force(e)))[\vec{b}/\vec{x}]$, we can finally assert $safe((\lambda(y)(force(e)))[\vec{b}/\vec{x}]{:}D^1\rightarrow E^1)$, $HP(E^1)$.

This concludes the proof of safety. □

Chapter 5
Classical Logic
+ Explicit Recursion
= Markov's Principle

So far, we appear to have been advocating a system of program extraction where the extracted program or witness is given explicitly in each step of a proof, as part of the rules of the system. In order to claim that something exists with a particular property (constructively), one must present such an object. But there is another way to present such an object: by exhaustive search. Of course, this only works when we have somehow obtained a guarantee that an unbounded search will terminate. This is known as *Markov's Principle*, and as we shall see in the third section, it makes essential use of both the classical logic facility that will be developed in the first section, and the explicit recursion facility that will be developed in the second section. Moreover, the use of *delay* to permit safe redundancy removal will be essential in preventing nontermination in some cases.

5.1 Classical Logic

There are various guises under which we could add classical reasoning to the system. We choose to add a rule for Double Negation Elimination.

FIGURE 34. Double negation elimination

$$\frac{\Gamma \vdash e : \neg\neg(A)}{\Gamma \vdash \mathit{empty} : A^{-}} \quad (\mathit{double\text{-}negation\text{-}elim})$$

Notice that this rule can cause the automatic marking process to fail because it forces all the positively occurring marks on subtypes to be set to zero. Very little needs to be done to adapt the automatic annotation algorithm to this new possibility. We merely give a constrained version of this rule, like we did for the other rules of the standard system, and show how to translate the high-level constraint into and equivalent set of low-level boolean constraints between individual marks. Here is the constrained rule:

$$\frac{\Gamma \vdash e : \neg\neg(A_1)}{\Gamma \vdash \mathit{empty} : A_2}, \; A_2 = A_1^{-}$$

And for each high-level constraint of the form $A_2 = A_1^{-}$, we generate an equivalent set of low-level constraints by the following definition:

$$\begin{aligned}
&\mathit{Minus}(A^{M},A^{-}) && \equiv \{\overline{M}\}, A \in \{\mathit{Nat}, \top, \bot\} \\
&\mathit{Minus}((D_1^{M} \wedge E_1^{N})^{P},(D_2 \wedge E_2)^{-}) && \equiv \{\overline{P}\} \cup \mathit{Minus}(D_1^{M},D_2^{-}) \cup \mathit{Minus}(E_1^{N},E_2^{-}) \\
&\mathit{Minus}((D_1^{M} \rightarrow E_1^{N})^{P},(D_2 \rightarrow E_2)^{-}) && \equiv \{\overline{P}\} \cup \mathit{Minus}(E_1^{N},E_2^{-})
\end{aligned}$$

With the addition of this rule, we can extend our proofs-as-types analogy to classical propositional logic.

5.2 General Recursion

In order to enrich our system to handle general recursion, we add a new form for terms, *fix((x)(f)b)*, where $x, f \in \mathit{Var}$ and $b \in \mathit{Term}$. Next, we extend the evaluator to give it the expected behavior.

FIGURE 35. Well-Founded Recursion - Evaluation Rule

$$\frac{}{fix((x)(f)b) \Downarrow \lambda(x)(\ b\ [\ \lambda(y)app(fix((x)(f)b),y)\ /\ f\ ]\ )}$$

To extract programs that make use of this potentially unbounded recursion, we must add a inference rule corresponding to well-founded induction. Over the naturals, this is also known as *course-of-values* induction, or *strong* induction.

FIGURE 36. Well-Founded Induction - Inference Rule:

$$\frac{\Gamma \vdash size : A^1 \rightarrow Nat^1 \quad \Gamma,\ x{:}A^1,\ f{:}\forall(y{:}A^1).size(y){<}size(x) \rightarrow B(y) \vdash b : B(x)^1}{\Gamma \vdash fix((x)(f)b) : \forall(x{:}A^1).B(x)^1}\ (fix)$$

Note that in the interest of getting the rule to fit onto the page, we have left some uses of *app* implicit. For example, *size(x)* should read *app(size,x)*. We shall use this abbreviation throughout the rest of this section.

As part of this definition, we must define the less-than relation on the naturals. That is the purpose of the following two zero-content rules:

$$\frac{\Gamma \vdash n : Nat}{\Gamma \vdash empty : n < succ(n)} \qquad \frac{\Gamma \vdash e\ : n < m}{\Gamma \vdash empty : n < succ(m)}$$

In order to add these rules without destroying the nice properties of the existing system, we need to prove subject reduction and safety. Together, these ensure that evaluation does not abort due to type-mismatch errors and that nontermination is avoided. It does not appear to be possible to prove nontermination within the propositional framework, since our reasoning will rely on induction over the shape of the values that result from evaluating the terms that appear in the statement of the rule. That is, the proof of termination will proceed by well-founded induction at the meta-level, mirroring what is being expressed in the object language. Since we have not shown subject reduction and safety for a dependent type system, what follows can only be considered as a detailed

sketch of how one might proceed. Nevertheless, it should provide some evidence for the plausibility of our claim that we can extract explicitly recursive terms via a well-founded induction rule while preserving the subject reduction and safe evaluation properties.

First we prove that evaluation of a *fix* preserves the type derivation. Assume that the evaluation derivation ends with the *fix* rule, above. As in all the previous proof of subject reduction, it will be necessary to appeal to an inner induction to take care of the possibility that the last inference rule applied in the type derivation was one of the ambiguous rules. In the base case of the inner induction, the last rule that was applied must have been the *fix* rule. From a derivation ending in this way:

$$\Gamma^0 \vdash fix((x)(f)b) : \forall(x{:}A^1).B(x)^1$$

we can construct by weakening:

$$\Gamma^0 , y : A^1 , z : size(y){<}size(x) \vdash fix((x)(f)b) : \forall(x{:}A^1).B(x)^1$$

then by universal elimination:

$$\Gamma^0 , y : A^1 , z : size(y){<}size(x) \vdash app(fix((x)(f)b),y) : B(y)^1$$

and, finally, by discharging:

$$\Gamma^0 \vdash \lambda(y)(app(fix((x)(f)b),y)) : \forall y{:}A^1.size(y){<}size(x){\rightarrow}B(y)^1$$

And by substitution, we can construct from this together with the premise of original type derivation:

$$\Gamma^0, x{:}A^1 \vdash b[\ \lambda(y)(app(fix((x)(f)b),y))\ /f\ ] : B(x)^1$$

by discharging again, we get, as desired:

$$\Gamma^0 \vdash \lambda(x)(\ b\ [\ \lambda(y)(app(fix((x)(f)b),y))\ /f\ ]\ ) : \forall x{:}A^1.B(x)^1$$

This completes our sketch of how to extend of the proof of subject reduction to the recursion operator.

Next, we must show $safe(\ fix((x)(f)b) : \forall x{:}A^1.B(x)^1\ )$, that is:

for all a, if $safe(a{:}A^1)$ then $safe(app(fix((x)(f)b),a){:}B(a)^1)$

So we consider some a such that $safe(a{:}A^1)$. We proceed by course-of-values induction at the meta-level. We assume that for all a' such that $safe(a'{:}A^1)$ and $size(a') < size(a)$ it is the case that $safe(app(fix((x)(f)b),a'){:}B(a')^1)$. There are two technical points to note. First, by induction on the type derivation for $fix((x)(f)b)$, we know that $safe(size{:}A^1{\rightarrow}Nat^1)$ which ensures that $size(a)$ and $size(a')$ can be evaluated to numerals. Second, we require that our encoding of the less-than relation be sound, that is, if there is a derivation of $size(a') < size(a)$ then it does in fact hold.

Since we have:

$$\begin{aligned} app(fix((x)(f)b),a) &\approx app(\ \lambda(x)(\ b\ [\ \lambda(y)(app(fix((x)(f)b),y))\ /\ f\ ]\ )\ ,\ a\ ) \\ &\approx b\ [\ \lambda(y)(app(fix((x)(f)b),y))\ /\ f\ ;\ a\ /\ x\ ] \end{aligned}$$

It is sufficient to show $safe(\ b\ [\ \lambda(y)(app(fix((x)(f)b),y))\ /\ f\ ;\ a\ /\ x\ ] : B(a)^1\ )$. And by induction on the type derivation of $fix((x)(f)b)$, it is sufficient to demonstrate a and g such that $safe(a{:}A^1)$ and $safe(\ g : \forall y{:}A^1.size(y){<}size(a){\rightarrow}B(y)^1\ )$ in order to claim $safe(\ b[a/x,g/f] : B(a)^1\ )$. We already have such an a, and now we must show such a g, specifically:

$$safe(\ \lambda(y)(app(fix((x)(f)b),y)) : \forall y{:}A^1.size(y){<}size(a){\rightarrow}B(y)^1\ )$$

That is, we must show that for all a' such that $safe(a'{:}A^1)$ we have:

$$safe(\ app(\lambda(y)(app(fix((x)(f)b),y)),a') : size(a'){<}size(a){\rightarrow}B(a')^1\ )$$

That is, if $size(a') < size(a)$, then $safe(\ app(\lambda(y)(app(fix((x)(f)b),y)),a') : B(a')^1\ )$. And since $app(\lambda(y)(app(fix((x)(f)b),y)),a') \approx app(fix((x)(f)b),a')$, an appeal to the course-of-values induction hypothesis which states that $safe(app(fix((x)(f)b),a'):B(a')^1)$ is enough. This concludes our sketch of a proof that the explicitly recursive term extracted by the well-founded induction rule will in fact terminate.

