Incremental Computation via Partial Evaluation

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Raman Srinivas Sundaresh

May 1992

Abstract

Incremental Computation via Partial Evaluation

Raman Srinivas Sundaresh
Yale University
1992

Incremental programs – programs which do not recompute their answer from scratch when the input changes slightly – are an important component of a modern programming environment. There has been considerable research into building incremental programs for important problems (e.g. data flow analysis). As programming environments grow in size and complexity, there is a need for a general framework to build incremental programs. Other researchers have recognized this need and constructed frameworks based on function caching, attribute grammars etc. We present a framework based on partial evaluation.

We show a relationship between incremental programs and partial evaluation. The theoretical foundations of this framework are explored. The basis of our approach is the operation of combining residual functions (the results of partial evaluation). The algebraic properties of such an operator are identified. We then present an implementation based on the partial evaluator *Schism*. The implementation is presented by means of a series of examples with performance results.

Contents

Acknowledgements

I would like to thank my advisor Paul Hudak for his support and encouragement throughout my graduate student years. His insight and guidance have been invaluable.

Thanks are also due to my readers: Olivier Danvy and Marina Chen, for their meticulous reading and thoughtful suggestions. Charles Consel introduced me to partial evaluation and provided me with his partial evaluator Schism.

My friends at the Yale Computer Science Department made my years at Yale a little more bearable. They are too numerous to name, but they know who they are. Kannan Parthasarathy and Abhay Deshpande helped me spend many enjoyable hours outside of Yale.

None of this would have been possible without the help of my family: My mother, father, my sister Suchitra and my wife Aparna.

Finally I would like to thank my old friends G. Ramalingam and S. Narayan. They helped me pull through when the going got tough.

This research was supported in part by grant DCR-8451415 from the National Science Foundation and an IBM graduate fellowship.

Chapter 1

Introduction

1.1 Overview

It is a common occurrence in a programming environment to apply a software tool to a series of similar inputs. Examples include compilers, interpreters, text formatters, etc., whose inputs are usually incrementally modified text files. Thus programming environment researchers have recognized the importance of building *incremental* versions of these tools — i.e. ones which can efficiently update the result of a computation when the input changes only slightly.

There seems to be some consensus that incremental algorithms are hard to derive, debug and maintain [Pug88, YS91, FT90], and as we attempt to create incremental programs for larger tasks, this problem will only get worse. Thus there is an increasing need for a *framework* for incremental computation which will facilitate the construction of incremental programs. We have developed such a framework based on *partial evaluation* (a.k.a. program specialization). Partial evaluation is a general program transformation technique used to produce specialized versions of programs when given some of the program's inputs. We describe an incremental program as a combination of a non-incremental program and a partition of the input data structure. Then we describe a methodology to generate an incremental program from this description. Thus an important design step in this process is to determine a partition of the input domain. This controls both the "granularity" of the incrementality as well as overall efficiency. We also provide a formal algebraic basis for *reasoning* about the correctness of the incremental programs thus generated. The framework relies partially on the notion of a *Brouwerian algebra.* Finally we describe an implementation of this framework incorporating methods to overcome the overhead of "incremental interpretation," and generate "compiled" incremental programs using *Futamura projections.*

This chapter describes the concept of partial evaluation, briefly discusses its connection with incremental computation, discusses related work and outlines the rest of the thesis.

1.2 Partial Evaluation

Consider a program which accepts many inputs. If this program is used repeatedly with one of the inputs set to a fixed value, then it is advantageous to precompute a "specialized" program – one in which computations depending on the fixed input are carried out ahead of time. The goal of partial evaluation is to produce (automatically) such a specialized program.

Many applications are more naturally expressed in an "interpretive" style – one where all the arguments are presented at the same time. While this results in clearer programs, efficiency is sacrificed because many more computations are performed at run time. Partial evaluation attempts to present us with the best of both worlds: we retain clarity and maintainability by writing highly parametrized programs, and obtain efficiency by automatically specializing these programs using known inputs. The result of partial evaluation is called a *residual program.* The arguments supplied ahead of time are referred to as *static.* The arguments supplied at run time are referred to as *dynamic.*

The correctness criterion for a partial evaluator is a simple one: the result of running the program when given all its inputs must be the same as running the *residual program* on the dynamic input. We call a program representing function f as $\underline{f}$. The partial evaluator will be called $\mathcal{PE}$. So the correctness criterion can be stated as:

$$\mathit{prog}'\ y = \mathit{prog}\ x\ y \text{ where } \underline{\mathit{prog}'} = \mathcal{PE}\ \underline{\mathit{prog}}\ \underline{x}$$

The earliest use of the term "partial evaluation" was in a paper by Lombardi and Raphael [LR64]. Since then there has been a great deal of activity in the area. For a good survey of the field, see [JSS89]. Because of the general nature of partial evaluation, applications have been found in diverse fields, ranging from artificial intelligence to computer graphics. One of the more interesting applications of partial evaluation was put forth by Futamura [Fut71] when he suggested a method of deriving compilers from interpreters. Consider an interpreter *int* for a language L. Assume that *int* is written in language M. Furthermore, let $\mathcal{PE}$ be a partial evaluator for programs in language M. Then, consider the *first Futamura projection*:

$$\underline{\mathit{obj}} = \mathcal{PE}\ \underline{\mathit{int}}\ \underline{\mathit{prog}}$$

where *prog* is a program written in L. By the correctness criterion for partial evaluation,

$$\mathit{obj}\ \mathit{data} = \mathit{int}\ \underline{\mathit{prog}}\ \mathit{data}$$

This can be interpreted as saying that *obj* is a "compiled" version of *prog*. Consider the *second Futamura projection*: (note that this requires that the partial evaluator be written in the same language that it specializes)

$$\underline{\mathit{comp}} = \mathcal{PE}\ \underline{\mathcal{PE}}\ \underline{\mathit{int}}$$

Again, the correctness criterion for partial evaluation states:

$$\mathit{comp}\ \underline{\mathit{prog}} = \mathcal{PE}\ \underline{\mathit{int}}\ \underline{\mathit{prog}} = \underline{\mathit{obj}}$$

This can be interpreted as saying that *comp* is compiler from language L to language M. The *third Futamura projection* states:

$$\underline{cocom} = \mathcal{PE}\ \underline{\mathcal{PE}}\ \underline{\mathcal{PE}}$$

Again, the correctness criterion for partial evaluation states:

$$cocom\ \underline{int} = \mathcal{PE}\ \underline{\mathcal{PE}}\ \underline{int} = \underline{comp}$$

This can be interpreted as saying that *cocom* is a compiler generator for interpreters written in language M generating compilers which compile into language M. Note that we have not touched upon the topic of implementation of partial evaluation. The techniques necessary to achieve the Futamura projections are non trivial and were first discovered by Neil Jones and his associates at DIKU [JSS89].

1.3 Example

In this section we explain some of the techniques used to implement self-applicable partial evaluation (i.e. partial evaluators capable of carrying out the *Futamura projections* by means of an example. Partial evaluators are usually split into two stages: a preprocessing stage where program transformation decisions are made for each subexpression in the program and a specialization stage where these transformations are actually carried out. Just what are these transformations? The basic techniques of partial evaluation for a functional language are:

- Symbolic Computation: manipulating symbolic expressions even if all the subcomponents are not static.
- Function Call Unfolding: replacing a call to a function by its body with arguments substituted.
- Function Specialization: replacing a function with a version *specialized* on some arguments.

We use the example of Ackerman's function to illustrate these techniques:

```
ack m n = if (m == 0)
          then n+1
          else if (n == 0)
               then ack (m-1) 1
               else ack (m-1) (ack m (n-1))
```

If we partially evaluate ack with m static (equal to 2) and n as dynamic, then we obtain the following:

```
ack2 n = if (n == 0)
         then 3
         else ack1 (ack2 (n-1))

ack1 n = if (n == 0)
         then 2
         else ack0 (ack1 (n-1))

ack0 n = n+1
```

Each one of the bodies is obtained by simplifying the right hand side with the static value of m (namely 2, 1 and 0). Simplifications are performed at each stage. These are either symbolic computations (e.g. reducing conditional statements) or function unfolding. Specialization is used to produce ack2, ack1 and ack0.

1.4 Incremental Computation

Lombardi and Raphael's pioneering paper [LR64] coined the term "partial evaluation" for the purpose of "incremental computation." However, their notion of incremental computation was to monotonically add information about the input to a function, and to do as much computation as possible at each step. Our definition of incremental computation (and what is generally understood by the term today) is more general than this in that changes to the input need not always be in the form of adding information — information can change non-monotonically.

Also, the partial evaluation referred to by Lombardi and Raphael was restricted to intra-procedural constant propagation.

Consider the simple example of the exponentiation function:

```
power x n = if (n == 0)
            then 1
            else if (even? n)
                 then (power x (n/2)) ** 2
                 else x * (power x (n-1))
```

We examine the efficient recomputation of this function as x and n change. First let us assume that n remains fixed (say 3) and that x varies. What sort of information would we like like to store so as not to redo the computations which depend only on n? We suggest that this information take the form of a residual function – the result of partial evaluation. Specifically, consider the result of specializing power with x as unknown and n as known (equal to 3):

```
power3 x = x * ( x * 1 ) ** 2
```

Now if the value of x varies, we can recompute power by simply applying the residual function to the new value of x. This saves the computation which depends only on the value of n. Similarly we can compute another residual function to handle the case of x being fixed (equal to 5) and n varying.

```
power5 n = if (n == 0)
           then 1
           else if (even? n)
                then (power5 (n/2)) ** 2
                else 5 * (power5 (n-1))
```

Note that there may be more efficient ways of carrying out the incremental computation in this case. These would take advantage of the following property of power:

```
(power x n) * (power x m) = power x (m+n)
```

We do not make use of this, but rather we concentrate on saving those computations in the function which depend only on the unchanged portion. In this example, obtaining the answer from the residual function was easy; one simply needed to apply the residual function to the changed argument. But consider the following function:

```
f x n y m = (power x n) + (power y m)
```

In this case we would have four different residual functions, one for each argument being known and all the others being unknown. How do we synthesize the answer from four different residuals? This is one of the important questions that this thesis will address.

1.5 Other Applications of Partial Evaluation

Partial Evaluation is a general program specialization technique which has found application in a variety of areas. To give some idea of the generality of the concept, we describe some of these applications. We have already looked at partial evaluation as applied to compiler generation [Fut71]. This area has received a great deal of attention [JSS89]. The DIKU group, led by Neil Jones, was the first group to be achieve the *Futamura projections.* They worked with a purely functional subset of Lisp, and were able to transform interpreters into compilers for a number of small example languages. The compiled code produced good speedup (5 to 20) over the interpreted code.

Partial evaluation had been used in specific applications such as theorem proving as early as in 1975 [BM75]. One of the early attempts to provide a general purpose partial evaluator was by a team from Linköping University who built a partial evaluator for Lisp [BHOS76]. This partial evaluator utilized simple on the fly techniques, and did not use any static program analysis. Consequently it was not self-applicable.

Emanuelson and Haraldsson used partial evaluation to compile extensions to Lisp [EH80]. The method used was to construct an interpreter for the extensions and then specialize programs in the extended language. The paper details an experiment carried out in adding pattern matching to Lisp. Their results compared favorably with those of a commercial compiler.

Mogensen [Mog86] describes the use of partial evaluation in optimizing raytracing programs. A raytracer takes two arguments: a scene description and a viewpoint. The output of the raytracer is a picture of the scene as it appears from the desired viewpoint. Specializing the ray tracer with respect to a given scene produces a specialized ray tracer which (relatively) inexpensively computes the picture of a *given* scene from various viewpoints. Interestingly, Mogensen found that speedups were produced even if only one view was necessary. The reason for this is that in the original ray tracer, the same scene is examined many times, entailing repeated computations. This does not occur in the specialized version.

Consel and Danvy [CD89] describe an experiment where a naive (but easy to understand) pattern matcher is specialized with a fixed text string. They show how to make this specialization process achieve the automatic generation of the Knuth-Morris-Pratt algorithm.

1.6 Related Work in Incremental Computation

Readers familiar with the literature of partial evaluation will recall Lombardi and Raphael's pioneering paper [LR64], which coined the term "partial evaluation" in the context of doing "incremental computation." However, their notion of incremental computation was to monotonically add information about the input to a function, and to do as much computation as possible at each step. This is achieved by computing a series of residual functions each of which is the result of partially evaluating the previous one with the additional input. Our definition of incremental computation (and what is generally understood by the term today) is more general than this in that changes to the input need not always be in the form of adding information — information can change non-monotonically. Thus our work can be seen as a generalization of the work which originally introduced partial evaluation.

Modern programming environments incorporate incremental techniques in many different components. An important incremental activity is the programming activity itself. Frequently programs are modified many times before the program is deemed correct. Thus compilers are frequently presented with a series of closely related inputs. The most well known example of this is the Cornell Synthesizer Generator [Rep84, RTD83], which incorporates a general attribute reevaluation algorithm to incrementally reattribute a modified parse tree. Another compiler related area is data flow analysis, a task for which many incremental algorithms have been developed (e.g. [MR90]). This area has given rise to many general approaches to the incremental computation problem [Ryd82, RC86, RP86]. Other compiler related areas are incremental parser generation [HKR89].

Spreadsheets have also made use of incremental evaluation to take advantage of the common pattern of usage – compute answer, change constraints, compute answer, change constraints ... . Other areas where there has been work done in designing incremental programs includes: document formatting [Bro88], graphic editing [Ase87, Nel85]. In each of these areas, the output is a visual object while the input is a instruction sequence. Every time the instruction sequence is modified, the output needs to be recomputed. There has been work done on general systems – ones which are not dedicated to a single task. The Cornell Synthesizer Generator is one of these. The VisiProg environment [HW88] also provides incrementality. Constraint languages [Lel88] where constraints can be retracted face the same problem.

Considerable work has gone into building incremental programs using the dependency graph approach. Here nodes represent computations and directed edges represent dependencies between the computations. The goal of the incremental evaluator here is to reevaluate the smallest number of nodes possible when the input changes [AHR$^+$90]. The emphasis here is on very efficient and accurate change propagation, not on reusing the computations within a node. [Pug88] has pointed out certain restrictions imposed by this approach – the framework is not expressive enough to handle problems whose solution requires more than linear time.

In [Pai86, PK82] an approach to building incremental programs called *finite differencing* is presented. This approach has its origins in the compiler optimization technique of strength reduction. Under this framework a function is made incremental by applying a series of transformations chosen from a prescribed set. The transformations can be linked together using the *chain rule.*

[HT86b] describes extensions to the Cornell Synthesizer which enable the incremental updating of relational database views. This is important since many programming environment tasks can make profitable use of a relational database. Another extension to the Cornell Synthesizer Generator is the work described in [HT86a], where the aggregate update problem is tackled. The aggregate update problem occurs when a small change to an aggregate data structure causes an unnecessarily large amount of recomputation.

Yellin and Strom [YS91] describe a restricted functional language for incremental computation. The main data structure in the language is a *bag.* Programs written in the language make use of certain combining forms which are guaranteed to have efficient incremental performance. The emphasis in this work has been to design efficient incremental algorithms for the various operations in the language. These algorithms are expressed in the form of compile time transformations. The framework incorporates techniques to evaluate the *incremental complexity* of programs.

Pugh [Pug88] tackles the problem of caching results of function calls to achieve incrementality. The scheme depends crucially on clever run-time support. Programs are written using *stable decompositions* [Pug88] of data structures to obtain good performance. To this end [Pug88] presents techniques for representing sets and sequences in an applicative manner and also presents techniques to efficiently update and test for equality.

In [FT90] Field and Teitelbaum construct an incremental evaluator for the λ-calculus. It keeps track of exactly which reductions did not depend on the part of the λ-term which changed. For this purpose the parts of the term which can change are marked. A notion of *overlapping* reductions on similar terms is defined. This enables [FT90] to present an optimality criterion: The incremental evaluator is guaranteed to perform non-overlapping reductions.

1.7 Outline

- Chapter 2 describes the notation for partial evaluation which will be used in the rest of the thesis and introduces the relationship between incremental computation and partial evaluation.

- Chapter 3 presents the formal basis for our framework by describing interesting algebraic properties of residual functions. This makes the framework described in Chapter 2 more precise.

- Chapter 4 presents algorithms for combining residual functions. This forms the computational core of the framework. The correctness and efficiency of these algorithms are also discussed.

- Chapter 5 describes the implementation by means of examples. Performance results for these examples are also presented.

- Chapter 6: Conclusions, Related work and Future Work.

Chapter 2

Incremental Computation and Partial Evaluation

This chapter describes the relationship between incremental computation and partial evaluation. This relationship forms the basis of our framework. In the process of presenting this relationship we raise a number of questions. The answers to these questions form the subject matter of later chapters. First we introduce the formal notation for partial evaluation which we will be using in this thesis. This framework was introduced by John Launchbury [Lau88]. In this framework *projections* are used to represent partially static data.

After having introduced the formal framework for partial evaluation, we present an overview of our methodology to build incremental programs. First this is done informally, then we formalize the framework in the form of an *Incremental Interpreter.* This interpreter takes specially crafted non incremental programs, information about how to make them incremental, and runs these programs incrementally. The additional information is in the form of a partition of the input data structure.

2.1 Partial Evaluation

Following Launchbury [Lau88], we give a precise definition of partial evaluation using *projections.*

For purposes of generality, all functions are assumed to take just one argument. For first order functions which take more than one argument, this condition can be satisfied by simply tupling the arguments together. Projections are then used to capture the information about the *static* portion of the argument.