5.3 Markov's Principle

We shall now see how we can combine our ability to reason classically with our ability to extract recursive programs in order to extract an unbounded search for a number with a particular property, Q, given only a classical assurance of the existence of such a number. For the purposes of this example, we shall assume that we have disjunctions available – either directly or through the standard encoding. Start by making the following definitions, in the interest of a concise presentation:

let Q be a one-place predicate on the naturals
let $H(x) \equiv \forall(w{:}Nat).w{<}x{\rightarrow}\neg Q(w)$
let $P(x) \equiv H(x){\rightarrow}\exists(z{:}Nat).Q(z)$

Here, $H(x)$ holds if there is no number smaller than x for which Q holds. P(x) will play a role analogous to that of a loop invariant, in Floyd/Hoare terminology. It asserts that if there is no smaller value of x for which Q holds, then we must be able to find one. Of course, for this to be true, we must know that a value satisfying Q exists. What is interesting is that we shall assume we know this fact classically:

$$\neg\neg\exists(x{:}Nat).Q(x)$$

And assuming that Q is a decidable predicate, i.e. we have some function *test*, such that:

$$test : \forall(x{:}Nat^1).(Q(x)^0{\vee}\neg Q(x)^0)^1$$

Recalling that disjunction is just shorthand for a tuple of a natural number with another type, we note that the marking shown on the disjunction in the above type indicates that only the value of this *tag* is provided. Then, by using the non-constructive double-negation elimination rule, we can speak of the zero-marked witness of the existential. In fact, we can use it to construct a zero-marked size function for use in the well-founded induction rule. The key idea here is that we ought to be able to prove that the predecessor of a number is smaller than that number *regardless* of what its value is. This allows us to extract:

$$MP \equiv fix((x)(f)case(app(test,x),x,app(f,(succ(x))))) : \forall(x{:}Nat^1).H(x)^0 \rightarrow \exists(y{:}Nat^1).Q(y)^0$$

Recall that *case(•,•,•)* is just a disjunction-elimination, that is, a "case-statement". In this example, it plays the role of a simple if-then-else since neither disjunct has any computational content of its own.

Now, certainly *H(0)* is provable, but we cannot prove *H(succ(0))*. Nonetheless, we can extract an unsafe term if we just use the extended rules without using *delay*s where necessary:

$$app(MP,succ(0)) : H(succ(0))^0 \rightarrow \exists(y{:}Nat^1).Q(y)^0$$

But what if Q is in fact only true of 0? Then the function will search forever. This completely violates the oft-touted advantage of total-correctness logics like type theory.

Of course, we would like to extract:

$$delay(app(MP,succ(0))) : H(succ(0))^0 \rightarrow \exists(y{:}Nat^1).Q(y)^0$$

And this is exactly what the improved *delay/force* rules will extract. We can prove *H(0)*, that is, $\forall(w{:}Nat).w{<}0 \rightarrow \neg Q(w)$. And thus we are able to obtain:

$$app(MP,0) : \exists(y{:}Nat^1).Q(y)^0$$

where the outermost force-delay has been removed by our peephole optimizer.

Chapter 6
Relation to Other Work

In this chapter we relate our results to other work in the field of type theory and the extraction of programs from proofs. In the first section, we examine the standard formulation of the so-called *subset type* whose influence appears in each of the best-known program extraction systems: Nuprl, Coq, PX, and ALF. In the second section, we say a few words about each of these systems in turn. Then we take a closer look at two nonstandard opinions: Simon Thompson's assertion that subset types – of which my half-constructive pairs/conjunctions are a generalization – are unnecessary, and Tim Griffin and Chet Murthy's investigation of an operational interpretation for classical logic. Because the approach to classical logic taken by Griffin and Murthy is so different than what preceded it, we shall go so far as to present a worked example: the law of the excluded middle. We hope that some perspective can be gained by relating our own work to the existing body of literature.

6.1 The Subset Type

A standard extension to the naive formulation of type theory is the addition of so-called *subset* types. For example, $\{\ x{:}A \mid B(x)\ \}$ indicates the subset of A such that B holds of all its elements. It is a degenerate form of the existential quantifier and thus – by dependent type erasure – of conjunction. The extracted program is only the first component of the

pair that would have been extracted from the fully-constructive existential or conjunction. That is, we have the following introduction rule:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B(a)}{\Gamma \vdash a : \{x{:}A | B(x)\}}$$

The left-hand elimination rule is straightforward:

$$\frac{\Gamma \vdash a : \{x{:}A | B(x)\}}{\Gamma \vdash a : A}$$

However, the right-hand elimination rule is more indirect, and requires side-conditions:

$$\frac{\Gamma \vdash a : \{x{:}A | B(x)\} \qquad \Gamma , x{:}A , y{:}B(x) \vdash d : D(x)}{\Gamma \vdash d[a/x] : D(a)} \quad , y \notin d, y \notin D(x)$$

Certainly, this connective is available in the extended propositional type system proposed in this thesis, via the erasure of predicate logic to propositional logic. But our system also provides the symmetric sibling: the degenerate pair where only the second component is present. Thus the system described in this thesis is more expressive than mere subset types. As an example, in our system we can write $\exists x{:}A^1.\exists y{:}B^0.D(x,y)^0$ which can be written as $\{ x{:}A^1 \mid \exists y{:}B^0.D(x,y)^0 \}$ with the subset type. In both cases, the extracted program will be an element of A. But we can also write $\exists x{:}A^0.\exists y{:}B^1.D(x,y)^0$ whose computational content is an element of B and which cannot be expressed by the subset type, due to its inherent asymmetry; computational content is permitted only on the left, not on the right. Few authors have commented on this fact, but it certainly forces the user to take more care when formulating their specification. And even if some automation is present to help with this aspect, it will be stymied by the lack of the symmetric version of this connective.

6.2 Existing Program Extraction Systems

ALF is the system under development by the Goteborg group [14]. It follows very closely to orthodox Martin Löf type theory. It uses a lazy evaluation strategy. The result of evaluating the term extracted by their counterpart to our *(abort)* rule is undefined.

Nuprl is probably the most well-known of the program extraction systems described in the literature and it continues to be actively developed. It is described in [4]. Although it has Martin Löf's type theory as its basis, quite a few extensions have been made. Subset types are supported and are used to avoid extracting programs with redundant components. The system includes an evaluator which is lazy and is described as "nondeterministic". There is a rule, corresponding to our *(abort)* rule. The evaluation of the term that is extracted from a use of this rule is not defined. Thus a strict evaluator would fail. This could be remedied in the same way that we have chosen, that is, by extracting a dummy term of the correct shape to ensure that the evaluator never "gets stuck". More seriously, Nuprl permits general recursive programs to be extracted. Again, if combined with automatic pruning, this could cause a strict evaluator to fail.

Coq is based on the Calculus of Inductive Constructions [19]. It makes a fundamental distinction between *Propositions* and *Sets*. The former correspond to our zero-marked types, and the latter to our one-marked types. However, unlike the system advocated here, Coq requires that all steps of a program derivation say explicitly which version of the various connectives are intended, and which inference rules are to be used [15]. This puts control over every minute detail of redundancy removal in the hands of the user, who must be completely aware of the exact computational content of their proof at each step. The extracted programs are Lazy-ML programs. As the name indicates, it executes programs by means of lazy evaluation.

PX is based on a notion of realizability [10]. It extracts programs in Lisp, which has a strict evaluation strategy. Thus, of the systems described here, it is the only one whose underlying execution mechanism is comparable to the one examined in this thesis. It appears to be the only system described in the literature as making use of a strict evaluator. It supports a connective which serves the same purpose as the subset type, with its attendant limitations. Takayama [20] proposes an algorithm for pruning redundant components from proofs. The algorithm appears to be aimed at the PX system. Berardi [3] comments that his pruning algorithm is better than Takayama's, since it removes redundant function formations and applications, which Takayama's does not. One may speculate that the reason Takayama did not take this extra step is because such optimizations are exactly the ones which can cause a strict evaluation strategy, such as PX's, to fail. In this sense, it is Takayama's algorithm that is better than Berardi's. Is there a middle ground? Can we optimize some redundant function abstractions and applications while retaining the ability to execute safely under a strict evaluation strategy? As we have shown in the preceding chapters, the answer is Yes.