Definition 2.1.1 *A* projection *on a domain $\mathcal{D}$ is a continuous mapping $p : \mathcal{D} \rightarrow \mathcal{D}$ such that:*

- $p \sqsubseteq ID$ *(no information addition)*
- $p \circ p = p$ *(idempotence)*

Note that *ID* (the identity function) is the greatest projection and *ABSENT* (the constant function with value $\perp$) the least (under the standard information ordering on functions). The first condition in the above definition simply states that projections always remove information from their argument. In our context this is interpreted as saying that projections remove the *dynamic* parts of their arguments and leave behind the *static* parts. The second condition states that the projection removes all the dynamic information in a single application to its argument. Projections are a special case of a more general class of functions called *retractions*, which are simply continuous idempotent functions. Projections have been also been used to carry out backwards strictness analysis of functional programs [Hug88, HW87].

Definition 2.1.2 *If p and q are projections and $p \sqcup q = ID$, then q is a* complement *of p.*

Note that by the above definition the complement of a projection may not be unique (for example, *ID* is a complement of *every* projection). We tighten this definition in Chapter 3 to achieve uniqueness by choosing the "least" of these projections. Indeed, a major goal of that chapter is to define domains of projections where such a construction always exists. We write $\bar{p}$ to denote the unique (to be defined later) complement of p. Intuitively projection complements represent the dynamic information which a residual function needs at run time. In [Lau90] Launchbury has extended this framework by showing how the *type* of the residual function depends on the static value used to produce it. Using the notion of dependent sum and dependent product domain constructions, he shows how the residual function can be made to take as little information as possible.

We now define partial evaluation in terms of projections:

Definition 2.1.3 *A* partial evaluator $\mathcal{PE}$ *is a function which takes representations of a function f, a projection p, and a value a, and produces a representation of the residual function, f_{pa}, defined as follows:*

$$\mathcal{PE}\ \underline{f}\ \underline{p}\ \underline{a} \ = \ \underline{f_{pa}}$$

such that

$$f_{pa}\ \bar{p}\ a = \ f\ a$$

This can be seen extensionally as a restatement of Kleene's S^m_n theorem from recursive function theory.

When there is no ambiguity we use r_p to denote $\mathcal{PE}\ f\ p\ a$. ID is the identity projection (function). $ABSENT$ is the projection which removes all information. Given this notation, note that $r_{ID} = f\ a$ and $r_{ABSENT} = f$.

Although the partial evaluator really takes *representations* of its arguments and not actual *values*, hereafter we treat it as taking values as arguments (primarily to avoid having to propagate underlinings). On the other hand, the algorithm for "combining" residual functions in Subsection 5 depends crucially on manipulating the representations.

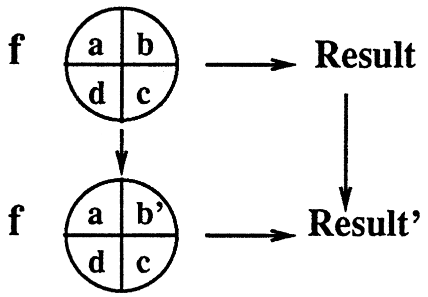

Figure 2.1: Incremental computation

2.2 Incremental Computation

Returning now to the problem of incremental computation, we can summarize the situation as in Figure 2.1. Here the function f (which may be a compiler, text formatter, etc.) is being applied to a structured argument to give the result. If only part of the argument changes, such as part **b**, we would like to compute the new result without having to redo the entire computation; in other words, we would like to avoid having f reprocess parts **a**, **c**, and **d**.

Now, here's the connection to partial evaluation, and the basis of our framework: The partitioning of the input domain can be described using a set of *projections* as defined in the previous section; let's call them p_a, p_b, p_c, and p_d for the example in Figure 2.1. If we then compute the residual functions r_{p_a}, r_{p_b}, r_{p_c}, and r_{p_d}, we have essentially "cached" those portions of the computation that depend only on parts **a**, **b**, **c**, and **d** of the input, respectively.

Recalling that $r_{ID} = apply\ f\ a$, all we need now to compute the final result is a (presumably efficient) way to construct r_{ID} from the set of residual functions — for now, let's assume that such a technique exists. If part of the input were to change, say **b** changes to **b'**, then all we have to do is recompute r_{p_b}; computation of r_{ID} then takes place with this new residual function in place.

An alternative way to describe this process is as follows: At the point when **b** changes to **b'**, suppose we had by some means already computed $r_{\bar{p}_b}$ — then all we need to do to compute the new result is to apply $r_{\bar{p}_b}$ to **b'**. We can thus view the problem as an attempt to find (at least a conservative approximation to) $r_{\bar{p}_b}$ by combining existing residual functions.

To summarize: all incremental methods maintain some form of auxiliary information. This auxiliary information represents the results of subcomputations of the original program. We propose that this auxiliary information be stored in the form of the results of partial evaluation: residual functions. We have raised many questions, which we frame after defining the approach more formally. First we formalize the notion of a partition as a set of projections:

Definition 2.2.1 *A* partition P *of a domain* $\mathcal{D}$ *is a set of projections* $\{p_i\}$ *on* $\mathcal{D}$ *such that* $\sqcup\{p_i\} = ID$.

Given this definition of a partition, we can capture all the information needed to specify an incremental program.

Definition 2.2.2 *An* incremental program specification *is a pair* $\langle f, P\rangle$ *where* $f : \mathcal{D} \rightarrow \mathcal{E}$ *is the function to be incrementalized and* P *is a partition of* $\mathcal{D}$.

Note that the program to be incrementalized may need to be specially crafted in certain ways. This is the topic of discussion in the chapter on applications. Now we are ready to formalize the intuitive description of our methodology. We do this in the form of an "incremental interpreter," denoted $\mathcal{I}$. $\mathcal{I}$ has functionality:

$$\mathcal{I} : \langle f, P\rangle \;\rightarrow\; a_0 \;\rightarrow\; \langle \delta_0, \delta_1, \ldots\rangle \;\rightarrow\; \langle b_0, b_1, \ldots\rangle$$

$\langle f, P\rangle$ is the incremental program specification, and a_0 is the initial argument. The δs are functions capturing "small" changes to the input, and the bs are the successive output results.

The main purpose of $\mathcal{I}$ is to maintain the invariant: $r_{p_i} = \mathcal{PE}\ f\ p_i\ a$ for all $p_i \in P$, and in so doing satisfies the following correctness criterion:

$$b_i = f\ a_i \text{ where } a_i = \delta_{i-1}\ a_{i-1}$$

We now give a high level description of $\mathcal{I}$. Note that some of the parts of $\mathcal{I}$ have been left unspecified.

Algorithm $\mathcal{I}$:

- **Setup:** Compute $r_{p_i} = \mathcal{PE}\ f\ p_i\ a$ for each p_i in the partition P.

- **Reestablish:** If a changes to a' $(= \delta\ a)$, recompute all r_{p_i} for which $p_i\ a \neq p_i\ a'$.

- **Combine:** The new result r_{ID} is obtained from $\{r_{p_i}\}$ using appropriate combining operations.

We have so far made many assumptions that require fleshing out. In particular:

1. How do we combine residual functions to get "larger" ones? Does the construction even exist? If so, is it unique? What are the formal properties of such a construction? This is the subject matter of Chapter 3.

2. Assuming that the construction to combine residual functions exists, can we design correct and efficient algorithms to actually carry out the combination. In particular, the efficiency requirement is crucial – we must make sure that in combining two residual functions we do not redo the computations done in either one of them. This is crucial for incremental performance in the system. Chapter 4 discusses this issue.

3. What is the basis for choosing a good partition of the input domain? If it is too coarse, even small changes trigger massive recomputation; if too fine, the "stored" residual functions each capture very little computation and excessive work will be done in the combining phase. We examine this in Chapter 5 by means of examples.

Chapter 3

Algebraic Properties of Residual Functions

In this chapter we address the first of the questions we asked at the end of the last chapter: How do we combine residual functions to get "larger" ones? Does the construction even exist? If so, is it unique? What are the formal properties of such a construction? We can rephrase the question as follows: What are the properties of the combining operator which given r_p and r_q produces $r_{p \sqcup q}$?

To answer this question we first examine the domains of projections we will be using. The aim of this to make sure that these domains contain suitable well-defined combining operators. To do this we consider various domains of projections, finally choosing one based on the intuition that we need projections which depend only on the structure of their arguments and not on the actual values of the subcomponents. Having chosen this domain we investigate algebraic properties of projections belonging to the domain. Then we see how partial evaluation induces the same properties in the domain of residual functions corresponding (via partial evaluation) to this (chosen) domain of projections.

3.1 Projection Algebras

To construct domains of projections, we must first define the underlying domain of objects under consideration. We begin with a set of domains and domain formers that are adequate in capturing most of the domains found in conventional programming languages.

t ::=	**1**(the unit domain), **Nat**,...	base domains
|	τ_i	type parameter
|	$t_1 + t_2$	separated sum
|	$t_1 \times t_2$	non-strict product
|	$\mu\tau_1.\ t$	recursive domain

The structure described above can be easily generalized to more than one type parameter. In what follows we assume that standard domains such as lists, pairs, and natural numbers have been pre-defined, and we use conventional notation (*Nil*, *Cons*, 1, 2 etc.) when referring to elements of the domains. For example, polymorphic lists and pairs can be defined by:

$$
\begin{aligned}
List(\tau_1) &= \mu\tau_2.\ \mathbf{1} + (\tau_1 \times \tau_2) \\
Pair(\tau_1, \tau_2) &= \tau_1 \times \tau_2
\end{aligned}
$$

3.1.1 Projection Domains

Consider a domain of projections [Lau90] under the standard information ordering of functions — this domain is not closed with respect to greatest lower bound. This is because $p_1 \sqcap p_2 = \lambda x.\ (p_1 x) \sqcap (p_2 x)$ is not necessarily idempotent, and therefore may not be a projection. A simple example should convince the reader of this:

Example 3.1.1 *Consider projections on the domain of pairs of natural numbers. Let p_1 and p_2 be defined as follows:*

$$p_1\ (x, y) \ = \ \mathit{if}\ x = 2\ \mathit{then}\ (\bot, y)\ \mathit{else}\ (x, y)$$

$$p_2\ (x,y)\ \ =\ \ \mathit{if}\ x=\perp\ \mathit{then}\ (x,\perp)\ \mathit{else}\ (x,y)$$

Let p be $\lambda x.(p_1 x) \sqcap (p_2 x)$. It is easy to verify that $p\ (2,3) = (\perp, 3)$ which is not the same as $p \circ p\ (2,3) = (\perp,\perp)$. Thus p is not idempotent, and is therefore not a projection.

This problem arises because the domain of all projections is too large. We are interested (for purposes of partial evaluation) in projections which only depend on the *structure* of the object they are manipulating and not on the *values* of its components. Launchbury [Lau88] describes a smaller (finite) domain of projections, but his domain does not serve our purpose because it does not contain useful projections such as the following one on the domain of lists:

$p\ Nil = Nil$

$p\ (Cons\ x\ xs) = Cons\ \perp\ xs$

Here is a first attempt at constructing a domain of projections small enough to possess properties of interest to us yet large enough to contain examples such as the one above.

Definition 3.1.1 *A* polymorphic projection *on a domain $F(\tau)$ is a collection of instances $f_A : F(A) \rightarrow F(A)$, such that for any* strict *function $\alpha : A \rightarrow B$, the diagram in Figure 3.1 commutes; i.e. $f_B \circ mapF(\alpha) = mapF(\alpha) \circ f_A$. By $mapF$ we mean the appropriate map function for the datatype F.*

Example 3.1.2 *Define two projections over the domain of pairs:*

$p\ (\perp, b) = (\perp,\perp)$,

$p\ (a,b) = (a,b)$;

$LEFT\ (x,y) = (x,\perp)$.

p is not polymorphic (as can be seen from the fact that the diagram in Figure 3.2 does not commute) , but $LEFT$ is.

$$\begin{tikzcd}
\mathbf{F(A)} \arrow[r, "\mathbf{f_A}"] \arrow[d, swap, "\mathbf{mapF(\alpha)}"] & \mathbf{F(A)} \arrow[d, "\mathbf{mapF(\alpha)}"] \\
\mathbf{F(B)} \arrow[r, swap, "\mathbf{f_B}"] & \mathbf{F(B)}
\end{tikzcd}$$

Figure 3.1: Polymorphic projections

$$\begin{tikzcd}
(2,3) \arrow[r, "\mathbf{p}"] \arrow[dd, swap, "\mathbf{mapPair(g)}"] & (2,3) \arrow[d, "\mathbf{mapPair(g)}"] \\
 & (\perp,3) \\
(\perp,3) \arrow[r, swap, "\mathbf{p}"] & (\perp,\perp) \quad ?
\end{tikzcd}$$

Figure 3.2: p is not polymorphic, `g n = if (n==2) then ⊥ else n`

Example 3.1.3 *Consider the domain of lists defined earlier. Projections which can be formed using ID, $ABSENT$, $PCons$ and $PMap$ (defined below) are polymorphic:*

$$\begin{aligned}
ID\ x &= x \\
ABSENT\ x &= \bot \\
(PCons\ p\ q)\ Nil &= Nil \\
(PCons\ p\ q)\ (Cons\ x\ xs) &= Cons\ (p\ x)\ (q\ xs) \\
(PMap\ p)\ Nil &= Nil \\
(PMap\ p)\ (Cons\ x\ xs) &= Cons\ (p\ x)\ ((PMap\ p)\ xs)
\end{aligned}$$

The first argument to $PCons$ is a projection on the domain of the list element (for our purposes either ID or $ABSENT$). The second argument is a polymorphic projection on lists. Thus $PCons$ constructs new polymorphic projections from known ones. Polymorphic projections are exactly those projections which can differentiate only between elements of different "structure", and not the values that make up the structure.

Unfortunately, the domain of polymorphic projections does not exactly capture the intuition of depending only on the structure of the input. Consider:
$g\ Nil = Nil$,
$g\ (Cons\ x\ \bot) =\ Cons\ \bot\ \bot$,
$g\ x = x$

g is a polymorphic projection (as can be seen by observing that there is no α in the definition of polymorphic projections which can cause the diagram not to commute) but still depends on the values of subcomponents of its input (in this case $\bot$ as the tail).

To eliminate projections such as g, we take a completely different approach to defining the projection domain, building it constructively rather than placing constraints on the full function space:

$$\begin{aligned}
\mathcal{P}\ (\mathrm{d}) &= \{\ ID_d,\ ABSENT_d\ \}\ (\text{d is a base domain}) \\
\mathcal{P}\ (\tau) &= \{\ ID_\tau,\ ABSENT_\tau\ \} \\
\mathcal{P}\ (d_1 + d_2) &= \{\ p_1 + p_2 \mid p_1 \in \mathcal{P}(d_1),\ p_2 \in \mathcal{P}(d_2)\ \} \\
\mathcal{P}\ (d_1 \times d_2) &= \{\ p_1 \times p_2 \mid p_1 \in \mathcal{P}(d_1),\ p_2 \in \mathcal{P}(d_2)\ \} \\
\mathcal{P}\ (\mu\tau.T(\tau)) &= \mathcal{P}(T(\mu\tau.T(\tau))) \cup \{\ ABSENT_{\mu\tau.T(\tau)}\ \}
\end{aligned}$$

$\mathcal{P}$ maps from domains to domains. Any domain that is constructed using $\mathcal{P}$, starting with base domains d, is a valid projection domain. The subscript to ID or $ABSENT$ refers to the domain of definition. + and × are defined for functions as follows: $(f \times g)(a,b) = (f\ a, g\ b)$ and $(f+g)\bot = \bot$, $(f+g)(inl\ a) = inl(f\ a)$, $(f+g)(inr\ b) = inr(g\ b)$. Note the case of the recursive domain where the $ABSENT$ projection can be invoked (say) on any tail of the list. In general, this allows us to treat different levels of a list differently.

We have discussed the case of first order languages. We do not know of any extension of Launchbury's work to higher order languages.

We use $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ to denote the domain defined above (using $\mathcal{P}$), under the standard information ordering. In the next section we show how the domain $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ enjoys many properties like least upper bound, greatest lower bound etc. Based on this we can show that $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ is a pointed complete partial order.

Lemma 3.1.1 *$\mathcal{D} \xrightarrow{proj} \mathcal{D}$ is a pointed cpo.*

Proof: First the existence of a least element is established ($ABSENT$). Next we need to prove that every chain has a least upper bound in the domain. Proof by contradiction: Assume that a chain x_i has two incomparable upper bounds r_1 and r_2. Further assume that there is no r such that $r \sqsubseteq r_1$ and $r \sqsubseteq r_2$. Now glbs exist in the domain (proof is in the next section), and thus $r_1 \sqcap r_2$ is an r which is an upper bound less than both r_1 and r_2. Contradiction. If there are an infinite number of r_i, consider the decreasing chain, r_1, $r_1 \sqcap r_2$, $r_1 \sqcap r_2 \sqcap r_3$,.... Since the domain has no infinite decreasing chains, the infinite glb exists. Contradiction. □

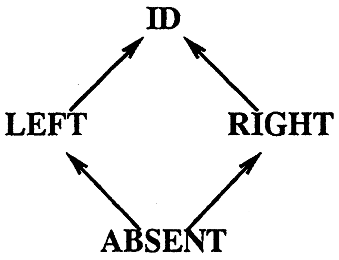

Figure 3.3: Projections on the domain of pairs

Note that any element of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ is guaranteed to be a polymorphic projection. This can be seen from the fact that the projections as defined by $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ do not examine the substructure modified by the strict function α (in the definition of polymorphism).

Example 3.1.4 *Consider projections on the domain of pairs.* $Pair(\tau_1, \tau_2) \xrightarrow{proj} Pair(\tau_1, \tau_2)$ *is shown in Figure 3.3, where* $LEFT(x,y) = (x, \perp)$ *and* $RIGHT(x,y) = (\perp, y)$.