6.3 Are Subset Types Necessary?

In Sections 7.1-7.4 of [21], an expanded version of which appears in [22], Thompson puts forth an argument about the source of computationally redundant portions of programs extracted from constructive proofs. His view is that the problem has its roots in the standard conception of a "specification" in the constructive methodology. As we have seen already, a specification for a function is usually considered to be a logical statement of the form:

$$\forall(x{:}\mathit{Input}).\exists(y{:}\mathit{Output}).R(x,y)$$

Where $R(x,y)$ describes the desired relationship between the input, x, and the output, y. Often, this will take the form of an implication which requires some preconditions to be true of the input, x, and in return guarantees some postconditions relating x and y:

$$\forall(x{:}Input).\exists(y{:}Output).Pre(x)\rightarrow Post(x,y)$$

Thompson's view is that the root of our difficulties lies in the fact that the constructed function is implicit. Instead, he proposes that a specification be an explicit existential statement, asserting the existence of a function adhering to the desired relation between inputs and outputs. For example:

$$\exists(f{:}Input\rightarrow Output).\forall(x{:}Input).Pre(x)\rightarrow Post(x,app(f,x))$$

Thompson proposes that we make use of the fact that every instance of the axiom of choice is provable in Martin Löf's type theory in order to go from a derivation ending in the former type to one ending in the latter type. However, in going from the "implicit" to the "explicit" statement we do not achieve any optimization of the extracted term – in fact, just the opposite is true: we make the term larger. For example, given a derivation ending with:

$$g : \forall(x{:}Input).\exists(y{:}Output).R(x,y)$$

an invocation of the axiom of choice derivation gives:

$$<\lambda(x)fst(app(g,x)),\lambda(x)snd(app(g,x))> : \exists(f{:}Input\rightarrow Output).\forall(x{:}Input){:}R(x,y)$$

Now that we have explicitly isolated the function that we desire, we apply the left-hand existential elimination rule to extract:

$$fst(<\lambda(x)fst(app(g,x)),\lambda(x)snd(app(g,x))>) : Input\rightarrow Output$$

Thompson realizes that these terms are still full of redundant subterms which a strict evaluator, like the one we have considered, will evaluate to normal form. But he does not appear to have been aware of a pruning algorithms such as those of Berardi and Takayama. Instead, in order to avoid having the evaluator perform wasted work at runtime, Thompson proposes lazy evaluation, and offers [16] as evidence that this is a viable alternative to strict evaluation. It is true that this reference provides a description of an implementation technique (graph reduction) which has made lazy evaluation practical. But no sooner than page 5 of [16] does Peyton-Jones explicate the matter: "strictness analysis [is] a compile-time program analysis method which has been the subject of much recent work, and which is crucial to many of the optimizations ...." This is especially revealing, as it is an observation made by one of the foremost *proponents* of lazy evaluation. Lazy evaluation is not the silver bullet that will make our programs from proofs execute efficiently. On this point, we agree with Peyton-Jones: the best way to make lazy evaluation efficient is to determine when and where it is safe to use strict evaluation instead.

Thompson is also aware of the problem of applying the axiom of choice: that the extracted term grows in size. As a solution, he proposes abstract interpretation and partial evaluation. All these analysis techniques are done after-the-fact and operate on the raw program term extracted from a proof. Since they are outside the proof system, they operate completely without benefit of any of the important, useful information contained not only the overall type of the extracted program term, but the whole derivation thereof. Basically, "abstract interpretation" is the process of trying to prove theorems about program terms. But why relegate such an important and difficult process to an outsider that must operate on a term totally out of context of its derivation? Doesn't this kind of reasoning belong as an integral part of the constructive proof system itself?

Thompson argues that the orthodox view of a specification in type theory is not explicit enough. Some might say that Thompson's view is *too* explicit, and that it negates the transparency of proofs-as-programs. Nevertheless, it is certainly the case that Thompson's style is still possible inside the system proposed in this thesis. Thus Thompson's proposal appears to be strictly weaker than the one described here.

6.4 An Operational Interpretation of Classical Reasoning

In this section, we describe an exciting new development in the ongoing process of trying to extract computational content from proofs which are not strictly intuitionistic, that is, proofs containing some classical steps. We shall examine this research direction in some detail, since the underlying idea is so different than the sort of proposals that have been made by others heretofore, most of which are of the flavor of the proposal made in this thesis.

6.4.1 Background

In [8], Griffin describes a way to give an operational meaning to certain classical reasoning steps, via an extension to lambda-calculus known as *call-with-current-continuation*, which is usually abbreviated as *call/cc*. It is an operation provided in the programming language, Scheme, which is a popular, modern dialect of Lisp. Originally, it was inspired by the so-called "throw/catch" facilities of various Lisp dialects which were invented to allow non-local control transfers, for example, to facilitate exception-handling. In the small examples we shall see here, one can also detect a strong similarity to the backtracking behavior of programs in the popular logic programming language, Prolog.

It turns out that uses of the *call/cc* operator can be simulated by a systematic translation of the program, usually known in the functional programming community as a *continuation-*

passing transformation or *CPS conversion*. The type of the transformed program turns out to be classically equivalent to the type of the original program, although it is now typable in a purely intuitionistic system. Such transformations, at the logical level, are well-known to logicians, and several have been proposed. They provide a way of embedding classical logic inside intuitionistic logic, such that if a sentence is provable classically, then some translation of it is provable intuitionistically, and furthermore, the translation is classically equivalent to the original. Typically, such translations involve adding a lot of double-negations to a formula. Thus, they are often known collectively in the logic community as *double negation translations*. In [13] Murthy gives an overview of his Ph.D. thesis work, which extended Griffin's initial work by demonstrating that several different logical transformations of this kind each induced a continuation-passing transformation on the program extracted from the proof.

To begin, we extend the term language with two "control" operators: C and A. C is an idealized version of the *call/cc* operator. A plays a subsidiary role, and can actually be defined as a degenerate version of C.

First, we must alter the usual lambda-calculus notion of reduction to be able to speak of "reduction in an evaluation context". Evaluation contexts will be written with square brackets: $E[\bullet]$. The purpose of an evaluation context is to indicate which subterm the evaluator has turned its attention on. This allows an evaluator to choose between the different redexes that may be present in a term, thus avoiding the ambiguity that might

otherwise result in a non-confluent rewrite system. Consider an evaluator with the following reduction rules:

$$E[\ (\lambda x.M)N\ ] \rightarrow_\beta\ E[\ M[N/x]\ ]$$

$$E[\ C(M)\ ] \rightarrow_C\ M(\lambda x.A(E[\ x\ ])),\ x \text{ is fresh}$$

$$E[\ A(M)\ ] \rightarrow_A\ M$$

The first rule expresses the standard function application rule familiar from lambda-calculus. Hopefully the overloaded use of square brackets is not too confusing. The second rule expresses an idealized form of *call/cc*: bundle-up the current evaluation context and pass it as an argument. The variable, x, is a fresh variable that does not appear anywhere in the left-hand side of the rule. The third rule simply indicates that the current evaluation context is to be discarded, and replaced by the indicated term. It can be defined in terms of C as:

$$A(M) \equiv C(\lambda x.M),\ x \text{ is fresh}$$

When the evaluation gets to a pair, it may choose to direct its focus to either component:

$$E[<a,b>] \rightarrow_L E'[a],\ \textit{where}\ E' = E[<\bullet,b>]$$

$$E[<a,b>] \rightarrow_R E'[b],\ \textit{where}\ E' = E[<a,\bullet>]$$

There is something unusual about *call/cc*, though: adding it to the lambda-calculus makes it non-Church-Rosser. In other words, if there is more than one way to reduce a term (i.e. it contains more than one redex), it might matter which one is chosen. For example consider the following pair of reduction sequences, both starting from the same term:

$$E[<C(\lambda x.a),C(\lambda y.b)>] \rightarrow_L E'[C(\lambda x.a)] \rightarrow_C (\lambda x.a)(\lambda z.A(E'[z])) \rightarrow_\beta a$$

$$E[<C(\lambda x.a),C(\lambda y.b)>] \rightarrow_R E''[C(\lambda y.b)] \rightarrow_C (\lambda y.b)(\lambda z.A(E''[z])) \rightarrow_\beta b$$

In both cases, the first step of the reduction sequence is to "refocus" the evaluator, by adjusting the evaluation context. In the first example, the evaluator chooses to focus its attention on the first component of the pair, and vice versa in the second example. But either redex could discard the current evaluation context and basically do whatever it wants. Thus, it is no surprise that this extension of the lambda calculus fails to be confluent.

It turns out that C can be consistently given the type of double-negation elimination, as embodied in the following type inference rule:

$$\frac{\Gamma \vdash a : \neg\neg A}{\Gamma \vdash C(a) : A} \text{ (Double Negation Elimination)}$$

(Recall that we defined $\neg A \equiv A \rightarrow \bot$, and thus $\neg\neg A \equiv (A \rightarrow \bot) \rightarrow \bot$.)

One strange thing about Griffin's proposal is that it left the overall type of the program as contradiction, i.e. $\bot$. Murthy solved this problem – placing Griffin's work on a more solid logical footing – by borrowing another trick from the logicians, in the form of an "A-translation". Basically, if we prove contradiction in a minimal logic, then we could have proved anything (i.e. in minimal logic, contradiction is simply defined to be an arbitrary formula). As a result, we can replace every occurrence of contradiction in a double-negation-translated formula by an arbitrary formula (i.e. the desired "answer type") and the resulting formula is still classically equivalent to the original.