3.1.2 Properties of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$

In this section we investigate algebraic properties of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ like commutativity, distributivity, and so on. This enables us to establish the existence of operators like $\sqcup$, $\sqcap$ and to provide a precise definition of the unique complement.

Definition 3.1.2 *A* commutative *function domain is one whose elements commute wrt function composition.*

Lemma 3.1.2 $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ *is a commutative domain for any domain* $\mathcal{D}$.

Proof: The proof is by induction on the structure of domains. The base projection domains are commutative (because they only contain projections ID and ABSENT). Assume that $d_1 \xrightarrow{proj} d_1$ and $d_2 \xrightarrow{proj} d_2$ are commutative domains, $p_1, p_3 \in d_1 \xrightarrow{proj} d_1$; $p_2, p_4 \in d_2 \xrightarrow{proj} d_2$.

- $(p_1 \times p_2) \circ (p_3 \times p_4) = (p_1 \circ p_3) \times (p_2 \circ p_4) = (p_3 \circ p_1) \times (p_4 \circ p_2) = (p_3 \times p_4) \circ (p_1 \times p_2)$. Thus $d_1 \times d_2 \xrightarrow{proj} d_1 \times d_2$ is commutative.
- $(p_1+p_2)\circ(p_3+p_4) = (p_1\circ p_3)+(p_2\circ p_4) = (p_3\circ p_1)+(p_4\circ p_2) = (p_3+p_4)\circ(p_1+p_2)$. Thus $d_1 + d_2 \xrightarrow{proj} d_1 + d_2$ is commutative.

In the case of the recursive domain, if one of the projections chosen is $ABSENT$ commutativity holds since $p \circ q$ is $ABSENT$ if either p or q is $ABSENT$. If both of the projections are in $\mathcal{P}(\mu\tau.T(\tau))$ (i.e. in the branch other than $ABSENT$ in the recursive domain definition of $\mathcal{P}$), then whether or not the projections commute depends on whether the elements of $\mathcal{P}(\mu\tau.T(\tau))$ chosen commute. By the induction hypothesis, $T(\tau)$ commutes whenever τ does. Repeating the earlier argument, if $ABSENT$ is chosen for either of the projections, we are done. By repeating this argument, as long as $ABSENT$ is chosen at some level of recursion, the projections commute. The other case is when $ABSENT$ is never chosen. This represents the single projection $\mu p.T(p)$. In this case since both the projections have to be the same, they commute. □

We prove the following properties (Lemmas 3.1.2 through 3.1.4) for any p, q from a commutative projection domain.

Lemma 3.1.3 *$p \circ q$ is a projection.*

Proof: No information addition: $p \sqsubseteq ID$ and $q \sqsubseteq ID \;\Rightarrow\; p \circ q \sqsubseteq ID$. Idempotence: $(p \circ q) \circ (p \circ q) = p \circ (q \circ p) \circ q = p \circ (p \circ q) \circ q = (p \circ p) \circ (q \circ q) = p \circ q$. □

Lemma 3.1.4 *The greatest lower bound (glb) of p and q exists and is $p \circ q$.*

Proof: Note that since $p \sqsubseteq ID$ and $q \sqsubseteq ID$, $p \circ q \sqsubseteq p$ and $p \circ q \sqsubseteq q$. Let r be any projection such that $r \sqsubseteq p$ and $r \sqsubseteq q$. Then by monotonicity, $r \circ r \sqsubseteq p \circ q$ and by the definition of projection, $r \sqsubseteq p \circ q$. □

Lemma 3.1.5 *The least upper bound (lub) exists.*

Proof: (By contradiction.) Suppose that there exists p and q such that $p \sqcup q$

does not exist. Since ID is the top element, this implies that there exist two incomparable upper bounds l_1 and l_2, and that there is no upper bound less than both l_1 and l_2.

(This is because there are no infinite descending chains in $\mathcal{D} \xrightarrow{proj} \mathcal{D}$: From the definition of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$, it can be shown that for any projection p, there are only finitely many projections q such that q is $\sqsubseteq p$. Proof by induction: Base case is the case of primitive domain, where the domains involved are finite. In the case of sums, if the proposition holds for $\mathcal{P}(d_1)$ and $\mathcal{P}(d_2)$, it must hold for $\mathcal{P}(d_1 + d_2)$ because $p_1 + p_2 \sqsubseteq p_3 + p_4$ iff $p_1 \sqsubseteq p_3$ and $p_2 \sqsubseteq p_4$. Similarly for products. In the case of the recursive domain, any projection is constructed with a finite number of applications of the recursive application followed by a single insertion of the $ABSENT$ projection. Thus there can only be finitely many projections less than a given projection.)

But since glbs exist, $l_1 \circ l_2$ is an upper bound of p and q which is (by definition of glb) less than both l_1 and l_2. Contradiction. □

Commutative domains with the additional property of distributivity are of special interest.

Definition 3.1.3 *A domain is said to be* distributive *iff for all elements p, q, r of the domain, $p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$ and $p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$.*

Lemma 3.1.6 *$\mathcal{D} \xrightarrow{proj} \mathcal{D}$ is a distributive domain for any domain $\mathcal{D}$.*

Proof: The proof is by induction on the domain structure. We detail the proof of $p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$; the proof of $p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$ is similar. The distributivity properties of the base projection domains are easy to verify because they only conyain ID and $ABSENT$. Assume that $d_1 \xrightarrow{proj} d_1$ and $d_2 \xrightarrow{proj} d_2$ are distributive, and p_1, q_1, $r_1 \in d_1$ and p_2, q_2, $r_2 \in d_2$. In what follows we write (f, g) for $f \times g$.

- $(p_1, p_2) \sqcup ((q_1, q_2) \sqcap (r_1, r_2)) = (p_1, p_2) \sqcup (q_1 \circ r_1, q_2 \circ r_2) = (p_1 \sqcup (q_1 \circ r_1), p_2 \sqcup (q_2 \circ r_2)) = ((p_1 \sqcup q_1) \sqcap (p_1 \sqcup r_1), (p_2 \sqcup q_2) \sqcap (p_2 \sqcup r_2)) = (p_1 \sqcup$

$q_1, p_2 \sqcup q_2) \sqcap (p_1 \sqcup r_1, p_2 \sqcup r_2) = ((p_1, p_2) \sqcup (q_1, q_2)) \sqcap ((p_1, p_2) \sqcup (r_1, r_2))$. Thus $d_1 \times d_2 \xrightarrow{proj} d_1 \times d_2$ is distributive.

- The proof for $d_1 + d_2$ works similarly.

- In the case of the recursive domain construction, first note that if any of p, q or r is chosen as $ABSENT$, the distributivity property holds. If none of them are $ABSENT$, then each of them must be in $\mathcal{P}(\mu\tau.T(\tau))$ (definition of the recursive domain in $\mathcal{P}$. Now by the induction hypothesis, , we know that $T(\tau)$ commutes whenever τ does. Thus if any of the τs are chosen as $ABSENT$ then we are done. We can repeat this argument to show that as long as $ABSENT$ is chosen at some level of recursion, commutativity holds. The other case (when $ABSENT$ is never chosen) represents the single projection $\mu p.T(p)$. In this case since the three projections are the same, distributivity holds.

□

Lemma 3.1.7 *For any p, q in a commutative distributive domain, there exists a least r such that $p \sqsubseteq q \sqcup r$.*

Proof: Clearly there is always at least one r which satisfies the definition (take $r = ID$). Assume we have two incomparable elements r_1 and r_2 such that $p \sqsubseteq q \sqcup r_1$ and $p \sqsubseteq q \sqcup r_2$. Also assume that there is no element smaller than both r_1 and r_2 satisfying the difference condition. Then $p \sqsubseteq q \sqcup r_1$ and $p \sqsubseteq q \sqcup r_2$, implies (from definition of glb) $p \sqsubseteq (q \sqcup r_1) \sqcap (q \sqcup r_2)$ and by distributivity: $p \sqsubseteq q \sqcup (r_1 \sqcap r_2)$. But $r_1 \sqcap r_2$ is less than r_1 and r_2 and it satisfies the difference equation. Contradiction. (In case there are an infinite number of r_i satisfying the above condition, the existence of the infinite glb needs to be shown. The infinite glb exists because we know that the domain has no infinite decreasing chains and r_1, $r_1 \sqcap r_2$, $r_1 \sqcap r_2 \sqcap r_3$,... is a decreasing chain. It is also easy to see that lub distributes over the infinite glb.) □

For domains which are distributive in addition to being commutative, we define the difference operation as follows:

Definition 3.1.4 *If p and q are elements of a commutative distributive domain, the* difference *of p and q (written $p - q$) is the least r such that $p \sqsubseteq q \sqcup r$.*

Lemma 3.1.6 ensures that the difference is uniquely and well defined. This leads us, as promised, to a unique definition of complement.

Definition 3.1.5 *The* complement $\bar{p}$ *of an element p in a commutative distributive domain is $ID - p$.*

3.1.3 Algebraic Properties of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$

The properties that we have thus far defined were motivated by our context. Interestingly, they form what is known as a *Brouwerian algebra.*

Definition 3.1.6 *A* Brouwerian algebra *is a 5-tuple $\langle L, \sqcup, \sqcap, \dot{-}, \top \rangle$ where $\langle L, \sqcup, \sqcap \rangle$ is a lattice with greatest element $\top$, L is closed under $\dot{-}$, and $a \dot{-} b \sqsubseteq c$ iff $a \sqsubseteq b \sqcup c$.*

A Brouwerian algebra can be seen as a generalization of a Boolean algebra where the following equation need not hold: $ID - (ID - p) = p$. This allows us to succintly summarize the properties of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$.

Theorem 3.1.1 *$\langle \mathcal{D} \xrightarrow{proj} \mathcal{D}, \sqcup, \sqcap, -, ID \rangle$ is a Brouwerian algebra.*
Proof: Follows directly from lemmas 3.1.3 through 3.1.6. □

Knowing that we are dealing with a Brouwerian algebra allows us to use known properties of such algebras for reasoning about our projection domains. For example, when specifying an incremental program, the partition we are interested in may only be a subset of $\mathcal{D} \xrightarrow{proj} \mathcal{D}$, and thus we may need to extend the domain to make it Brouwerian; it is a known theorem that such "completions" always exist. Our use of Brouwerian algebras is inspired by an interesting and much more extensive application of the concept: the modelling of *program integration*

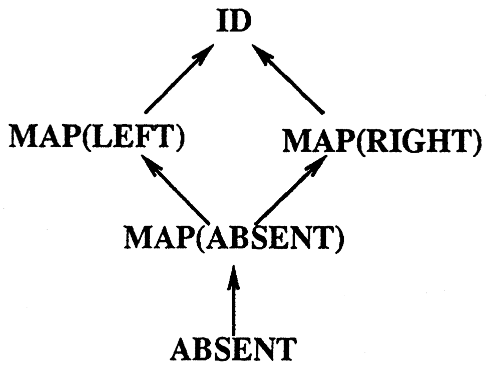

Figure 3.4: Projections on the domain of lists of pairs

[Rep90]. The following theorem partially addresses this problem, quoted without proof from [MT46]:

Theorem 3.1.2 *If* $\langle \mathcal{L}, \sqcup, \sqcap, -, \top \rangle$ *is a Brouwerian algebra, and* $\mathcal{M}$ *is a finite subset of* $\mathcal{L}$ *containing* n *elements, then there exists a subset* $\mathcal{L}'$ *of* $\mathcal{L}$ *and an operation* $-'$ *with the following properties:*

- $\langle \mathcal{L}', \sqcup, \sqcap, -', \top \rangle$ *is a Brouwerian algebra.*
- $\mathcal{L}'$ *contains at most* 2^{2^n} *elements.*
- $\mathcal{M}$ *is a subset of* $\mathcal{L}'$.
- *If* x, y *and* $x - y$ *are in* $\mathcal{L}'$, *then* $x -' y = x - y$.

One possible $\mathcal{L}'$ is the set of all elements of $\mathcal{L}$ which are expressible as $\sqcup$s and $\sqcap$s of elements of $\mathcal{M}$.

Example 3.1.5 *Consider projections on the domain of lists of pairs.* $\mathcal{D} = List(Pair(a, b))$. *A finite subdomain of* $\mathcal{D} \xrightarrow{proj} \mathcal{D}$ *is shown in Figure 3.4. LEFT, RIGHT and ABSENT are projections on pairs. Here* $\mathcal{L} = \mathcal{D} \xrightarrow{proj} \mathcal{D}$ *and* $\mathcal{L}'$ *and* $\mathcal{M}$ *are the subdomain shown in the figure. However, if ID were not one of the projections, then the algebra would not be Brouwerian.*

Restricting ourselves to polymorphic projections enables us to show other simple and desirable properties of partial evaluation, for example that the order of partial evaluation is immaterial:

Theorem 3.1.3 *If partial evaluation terminates, the order of partial evaluation is immaterial: If projections p and q belong to a commutative distributive domain,* $\mathcal{PE}(\mathcal{PE}\ f\ p\ a)q\ a = \mathcal{PE}(\mathcal{PE}\ f\ q\ a)p\ a$, *where f is the function being specialized, and a is its argument.*

Proof: By the definition of partial evaluation the two residual functions in the equation above (call them r_1 and r_2) (assuming all the specializations terminate) need to satisfy: $r_1\ ((\bar{q} \circ \bar{p})a) = f\ a$ and $r_2\ ((\bar{p} \circ \bar{q})a) = f\ a$. But, since our projections commute, $\bar{q} \circ \bar{p} = \bar{p} \circ \bar{q}$. Thus $r_1 = r_2$. □

3.2 Residual Function Algebras

Now that we have obtained these properties about the domain of projections, let us see how they affect the domain of residual functions. We demonstrate that the domain of projections in the last section induces an homomorphic domain of residual functions. This makes precise the notion of "combining" residual functions which was referred to earlier. Before we can proceed, we define the domain of residual functions under consideration:

Definition 3.2.1 *The domain* $\mathcal{R}$ *of residual functions for a function f, its argument a, and a commutative, distributive domain of projections* $\mathcal{P}$ *is defined as* $\mathcal{R} = \{r \mid r = \mathcal{PE}\ f\ p\ a,\ p \in \mathcal{P}\ \}$. *The ordering relation on* $\mathcal{R}$ *is defined as follows:* $r_p \sqsubseteq r_q$ *iff* $p \sqsubseteq q$.

It is easy to verify that the ordering on residuals is a partial order. This ordering is intimately related to the standard information ordering. If the type of f is $A \rightarrow B$, the type of a residual function for a fixed argument a is $A' \rightarrow B$ where A' is the subdomain of A of elements $\sqsubseteq a$. The monotonicity of the $\mathcal{PE}$ implies:

$$p \sqsubseteq q \;\Rightarrow\; r_p \sqsubseteq r_q$$

Thus, the least upper bound, greatest lower bound and the difference of residual functions are:

$$\begin{aligned} r_p \sqcup r_q &= r_{p \sqcup q}, \\ r_p \sqcap r_q &= r_{p \sqcap q} \; and \\ r_p - r_q &= r_{p-q} \end{aligned}$$

We now relate the domain of projections and the domain of residual functions using partial evaluation:

Lemma 3.2.1 *Given a commutative, distributive projection domain $\mathcal{P}$ and its corresponding domain of residual functions $\mathcal{R}$, $\lambda p.\ \mathcal{PE}\ f\ p\ a$ is a homomorphism from $\langle \mathcal{P}, \sqcup, \sqcap, -, ID \rangle$ to $\langle \mathcal{R}, \sqcup, \sqcap, -, r_{ID} \rangle$. Here f is a function to be specialized and a is an element of its domain.*

Proof: Clearly $\mathcal{PE}$ preserves the identities of $\sqcup$, $\sqcap$ and $-$. It follows directly from the lub, glb and difference identities that $\mathcal{PE}$ satisfies:

$$\begin{aligned} \mathcal{PE}\ f\ (p \sqcup q)\ a &= (\mathcal{PE}\ f\ p\ a) \sqcup (\mathcal{PE}\ f\ q\ a) \\ \mathcal{PE}\ f\ (p \sqcap q)\ a &= (\mathcal{PE}\ f\ p\ a) \sqcap (\mathcal{PE}\ f\ q\ a) \\ \mathcal{PE}\ f\ (p - q)\ a &= (\mathcal{PE}\ f\ p\ a) - (\mathcal{PE}\ f\ q\ a) \end{aligned}$$

□

To complete the sequence, we quote the following theorem from [MT46]:

Theorem 3.2.1 *Any homomorphic image of a Brouwerian algebra is a Brouwerian algebra.*

Since the domain of residuals is a homomorphic image of the projection domain, we can state that:

Corollary 3.2.1 $\langle \mathcal{R}, \sqcup, \sqcap, -, r_{ID} \rangle$ *is a Brouwerian algebra.*

Thus the domain of residual functions enjoys the same properties as $\mathcal{D} \xrightarrow{proj} \mathcal{D}$. In our context, namely the incremental interpreter, we identify the $\sqcup$ operation with the combining operation on residual functions. This not only assures us that the operation is well defined and unique, but also that the operation is commutative and associative. This implies that when the interpreter carries out combining operations in the cache of residual functions, it can carry them out in any order it chooses. [1] Also we can view the problem of incremental computation in a different light: trying to approximate the difference operator using the least upper bound. This is so since if the difference operator on residual functions is available, then when the input changes, we can simply apply the appropriately "differenced" residual function to the changed input: If projection p describes the change in the input, then we need to apply r_{ID-p} to the changed input.