6.4.2 Example: The Law of the Excluded Middle

In order to see this interpretation of classical logic at work, let us examine a simple example. Probably the most well-known of the classical truths which are rejected by intuitionism is the law of the excluded middle, which states that any proposition is either true or false. This is certainly acceptable if one takes the semantic function of a logical

proposition to be a total, truth-valued function. But, as we have seen, intuitionism attaches a more complex meaning to a proposition, specifically, the set of its proofs. Thus, we cannot assert that $(A \vee \neg A)$ without a proof of A or $\neg A$. (Note, of course, that $\neg\neg(A \vee \neg A)$ is provable.) But, since this is classically provable by using the double-negation elimination rule, above, we can extract a program containing *call/cc*, and examine its operational behavior. Here is a derivation of $(A \vee \neg A)$ for an arbitrary formula, A:

$$
\begin{array}{ll}
k{:}\neg(A \vee \neg A)\ ,\ x{:}A & \vdash\ x : A \\
k{:}\neg(A \vee \neg A)\ ,\ x{:}A & \vdash\ inl(x) : A \vee \neg A \\
k{:}\neg(A \vee \neg A)\ ,\ x{:}A & \vdash\ k(inl(x)) : \bot \\
k{:}\neg(A \vee \neg A) & \vdash\ \lambda x.k(inl(x)) : \neg A \\
k{:}\neg(A \vee \neg A) & \vdash\ inr(\lambda x.k(inl(x))) : A \vee \neg A \\
k{:}\neg(A \vee \neg A) & \vdash\ k(inr(\lambda x.k(inl(x)))) : \bot \\
[\,] & \vdash\ \lambda k.k(inr(\lambda x.k(inl(x)))) : \neg\neg(A \vee \neg A) \\
[\,] & \vdash\ C(\lambda k.k(inr(\lambda x.k(inl(x))))) : A \vee \neg A
\end{array}
$$

Now let us examine its operational behavior in some evaluation context, *E[]*:

$$E[C(\lambda k.k(inr(\lambda x.k(inl(x)))))]$$

$$\rightarrow_C (\lambda k.k(inr(\lambda x.k(inl(x)))))(\lambda y.A(E[y]))$$

$$\rightarrow_\beta Q(inr(\lambda x.Q(inl(x)))),\ where\ Q \equiv \lambda y.A(E[y])$$

$$\equiv (\lambda y.A(E[y]))(inr(\lambda x.Q(inl(x))))$$

$$\rightarrow_\beta A(E[inr(\lambda x.Q(inl(x)))])$$

$$\rightarrow_A E[inr(\lambda x.Q(inl(x)))]$$

Thus, *E[]* thinks it has been given a value which proves the disjunction. In fact, *E[]* is free to perform case analysis on $inr(\lambda x.Q(inl(x))) : A \vee \neg A$ in order to extract $\lambda x.Q(inl(x)) : \neg A$ and nothing unusual happens unless an attempt is made to actually use

this object, i.e. to apply it to a proof $a{:}A$ in order to obtain $\perp$. But watch what happens if it does (in this slightly changed evaluation context, $E'[]$):

$$E'[(\lambda x.Q(inl(x)))(a)]$$

$$\rightarrow_\beta E'[Q(inl(a))]$$

$$\equiv E'[(\lambda y.A(E[y]))(inl(a))]$$

$$\rightarrow_\beta E'[A(E[inl(a)]]$$

$$\rightarrow_A E[inl(a)]$$

What has happened? The new evaluation context, $E'[]$, has been discarded and the computation has "backtracked" to the point at which it "guessed" that there was an element of $\neg A$. As soon as this guess was refuted by the existence of an element a:A, the original evaluation (in the original context, $E[]$) was restarted with $inl(a){:}A \vee \neg A$. This concludes the example.

Notice the apparent asymmetry here: the first "guess" was the negated disjunct, and only once a refutation – in the form of an element/proof of the positive disjunct – was supplied did the computation backtrack. Is this negative-before-positive behavior avoidable? Yes, if we redo the proof, starting with the opposite assumption. Note however, that we will require two uses of the *call/cc* operator, not just one. And the operational behavior of the

resulting term is even more complex than the one we have just seen. Here is the derivation of the positive-before-negative version:

$$
\begin{array}{lll}
k{:}\neg(A\vee\neg A)\ ,\ x{:}\neg A & \vdash & x : \neg A \\
k{:}\neg(A\vee\neg A)\ ,\ x{:}\neg A & \vdash & inr(x) : A\vee\neg A \\
k{:}\neg(A\vee\neg A)\ ,\ x{:}\neg A & \vdash & k(inr(x)) : \bot \\
k{:}\neg(A\vee\neg A) & \vdash & \lambda x.k(inr(x)) : \neg\neg A \\
k{:}\neg(A\vee\neg A) & \vdash & C(\lambda x.k(inr(x))) : A \\
k{:}\neg(A\vee\neg A) & \vdash & inl(C(\lambda x.k(inr(x)))) : A\vee\neg A \\
k{:}\neg(A\vee\neg A) & \vdash & k(inl(C(\lambda x.k(inr(x))))) : \bot \\
[] & \vdash & \lambda k.k(inl(C(\lambda x.k(inr(x))))) : \neg\neg(A\vee\neg A) \\
[] & \vdash & C(\lambda k.k(inl(C(\lambda x.k(inr(x)))))) : A\vee\neg A
\end{array}
$$

Note the two uses of C. Now, let's see what happens when we place this term in an evaluation context, $E[]$:

$$E[C(\lambda k.k(inl(C(\lambda x.k(inr(x))))))]$$

$$\rightarrow_C (\lambda k.k(inl(C(\lambda x.k(inr(x))))))(\lambda y.A(E[y]))$$

$$\rightarrow_\beta Q(inl(C(\lambda x.Q(inr(x))))),\ where\ Q{\equiv}\lambda y.A(E[y])$$

$$\equiv (\lambda y.A(E[y]))(inl(C(\lambda x.Q(inr(x)))))$$

$$\rightarrow_\beta A(E[inl(C(\lambda x.Q(inr(x))))])$$

$$\rightarrow_A E[inl(C(\lambda x.Q(inr(x))))]$$

This time, we have $inl(C(\lambda x.Q(inr(x))))$ purporting to be an element of $A \vee \neg A$ constructed from an element of A. But what happens if we perform case analysis on this element (thus placing us in a slightly different evaluation context, $E'[\,]$)?

$$E'[C(\lambda x.Q(inr(x)))]$$

$$\rightarrow_C (\lambda x.Q(inr(x)))(\lambda z.A(E'[z]))$$

$$\rightarrow_\beta Q(inr(R)), \text{ where } R \equiv \lambda z.A(E'[z])$$

$$\equiv (\lambda y.A(E[y]))(inr(R))$$

$$\rightarrow_\beta A(E[inr(R)]$$

$$\rightarrow_A E[inr(R)]$$

As in the previous example, we have discarded the new evaluation context, $E'[\,]$, and have backtracked to the original evaluation context, $E[\,]$, in order to "guess again". Basically, what has happened is that the proof of the law of the excluded middle guarantees that there is a proof of one or the other of the disjuncts. And this version first predicts that it is the positive disjunct which is provable. But when asked to produce the element/proof, it cannot proceed and has to backtrack to the point where the prediction was made, in order to retry. At this point, it claims to have delivered an element of $A \vee \neg A$ constructed from an element of $\neg A$. As in the previous example, the evaluation context, $E[\,]$, is free to perform case analysis to extract this object and pass it around, but if it is ever used (i.e. applied to an element $a{:}A$) here is what happens (in a new evaluation context $E''[\,]$):

$$E''[R(a)]$$

$$\equiv (\lambda z.A(E'[z]))(a)$$

$$\rightarrow_\beta A(E'[a])$$

$$\rightarrow_A E'[a]$$

This time, we have discarded the latest evaluation context, *E"[]*, but instead of restarting in the original context, *E[]*, we have only backtracked to the second context, *E'[]*, which was expecting an element *a:A*. When we first arrived in this context, our proof was unable to explicitly provide such an element, and had to backtrack. But when such an object was finally provided, we returned to the context that was expecting it. This concludes the second example of extracting computational content from a classical proof of the law of the excluded middle.

6.4.3 Relationship to Continuation Passing Style

We have seen how the *call/cc* operator of Scheme can give a kind of operational interpretation of proofs which make use of the classical technique of Double Negation Elimination. However, the computations that are extracted are quite unusual. To begin with, the lambda calculus extended with the *C* operator is no longer confluent. The result of a computation may depend on the evaluation order that is used. Thus it is vital to choose an order of evaluation if we want our programs to be deterministic.

Here is an example of a continuation-passing transformation on terms in the extended calculus that does exactly that:

$$\begin{aligned}
\overline{x} &\equiv \lambda k.kx \\
\overline{\lambda x.M} &\equiv \lambda k.k(\lambda x.\overline{M}) \\
\overline{M(N)} &\equiv \lambda k.\overline{M}(\lambda m.\overline{N}(\lambda n.mnk)) \\
\overline{C(M)} &\equiv \lambda k.\overline{M}(\lambda m.m(\lambda z.\lambda d.kz)(\lambda x.x))
\end{aligned}$$

By inspection of the rules defining this translation, we see that the uses of the *C* operator have been replaced by the standard devices of abstraction and application. Thus, the resulting term can be executed by an evaluator for the standard lambda calculus. Also, we notice that all the extra "continuation" parameters have the effect of removing redexes

where they might have existed in the original term. For example, in the first rule, a variable, which may have been part of a redex, is replaced by an abstraction, the body of which contains no redexes. In fact, we will have to apply the CPS translation of the whole term to the identity function in order to create a redex and "kickstart" the evaluation process.

The whole idea of this translation may seem strange, since we have gone from a term in the extended, non-confluent lambda calculus to a term in the normal, confluent lambda calculus. How can this be? By virtue of the redex-delaying behavior just described, the transformation has fixed the order of evaluation. At every step in any evaluation of the resulting term there will be exactly one redex. Thus, it does not even matter what strategy the evaluator uses to choose redexes, since there is never any choice. Parenthetically, we note that translations of this sort make it is possible to simulate call-by-name using call-by-value, and vice versa, as originally demonstrated in [17].

This sort of embedding of the extended lambda calculus into the normal lambda calculus is mirrored in the types. When a term undergoes the CPS translation just described, its type also undergoes a translation, as defined by these rules:

$$\begin{aligned} \overline{A} &\equiv A, \text{ if } A \text{ is atomic} \\ \overline{A \to B} &\equiv \overline{A} \to (\overline{B} \to \bot) \to \bot \equiv \overline{A} \to \neg\neg\overline{B} \end{aligned}$$

And this transformation on types is also an embedding from the original type derivation in the system extended with Double Negation Elimination into the purely intuitionistic system. Moreover, the resulting type is classically equivalent to the original type. It appears that each of possible double-negation translations induces a particular evaluation order (e.g. argument before function, or function before argument; left element of pair before right element or vice versa, etc.).

6.4.4 Other Results

Another nice result of Griffin's is that we can derive the expected operational meaning of conjunction and disjunction in a "classical" system with just implication and contradiction and double-negation-elimination. In fact, this is accomplished with the standard classical definitions of conjunction and disjunction in terms implication and negation:

$$A \wedge B \equiv \neg(A \rightarrow \neg B)$$
$$A \vee B \equiv \neg A \rightarrow B$$

Moreover, if we add universal quantification, we can derive a term with the behavior we would expect from an intuitionistic existential quantifier, once again making use of the standard definition using negation:

$$\exists x.A \equiv \neg \forall x.\neg A$$

This in itself is a surprising and pleasant result: until very recently, all the literature on computational interpretations of constructive proofs has stuck very closely to the intuitionistic set of connectives. In fact, one might reasonably have drawn the conclusion that a minimal set of connectives for interpreting proofs as programs (without leaving first-order logic) would include conjunction, for example.

To conclude this brief examination of CPS translation, we mention some caveats. The translation is only valid for a limited class of sentences (specifically, the Π_2^0 sentences). Also, it appears to be dangerous to mix this "guess-and-backtrack" behavior with explicit recursion. Consider a program extracted from a classical proof of Markov's principle: given the fact that there must exist a number with a property, and the fact that every number either has the property or does not (the law of the excluded middle, proved classically) we ought to be able to exhaustively search for such a number. At each step of the search, the extracted program might "guess" that the number we are searching for

appears after the current number, without testing it for the property. We saw this kind of behavior in the examples, but in each of them we assumed that the incorrect guess was eventually contradicted, thus causing evaluation to restart from the point where the guess was made. But in the current example, the recursive search will unwind forever without ever testing any numbers for the desired property. Certainly, this is not the sort of behavior we want from our programs! This, then is the danger of "guessing": if you guess wrong, and explicit recursion is allowed, then you may non-terminate and never get a chance to backtrack to the correct alternative. Worse yet, if you redo the proof, for example, with the left-guessing version of the law of the excluded middle – instead of the right-guessing version – then your program may terminate.

In conclusion, this is a very interesting alternative to extracting programs from purely intuitionistic proofs. It is currently an area of active research. Of course, because it is so different from the approach we advocated in the previous chapter for integrating classical logic into the programs-from-proofs paradigm, it is difficult to say exactly what the relationship between the two might turn out to be. As it stands, they appear to be as different as night and day: in the approach presented in the previous chapter, proofs using classical reasoning lead to the empty program, whereas in the approach just described, they lead to programs in a non-confluent extension of the lambda calculus.

Chapter 7
Conclusion

In this chapter we recap the results of this thesis and point out the main contribution. We also describe briefly a partial implementation of the automatic annotation and program extraction system we have been advocating. Finally, we outline possible extensions of this work and directions for further investigations.

7.1 Conclusion

In this thesis we have considered the problem of extracting more realistic programs from proofs in a typed first order predicate logic. This was accomplished by developing and proving the desired properties for a version of propositional logic extended with annotations to express the computational content of the various types. These results were extended to programs extracted from predicate logic proofs by a simple erasure map. We attacked four of the most common complaints about the process of extracting programs from proofs: that the extracted programs contain redundant components, that classical reasoning is not permitted, that explicit recursion is not permitted, and that the extracted programs are typically executed with a lazy evaluation strategy.

As we discovered, these problems are intertwined. Lazy evaluation is usually used to avoid the overhead of evaluating the computationally redundant portions of the extracted programs. By removing redundant components of the extracted programs, it became possible to use classical reasoning for portions of the proof. But this makes strict

evaluation unsafe. For an inference system without well-founded induction/recursion, this problem can be solved by extracting dummy terms whose evaluation is very inexpensive and which do not interfere with the evaluation process. These dummy terms have an execution cost that is probably comparable to the overhead of the lazy evaluator which would have to create run-time data structures to represent unevaluated expressions. But as soon as we permit well-founded recursion – which provides a nice opportunity for the use of classical reasoning in its termination proof – we are again in a position where the strict evaluator may fail.

In order to ensure that the programs extracted from proofs could be executed safely by a strict evaluator, we had to make two adaptations, related to the two different ways in which evaluation can fail. The first way in which evaluation may fail is by a so-called *type-mismatch* where a sub-evaluation of one of the elimination constructs (projections, applications, and forces) results in a value of the wrong shape. This can be prevented by altering the contradiction rule to extract a dummy term of the correct shape for the type in question. The second cause of failure – non-termination – is not so easily handled. Luckily, it does not even arise unless we permit a well-founded induction rule which can extract a explicitly recursive program. On the other hand, the inability to extract such programs is often cited as a disadvantage of the usual simple presentations of the proofs-as-programs paradigm.

The main original contribution of this thesis is an improved propositional inference system which keeps track of subprograms whose execution must be delayed in order to prevent the possibility of non-termination under a strict evaluation strategy. It does this in a way that prevents chains of delays and forces from being accumulated in the extracted program. Thus, it does less optimization than Berardi's algorithm [3], but more optimization than Takayama's algorithm [20]. But unlike Berardi's algorithm, the

extracted programs can be executed safely on a strict evaluator. Of course, it has the advantage of Berardi's and Takayama's algorithms: it is automatic. In this sense, it is an improvement over manual redundancy removal techniques exemplified by the use of the subset type in other program extraction systems [1], [4], [10], [14], [15], [19].

7.2 Implementation

We have constructed, in Prolog, a system which takes as input a derivation in the standard propositional type system, extended with support for explicit disjunctions. It then fleshes out the derivation, first with variables standing for marks, and then with the high-level constraints generated by the constrained version of the inference rules. It performs the decomposition of the high-level constraints into the equivalent set of low-level boolean constraints between individual marks, as we have described. Then, starting from an initial set of marks on atomic types that are to be set to 1, it iteratively solves the constraints for the smallest solution.

Once a solution to the constraints has been calculated, there are routines that implement each of the program extraction systems we have described: the standard system, the extended system, the improved system, and the improved system with peephole optimization to remove redundant force-delay pairs. With this implementation we have accomplished more significant examples of redundancy-removal than we have demonstrated here, but considering the attention that this problem has already received from others, we have chosen to focus on the issue of safe strict evaluation. See the Examples Appendix for a sample session with our automated system.

7.3 Future Work

This thesis has examined several problems with extracting programs from proofs. The framework used was an annotated propositional logic, with conjunction, implications, and

natural numbers. Currently, disjunction is treated as a second-class citizen and is handled through a cumbersome encoding. It would be nice to support it directly. We attempted to extend our results for the propositional logic system to a dependent type system based on predicate logic system via an erasure map which preserves derivability. This approach is rather limited, however, since it does not even support the use of Peano's fourth axiom to the effect that zero is not the successor of any number. It is therefore urgent to extend our results of subject reduction and safe evaluation directly to a system of dependent types based on predicate logic. However, even once this is done the fact remains that our results are limited to first order logic and it would be nice to support higher-order quantification over predicates/types, as some systems do [4], [14], [19]. Also, it would be a good idea to extend the treatment of datatypes like the natural numbers to general inductive definitions, as are permitted by some systems [4], [19].

The above possibilities for further work are all theoretical in nature. And as with all theoretical work, the best possible extension is a practical application. The most appropriate test of the usefulness of the system described in this thesis would be to add it to an existing implementation which extracts programs from proofs.

The system of marking, pruning, delaying, and forcing that is described in this thesis bears some relation to a wider circle of work which includes work in the logic community as well as in the programming languages community. On the logical side of the fence, the content/no-content distinction that is made in this thesis is just one example of adding a notion of resource accounting to a logic. Perhaps linear types [6], relevance logic, single-threadedness, etc. could be considered as well. On the programming languages side of the fence, this thesis bears some relation to work concerning strictness analysis, dead code elimination, and partial evaluation. It is possible that the system we have advocated here –

and more generally, the techniques we have used to develop it – may have some application in these other areas.

References

[1] Backhouse, R., *et. al.* "Do-it-Yourself Type Theory". In *Formal Aspects of Computing*, 1:19-84, 1989.

[2] Beeson, M.J. *Foundations of Constructive Mathematics*. Springer-Verlag, Berlin, 1985.

[3] Berardi, S. "'Pruning' simply typed lambda-terms". Technical Report, Dipartimento di Informatica dell'Universita' di Torino (University of Turin, Italy).

[4] Constable, R.L., *et. al. Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall Inc., New Jersey, 1986.

[5] Gallier, J. *Constructive Logics. Part I: A Tutorial on Proof Systems and Typed Lambda-Calculi*. Unpublished. Available by anonymous ftp from ftp.cis.upenn.edu.

[6] Girard, J.Y., Lafont, Y., Taylor, P. *Proofs and Types*. Cambridge University Press, Cambridge, 1989.

[7] Gries, D. *The Science of Programming*. Springer-Verlag New York Inc., 1981.

[8] Griffin, T.G. "A Formulae-as-Types Notion of Control". In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

[9] Gunter, C.A. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, Cambridge Massachusetts, 1993.

[10] Hayashi, S., Nakano, H. *PX: A Computational Logic*. MIT Press, Cambridge, 1988.

[11] Howard, W.A. "The formulae-as-types notion of constructions". In*To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* J.P. Seldin and J.R. Hindley, (Eds). Academic Press, 1980.