[1] Based on a similar property, C. Consel and O. Danvy have investigated partial evaluation in parallel (i.e. on a multiprocessor) [CD90b]

Chapter 4

Combining Residual Functions

In this chapter we present an algorithm to combine residual expressions and present proofs of its correctness and efficiency. The last chapter described three binary operations on residual functions: $\sqcup$, $\sqcap$ and $-$, but did not describe algorithms for them. If we had an efficient algorithm for $-$, the whole problem of incremental computation as we have stated it would be solved!

But this is a difficult operation to compute since it involves "backing up" of computation: identifying which computations have to be undone, and moving to a state where these computations have not yet been performed.

Our methods can be seen as trying to approximate $-$ by using the other two operations. In this thesis (and in all applications that we have investigated to date) we only use $\sqcup$, and thus in this chapter we develop an efficient algorithm for it.

Since we know that $\mathcal{PE}$ is a homomorphism from the domain of projections to the domain of residual functions, the following equality holds:

$$\mathcal{PE}\ f\ (p \sqcup q)\ a = (\mathcal{PE}\ f\ p\ a) \sqcup (\mathcal{PE}\ f\ q\ a)$$

While this gives us a simple method to compute the lub, it is obviously inefficient, since it ignores the work already done in computing r_p and r_q. A good algorithm avoids redoing any reductions already done to compute r_p and r_q. Indeed if our incremental interpreter is to achieve good performance, this is essential.

$$\begin{tikzcd}
 & r_{p \sqcup q} & \\
r_q \arrow[ur] & & r_p \arrow[ul] \\
 & r_{ABSENT} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 4.1: Least upper bound of residual functions

In what follows, we assume that the partial evaluator is implemented using binding time analysis. This technique has been shown to be crucial in achieving self-application of partial evaluators [JSS89]. Thus we begin this chapter with an overview of the *binding time analysis* technique. We also describe an interpretation of the binding time analysis information: *actions* [CD90a]. We then describe the least upper bound algorithm based on the binding time analysis information. Finally we close with proofs of correctness of the algorithm for the first order and higher order cases. Note that while the original projection based framework fits only first order programs, the least upper bound algorithm itself can be stated for arbitrary lambda expressions. Indeed, the correctness criterion for the higher order case is stated without any reference to projections. The correctness proofs are stated using a "two level language notation" (borrowed from [Nie89]). This notation is more conducive to proofs. The action based algorithm is restated in this framework both for first order and for higher order languages.

4.1 Binding time analysis

It has been found useful to split the process of partial evaluation into two stages [JSS89]. The first stage is a preprocessing stage called *binding time analysis.* The second stage is the *specialization* phase. This two fold separation has been found, for example, to be essential in achieving efficient self-application.

The only information known during binding time analysis is: *which arguments are static* ?. The actual values of the static arguments are not known. The main aims of the analysis are to discover all program variables and expressions whose values depend only on static data and to discover which operations can be performed using only the static data.

The result of the binding time analysis phase is an annotated source program – annotated with binding time information about which operations can be executed at specialization time. This information is meant to be used by the specializer. The specializer takes two inputs: the annotated subject program and the static program inputs.

In [CD90a], this framework is extended by annotating the source program with an *interpretation* of the binding time information. These interpretations are in the form of program transformations and the binding time information is used to actually choose a program transformation prior to the actual specialization. [CD90a] has called these transformations *actions*. In what follows we describe a subset of the actions actually used in the partial evaluator SCHISM [Con90a, Con90b, Con88] which has been built using the action framework.

The source program (a λ-term) is represented as a tree with the following kinds of nodes: application, lambda abstraction, variable, constant and conditional. Given a source program and a description of the binding time of its input (in the form of a projection), the result of binding time analysis is an *action tree*, isomorphic to the source program, with the following nodes:

- **Reduce** An action tree which says "process the children of the syntax tree according to the action subtrees rooted at this node and then **Reduce** the node."

- **Rebuild** An action tree which says "process the children of the syntax tree according to the action subtrees rooted at this node and then **Rebuild** the node."

- **Eval.** Fully evaluate the subtree rooted at this node (complete reduction).
- **Id.** Rebuild completely the subtree rooted at this node (no reductions).

The specializer simply processes each node of the source program by executing the corresponding action.

Eval and **Id** are not strictly necessary (other than for optimization purposes), since a subtree with root marked **Eval** can be translated into a subtree with all nodes marked **Reduce** and similarly for **Id**. Thus in the discussion which follows we only consider **Reduce** and **Rebuild**.

4.2 Algorithm for Least Upper Bound of Residual Expressions

We now present an algorithm for the least upper bound of two residual functions. We assume that the binding time information for each function (in the form of an action tree) is available. We describe the algorithm in the context of the call-by-value λ-calculus; it can easily be adapted to languages with more syntactic sugar. Also, we assume that the functions have first been α-converted to avoid any name clashes.

A residual function is described by a triple $\langle r, a, e \rangle$ where r is the λ-term representing the residual function, a is the isomorphic action tree which produced the residual function, and e is an associated environment which is initially empty. The environment maps variables to pairs of the form $\langle t, a \rangle$, where t is a λ-term and a is the associated action tree.

Algorithm LUB($\langle r_1, a_1, e_1 \rangle$, $\langle r_2, a_2, e_2 \rangle$ **)**

- Apply rewrite rules in Figures 4.2 and 4.3 to $\langle r_1, a_1, e_1 \rangle \sqcup \langle r_2, a_2, e_2 \rangle$ until all $\sqcup$ symbols are removed from the term.
- Reduce nodes all of whose children are reduced. This is needed to perform reductions not performed by either of the two residual functions but made

possible by their combination.

To reduce the number of rules, symmetric cases have been omitted from Figures 4.2 and 4.3. *The main motivation behind the construction of the rules is to avoid doing any reduction which has already been done in any one of the other residual functions.* For example in rule 1, an application has been reduced in one residual function but has been left residual in another. The rule reduces this to taking the $\sqcup$ of the bodies of the abstractions, but also updates the environment of the second residual function so as not to lose reductions performed in the argument. The rules are designed so as to pattern match on action trees corresponding to each residual function. Each rule chooses the more evaluated over the less evaluated so as to construct the least upper bound from fragments of each residual function. The question marks in some of the rules indicate "dont cares" conditions. Constants in the rules are denoted by c.

The main complication arises from the case where an application is reduced in one residual and rebuilt in another. In this case it does not suffice to simply choose the more evaluated term. This is because the argument portion of the rebuilt application may incorporate some reductions which we do not wish to lose. This can be achieved by saving the argument in an environment. The environment is looked up when the variable is reached in the least upper bound process.

The rules for conditional reflect the fact that for the algorithm to proceed, we need to know which way the condition was resolved. This information can be stored by the partial evaluator during specialization.

Finally, there is a post-processing step to be executed, which we explain by an example:

```
g (x,y) = if (x == 0) then f (x,y) else f (x,y) - 5
f (x,y) = x*x - y*y
```

Specializing g with the projection $LEFT$ and argument `(0,2)` gives:[1]

[1] Note that since residual functions have the same type as the function being specialized, g1

```
  g1 (x,y) = 0 - y*y
```

Specializing g with the projection *RIGHT* and the same argument (0,2) gives:

```
  g2 (x,y) = if (x == 0) then x*x - 4 else x*x - 9
```

The action tree for g corresponding to g1 (written as an annotation to the source code) is:

```
if Reduce  (== Reduce x Reduce 0 Reduce)
           (f Reduce x Reduce y Rebuild)
           (- Rebuild (f Reduce x Reduce y Rebuild)
                      5 Reduce)
```

The action tree for g corresponding to g2 (written as an annotation to the source code) is:

```
if Rebuild  (== Rebuild x Rebuild 0 Reduce)
            (f Reduce x Rebuild y Reduce)
            (- Rebuild (f Reduce x Rebuild y Reduce)
                       5 Reduce)
```

g1 $\sqcup$ g2 according to the above algorithm is: (the action tree corresponding to g1 marks the conditional as *STATIC* and the action tree corresponding to g2 marks it as *DYNAMIC*)

```
  g12 (x,y) = 0 - 4
```

The first step in applying the **LUB** algorithm is to use the rule for the conditional. This rule forces us to choose the left hand branch of both the conditionals, because g1 resolves the conditional to the consequent branch. The two residuals at this point of the computation are as follows:

```
g1: 0 - y*y
g2: x*x - 4
```

still takes a pair as argument. It simply ignores the left element of the pair. A similar comment applies to g2.

The next step is to use the function application rule. Since both the residuals unroll the function application, the algorithm moves to the body of `f`. At this point, the algorithm rebuilds the - since it is rebuilt in both the residuals. The process moves on to comparing the arms of the -, selecting one arm from each residual. At this point in the computation, there are two least upper bound processes:

```
g1: 0 ; g2: x*x
g1: y*y ; g2: 4
```

Note how the algorithm selects the more evaluated parts of each residual: the conditional from g2, the `x*x` from g1 and so on.

Clearly this can be further reduced; i.e., there may be reductions in the least upper bound which are neither in g1 nor g2. These can be performed through another phase of partial evaluation with no arguments.

4.3 Proofs of Correctness and Efficiency

Note that as long as there is still a $\sqcup$ in a term, one of the rules apply. Termination of the first phase of the algorithm is not difficult to verify: each rule reduces the size of the λ-term under consideration, until the base case is reached. The second phase may not terminate when the source program contains non-terminating computations, but this is common among partial evaluators, so we won't be concerned about it here.

Theorem 4.3.1 *Algorithm* **LUB** *correctly computes the least upper bound.*
Proof:

To carry out these proofs we use the two level lambda calculus notation first used by Nielson [Nie89], which is more conducive to proofs. To carry out the proofs, we restate the algorithms in the two level framework. The proofs are then carried out for the first order and the higher order cases. The proof is given

- $< c,\ \mathbf{Reduce}\ c,\ \rho_1 > \sqcup < c,\ \mathbf{Reduce}\ c,\ \rho_2 > \Rightarrow c$

- $< c,\ \mathbf{Reduce}\ c,\ \rho_1 > \sqcup < c,\ \mathbf{Rebuild}\ c,\ \rho_2 > \Rightarrow c$

- $< c,\ \mathbf{Rebuild}\ c,\ \rho_1 > \sqcup < c,\ \mathbf{Rebuild}\ c,\ \rho_2 > \Rightarrow c$

- $\langle\ r_1,\ \mathbf{Reduce}(a_{11})(a_{12}),\ e_1\ \rangle \sqcup$
  $\langle\ t_{21}t_{22},\ \mathbf{Rebuild}(a_{21})(a_{22}),\ e_2\ \rangle$
  $\Longrightarrow\ \langle\ \lambda ?.r_1,\ a_{11},\ e_1\ \rangle \sqcup \langle\ t_{21},\ a_{21},\ e_2[?/\langle\ t_{22}, a_{22}\ \rangle]\ \rangle$

- $\langle\ \lambda ?.r_1,\ \mathbf{Reduce}(a_1),\ e_1\ \rangle \sqcup$
  $\langle\ \lambda x.t_2,\ \mathbf{Rebuild}(a_2),\ e_2[?/\langle\ t_{22}, a_{22}\ \rangle]\rangle$
  $\Longrightarrow\ \langle\ r_1,\ a_1,\ e_1\ \rangle \sqcup \langle\ t_2,\ a_2,\ e_2[x/\langle\ t_{22}, a_{22}\ \rangle]\ \rangle$

- $\langle\ r_1,\ \mathbf{Reduce}(a_{11})(a_{12}),\ e_1[x/\langle\ t_1, a_1\ \rangle]\rangle \sqcup$
  $\langle\ x,\ \mathbf{Rebuild}()(),\ e_2[x/\langle\ t_2, a_2\ \rangle]\ \rangle$
  $\Longrightarrow \langle\ r_1, a_1,\ e_1\ \rangle \sqcup \langle\ t_2,\ a_2,\ e_2\rangle$

- $\langle\ r_1,\ \mathbf{Reduce}(a_{11})(a_{12}),\ e_1\ \rangle \sqcup$
  $\langle\ r_2,\ \mathbf{Reduce}(a_{21})(a_{22}),\ e_2\ \rangle$
  $\Longrightarrow\ \langle\ r_1,\ a_{11},\ e_1\ \rangle \sqcup \langle\ r_2,\ a_{21},\ e_2\ \rangle$ (If the node reduced was an application)

- $\langle\ r_1,\ \mathbf{Reduce}(a_{11})(a_{12}),\ e_1[x/\langle\ t_1, a_1\ \rangle]\ \rangle \sqcup$
  $\langle\ r_2,\ \mathbf{Reduce}(a_{21})(a_{22}),\ e_2[x/\langle\ t_2, a_2\ \rangle]\ \rangle$
  $\Longrightarrow\ \langle\ r_1,\ a_1,\ e_1\ \rangle \sqcup \langle\ r_2,\ a_2,\ e_2\ \rangle$(If the node reduced was a variable x)

- $\langle\ t_{11}t_{12},\ \mathbf{Rebuild}(a_{11})(a_{12}),\ e_1\ \rangle \sqcup$
  $\langle\ t_{21}t_{22},\ \mathbf{Rebuild}(a_{21})(a_{22}),\ e_2\ \rangle$
  $\Longrightarrow (\langle\ t_{11},\ a_{11},\ e_1\ \rangle \sqcup \langle\ t_{21},\ a_{21},\ e_2\ \rangle)\ (\langle\ t_{12},\ a_{12},\ e_1\ \rangle \sqcup \langle t_{22},\ a_{22},\ e_2\ \rangle)$

Figure 4.2: Rewrite rules for algorithm **LUB**

- $\langle\ x,\ \textbf{Rebuild()()},\ e_1[x/\langle t_1, a_1\rangle]\ \rangle \sqcup$
  $\langle\ x,\ \textbf{Rebuild()()},\ e_2[x/\langle t_2, a_2\rangle]\ \rangle \Longrightarrow \langle t_1, a_1, e_1\rangle \sqcup \langle t_2, a_2, e_2\rangle$

- $\langle\ x,\ \textbf{Rebuild()()},\ e_1\ \rangle \sqcup$
  $\langle\ x,\ \textbf{Rebuild()()},\ e_2\ \rangle \Longrightarrow x$ (if both e_1 and e_2 do not have bindings for x).

- $\langle\ \lambda x.t_1,\ \textbf{Rebuild}(a_1),\ e_1\ \rangle \sqcup \langle\ \lambda x.t_2,\ \textbf{Rebuild}(a_2),\ e_2\ \rangle$
  $\Longrightarrow \lambda x.(\langle\ t_1, a_1, e_1\rangle\ \sqcup\ \langle\ t_2, a_2, e_2\rangle)$

- $\langle\ \lambda x.t_1,\ \textbf{Reduce}(a_1),\ e_1\ \rangle \sqcup \langle\ \lambda x.t_2,\ \textbf{Reduce}(a_2),\ e_2\rangle$
  $\Longrightarrow \lambda x.(\langle\ t_1, a_1, e_1\rangle\ \sqcup\ \langle\ t_2, a_2, e_2\rangle)$

- $\langle r_1,\ \textbf{Reduce}\ e_{11}\ e_{12}\ e_{13},\ \rho_1\rangle \sqcup \langle r_2,\ \textbf{Reduce}\ e_{21}\ e_{22}\ e_{23},\ \rho_2\rangle \Rightarrow$
  $\langle r_1,\ choose(e_{12}, e_{13}),\ \rho_1\rangle \sqcup \langle r_2,\ choose(e_{22}, e_{23}),\ \rho_2\rangle$

- $\langle r_1, \textbf{Reduce}\ e_{11}\ e_{12}\ e_{13},\ \rho_1\rangle \sqcup \langle if\ r_{21}\ r_{22}\ r_{23}, \textbf{Rebuild}\ e_{21}\ e_{22}\ e_{23},\ \rho_2\rangle \Rightarrow$
  $\langle r_1,\ choose(e_{12}, e_{13}),\ \rho_1\rangle \sqcup \langle choose(r_{22}, r_{23}),\ choose(e_{22}, e_{23}),\ \rho_2\rangle$

- $\langle if\ r_{11}\ r_{12}\ r_{13}, \textbf{Rebuild}\ e_{11}\ e_{12}\ e_{13},\ \rho_1\rangle \sqcup$
  $\langle if\ r_{21}\ r_{22}\ r_{23}, \textbf{Rebuild}\ e_{21}\ e_{22}\ e_{23},\ \rho_2\rangle \Rightarrow$
  $if\ (\langle r_{11}, e_{11}, \rho_1\rangle\ \sqcup\ \langle r_{21}, e_{21}, \rho_2\rangle)$
  $(\langle r_{12}, e_{12}, \rho_1\rangle\ \sqcup\ \langle r_{22}, e_{22}, \rho_2\rangle)$
  $(\langle r_{13}, e_{13}, \rho_1\rangle\ \sqcup\ \langle r_{23}, e_{23}, \rho_2\rangle)$

Figure 4.3: Rewrite rules for algorithm **LUB**

for two different cases: for a first order language and for a higher order language. Although the higher order case subsumes the first order case, we include both for completeness. For the first order language, we first restate the rules for algorithm **LUB**. The proofs are carried out by structural induction.

4.3.1 A Two-Level First Order Language

The abstract syntax for a first order language is given below. Note that each expression arises in two forms, underlined and plain. The underlined expressions represent expressions which are rebuilt while plain expressions represent reduced expressions. The result of the binding time analysis phase is such a two-level expression. This notation is borrowed from the two level lambda calculus notation found in [JGB+90].

$$
\begin{array}{lcll}
c & \in & Con & constants \\
x & \in & Bv & bound\ variables \\
p & \in & Pf & primitive\ functions \\
f & \in & Fv & function\ variables \\
e & \in & Exp & expressions,\ where \\
 & & & e \;=\; c \mid x \mid p(e_1, ..., e_n) \mid if\ e_1\ e_2\ e_3 \mid f(e_1, ..., e_n) \\
 & & & \qquad \underline{c} \mid \underline{x} \mid \underline{p}(e_1, ..., e_n) \mid \underline{if}\ e_1\ e_2\ e_3 \mid \underline{f}(e_1, ..., e_n) \\
pr & \in & Prog & programs,\ where \\
 & & & pr \;=\; \{f_i(x_i, ..., x_n) = e_i\}
\end{array}
$$

4.3.2 The LUB Algorithm

In this section we state the rules for the **LUB** algorithm for the first order language. We begin with the case of constants:

$$
\begin{array}{lcll}
< c,\ c,\ \rho_1 > & \sqcup & < c,\ c,\ \rho_2 > & \Rightarrow\ c \\
< c,\ c,\ \rho_1 > & \sqcup & < c,\ \underline{c},\ \rho_2 > & \Rightarrow\ c \\
< c,\ \underline{c},\ \rho_1 > & \sqcup & < c,\ \underline{c},\ \rho_2 > & \Rightarrow\ c
\end{array}
$$

We then present the rules for variables:

$$\begin{array}{lll}
< r_1,\ x,\ \rho_1[x/(t_1,c_1)] > & \sqcup & < r_2,\ x,\ \rho_2[x/(t_2,c_2)] > \ \Rightarrow \\
< r_1,\ c_1,\ \rho_1 > & \sqcup & < r_2,\ c_2,\ \rho_2 > \\
< r_1,\ x,\ \rho_1[x/(t_1,c_1)] > & \sqcup & < r_2,\ \underline{x},\ \rho_2[x/(t_2,c_2)] > \ \Rightarrow \\
< r_1,\ c_1,\ \rho_1 > & \sqcup & < t_2,\ c_2,\ \rho_2 > \\
< r_1,\ \underline{x},\ \rho_1[x/(t_1,c_1)] > & \sqcup & < r_2,\ \underline{x},\ \rho_2[x/(t_2,c_2)] > \ \Rightarrow \ x
\end{array}$$

We make the assumption that a primitive can be reduced only if each of its arguments is completely static (free from underlining). Symmetric cases have been omitted.

$$\begin{array}{lll}
< r,\ p(c_1,...,c_n),\ \rho_1 > & \sqcup & < r,\ p(c_1,...,c_n),\ \rho_2 > \\
\Rightarrow r & & \\
< r_1,\ p(c_{11},...,c_{1n}),\ \rho_1 > & \sqcup & < r_2,\ \underline{p}(c_{21},...,c_{2n}),\ \rho_2 > \\
\Rightarrow r_1 & & \\
< p(e_{11},...,e_{1n}),\ \underline{p}(c_{11},...,c_{1n}),\ \rho_1 > & \sqcup & < p(e_{21},...,e_{2n}),\ \underline{p}(c_{21},...,c_{2n}),\ \rho_2 > \\
 & & \Rightarrow
\end{array}$$

$$p(< e_{11}, c_{11}, \rho_1 > \ \sqcup \ < e_{21}, c_{21}, \rho_2 > \ , \ ... \ , \ < e_{1n}, c_{1n}, \rho_1 > \ \sqcup \ < e_{2n}, c_{2n}, \rho_2 >)$$

The rules for the *if* construct are given below. It is assumed that the information about which way the *if* was reduced is available. This is reflected in the *choose* function:

$$\begin{array}{lcl}
< r_1,\ if\ e_{11}\ e_{12}\ e_{13},\ \rho_1 > & \sqcup & < r_2,\ if\ e_{21}\ e_{22}\ e_{23},\ \rho_2 > \\
 & \Rightarrow & \\
< r_1,\ choose(e_{12}, e_{13}),\ \rho_1 > & \sqcup & < r_2,\ choose(e_{22}, e_{23}),\ \rho_2 > \\
< r_1,\ if\ e_{11}\ e_{12}\ e_{13},\ \rho_1 > & \sqcup & < if\ r_{21}\ r_{22}\ r_{23},\ \underline{if}\ e_{21}\ e_{22}\ e_{23},\ \rho_2 > \\
 & \Rightarrow & \\
< r_1,\ choose(e_{12}, e_{13}),\ \rho_1 > & \sqcup & < choose(r_{22}, r_{23}),\ choose(e_{22}, e_{23}),\ \rho_2 > \\
< if\ r_{11}\ r_{12}\ r_{13},\ \underline{if}\ e_{11}\ e_{12}\ e_{13},\ \rho_1 > & \sqcup & < if\ r_{21}\ r_{22}\ r_{23},\ \underline{if}\ e_{21}\ e_{22}\ e_{23},\ \rho_2 > \\
 & \Rightarrow & \\
if\ (< r_{11}, e_{11}, \rho_1 > \ \sqcup\ < r_{21}, e_{21}, \rho_2 >) & & (< r_{12}, e_{12}, \rho_1 > \ \sqcup\ < r_{22}, e_{22}, \rho_2 >) \\
\quad (< r_{13}, e_{13}, \rho_1 > \ \sqcup\ < r_{23}, e_{23}, \rho_2 >) & &
\end{array}$$

exp_{1i} is the two level expression corresponding to the body of the function f_i using the first binding time information.

$$\begin{array}{lcl}
< r_1,\ f_i(e_{11}, ..., e_{1n}),\ \rho_1 > & \sqcup & < r_2,\ f_i(e_{21}, ..., e_{2n}),\ \rho_2 > \\
 & \Rightarrow & \\
< r_1,\ exp_{1i},\ \rho_1[x_i/(\_, e_{1i})] > & \sqcup & < r_2,\ exp_{2i},\ \rho_2[x_i/(\_, e_{2i})] > \\
< r_1,\ f_i(e_{11}, ..., e_{1n}),\ \rho_1 > & \sqcup & < f_i(r_{21}, ..., r_{2n}),\ \underline{f}_i(e_{21}, ..., e_{2n}),\ \rho_2 > \\
 & \Rightarrow & \\
< r_1,\ exp_{1i},\ \rho_1[x_i/(,e_{1i})] > & \sqcup & < exp_{2i},\ exp_{2i},\ \rho_2[x_i/(r_{2i}, e_{2i})] > \\
< f_i(r_{11}, ..., r_{1n}),\ \underline{f}_i(e_{11}, ..., e_{1n}),\ \rho_1 > & \sqcup & < f_i(r_{21}, ..., r_{2n}),\ \underline{f}_i(e_{21}, ..., e_{2n}),\ \rho_2 > \\
 & \Rightarrow & \\
f_i(< r_{11}, e_{11}, \rho_1 > \ \sqcup\ < r_{21}, e_{21}, \rho_2 > & , \ldots , & < r_{1n}, e_{1n}, \rho_1 > \ \sqcup\ < r_{2n}, e_{2n}, \rho_2 >)
\end{array}$$

4.3.3 Proof of Correctness

The function ϕ erases the underlinings of a two-level term. We first define a relation which helps us state the correctness criterion more precisely.

Definition 4.3.1 *We define a relation $\mathcal{L}$ defined as follows: $\mathcal{L}(t_1, t_2, \rho_1, \rho_2)$ if and only if*

$$< r_1, t_1, \rho_1 > \ \sqcup\ < r_2, t_2, \rho_2 > \ = \ \mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2)$$

provided

- $r_1 = \mathcal{PE}[\![t_1]\!]\rho_1$
- $r_2 = \mathcal{PE}[\![t_2]\!]\rho_2$
- $\phi(t_1) = \phi(t_2)$
- $t_1 \oplus t_2$ *is defined as the two level term such that* $\phi(t_1 \oplus t_2) = \phi(t_1) = \phi(t_2)$ *and having a subcomponent underlined if it is underlined both in* t_1 *and* t_2.
- *If both* ρ_1 *and* ρ_2 *have bindings for* x *then* $\mathcal{L}(snd(\rho_1(x)), snd(\rho_2(x)), \rho_1 \backslash x, \rho_2 \backslash x)$.
- $(\rho_1 \oplus \rho_2)(x) = if\ \rho_1(x) = error^2\ then\ \rho_2(x)\ elseif\ \rho_2(x) = error\ then\ \rho_1(x)\ else$ $(< fst(\rho_1(x)), snd(\rho_1(x)), \rho_1 \backslash x > \sqcup < fst(\rho_2(x)), snd(\rho_2(x)), \rho_2 \backslash x >)$

Now the correctness criterion we wish to prove is:

Theorem 4.3.2 *If* $\phi(t_1) = \phi(t_2)$ *then* $\mathcal{L}(t_1, t_2)$.

Proof: Follows from the following Lemmas.

The proof proceeds by structural induction. The base case is verified first. In each of the cases the induction hypothesis is assumed to hold: namely that the theorem hold true for the corresponding subcomponents. First we handle constants:

Lemma 4.3.1 *If* $\phi(t_1) = \phi(t_2) = c$ *then* $\mathcal{L}(t_1, t_2)$.

Proof: There are four possible cases:

1. $t_1 = c$ and $t_2 = c$.

   $< c, c, \rho_1 > \sqcup < c, c, \rho_2 > \Rightarrow$

   $c \Rightarrow$

   $\mathcal{PE}[\![c \oplus c]\!](\rho_1 \oplus \rho_2)$

[2] i.e. ρ does not have a binding for x

2. $t_1 = c$ and $t_2 = \underline{c}$.

   $< c, c, \rho_1 > \sqcup < c, \underline{c}, \rho_2 > \Rightarrow$

   $c \Rightarrow$

   $\mathcal{PE}[\![c \oplus \underline{c}]\!](\rho_1 \oplus \rho_2)$

3. $t_1 = \underline{c}$ and $t_2 = c$.

   Proof is analogous to the last case.

4. $t_1 = \underline{c}$ and $t_2 = \underline{c}$.

   $< c, \underline{c}, \rho_1 > \sqcup < c, \underline{c}, \rho_2 > \Rightarrow$

   $c \Rightarrow$

   $\mathcal{PE}[\![\underline{c} \oplus \underline{c}]\!](\rho_1 \oplus \rho_2)$

Then we handle variables:

Lemma 4.3.2 *If* $\phi(t_1) = \phi(t_2) = x$ *then* $\mathcal{L}(t_1, t_2)$.

Proof: There are four possible cases:

1. $t_1 = x$ and $t_2 = x$.

   $\mathcal{PE}[\![x \oplus x]\!]\ (\rho_1[x/(t_1, c_1)] \oplus \rho_2[x/(t_2, c_2)]) \Rightarrow$ (Definition of $\oplus$)

   $(\rho_1[x/(t_1, c_1)] \oplus \rho_2[x/(t_2, c_2)])x \Rightarrow$ (Definition of $\oplus$)

   $< t_1, c_1, \rho_1 \backslash\ x > \sqcup < t_2, c_2, \rho_2 \backslash\ x > \Rightarrow$ (Algorithm LUB)

   $< t_1, x, \rho_1[x/(t_1, c_1)] > \sqcup < t_2, x, \rho_2[x/(t_2, c_2)] >$

2. $t_1 = x$ and $t_2 = \underline{x}$.

   $\mathcal{PE}[\![x \oplus \underline{x}]\!]\ (\rho_1[x/(t_1, c_1)] \oplus \rho_2[x/(t_2, c_2)]) \Rightarrow$ (Definition of $\oplus$)

   $(\rho_1[x/(t_1, c_1)] \oplus \rho_2[x/(t_2, c_2)])x \Rightarrow$ (Definition of $\oplus$)

   $< t_1, c_1, \rho_1 \backslash\ x > \sqcup < t_2, c_2, \rho_2 \backslash\ x > \Rightarrow$ (Algorithm LUB)

   $< t_1, x, \rho_1[x/(t_1, c_1)] > \sqcup < x, \underline{x}, \rho_2[x/(t_2, c_2)] >$

3. $t_1 = \underline{x}$ and $t_2 = x$.

   Proof is analogous to the last case.

4. $t_1 = \underline{x}$ and $t_2 = \underline{x}$.

   $\mathcal{PE}[\![\underline{x} \oplus \underline{x}]\!]\ (\rho_1[x/(t_1, c_1)] \oplus \rho_2[x/(t_2, c_2)]) \Rightarrow$ (Definition of $\oplus$)

$x \Rightarrow$ (Algorithm LUB)
$< x, \underline{x}, \rho_1[x/(t_1, c_1)] > \ \sqcup \ < x, \underline{x}, \rho_2[x/(t_2, c_2)] >$

Define the function $\not\psi$ which erases underlinings at only one level. Next we handle primitives:

Lemma 4.3.3 *If* $\not\psi(t_1) = p(t_{11}, ..., t_{1n})$ *and* $\not\psi(t_2) = p(t_{21}, ..., t_{2n})$ *and* $\mathcal{L}(t_{1i}, t_{2i})$ *then* $\mathcal{L}(t_1, t_2)$.

Proof: There are four possible cases:

1. $t_1 = p(t_{11}, ..., t_{1n})$ and $t_2 = p(t_{21}, ..., t_{2n})$.
   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Since both t_1 and t_2 are free of underlinings)
   $c \Rightarrow$ (Algorithm LUB)
   $< c, p(t_{11}, ..., t_{1n}), \rho_1 > \ \sqcup \ < c, p(t_{21}, ..., t_{2n}), \rho_2 >$

2. $t_1 = p(t_{11}, ..., t_{1n})$ and $t_2 = \underline{p}(t_{21}, ..., t_{2n})$.
   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Since both t_1 is free of underlinings)
   $c \Rightarrow$ (Algorithm LUB)
   $< c, p(t_{11}, ..., t_{1n}), \rho_1 > \ \sqcup \ < r, \underline{p}(t_{21}, ..., t_{2n}), \rho_2 >$

3. $t_1 = \underline{p}(t_{11}, ..., t_{1n})$ and $t_2 = p(t_{21}, ..., t_{2n})$.
   Proof is analogous to the last case.

4. $t_1 = \underline{p}(t_{11}, ..., t_{1n})$ and $t_2 = \underline{p}(t_{21}, ..., t_{2n})$.
   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)
   $p(\mathcal{PE}[\![t_{11} \oplus t_{21}]\!](\rho_1 \oplus \rho_2), ..., \mathcal{PE}[\![t_{1n} \oplus t_{2n}]\!](\rho_1 \oplus \rho_2)) \Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$
   $p(< e_{11}, t_{11}, \rho_1 > \ \sqcup \ < e_{21}, t_{21}, \rho_2 >, ..., < e_{1n}, t_{1n}, \rho_1 > \ \sqcup \ < e_{2n}, t_{2n}, \rho_2 >)$
   $\Rightarrow$ (Algorithm LUB)
   $< p(e_{11}, ...e_{1n}), \underline{p}(t_{11}, ..., t_{1n}), \rho_1 > \ \sqcup \ < p(e_{21}, ..., e_{2n}), \underline{p}(t_{21}, ..., t_{2n}), \rho_2 >$

Next we handle conditional statements:

Lemma 4.3.4 *If* $\not\psi(t_1) = if\ t_{11}\, t_{12}\, t_{13}$ *and* $\not\psi(t_2) = if\ t_{21}\, t_{22}\, t_{23}$ *and* $\mathcal{L}(t_{1i}, t_{2i})$ *then* $\mathcal{L}(t_1, t_2)$.

Proof:

There are four possible cases:

1. $t_1 = \mathit{if}\ t_{11}\ t_{12}\ t_{13}$ and $t_2 = \mathit{if}\ t_{21}\ t_{22}\ t_{23}$.

   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)

   $\mathit{choose}(\mathcal{PE}[\![t_{12} \oplus t_{22}]\!](\rho_1 \oplus \rho_2), \mathcal{PE}[\![t_{13} \oplus t_{23}]\!](\rho_1 \oplus \rho_2)) \Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$

   $\mathit{choose}(< r_{12}, t_{12}, \rho_1 > \sqcup < r_{22}, t_{22}, \rho_2 >, < r_{13}, t_{13}, \rho_1 > \sqcup < r_{23}, t_{23}, \rho_2 >$

   $) \Rightarrow$

   $< r_1, \mathit{choose}(t_{12}, t_{22}), \rho_1 > \sqcup < r_2, \mathit{choose}(t_{13}, t_{23}), \rho_2 > \Rightarrow$ (Algorithm LUB)

   $< r_1, \mathit{if}\ t_{11}\ t_{12}\ t_{13}, \rho_1 > \sqcup < r_2, \mathit{if}\ t_{21}\ t_{22}\ t_{23}, \rho_2 >$

2. $t_1 = \mathit{if}\ t_{11}\ t_{12}\ t_{13}$ and $t_2 = \underline{\mathit{if}}\ t_{21}\ t_{22}\ t_{23}$.

   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)

   $\mathit{choose}(\mathcal{PE}[\![t_{12} \oplus t_{22}]\!](\rho_1 \oplus \rho_2), \mathcal{PE}[\![t_{13} \oplus t_{23}]\!](\rho_1 \oplus \rho_2)) \Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$

   $\mathit{choose}(< r_{12}, t_{12}, \rho_1 > \sqcup < r_{22}, t_{22}, \rho_2 >, < r_{13}, t_{13}, \rho_1 > \sqcup < r_{23}, t_{23}, \rho_2 >$

   $) \Rightarrow$

   $< r_1, \mathit{choose}(t_{12}, t_{13}), \rho_1 > \sqcup < \mathit{choose}(r_{22}, r_{23}), \mathit{choose}(t_{22}, t_{23}), \rho_2 > \Rightarrow$ (Algorithm LUB)

   $< r_1, \mathit{if}\ t_{11}\ t_{12}\ t_{13}, \rho_1 > \sqcup < \mathit{if}\ r_{21}\ r_{22}\ r_{23}, \underline{\mathit{if}}\ t_{21}\ t_{22}\ t_{23}, \rho_2 >$

3. $t_1 = \underline{\mathit{if}}\ t_{11}\ t_{12}\ t_{13}$ and $t_2 = \mathit{if}\ t_{21}\ t_{22}\ t_{23}$.