[12] Kleene, S.C. *Introduction to Metamathematics*. Van Nostrand, 1952.

[13] Murthy, C. "Classical Proofs as Programs: How, What and Why". In *Constructivity in Computer Science*. J.P. Myers, Jr., and M.J. O'Donell (Eds). LNCS 613, 1991.

[14] Nordstrom, B., Petersson, K., Smith, J.M. *Programming in Martin-Lof's Type Theory: An Introduction*. Oxford University Press, NewYork, 1990.

[15] Paulin-Mohring, Christine. "Extracting F-omega's Programs from Proofs in the Calculus of Constructions". In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.

[16] Peyton-Jones, S. *The Implementation of Functional Programming Languages*. Prentice-Hall International (UK) Ltd, 1987.

[17] Plotkin, G. "Call-by-name, call-by-value and the lambda calculus". In *Theoretical Computer Science*, 1:125-159, 1975.

[18] Prawitz, D. *Natural Deduction: A Proof-Theoretical Study*. Almquist and Wiksell, Stockholm, 1965.

[19] Projet Formel. *The Coq Proof Assistant User's Guide, Version 5.8*. Unpublished. Available by anonymous ftp from ftp.inria.fr.

[20] Takayama, Y. Extraction of Redundancy-free Programs from Constructive Natural Deduction Proofs. In *Journal of Symbolic Computation*, 12:29-69, 1989.

[21] Thompson, S. *Type Theory and Functional Programming*. Addison-Wesley Publishers, Ltd., 1991.

[22] Thompson, S. "Are Subsets Necessary in Martin-Lof Type Theory?". In *Constructivity in Computer Science*. J.P. Myers, Jr. and M.J. O'Donell (Eds.). LNCS 613, 1991.

[23] Troelstra, A.S. "Aspects of Constructive Mathematics". Chapter D.5 in *Handbook of Mathematical Logic*. J. Barwise (Ed.) North Holland Publishing Company, 1977.

[24] Troelstra, A.S., van Dalen, D. *Constructivism in Mathematics: An Introduction*. 2 vols. Elsevier Science Publishers B.V., Amsterdam, 1988.

Appendix A
Collected Rules

Type Formation Rules - Naturals

$$\frac{}{Nat^0:Type} \qquad \frac{}{Nat^1:Type}$$

Type Formation Rules - Truth and Falsity

$$\frac{}{\top^0 : Type} \qquad \frac{}{\bot^0 : Type}$$

Type Formation Rules - Conjunction

$$\frac{A^1:Type \quad B^1:Type}{(A^1 \wedge B^1)^1:Type} \qquad \frac{A^0:Type \quad B^0:Type}{(A^0 \wedge B^0)^0:Type}$$

$$\frac{A^1:Type \quad B^0:Type}{(A^1 \wedge B^0)^1:Type} \qquad \frac{A^0:Type \quad B^1:Type}{(A^0 \wedge B^1)^1:Type}$$

Type Formation Rules - Implications

$$\frac{A^1:Type \quad B^1:Type}{(A^1 \rightarrow B^1)^1:Type} \qquad \frac{A^0:Type \quad B^0:Type}{(A^0 \rightarrow B^0)^0:Type}$$

$$\frac{A^1:Type \quad B^0:Type}{(A^1 \rightarrow B^0)^0:Type} \qquad \frac{A^0:Type \quad B^1:Type}{(A^0 \rightarrow B^1)^1:Type}$$

Context Formation Rules

$$\frac{}{[] : Context}$$

$$\frac{\Gamma : Context \quad \Gamma \vdash A:Type}{\Gamma, x:A : Context}$$

Assumptions

$$\frac{\Gamma_1,x{:}A^{\,0},\Gamma_2 : \mathit{Context}}{\Gamma_1,x{:}A^{\,0},\Gamma_2 \vdash \mathit{empty}{:}A^{\,0}} \quad (\mathit{assume})$$

$$\frac{\Gamma_1,x{:}A^{\,1},\Gamma_2 : \mathit{Context}}{\Gamma_1,x{:}A^{\,1},\Gamma_2 \vdash x{:}A^{\,1}} \quad (\mathit{var})$$

$$\frac{\Gamma_1,x{:}A^{\,1},\Gamma_2 : \mathit{Context}}{\Gamma_1,x{:}A^{\,1},\Gamma_2 \vdash \mathit{empty}{:}A^{\,-}} \quad (\mathit{squash})$$

Truth

$$\frac{}{\Gamma \vdash \mathit{empty}{:}\top^0} \quad (\mathit{truth})$$

Conjunction Introduction

$$\frac{\Gamma \vdash a{:}A^1 \qquad \Gamma \vdash b{:}B^1}{\Gamma \vdash \langle a,b\rangle : (A^1 \wedge B^1)^1} \quad \mathit{and}^{AB}$$

$$\frac{\Gamma \vdash a{:}A^1 \qquad \Gamma \vdash b{:}B^0}{\Gamma \vdash a : (A^1 \wedge B^0)^1} \quad \mathit{and}^{Ab}$$

$$\frac{\Gamma \vdash a{:}A^0 \qquad \Gamma \vdash b{:}B^1}{\Gamma \vdash b : (A^0 \wedge B^1)^1} \quad \mathit{and}^{aB}$$

$$\frac{\Gamma \vdash a{:}A^0 \qquad \Gamma \vdash b{:}B^0}{\Gamma \vdash \mathit{empty} : (A^0 \wedge B^0)^0} \quad \mathit{and}^{ab}$$

Conjunction Elimination Left

$$\frac{\Gamma \vdash e : (A^1 \wedge B^1)^1}{\Gamma \vdash fst(e) : A^1} fst^{AB}$$

$$\frac{\Gamma \vdash e : (A^1 \wedge B^0)^1}{\Gamma \vdash e : A^1} fst^{Ab}$$

$$\frac{\Gamma \vdash e : (A^0 \wedge B^1)^1}{\Gamma \vdash empty : A^0} fst^{aB}$$

$$\frac{\Gamma \vdash e : (A^0 \wedge B^0)^0}{\Gamma \vdash empty : A^0} fst^{ab}$$

Conjunction Elimination Right

$$\frac{\Gamma \vdash e : (A^1 \wedge B^1)^1}{\Gamma \vdash snd(e) : B^1} snd^{AB}$$

$$\frac{\Gamma \vdash e : (A^1 \wedge B^0)^1}{\Gamma \vdash empty : B^0} snd^{Ab}$$

$$\frac{\Gamma \vdash e : (A^0 \wedge B^1)^1}{\Gamma \vdash e : B^1} snd^{aB}$$

$$\frac{\Gamma \vdash e : (A^0 \wedge B^0)^0}{\Gamma \vdash empty : B^0} snd^{ab}$$

Implication Introduction

$$\frac{\Gamma,x{:}A^{1} \vdash b{:}B^{1}}{\Gamma \vdash \lambda(x)b : (A^{1}\rightarrow B^{1})^{1}} \; imp^{AB}_{\neg HP(B^{1})}$$

$$\frac{\Gamma,x{:}A^{1} \vdash b{:}B^{1}}{\Gamma \vdash \lambda(x)(force(b)) : (A^{1}\rightarrow B^{1})^{1}} \; imp^{AB}_{HP(B^{1})}$$

$$\frac{\Gamma,x{:}A^{0} \vdash b{:}B^{1}}{\Gamma \vdash delay(b) : (A^{0}\rightarrow B^{1})^{1}} \; imp^{aB}_{\neg HP(B^{1})}$$

$$\frac{\Gamma,x{:}A^{0} \vdash b{:}B^{1}}{\Gamma \vdash b : (A^{0}\rightarrow B^{1})^{1}} \; imp^{aB}_{HP(B^{1})}$$

$$\frac{\Gamma,x{:}A \vdash b{:}B^{0}}{\Gamma \vdash empty : (A\rightarrow B)^{0}} \; imp^{ab}$$

Implication Elimination

$$\frac{\Gamma \vdash f{:}\,(A^{1}\rightarrow B^{1})^{1} \qquad \Gamma \vdash a : A^{1}}{\Gamma \vdash app(f,a) : B^{1}} \; app^{AB}_{\neg HP(B^{1})}$$

$$\frac{\Gamma \vdash f{:}\,(A^{1}\rightarrow B^{1})^{1} \qquad \Gamma \vdash a : A^{1}}{\Gamma \vdash delay(app(f,a)) : B^{1}} \; app^{AB}_{HP(B^{1})}$$

$$\frac{\Gamma \vdash f{:}\,(A^{0}\rightarrow B^{1})^{1} \qquad \Gamma \vdash a : A^{0}}{\Gamma \vdash force(f){:}\,B^{1}} \; app^{aB}_{\neg HP(B^{1})}$$

$$\frac{\Gamma \vdash f{:}\,(A^{0}\rightarrow B^{1})^{1} \qquad \Gamma \vdash a : A^{0}}{\Gamma \vdash f{:}\,B^{1}} \; app^{aB}_{HP(B^{1})}$$

$$\frac{\Gamma \vdash f{:}\,(A\rightarrow B)^{0} \qquad \Gamma \vdash a : A}{\Gamma \vdash empty : B^{0}} \; app^{ab}$$