   Proof is analogous to the last case.

4. $t_1 = \underline{\mathit{if}}\ t_{11}\ t_{12}\ t_{13}$ and $t_2 = \underline{\mathit{if}}\ t_{21}\ t_{22}\ t_{23}$.

   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)

   $\mathit{if}\ \mathcal{PE}[\![t_{11} \oplus t_{21}]\!](\rho_1 \oplus \rho_2)\ \mathcal{PE}[\![t_{12} \oplus t_{22}]\!](\rho_1 \oplus \rho_2)\ \mathcal{PE}[\![t_{13} \oplus t_{23}]\!](\rho_1 \oplus \rho_2)$

   $\Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$

   $\mathit{if}\ (< r_{11}, t_{11}, \rho_1 > \sqcup < r_{21}, t_{21}, \rho_2 >)\ (< r_{12}, t_{12}, \rho_1 > \sqcup < r_{22}, t_{22}, \rho_2 >)$

   $(< r_{13}, t_{13}, \rho_1 > \sqcup < r_{23}, t_{23}, \rho_2 >) \Rightarrow$ (Algorithm LUB)

   $< r_1, \underline{\mathit{if}}\ t_{11}\ t_{12}\ t_{13}, \rho_1 > \sqcup < r_2, \underline{\mathit{if}}\ t_{21}\ t_{22}\ t_{23}, \rho_2 >$

Finally we come to the case of function application:

Lemma 4.3.5 *If* $\psi(t_1) = f(t_{11}, ..., t_{1n})$ *and* $\psi(t_2) = f(t_{21}, ..., t_{2n})$ *and* $\mathcal{L}(t_{1i}, t_{2i})$ *then* $\mathcal{L}(t_1, t_2)$.

Proof:

There are four possible cases. In each of the cases we carry out a fixpoint induction. This is necessary since f may be recursively defined. We do not consider mutual recursion since it can be translated in a straightforward manner into direct recursion. F is the functional such that $f = fix\ F$. Note that the only function calls in the body of F are recursive calls.

1. $t_1 = f(t_{11}, ..., t1n)$ and $t_2 = f(t_{21}, ..., t_{2n})$.
   Base case: $\mathcal{L}((\lambda\ x_1, ..., x_n.\ \perp)(t_{11}, ..., t_{1n}), (\lambda\ x_1, ..., x_n.\ \perp)(t_{21}, ..., t_{2n}))$
   Proof:
   $<\perp, (\lambda\, x_1, ..., x_n.\ \perp)(t_{11}, ..., t_{1n}), \rho_1 > \sqcup <\perp, (\lambda\, x_1, ..., x_n.\ \perp)(t_{21}, ..., t_{2n}), \rho_2 >$
   $=$
   $\perp$
   $\mathcal{PE}((\lambda\ x_1, ..., x_n.\ \perp)(t_{11}, ..., t_{1n}) \ \oplus\ (\lambda\ x_1, ..., x_n.\ \perp)(t_{21}, ..., t_{2n}))$
   Induction Step: $\mathcal{L}(x(t_{11}, ..., t_{1n}), x(t_{21}, ..., t_{2n})) \Rightarrow$
   $\mathcal{L}(F_1(x)(t_{11}, ..., t_{1n}), F_2(x)(t_{21}, ..., t_{2n}))$ (where F_1 and F_2 are two differently underlined versions of F)
   Proof:
   $< r_1, F_1(x)(t_{11}, ..., t_{1n})), \rho_1 > \ \sqcup\ < r_2, F_2(x)(t_{21}, ..., t_{2n}), \rho_2 > \Rightarrow$ (Algorithm LUB)
   $< r_1, exp_1[f/x], \rho_1[x_i/(,t_{1i})] > \ \sqcup\ < r_2, exp_2[f/x], \rho_2[x_i/(,t_{2i})] > \Rightarrow$ (Induction hypotheses, note that the updated environments are still compatible since $\mathcal{L}(t_{1i}, t_{2i})$)
   $\mathcal{PE}(exp_1[f/x] \ \oplus\ exp_2[f/x])(\rho_1[x_i/(,t_{1i})] \ \oplus\ \rho_2[x_i/(,t_{2i})]) \Rightarrow$ (Definition of partial evaluation)
   $\mathcal{PE}(F(x)(t_{11}, ..., t_{1n}) \ \oplus\ F(x)(t_{21}, ..., t_{2n}))$

2. $t_1 = f(t_{11}, ..., t_{1n})$ and $t_2 = \underline{f}(t_{21}, ..., t_{2n})$.
   Base case: $\mathcal{L}((\lambda\ x_1, ..., x_n.\ \perp)(t_{11}, ..., t_{1n}), (\lambda\ x_1, ..., x_n.\ \perp)(t_{21}, ..., t_{2n}))$

Proof:

$<\perp, (\lambda\, x_1, ..., x_n.\ \perp)(t_{11}, ..., t_{1n}), \rho_1 > \sqcup <\perp, (\lambda\, x_1, ..., x_n.\ \perp)(t_{21}, ..., t_{2n}), \rho_2 >$

$=$

$\perp$

$\mathcal{PE}((\lambda\ x_1, ..., x_n.\ \perp)(t_{11}, ..., t_{1n})\ \oplus\ (\lambda\ x_1, ..., x_n.\ \perp)(t_{21}, ..., t_{2n}))$

Induction Step: $\mathcal{L}(x(t_{11}, ..., t_{1n}), \underline{x}(t_{21}, ..., t_{2n})) \Rightarrow$ $\mathcal{L}(F_1(x)(t_{11}, ..., t_{1n}), \underline{F_2}(x)(t_{21}, ..., t_{2n}))$ (where F_1 and F_2 are two differently underlined versions of F)

Proof:

$< r_1, F_1(x)(t_{11}, ..., t_{1n})), \rho_1 >\ \sqcup\ < r_2, \underline{F_2}(x)(t_{21}, ..., t_{2n}), \rho_2 > \Rightarrow$ (Algorithm LUB)

$< r_1, exp_1[f/x], \rho_1[x_i/(,t_{1i})] >\ \sqcup\ < exp_2[f/x], exp_2[f/x], \rho_2[x_i/(e_{2i}, t_{2i})] >$ $\Rightarrow$ (Induction hypotheses, note that the updated environments are still compatible since $\mathcal{L}(t_{1i}, t_{2i})$)

$\mathcal{PE}(exp_1[f/x]\ \oplus\ exp_2[f/x])(\rho_1[x_i/(,t_{1i})]\ \oplus\ \rho_2[x_i/(e_{2i}, t_{2i})]) \Rightarrow$ (Definition of partial evaluation)

$\mathcal{PE}(F(x)(t_{11}, ..., t_{1n})\ \oplus\ \underline{F(x)}(t_{21}, ..., t_{2n}))$

3. $t_1\ =\ f(t_{11}, ..., t1n)$ and $t_2\ =\ \underline{f}(t_{21}, ..., t_{2n})$.

   Proof is analogous to the last case.

4. $t_1\ =\ \underline{f}(t_{11}, ..., t1n)$ and $t_2\ =\ \underline{f}(t_{21}, ..., t_{2n})$.

   Base case: $\mathcal{L}((\lambda\ x_1, ..., x_n.\ \perp)(t_{11}, ..., t_{1n}), (\lambda\ x_1, ..., x_n.\ \perp)(t_{21}, ..., t_{2n}))$

   Proof:

   $<\perp, (\lambda\, x_1, ..., x_n.\ \perp)(t_{11}, ..., t_{1n}), \rho_1 > \sqcup <\perp, (\lambda\, x_1, ..., x_n.\ \perp)(t_{21}, ..., t_{2n}), \rho_2 >$

   $=$

   $\perp$

   $\mathcal{PE}((\lambda\ x_1, ..., x_n.\ \perp)(t_{11}, ..., t_{1n})\ \oplus\ (\lambda\ x_1, ..., x_n.\ \perp)(t_{21}, ..., t_{2n}))$

   Induction Step: $\mathcal{L}(\underline{x}(t_{11}, ..., t_{1n}), \underline{x}(t_{21}, ..., t_{2n})) \Rightarrow$ $\mathcal{L}(\underline{F_1}(x)(t_{11}, ..., t_{1n}), \underline{F_2}(x)(t_{21}, ..., t_{2n}))$ (where F_1 and F_2 are two differently underlined versions of F)

Proof:

$< r_1, \underline{F_1}(x)(t_{11}, ..., t_{1n})), \rho_1 > \ \sqcup \ < r_2, \underline{F_2}(x)(t_{21}, ..., t_{2n}), \rho_2 > \Rightarrow$ (Algorithm LUB)

$F(x)(< e_{11}, t_{11}, \rho_1 > \ \sqcup \ < e21, t_{21}, \rho_2 >, ..., < e_{1n}, t_{1n}, \rho_1 > \ \sqcup \ < e_{2n}, t_{2n}, \rho_2 >) \Rightarrow$ (Induction hypotheses)

$F(x)(\mathcal{PE}(t_{11} \oplus t_{21})(\rho_1 \oplus \rho_2), ..., \mathcal{PE}(t_{1n} \oplus t_{2n})(\rho_1 \oplus \rho_2)) \Rightarrow$ (Definition of partial evaluation)

$\mathcal{PE}(\underline{F(x)}(t_{11}, ..., t_{1n}) \oplus \underline{F(x)}(t_{21}, ..., t_{2n}))$

4.3.4 A Two-Level Higher Order Language

Although the projection framework is defined only for first order programs, the least upper bound algorithm itself can be stated for higher order expressions (as has already been done). In this section we prove the general higher order least upper bound algorithm correct. We now present the proof of correctness for Algorithm **LUB** for higher order programs. To do this we first define a higher order two level language (essentially a call-by-value lambda calculus with constants).

$c \in Con$ *constants*

$x \in Bv$ *bound variables*

$e \in Exp$ *expressions, where*

$$e = c \mid x \mid if\ e\ e\ e \mid \lambda x.\ e \mid e\ @\ e$$

$$\underline{c} \mid \underline{x} \mid \underline{if}\ e\ e\ e \mid \underline{\lambda} x.\ e \mid e\ \underline{@}\ e$$

The **LUB** algorithm for this case is presented in figures 4.2, 4.3. The algorithm can be read in two level notation by simply replacing the **Rebuilds** by underlinings leaving the **Reduced** expressions untouched.

4.3.5 Proof of Correctness

Again we define a relation $\mathcal{L}$ which enables us to state the correctness criterion in a manner more amenable to proof. The function ϕ erases the underlinings of a two-level term.

Definition 4.3.2 *We define a relation $\mathcal{L}$ defined as follows: $\mathcal{L}(t_1, t_2, \rho_1, \rho_2)$ if and only if*

$$< r_1, t_1, \rho_1 > \sqcup < r_2, t_2, \rho_2 > = \mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2)$$

provided

- $r_1 = \mathcal{PE}[\![t_1]\!]\rho_1$
- $r_2 = \mathcal{PE}[\![t_2]\!]\rho_2$
- $\phi(t_1) = \phi(t_2)$
- *$t_1 \oplus t_2$ is defined as the two level term such that $\phi(t_1 \oplus t_2) = \phi(t_1) = \phi(t_2)$ and having a subcomponent underlined if it is underlined both in t_1 and t_2.*
- *If both ρ_1 and ρ_2 have bindings for x then $\mathcal{L}(snd(\rho_1(x)), snd(\rho_2(x)), \rho_1 \backslash x, \rho_2 \backslash x)$.*
- $(\rho_1 \oplus \rho_2)(x) = if\ \rho_1(x) = error\ then\ \rho_2(x)\ elseif\ \rho_2(x) = error\ then\ \rho_1(x)\ else$ $(< fst(\rho_1(x)), snd(\rho_1(x)), \rho_1 \backslash x > \sqcup < fst(\rho_2(x)), snd(\rho_2(x)), \rho_2 \backslash x >)$

Theorem 4.3.3 *If $\phi(t_1) = \phi(t_2)$ then $\mathcal{L}(t_1, t_2)$.*
Proof: Follows from following Lemmas.

The proof proceeds by structural induction. The induction hypothesis in each case states that the theorem under consideration holds for the corresponding subcomponents of the current expressions. First we prove the base case for constants:

Lemma 4.3.6 *If $\phi(t_1) = \phi(t_2) = c$ then $\mathcal{L}(t_1, t_2)$.*
Proof: There are four possible cases:

1. $t_1 = c$ and $t_2 = c$.
   $< c, c, \rho_1 > \sqcup < c, c, \rho_2 > \Rightarrow$
   $c \Rightarrow$
   $\mathcal{PE}[\![c \oplus c]\!](\rho_1 \oplus \rho_2)$

2. $t_1 = c$ and $t_2 = \underline{c}$.

   $< c, c, \rho_1 > \sqcup < c, \underline{c}, \rho_2 > \Rightarrow$

   $c \Rightarrow$

   $\mathcal{PE}\llbracket c \oplus \underline{c} \rrbracket (\rho_1 \oplus \rho_2)$

3. $t_1 = \underline{c}$ and $t_2 = c$.

   Proof is analogous to the last case.

4. $t_1 = \underline{c}$ and $t_2 = \underline{c}$.

   $< c, \underline{c}, \rho_1 > \sqcup < c, \underline{c}, \rho_2 > \Rightarrow$

   $c \Rightarrow$

   $\mathcal{PE}\llbracket \underline{c} \oplus \underline{c} \rrbracket (\rho_1 \oplus \rho_2)$

Then we prove the base case for variables:

Lemma 4.3.7 *If $\phi(t_1) = \phi(t_2) = x$ then $\mathcal{L}(t_1, t_2)$.*

Proof: There are four possible cases:

1. $t_1 = x$ and $t_2 = x$.

   $\mathcal{PE}\llbracket x \oplus x \rrbracket \; (\rho_1[x/(t_1, c_1)] \oplus \rho_2[x/(t_2, c_2)]) \Rightarrow$ (Definition of $\oplus$)

   $(\rho_1[x/(t_1, c_1)] \oplus \rho_2[x/(t_2, c_2)])x \Rightarrow$ (Definition of $\oplus$)

   $< t_1, c_1, \rho_1 \backslash x > \sqcup < t_2, c_2, \rho_2 \backslash x > \Rightarrow$ (Algorithm LUB)

   $< t_1, x, \rho_1[x/(t_1, c_1)] > \sqcup < t_2, x, \rho_2[x/(t_2, c_2)] >$

2. $t_1 = x$ and $t_2 = \underline{x}$.

   $\mathcal{PE}\llbracket x \oplus \underline{x} \rrbracket \; (\rho_1[x/(t_1, c_1)] \oplus \rho_2[x/(t_2, c_2)]) \Rightarrow$ (Definition of $\oplus$)

   $(\rho_1[x/(t_1, c_1)] \oplus \rho_2[x/(t_2, c_2)])x \Rightarrow$ (Definition of $\oplus$)

   $< t_1, c_1, \rho_1 \backslash x > \sqcup < t_2, c_2, \rho_2 \backslash x > \Rightarrow$ (Algorithm LUB)

   $< t_1, x, \rho_1[x/(t_1, c_1)] > \sqcup < x, \underline{x}, \rho_2[x/(t_2, c_2)] >$

3. $t_1 = \underline{x}$ and $t_2 = x$.

   Proof is analogous to the last case.

4. $t_1 = \underline{x}$ and $t_2 = \underline{x}$.

   $\mathcal{PE}\llbracket \underline{x} \oplus \underline{x} \rrbracket \; (\rho_1[x/(t_1, c_1)] \oplus \rho_2[x/(t_2, c_2)]) \Rightarrow$ (Definition of $\oplus$)

$x \Rightarrow$ (Algorithm LUB)
$< x, \underline{x}, \rho_1[x/(t_1, c_1)] > \ \sqcup \ < x, \underline{x}, \rho_2[x/(t_2, c_2)] >$

Define the function ψ which erases underlinings at only one level. We then handle the case of abstractions:

Lemma 4.3.8 *If $\psi(t_1) = \lambda x.t_{11}$ and $\psi(t_2) = \lambda x.t_{21}$ and $\mathcal{L}(t_{11}, t_{21})$ then $\mathcal{L}(t_1, t_2)$.*

Proof: There are four possible cases:

1. $t_1 = \lambda x.t_{11}$ and $t_2 = \lambda x.t_{21}$
   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)
   $\lambda x.\mathcal{PE}[\![(t_{11} \oplus t_{21})]\!](\rho_1 \oplus \rho_2)\ \mathcal{PE}[\![(t_{12} \oplus t_{22})]\!](\rho_1 \oplus \rho_2) \Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$
   $\lambda x. < r_1, t_{11}, \rho_1 > \sqcup < r_2, t_{21}, \rho_2 > \Rightarrow$ (Algorithm LUB)
   $< r_1, \lambda x.t_{11}, \rho_1 > \sqcup < r_2, \lambda x.t_{21}, \rho_2 >$

2. $t_1 = \lambda x.t_{11}$ and $t_2 = \underline{\lambda} x.t_{21}$
   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)
   $\lambda x.\mathcal{PE}[\![(t_{11} \oplus t_{21})]\!](\rho_1 \oplus \rho_2) \Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$
   $\lambda x. < r_1, t_{11}, \rho_1 > \sqcup < t_{21}, t_{21}, \rho_2[?/ < t_{22}, c_{22} >] > \Rightarrow$ (Algorithm LUB)
   $< r_1, \lambda x.t_{11}, \rho_1 > \sqcup < r_2, \underline{\lambda} x.t_{21}, \rho_2 >$

3. $t_1 = \underline{\lambda} x.t_{11}$ and $t_2 = \lambda x.t_{21}$ (Proof analogous to the previous case)

4. $t_1 = \underline{\lambda} x.t_{11}$ and $t_2 = \underline{\lambda} x.t_{21}$
   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)
   $\lambda x.\mathcal{PE}[\![(t_{11} \oplus t_{21})]\!](\rho_1 \oplus \rho_2) \Rightarrow (\mathcal{L}(t_{11}, t_{21}))$
   $\lambda x. < r_{11}, t_{11}, \rho_1 > \sqcup < r_{21}, t_{21}, \rho_2 > \Rightarrow$ (Algorithm LUB)
   $< r_1, \underline{\lambda} x.t_{11}, \rho_1 > \sqcup < r_2, \underline{\lambda} x.t_{21}, \rho_2 >$

Next we handle conditional statements:

Lemma 4.3.9 *If $\psi(t_1) = if\ t_{11}\ t_{12}\ t_{13}$ and $\psi(t_2) = if\ t_{21}\ t_{22}\ t_{23}$ and $\mathcal{L}(t_{1i}, t_{2i})$ then $\mathcal{L}(t_1, t_2)$, $i \in 1, 2$.*

Proof: There are four possible cases:

1. $t_1 = if\ t_{11}\ t_{12}\ t_{13}$ and $t_2 = if\ t_{21}\ t_{22}\ t_{23}$.