Natural Introduction

$$\frac{}{\Gamma \vdash 0{:}Nat^1}\ \textit{zero}$$

$$\frac{\Gamma \vdash n{:}Nat^1}{\Gamma \vdash succ(n){:}Nat^1}\ \textit{succ}$$

$$\frac{}{\Gamma \vdash empty{:}Nat^0}\ \textit{natempty}$$

Natural Elimination

$$\frac{\Gamma \vdash n{:}Nat^1 \qquad \Gamma \vdash z{:}A^1 \qquad \Gamma, x{:}Nat^1, y{:}A^1 \vdash s{:}A^1}{\Gamma \vdash natrec(n,z,(x)(y)s){:}A^1}\ \textit{natrec}$$

$$\frac{\Gamma \vdash n{:}Nat^1 \qquad \Gamma \vdash z{:}A^1 \qquad \Gamma, x{:}Nat^0, y{:}A^1 \vdash s{:}A^1}{\Gamma \vdash natiter(n,z,(y)s){:}A^1}\ \textit{natiter}$$

$$\frac{\Gamma \vdash n{:}Nat^1 \qquad \Gamma \vdash z{:}A^1 \qquad \Gamma, x{:}Nat^1, y{:}A^0 \vdash s{:}A^1}{\Gamma \vdash natpred(n,z,(x)s){:}A^1}\ \textit{natpred}$$

$$\frac{\Gamma \vdash n{:}Nat^1 \qquad \Gamma \vdash z{:}A^1 \qquad \Gamma, x{:}Nat^0, y{:}A^0 \vdash s{:}A^1}{\Gamma \vdash natchoose(n,z,s){:}A^1}\ \textit{natchoose}$$

$$\frac{\Gamma \vdash n{:}Nat^0 \qquad \Gamma \vdash z{:}A^0 \qquad \Gamma, x{:}Nat^0, y{:}A^0 \vdash s{:}A^0}{\Gamma \vdash empty{:}A^0}\ \textit{natind}$$

Contradiction

$$\frac{\Gamma \vdash e : \bot}{\Gamma \vdash empty : A^0} \qquad \frac{\Gamma \vdash e : \bot}{\Gamma \vdash abort_{A^1} : A^1}\ \textit{abort}$$

HasPrecondition

$$\begin{aligned}
&\text{HasPrecondition}((D^0 \rightarrow E^1)^1) \equiv \text{true}\\
&\text{HasPrecondition}((D^1 \wedge E^0)^1) \equiv \text{HasPrecondition}(D^1)\\
&\text{HasPrecondition}((D^0 \wedge E^1)^1) \equiv \text{HasPrecondition}(E^1)\\
&\text{HasPrecondition}(A^1) \equiv \text{false, otherwise}
\end{aligned}$$

Abort

$$abort_{Nat^1} = 0$$

$$abort_{A^1 \wedge B^1} = \langle abort_A, abort_B \rangle$$

$$abort_{A^1 \wedge B^0} = abort_A$$

$$abort_{A^0 \wedge B^1} = abort_B$$

$$abort_{A^1 \to B^1} = \lambda x.abort_B \quad , if \neg HP(B^1)$$

$$abort_{A^1 \to B^1} = \lambda x.force(abort_B) \quad , if\ HP(B^1)$$

$$abort_{A^0 \to B^1} = delay(abort_B) \quad , if \neg HP(B^1)$$

$$abort_{A^0 \to B^1} = abort_B \quad , if\ HP(B^1)$$

Double Negation Elimination

$$\frac{\Gamma \vdash e : \neg\neg(A)}{\Gamma \vdash empty : A^-} \quad (double\text{-}negation\text{-}elim)$$

Well-Founded Induction

$$\frac{\Gamma \vdash size : A^1 \to Nat^1 \quad \Gamma, x{:}A^1, f{:}\forall(y{:}A^1).size(y) < size(x) \to B(y) \vdash b : B(x)^1}{\Gamma \vdash fix((x)(f)b) : \forall(x{:}A^1).B(x)^1} \quad (fix)$$

Where the less-than relation on naturals is defined thus:

$$\frac{\Gamma \vdash n : Nat}{\Gamma \vdash empty : n < succ(n)} \qquad \frac{\Gamma \vdash e : n < m}{\Gamma \vdash empty : n < succ(m)}$$

Appendix B
Examples

We provide this key for interpreting the terms as they appear in the following sample session with our prototype tool for automatic proof annotation and program extraction:

`x`	is x
`lam(`x`,`e`)`	is $\lambda(x)e$
`app(`f`,`a`)`	is *app*(f,a)
`pair(`a`,`b`)`	is <a,b>
`fst(`e`)`	is *fst*(e)
`snd(`e`)`	is *snd*(e)
`empty`	is *empty*
`abort`	is *abort*
`0`	is 0
`s(`n`)`	is *succ*(n)
`natrec(`n`,`z`,`x`,`y`,`s`)`	is *natrec*(n,z,(x)(y)s)
`inl(`e`)`	is *inl*(e)
`inr(`e`)`	is *inr*(e)
`case(`e`,`x`,`f`,`y`,`g`)`	is *case*(e,(x)f,(y)g)

In addition, for the examples making use of disjunction, there are degenerate versions of the left and right injections, written as `left` and `right`, respectively. These play the same role as the boolean true/false values of most programming languages. Applied to either of them, the case analysis term acts as an if-then-else. Currently, no distinction is

made between the various versions of natrec; adding support for them is simple but tedious.

Also, to interpret the types – which are written in prefix form rather than infix form – we provide this key:

`_xyz`	is	an unbound Prolog variable standing for any type
`nat`	is	Nat
`and(A,B)`	is	$A \wedge B$
`imp(A,B)`	is	$A \rightarrow B$
`or(A,B)#M`	is	$(A \vee B)^M$

Briefly, the structure of each example is a raw proof term – i.e. a derivation in the standard propositional type system, augmented with disjunctions – followed by its inferred type and then by its annotated type which has positive integers as names of marks which the user is permitted to request be set to one. Thereafter, each example includes one or more requests to set some subset of these marks to one, followed in each case by the term that would be extracted under each of the extraction schemes that have been described in this thesis. Note that in each case, the marks requested to be set to one may be a subset of the marks that the system infers must be set to one. This simply makes it easier to enter the desired marks, for the purpose of experimentation; this leniency could easily be removed from the system and it could report inexact matches between the inferred marking and the requested marking. Finally, we note that there is a little additional commentary in the body of the examples – specifically, for the predecessor function and the even/odd function.

```
Raw proof term:
lam(x,pair(fst(x),snd(x)))

Raw type: imp(and(_310,_311),and(_310,_311))

Annotated type: imp(and(1,2),and(4,5))
```

```
Request to set these marks to 1: [4]

Standard extraction:
lam(x,pair(fst(x),snd(x)))

Extended extraction:
lam(x,x)

Naive delay/force extraction:
lam(x,x)

Improved extraction:
lam(x,x)

Improved extraction + peephole optimization:
lam(x,x)

----------------------------------------

Request to set these marks to 1: [5]

Standard extraction:
lam(x,pair(fst(x),snd(x)))

Extended extraction:
lam(x,x)

Naive delay/force extraction:
lam(x,x)

Improved extraction:
lam(x,x)

Improved extraction + peephole optimization:
lam(x,x)

----------------------------------------

Request to set these marks to 1: [4,5]

Standard extraction:
lam(x,pair(fst(x),snd(x)))

Extended extraction:
lam(x,pair(fst(x),snd(x)))

Naive delay/force extraction:
lam(x,pair(fst(x),snd(x)))

Improved extraction:
```

```
lam(x,pair(fst(x),snd(x)))

Improved extraction + peephole optimization:
lam(x,pair(fst(x),snd(x)))

------------------------------------------------------------------------

Raw proof term:
lam(x1,lam(x2,lam(x3,lam(y1,lam(y2,lam(y3,pair(x1,y1)))))))

Raw type:
imp(_284,imp(_318,imp(_352,imp(_386,imp(_420,imp(_454,and(_284,_386))))
)))

Annotated type:
imp(1,imp(2,imp(3,imp(4,imp(5,imp(6,and(7,8)))))))

Request to set these marks to 1: [7]

Standard extraction:
lam(x1,lam(x2,lam(x3,lam(y1,lam(y2,lam(y3,pair(x1,y1)))))))

Extended extraction:
lam(x1,x1)

Naive delay/force extraction:
lam(x1,delay(x1))

Improved extraction:
lam(x1,force(delay(x1)))

Improved extraction + peephole optimization:
lam(x1,x1)

---------------------------------------

Request to set these marks to 1: [8]

Standard extraction:
lam(x1,lam(x2,lam(x3,lam(y1,lam(y2,lam(y3,pair(x1,y1)))))))

Extended extraction:
lam(y1,y1)

Naive delay/force extraction:
delay(lam(y1,delay(y1)))

Improved extraction:
delay(lam(y1,force(delay(y1))))

Improved extraction + peephole optimization:
```

```
delay(lam(y1,y1))

----------------------------------------

Request to set these marks to 1: [7,8]

Standard extraction:
lam(x1,lam(x2,lam(x3,lam(y1,lam(y2,lam(y3,pair(x1,y1)))))))

Extended extraction:
lam(x1,lam(y1,pair(x1,y1)))

Naive delay/force extraction:
lam(x1,delay(lam(y1,delay(pair(x1,y1)))))

Improved extraction:
lam(x1,force(delay(lam(y1,force(delay(pair(x1,y1)))))))

Improved extraction + peephole optimization:
lam(x1,lam(y1,pair(x1,y1)))

------------------------------------------------------------------------

Raw proof term:
app(lam(x,pair(fst(x),snd(x))),pair(0,s(0)))

Raw type: and(nat,nat)

Annotated type: and(1,2)

Request to set these marks to 1: [1]

Standard extraction:
app(lam(x,pair(fst(x),snd(x))),pair(0,succ(0)))

Extended extraction:
app(lam(x,x),0)

Naive delay/force extraction:
app(lam(x,x),0)

Improved extraction:
app(lam(x,x),0)

Improved extraction + peephole optimization:
app(lam(x,x),0)

----------------------------------------

Request to set these marks to 1: [2]
```

```
Standard extraction:
app(lam(x,pair(fst(x),snd(x))),pair(0,succ(0)))

Extended extraction:
app(lam(x,x),succ(0))

Naive delay/force extraction:
app(lam(x,x),succ(0))

Improved extraction:
app(lam(x,x),succ(0))

Improved extraction + peephole optimization:
app(lam(x,x),succ(0))

---------------------------------------

Request to set these marks to 1: [1,2]

Standard extraction:
app(lam(x,pair(fst(x),snd(x))),pair(0,succ(0)))

Extended extraction:
app(lam(x,pair(fst(x),snd(x))),pair(0,succ(0)))

Naive delay/force extraction:
app(lam(x,pair(fst(x),snd(x))),pair(0,succ(0)))

Improved extraction:
app(lam(x,pair(fst(x),snd(x))),pair(0,succ(0)))

Improved extraction + peephole optimization:
app(lam(x,pair(fst(x),snd(x))),pair(0,succ(0)))

-----------------------------------------------------------------------
```