   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)

   $choose(\mathcal{PE}[\![t_{12} \oplus t_{22}]\!](\rho_1 \oplus \rho_2), \mathcal{PE}[\![t_{13} \oplus t_{23}]\!](\rho_1 \oplus \rho_2)) \Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$

   $choose(< r_{12}, t_{12}, \rho_1 > \sqcup < r_{22}, t_{22}, \rho_2 >, < r_{13}, t_{13}, \rho_1 > \sqcup < r_{23}, t_{23}, \rho_2 >$
   $) \Rightarrow$

   $< r_1, choose(t_{12}, t_{22}), \rho_1 > \sqcup < r_2, choose(t_{13}, t_{23}), \rho_2 > \Rightarrow$ (Algorithm LUB)

   $< r_1, if\ t_{11}\ t_{12}\ t_{13}, \rho_1 > \sqcup < r_2, if\ t_{21}\ t_{22}\ t_{23}, \rho_2 >$

2. $t_1 = if\ t_{11}\ t_{12}\ t_{13}$ and $t_2 = \underline{if}\ t_{21}\ t_{22}\ t_{23}$.

   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)

   $choose(\mathcal{PE}[\![t_{12} \oplus t_{22}]\!](\rho_1 \oplus \rho_2), \mathcal{PE}[\![t_{13} \oplus t_{23}]\!](\rho_1 \oplus \rho_2)) \Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$

   $choose(< r_{12}, t_{12}, \rho_1 > \sqcup < r_{22}, t_{22}, \rho_2 >, < r_{13}, t_{13}, \rho_1 > \sqcup < r_{23}, t_{23}, \rho_2 >$
   $) \Rightarrow$

   $< r_1, choose(t_{12}, t_{13}), \rho_1 > \sqcup < choose(r_{22}, r_{23}), choose(t_{22}, t_{23}), \rho_2 > \Rightarrow$ (Algorithm LUB)

   $< r_1, if\ t_{11}\ t_{12}\ t_{13}, \rho_1 > \sqcup < if\ r_{21}\ r_{22}\ r_{23}, \underline{if}\ t_{21}\ t_{22}\ t_{23}, \rho_2 >$

3. $t_1 = \underline{if}\ t_{11}\ t_{12}\ t_{13}$ and $t_2 = if\ t_{21}\ t_{22}\ t_{23}$.

   Proof is analogous to the last case.

4. $t_1 = \underline{if}\ t_{11}\ t_{12}\ t_{13}$ and $t_2 = \underline{if}\ t_{21}\ t_{22}\ t_{23}$.

   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)

   $if\ \mathcal{PE}[\![t_{11} \oplus t_{21}]\!](\rho_1 \oplus \rho_2)\ \mathcal{PE}[\![t_{12} \oplus t_{22}]\!](\rho_1 \oplus \rho_2)\ \mathcal{PE}[\![t_{13} \oplus t_{23}]\!](\rho_1 \oplus \rho_2)$
   $\Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$

   $if\ (< r_{11}, t_{11}, \rho_1 > \sqcup < r_{21}, t_{21}, \rho_2 >)\ (< r_{12}, t_{12}, \rho_1 > \sqcup < r_{22}, t_{22}, \rho_2 >)$
   $(< r_{13}, t_{13}, \rho_1 > \sqcup < r_{23}, t_{23}, \rho_2 >) \Rightarrow$ (Algorithm LUB)

   $< r_1, \underline{if}\ t_{11}\ t_{12}\ t_{13}, \rho_1 > \sqcup < r_2, \underline{if}\ t_{21}\ t_{22}\ t_{23}, \rho_2 >$

Finally we handle the case of application:

Lemma 4.3.10 *If $\psi(t_1) = t_{11} @ t_{12}$ and $\psi(t_2) = t_{21} @ t_{22}$ and $\mathcal{L}(t_{1i}, t_{2i})$ then $\mathcal{L}(t_1, t_2)$.*

Proof: There are four possible cases:

1. $t_1 = t_{11} @ t_{12}$ and $t_2 = t_{21} @ t_{22}$

   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)

   $\mathcal{PE}[\![(t_{11} \oplus t_{21})]\!](\rho_1 \oplus \rho_2)\ \mathcal{PE}[\![(t_{12} \oplus t_{22})]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Substitution)

   $\mathcal{PE}[\![(t_{11} \oplus t_{21})]\!](\rho_{11} \oplus \rho_{21}) \Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$

   $< r_1, t_{11}, \rho_{11} > \sqcup < r_2, t_{21}, \rho_{21} > \Rightarrow$ (Algorithm LUB)

   $< r_1, t_{11} @ t_{12}, \rho_1 > \sqcup < r_2, t_{21} @ t_{22}, \rho_2 >$

2. $t_1 = t_{11} @ t_{12}$ and $t_2 = t_{21} \underline{@} t_{22}$

   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)

   $\mathcal{PE}[\![(t_{11} \oplus t_{21})]\!](\rho_1 \oplus \rho_2[?/ < t_{22}, c_{22} >]) \Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$

   $< r_1, t_{11}, \rho_1 > \sqcup < t_{21}, t_{21}, \rho_2[?/ < t_{22}, c_{22} >] > \Rightarrow$ (Algorithm LUB)

   $< r_1, t_{11} @ t_{12}, \rho_1 > \sqcup < c_{21} @ c_{22}, t_{21} \underline{@} t_{22}, \rho_2 >$

3. $t_1 = t_{11} \underline{@} t_{12}$ and $t_2 = t_{21} @ t_{22}$ (Proof analogous to the previous case)

4. $t_1 = t_{11} \underline{@} t_{12}$ and $t_2 = t_{21} \underline{@} t_{22}$

   $\mathcal{PE}[\![t_1 \oplus t_2]\!](\rho_1 \oplus \rho_2) \Rightarrow$ (Definition of partial evaluation)

   $\mathcal{PE}[\![(t_{11} \oplus t_{21})]\!](\rho_1 \oplus \rho_2) @ \mathcal{PE}[\![(t_{12} \oplus t_{22})]\!](\rho_1 \oplus \rho_2) \Rightarrow (\mathcal{L}(t_{1i}, t_{2i}))$

   $< r_{11}, t_{11}, \rho_1 > \sqcup < r_{21}, t_{21}, \rho_2 > @ < r_{12}, t_{12}, \rho_1 > \sqcup < r_{22}, t_{22}, \rho_2 > \Rightarrow$ (Algorithm LUB)

   $< r_1, t_{11} \underline{@} t_{12}, \rho_1 > \sqcup < r_2, t_{21} \underline{@} t_{22}, \rho_2 >$

Theorem 4.3.4 *Algorithm* **LUB** *does not re-perform any reduction already performed in the computation of either r_p or r_q.*

Proof: During the application of the rewrite rules, no reductions are done (all of them are done in the post-processing phase). At the end of this phase, the term has the action tree $at_p \sqcup at_q$. This means that all reductions in r_p and r_q are already incorporated. □

Chapter 5

Applications

In this chapter we present several examples of incremental programs. In each case we present the non-incremental program and the associated input partition. We also discuss the special transformations required to bring the non-incremental program into a form amenable to partial evaluation. Anyone who has used a present day partial evaluator is familiar with the fact that the source program frequently needs to undergo certain meaning preserving transformations in order to specialize well; understanding and automating these transformations is the subject of ongoing work (see [CD91] for some progress). Once these transformations are done, the incremental program is derived automatically using the techniques described in previous chapters. The implementation is carried out using the Schism partial evaluator [Con90a, Con90b, Con88]. Performance figures are also presented for each of these programs. Before we proceed, we give an introduction to Schism – in particular we discuss the user annotation mechanism called *filters*, and an intoduction to our source language *Haskell.*

5.1 Schism

Schism [Con90a, Con90b, Con88] is a partial evaluator for a side-effect free subset of Scheme. It is able to specialize programs containing higher order functions and data structures. Schism is self-applicable – compilers can be obtained by specializing Schism with respect to a program. The system has three phases: preprocessing, specialization and postprocessing. Preprocessing has been discussed in the last chapter. This phase produces an annotated source program where each subexpression has been marked with an *action.*

Schism incorporates a feature called *filters* which specifies how to treat calls to a procedure; i.e., should the call be unfolded or specialized? If a call is to be made residual, the filter also specifies the arguments with respect to which the specialization is to take place. Thus a filter has two components: the first decides how to treat function calls and the second decides which arguments are to be used in the specialization process. As an example, consider the function append:

```
(define (append l1 l2)
  (filter (if (stat? l1) UNFOLD SPECIALIZE) (list l1 l2))
  (if (null? l1)
      l2
      (cons (car l1)
            (append (cdr l1) l2))))
```

The predicate stat? returns true if its argument is static, and false otherwise. The predicate dyn? returns true if its argument is dynamic and false otherwise. The filter in the example above states that calls to append must be unfolded when the first argument is static. Otherwise it states that the call must be specialized with respect to both the arguments. If it is desired not to specialize a call with respect to an argument, then the filter simply states DYNAMIC corresponding to the argument.

The implementation we are about to describe works on Haskell programs. Haskell programs are first translated into stand alone Schism programs, and are then specialized using Schism. The translation is semantics preserving in the sense that laziness is preserved by adding *delays.* The delay and force mechanism is implemented using the Schism primitive mechanism. This avoids the use of explicit lambdas to represent the delays, which is desirable since lambdas being higher order objects cannot be tested for static or dynamic nature in Schism filters. Filters can be attached to specific Haskell functions. Thus, while the implementation itself is based on Schism, we see no reason why the implementation should not be based on a partial evaluator which performs source to source transformations on Haskell.[1]

The binding time analysis of Schism is *monovariant*; i.e., each function can only have one binding time signature.[2] This loss of information is unacceptable in our case where we wish to do binding time analysis with one of the components known and the rest being unknown. This problem can be solved by replicating the function for each binding time pattern of its arguments.

The filters in Schism can only test for completely static or dynamic information. Therefore, partially static arguments may sometimes have to be split into static and dynamic components for purposes of writing the appropriate filters.

Since the time of these experiments, Schism is being extended to provide the following features:

- Predicates which make it unnecessary to separate binding times and split partially static objects.
- Polyvariant binding time analysis, which will be more convenient for its users since they won't need to duplicate source procedures by hand as we needed to.

In the rest of the chapter we use the terms "known" and `STATIC` interchangeably, and similarly the terms "unknown" and `DYNAMIC`.

[1] The problem of avoiding non-termination during specialization for lazy languages is still an open one.

[2] A binding time signature is a description of the binding times for the arguments and the result of a function.

5.2 An Overview of Haskell

Haskell is a new functional programming language, named after the logician Haskell B. Curry [HWe90]. It is a purely functional language, and has *non-strict semantics* (i.e., lazy evaluation) ,and a *rich type system* including user-defined concrete and abstract datatypes and strong static type inference. Non-strict semantics was popularized by languages such as SASL [Tur81, Tur82], and the type discipline was popularized by languages such as ML [Mil84, Wik88].

Haskell is *purely functional*, which means that it has no constructs inducing side effects to an implicit store (such as found in "almost-functional" languages such as ML). The language contains many of the recent innovations in programming language research, including higher-order functions, non-strict functions and data structures, static polymorphic type inference, user-definable concrete and abstract datatypes, pattern-matching, list comprehensions, a module system, and a rich set of primitive datatypes, including arbitrary and fixed precision integers, and complex, rational, and floating-point numbers. In addition it has several novel features that give it additional expressiveness, including an elegant form of overloading using a notion of *type classes*, a flexible I/O system that unifies the two most popular functional I/O models, and an array datatype that allows purely functional, monolithic arrays to be constructed using "array comprehensions."

Syntactically, Haskell has an "equational feel." A function is defined by a set of equations, each stating a different set of constraints on the arguments for the equation to be valid. These constraints primarily concern the structure of the arguments, and thus the process is called *pattern matching*. For example, lists are written `[a,b,c]` with `[]` being the empty list, and a list whose first element is `x` and whose rest is `xs` is denoted `x:xs`. Thus to define a function that tests for membership in a list, we can write:

```
member x []     = False
member x (y:ys) = if x==y then True
                          else member x ys
```

Given this definition, the expression `member 2 [1,2,3]` returns `True`, whereas the expression `member 0 [1,2,3]` returns `False`. Any data structure may be pattern-matched against, including user-defined ones. Also note that function application is "curried" and associates to the left; in a conventional language without currying one might write `member(2,[1,2,3])`.

A function `f x = x+1` may also be defined "anonymously" with a *lambda abstraction* having form `\x -> x+1`; thus `(\x -> x+1) 2` returns `3`. In addition, any infix operator may be turned into a value by surrounding it in parentheses; thus, for example, the following equivalence holds:

```
(+) x y   ==   x + y
```

List comprehensions are a concise way to define lists and are best explained by example:

```
[ (x,y) | x<-xs, y<-ys, x>y ]
```

This expression designates the list of all pairs whose first element is from `xs` and second is from `ys`, but such that the first element is always greater than the second. Pairs are examples of *tuples*, which in Haskell are constructed in arbitrary but finite length ≥ 2 by writing "`(a,b, ..., c)`" (the parentheses are mandatory). Tuples may be pattern-matched like lists.

"Infinite lists" may also be defined, and thanks to lazy evaluation, only that portion of the list that is needed by some other part of the program is actually computed. For example, the infinite list of ones can be defined by

```
ones = 1 : ones
```

Thus `member 1 ones` returns `True`, whereas `member 2 ones` does not terminate.

The notation `[a..b]` denotes the list of integers from `a` to `b`, inclusive, and `[a..]` is the infinite ascending list of integers beginning with `a`. Two lists may be appended together using the infix operator `++`, as in `l1++l2`. There are also many other standard utility functions defined on lists. Aside from the ones already discussed, the ones we need in this dissertation are the following:

```
head (x:xs) = x                   -- head and tail of list
tail (x:xs) = xs

fst  (x,y)  = x                   -- first and second of pair
snd  (x,y)  = y

map f  []    = []                 -- maps function f down list
map f (x:xs) = f x : map f xs

[]     ++ ys = ys                 -- infix append
(x:xs) ++ ys = x : (xs++ys)

foldl f a  []    = a              -- folds list from left
foldl f a (x:xs) = foldl f (f a x) xs

foldr f a  []    = a              -- folds list from right
foldr f a (x:xs) = f x (foldr f a xs)
```

Note that comments in Haskell are preceeded with "`--`" and continue to the end of the line. Function application always has higher precedence than any infix operator; thus "`f x :  map f xs`" is parsed as "`(f x) :  (map f xs)`." Note that for `foldl` and `foldr` the following relationships hold:

```
foldl f a [x1, x2, ..., xn]  ==>
          (f ... (f (f a x1) x2) ... xn)
foldr f a [x1, x2, ..., xn]  ==>
          (f x1 (f x2 ... (f xn a) ... ))
```

Haskell's fundamental block structuring mechanism (aside from those introduced by modules, which are beyond the scope of our discussion) is the `where` clause. For example,

```
x where x = y
        y = 1
```

The declarations in a `where` clause are lexically scoped and mutually recursive; thus their order does not matter, and the above example evaluates to 1. Also note the use of a "layout" strategy for parsing declarations. There is no need for a semicolon, or some other syntactic device, to terminate a declaration. The simple rule is that the first characters in the declarations must line up vertically with each other. For example,

```
x+y where x = a+b
              where a = 1
                    b = 2
          y = 3
```

parses in the "natural way," and evaluates to 6.

For more information on Haskell, see [HWe90].