This example shows the predecessor function which we used as our running example throughout the body of the thesis. The following two examples show what is extracted when it is applied, first to some data, then to a proof that the data meets the required precondition.

```
Raw proof term:
lam(n,lam(p,natrec(n,abort,x,y,pair(x,empty))))

Raw type: imp(nat,imp(_312,and(nat,_451)))

Annotated type: imp(1,imp(2,and(3,4)))
```

```
Request to set these marks to 1: [3]

Standard extraction:
lam(n,lam(p,natrec(n,abort,x,y,pair(x,empty))))

Extended extraction:
lam(n,natrec(n,abort,x,y,x))

Naive delay/force extraction:
lam(n,delay(natrec(n,0,x,y,x)))

Improved extraction:
lam(n,force(delay(natrec(n,0,x,y,x))))

Improved extraction + peephole optimization:
lam(n,natrec(n,0,x,y,x))

------------------------------------------------------------------------

Raw proof term:
app(lam(n,lam(p,natrec(n,abort,x,y,pair(x,empty)))),s(0))

Raw type: imp(_352,and(nat,_491))

Annotated type: imp(1,and(2,3))

Request to set these marks to 1: [2]

Standard extraction:
app(lam(n,lam(p,natrec(n,abort,x,y,pair(x,empty)))),s(0))

Extended extraction:
app(lam(n,natrec(n,abort,x,y,x)),s(0))

Naive delay/force extraction:
app(lam(n,delay(natrec(n,0,x,y,x))),s(0))

Improved extraction:
delay(app(lam(n,force(delay(natrec(n,0,x,y,x)))),s(0)))

Improved extraction + peephole optimization:
delay(app(lam(n,natrec(n,0,x,y,x)),s(0)))

------------------------------------------------------------------------

Raw proof term:
app(app(lam(n,lam(p,natrec(n,abort,x,y,pair(x,empty)))),s(0)),empty)

Raw type: and(nat,_531)
```

```
Annotated type: and(1,2)

Request to set these marks to 1: [1]

Standard extraction:
app(app(lam(n,lam(p,natrec(n,abort,x,y,pair(x,abort)))),s(0)),empty)

Extended extraction:
app(lam(n,natrec(n,abort,x,y,x)),s(0))

Naive delay/force extraction:
force(app(lam(n,delay(natrec(n,0,x,y,x))),s(0)))

Improved extraction:
force(delay(app(lam(n,force(delay(natrec(n,0,x,y,x)))),s(0))))

Improved extraction + peephole optimization:
app(lam(n,natrec(n,0,x,y,x)),s(0))

-----------------------------------------------------------------------
```

Here we have the even/odd function with the following dependent type:

$$\forall(n{:}Nat).\exists(m{:}Nat).((n = 2m) \vee (n = 2m{+}1))$$

That is, given a natural number, it will return a pair containing a number which is half of the argument, together with a left/right indication of whether the input was even or odd, respectively. As it turns out, we can choose to request content from only the left/right part of the disjunction and in return we get an optimized version of the program.

```
Raw proof term:
lam(x,natrec(x,pair(0,inl(empty)),y,z,case(snd(z),l,pair(fst(z),inr(emp
ty)),r,pair(s(fst(z)),inl(empty)))))

Raw type: imp(nat,and(nat,or(_458,_459)))

Annotated type: imp(1,and(2,or(3,4)#5))

Request to set these marks to 1: [2]

Standard extraction:
lam(x,natrec(x,pair(0,inl(empty)),y,z,case(snd(z),l,pair(fst(z),inr(emp
ty)),r,pair(succ(fst(z)),inl(empty)))))

Extended extraction:
lam(x,natrec(x,pair(0,left),y,z,case(snd(z),l,pair(fst(z),right),r,pair
(succ(fst(z)),left))))
```

```
Naive delay/force extraction:
lam(x,natrec(x,pair(0,left),y,z,case(snd(z),l,pair(fst(z),right),r,pair
(succ(fst(z)),left))))

Improved extraction:
lam(x,natrec(x,pair(0,left),y,z,case(snd(z),l,pair(fst(z),right),r,pair
(succ(fst(z)),left))))

Improved extraction + peephole optimization:
lam(x,natrec(x,pair(0,left),y,z,case(snd(z),l,pair(fst(z),right),r,pair
(succ(fst(z)),left))))

----------------------------------------

Request to set these marks to 1: [5]

Standard extraction:
lam(x,natrec(x,pair(0,inl(empty)),y,z,case(snd(z),l,pair(fst(z),inr(emp
ty)),r,pair(succ(fst(z)),inl(empty)))))

Extended extraction:
lam(x,natrec(x,left,y,z,case(z,l,right,r,left)))

Naive delay/force extraction:
lam(x,natrec(x,left,y,z,case(z,l,right,r,left)))

Improved extraction:
lam(x,natrec(x,left,y,z,case(z,l,right,r,left)))

Improved extraction + peephole optimization:
lam(x,natrec(x,left,y,z,case(z,l,right,r,left)))

------------------------------------------------------------------------
```

Here we have a second version of the even/odd function with the following dependent type, differing slightly from the first, yet still capable of being similarly optimized:

$$\forall(n{:}Nat).((\exists(m{:}Nat).(n = 2m)) \vee (\exists(m{:}Nat).(n = 2m{+}1))))$$

This is intended to demonstrate the flexibility of our approach.

```
Raw proof term:
lam(x,natrec(x,inl(pair(0,empty)),y,z,case(z,l,inr(pair(fst(l),empty)),
r,inl(pair(s(fst(r)),empty)))))

Raw type: imp(nat,or(and(nat,_444),and(nat,_612)))
```

```
Annotated type: imp(1,or(and(2,3),and(5,6))#8)

Request to set these marks to 1: [2,5,8]

Standard extraction:
lam(x,natrec(x,inl(pair(0,empty)),y,z,case(z,l,inr(pair(fst(l),empty)),
r,inl(pair(succ(fst(r)),empty)))))

Extended extraction:
lam(x,natrec(x,inl(0),y,z,case(z,l,inr(l),r,inl(succ(r)))))

Naive delay/force extraction:
lam(x,natrec(x,inl(0),y,z,case(z,l,inr(l),r,inl(succ(r)))))

Improved extraction:
lam(x,natrec(x,inl(0),y,z,case(z,l,inr(l),r,inl(succ(r)))))

Improved extraction + peephole optimization:
lam(x,natrec(x,inl(0),y,z,case(z,l,inr(l),r,inl(succ(r)))))

----------------------------------------

Request to set these marks to 1: [8]

Standard extraction:
lam(x,natrec(x,inl(pair(0,empty)),y,z,case(z,l,inr(pair(fst(l),empty)),
r,inl(pair(succ(fst(r)),empty)))))

Extended extraction:
lam(x,natrec(x,left,y,z,case(z,l,right,r,left)))

Naive delay/force extraction:
lam(x,natrec(x,left,y,z,case(z,l,right,r,left)))

Improved extraction:
lam(x,natrec(x,left,y,z,case(z,l,right,r,left)))

Improved extraction + peephole optimization:
lam(x,natrec(x,left,y,z,case(z,l,right,r,left)))
```

VITA

Surname: Knight

Given Names: Brent Eric

Place of Birth: Edmonton, Alberta, Canada

Date of Birth: November 15, 1970

Educational Institutions Attended:

University of Victoria	1992 to 1994
University of Alberta	1988 to 1992

Degrees Awarded:

B.Sc. (Honours)	University of Alberta	1992

Honours and Awards:

Canada Scholar	1988 to 1992
I.P. Sharp and Associates Scholarship	1990 to 1991
Louise McKinney Post-Secondary Scholarship	1991 to 1992
Dean's Silver Medal in Science, University of Alberta	1992
B.C. Advanced Systems Institute Graduate Scholarship	1992 to 1993
University of Victoria President's Scholarship	1992 to 1994
NSERC 1967 Scholarship	1992 to 1996

Publications:

A world championship caliber checkers program. J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, D. Szafron. In *Artificial Intelligence (Netherlands)*, vol. 53, no. 2-3, pp. 273-89 (Feb. 1992).

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Safe Strict Evaluation of Redundancy-Free Programs from Proofs

Author: [redacted]
(Signature)

BRENT KNIGHT
(Name in Block Letters)

Wed. July 27, 1994
(Date)