5.3 Compiled Incremental Programs

Shown below is an *incremental interpreter*, i.e. it takes a specification of an incremental program (a non-incremental program and a partition of the input), an initial argument and a series of small changes to the input and "incrementally" produces a series of answers. The function setup computes the residual functions corresponding to prog on each element of the partition. reestablish takes a set of residual functions, a small change to the argument of the program and recomputes only those residuals affected by the change.

```
(define (Inc prog part init_arg deltas)
  (I (setup prog part init_arg) deltas))

(define (I residuals deltas)
  (cond
    ((null? deltas)
     (cons (eval (ast->t (combine residuals))
                 (repl-env)) '()))
    (else
      (let* ((delta (car deltas))
             (residuals' (reestablish residuals delta)))
             (cons
               (eval (ast->t (combine residuals')) (repl-env))
               (I residuals' (cdr deltas))))))
```

That the use of an interpreter entails a performance overhead is well known. Partial evaluation has been used to capture the essence of compilation by specializing an interpreter with respect to a program yielding a compiled program. We can use this idea profitably to specialize the incremental interpreter (`Inc`) wrt a incremental program specification (`prog` and `part`). This yields a "compiled incremental program" which can yield substantial performance benefits over the interpreted version. We have actually carried out such a specialization for many problems. This can be described precisely as follows: If f is the function to be incrementalized, P is the input partition, $\mathcal{I}$ is the incremental interpreter, f_{inc} is the "compiled" incremental program, and $[\![p]\!]$ denotes the function corresponding to the program p, then

$$[\![\mathcal{PE}]\!]\ \mathcal{I}\ \langle f, P\rangle = f_{inc}$$

Since the incremental interpreter itself makes liberal use of partial evaluation (in the **Setup** and **Reestablish** phases), the abovementioned specialization entails self application of the partial evaluator.

How exactly is increased performance achieved by this specialization? First, binding time analysis of `prog` on each element of `part` can be carried out at specialization time. Also, the specializer can be specialized with each result of the binding time analyses to yield specialized specializers (the second Futamura projection [Fut71]). The **Combine** phase does not benefit from this specialization because the conditional environment arguments to the `lub` function (`cenv1` and `cenv2`) are both `DYNAMIC` (see Appendix A). The `lub` function has to be left residual because the information about which way the conditionals in the programs have been resolved is not statically available. It is available only at partial evaluation time. The whole process is summarized in Figure 5.1.

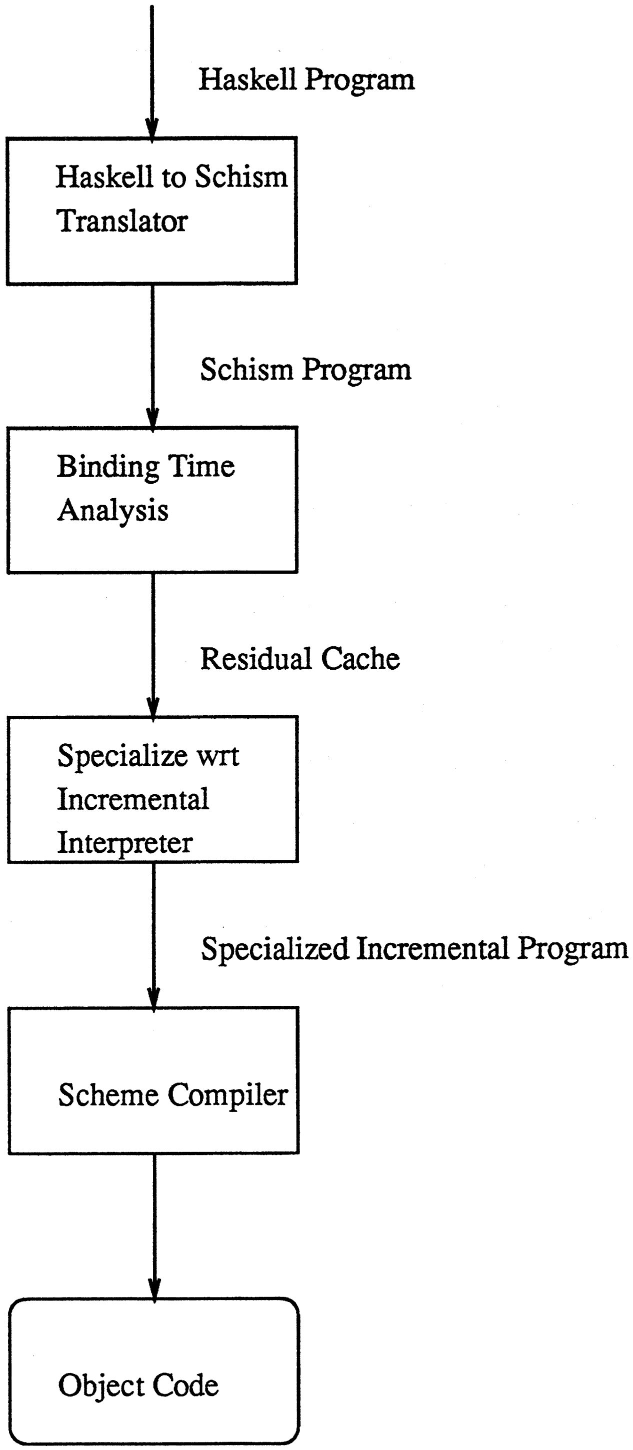
Haskell Program
Haskell to Schism Translator
Schism Program
Binding Time Analysis
Residual Cache
Specialize wrt Incremental Interpreter
Specialized Incremental Program
Scheme Compiler
Object Code

Figure 5.1: Incremental program generation.

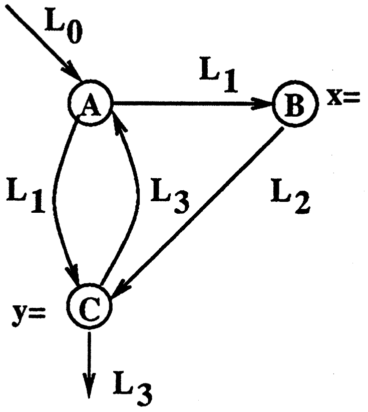

Figure 5.2: Example flow graph

5.4 Data Flow Analysis

As an example of compiler data flow analysis, consider the problem of determining the set of "reaching definitions" at every program point. A definition of a variable is said to "reach" a program point if there exists a path from the definition to the program point which does not pass through a redefinition of the same variable. For example, consider the flow graph in Figure 5.2 (taken from [MR90]). Each of the circles denotes a basic block, where a label "$x =$" means that x is assigned a value in that block. Arcs are labelled with sets L_i of definitions which reach that arc. For example, the set $L_0 = \{(x, e1), (y, e2)\}$ means that the definition of x at program point $e1$ and y at $e2$ reach the arc labelled L_0. For Figure 5.1, we can then write the following equations ("?" refers to a wildcard, meaning that it matches any name):

$$\begin{aligned}
L_0 &= \{(x, e1), (y, e2)\} \\
L_1 &= L_0 \cup L_3 \\
L_2 &= L_1 - \{(x, ?)\} \cup \{(x, B)\} \\
L_3 &= (L_1 \cup L_2) - \{(y, ?)\} \cup \{(y, C)\}
\end{aligned}$$

The solution to these set equations is defined by a least fixpoint construction in the obvious manner, yielding:

$$\begin{aligned}
L_0 &= \{(x, e1), (y, e2)\} \\
L_1 &= \{(x, e1), (y, e2), (x, B), (y, C)\} \\
L_2 &= \{(y, e2), (y, C), (x, B)\} \\
L_3 &= \{(x, e1), (x, B), (y, C)\}
\end{aligned}$$

Recall that an incremental algorithm specification consists of a non-incremental program plus a partition of the input domain. Therefore we first need to describe a non-incremental algorithm. We assume the input to the algorithm to be a list of strongly connected components of the set of data flow equations in topological order (which can be produced by a standard dependency analysis). The solution is then defined by the Haskell program in Figure 5.3.

`solve` is a function which takes an initial (empty) environment (mapping arc labels to sets) and a set of mutually recursive equations. It then computes the least fixpoint of the equations using the initial environment as the first approximation. Function `nextapprox` recomputes the identifiers defined by the equations `eqns` using the old environment `env` to produce a new environment `env'`.

To complete the incremental program specification we specify the following input partition. $P = \{p_i\}$ such that:

$$\begin{aligned}
p_0\ [] &= [], p_0\ (x : xs) &= \bot \\
p_1\ [] &= \bot, p_1\ (x : xs) &= (x : \bot) \\
p_i\ [] &= \bot, p_i\ (x : xs) &= (\bot : (p_{i-1}\ xs))
\end{aligned}$$

This means that each strongly connected component is in a separate element of the partition. This partition enables the fixpoint computation to go to completion on each element of the partition (as will be seen in the following sections).

In the **Setup** and **Reestablish** phases of the incremental interpreter, the computation of r_{p_i} is the main activity. In this section we see how to do this. Before we use the incremental interpreter defined previously, we need to transform the definition into a form amenable to partial evaluation. The projections which form the partition describe partially static input. To make sure that the binding

```
-- Data flow analysis

-- Expression data type representing the right hand side of a data
-- flow equation.
data Expr = Arc String
          | Union Expr Expr
          | Diff  Expr Expr
          | Const [ (String, String) ]
type Graph = [ Comp ]
type Comp = [ Eqn ]
type Eqn = (Arc, Expr)
type Env = [ Binding ]
type Binding = (Arc, [(String, String)])

dfa :: Graph -> Env -> Env
dfa graph env0 = foldl solve env0 graph

-- fix point finder
solve :: Env -> Comp -> Env
solve env scc =
      if (env == env') then env
                       else solve env' scc
       where env' = nextapprox scc env

nextapprox :: Comp -> Env -> Env
nextapprox scc env = envupdate env bindings
                     where bindings = map (compute env) scc

compute :: Env -> Eqn -> Binding
compute env (arc, exp) = (arc, eval exp env)

-- evaluate an equation right hand side with an environment
eval :: Expr -> Env -> Val
eval (Arc s) env = envlookup s env
eval (Union e e2) env = listunion (eval e env) (eval e2 env)
eval (Diff e e2) env = listdiff (eval e env) (eval e2 env)
eval (Const l) env = l
```

Figure 5.3: Data Flow Analysis.

times for arguments to the main function are not partially static, we carry out a process similar to *arity raising* [Rom88, Ses86]. This implies fixing the number of connected components *a priori.* If the graph grows to exceed this limit, we may either redo this arity raising with a higher number of arguments or we may group more connected components into a single argument. The latter solution ends up using a "coarser" partition of the input and may yield poorer incremental performance. In practice one may choose a sufficiently large arity to avoid this problem. For the example here we choose to convert the list of components into three arguments (The principles involved for a larger number of arguments remain the same):

```
dfa scc1 scc2 scc3 env0 =
    solve scc3
          (solve scc2
                 (solve scc1 env0))
```

During specialization, if the first argument to the function `solve` is `STATIC`, we know that the solve function can be completely unrolled. This is because the termination of the fixpoint computation does not depend on the environment argument. To convey this information to Schism we make use of the *filter* mechanism. Filters in Schism provide the user with control over the specialization process. First the environment is split into two components: a static and dynamic component. This is done since filters can only contain tests for either `STATIC` or `DYNAMIC`. The fixpoint iteration is always started with the static environment empty. The dynamic environment is the solution to the problem flowing in to the component. Thus the termination of the unfolding process is made to depend only on static information since the dynamic environment does not change during iteration within this component. Given this knowledge, the filter specifies that `solve` can be unfolded only if both `scc` and `env_s` are known:

```
solve env_s env_d scc =
      if (enveq env_s env_s')
      then envunion env_d env_s
      else solve env_s' env_d' scc
      where env_s' = nextapprox scc env_s
            env_d' = nextapprox' scc env_d

nextapprox scc env = envupdate env bindings
                     where bindings = map (compute env) scc
```

The filter associated with solve is (this filter is added manually to the translated Haskell program):

```
(filter (if (and (stat? scc) (stat? env_s)) UNFOLD SPECIALIZE)
        (list scc env_s env_d))
```

To avoid loss of information due to the monovariant nature of Schism, each call to solve is made to a differently named copy. This entails duplication of code, but can be done in a straightforward manner. The function applied to the ith component is named solvei.

The result of specializing dfa with scc2 and env0 as STATIC is shown below. The value used for scc2 is the graph shown in Figure 5.2 and the value used for env0 is the empty environment. [3] Note that solve2 has been completely unrolled. The Expr data type is completely translated into Schism and appears in the residual program. The static part of the result is completely computed. The dynamic environment (flowing in from the result of the first component) is processed before being combined with the static environment to yield the environment to be passed to the next component in the topological order (note that the residual function is stored in Scheme form because the Haskell program is first translated into a Schism program, the solveis are the renamed versions of solve):

```
(lambda (scc1 scc3)
  (solve3 scc3 (map31 (car scc3))
    (env-union2
      (next-approx scc2
        (next-approx scc2
```

[3] The empty environment binds each name to a special placeholder.

```
          (next-approx scc2
            (solve1 scc1 (map11 (car scc1))
                    '((l0) (l1) (l2) (l3))))))
                    '((l0 (x . e1) (y . e2))
                      (l1 (x . e1) (y . e2)
                          (x . b) (y . c))
                      (l2 (y . e2) (y . c)
                          (x . b))
                      (l3 (x . e1) (x . b)
                          (y . c))))))
```

The residual function corresponding to scc1 and env0 being STATIC is shown below. Note the following:

- solve1 is completely unfolded since all its arguments were static (the values used are the same as in the last case).
- The Expr data type in the *Haskell* program appears as a list in the Scheme code ((x . e1) etc.). Only the Const portion of the data type is left.

```
(lambda (scc2 scc3)
  (solve3 scc3 (map31 (car scc3))
          (solve2 scc2 (map21 (car scc2))
                  '((l0 (x . e1) (y . e2))
                    (l1 (x . e1) (y . e2)
                        (x . b) (y . c))
                    (l2 (y . e2) (y . c)
                        (x . b))
                    (l3 (x . e1) (x . b)
                        (y . c))))))
```

During the **Setup** phase of the incremental interpreter, residual functions (including the ones shown above) are computed corresponding to each element of the partition. The **Reestablish** phase simply entails recomputing those residual functions affected by changes in the input.

During the third phase (**Combine**), the result is obtained by computing the lub of all the residual functions, which effectively propagates information in topological order between the components. For example, the predecessor (in topological order) of the component above supplies the value of L_0.

The cost of carrying fixpoint iteration (for this problem) to completion is well known to be $\mathcal{O}(B \times V)$ where B is the size of the component and V is the number of variables of interest. For a fixed number of variables the cost is $\mathcal{O}(B)$. Phase three simply combines residual functions using the $\sqcup$ operation. No fixpoint iteration is performed. The partition used in this example is based on an incremental data flow analysis algorithm described in [MR90].

5.4.1 Comparison with known algorithms

An algorithm in the literature which is closely related to ours is the one described in [MR90]. This algorithm exhibits better performance than ours in the following respects: When a change is made to the input flow graph, the algorithm in [MR90] only reexamines those components which are actually affected by the change, and only these have their solutions recomputed. In our algorithm, while the fix point computation for unaffected components is not redone, the **Combine** phase is too heavy handed in that it propagates change information throughout the graph. This is because the information stored in the form of residual functions only incorporates information local to an element of the partition. This causes the final solution for *every* component to be recomputed even if only one changes. For example, in the first residual function we showed, observe the dependencies on both `scc1` and `scc3`. This implies that if any of the other elements of the partition change, this residual function causes recomputation. Also, the algorithm in [MR90] can handle non structural changes, whereas our algorithm cannot. This is because the entire partition may change due to a non-structural change, causing recomputation from scratch.

5.4.2 Performance

We now describe the performance results for the data flow analysis problem discussed above. First we make some general comments which also apply to the examples discussed later. The implementation runs under the T system (a dialect of Scheme) using the partial evaluator Schism. In each case we compare the performance of the "compiled incremental program" (as described in the last section) with the non-incremental (batch) version.

The choice of input data has been done as follows. It is possible to show very good performance by making the changing element of the partition very small and the rest of the input arbitrarily large. Correspondingly it is possible to make an incremental program perform poorly by making the changes very large, so that the overhead of recomputing the solution becomes comparable to the batch version. To present a more accurate picture, we have chosen input data where the elements of the partition are of equal size. Thus if there are n elements in the partition, one would expect the incremental version to show an n-fold speedup over the batch version for changes which only affect one element of the partition. In practice we observe that the overhead associated with the methodology gives us speedups less than n (because of the overheads associated with the **Combine** step).

Shown in Figure 5.4, 5.5 are the performance figures. The tables show the costs of recomputing the result in a non-incremental (batch) mode and using the incremental program. In Figure 5.4, since we use a partition of three equal sized components, we expect an ideal speedup of three. The overhead of the **Combine** step forces us to settle for a lower speedup. Each row of the tables represents a small change to the input in the previous row. The cost of the **Combine** step grows slower than the cost of the **Reestablish** step. Thus as the input partition size gets larger (more components in the data flow analysis example), the speedup also increases. This can be seen from Figure 5.5 where we show the results of running the same program with a partition containing five elements.

Batch	*Reestablish*	*Combine*	*Total*	*Speedup*
115.32	31.24	26.39	57.63	2.00
116.97	29.26	24.48	53.74	2.18
118.35	29.57	24.90	54.47	2.17
117.26	27.74	26.30	54.04	2.17

Figure 5.4: Data Flow Analysis: Reaching Definitions. A 120 node graph with 3 connected components of approximately equal size.

Batch	*Reestablish*	*Combine*	*Total*	*Speedup*
231.56	28.47	28.52	56.99	4.06
229.82	27.63	26.71	54.34	4.23
236.03	31.02	26.29	57.31	4.12
233.98	29.73	28.81	58.54	3.99

Figure 5.5: Data Flow Analysis: Reaching Definitions. A 200 node graph with 5 connected components of approximately equal size.

The experiments could not be run for examples with partition sizes greater than five, because the Schism programs thus generated were too large for the Scheme compiler to handle. Although there is considerable overlap among the residual functions and this provides many opportunities to share code, our system does not perform this optimization. Therefore the code size is unnecessarily large. We discuss this in the last chapter.

5.5 Attribute Evaluation

The use of attribute grammars [Knu68] to describe language specifications is well known. These specifications have also been used as the starting point for

$$\begin{array}{llllll}
N & \rightarrow & SL & \{\, L.scale & = & 0 \\
 & & & N.val & = & if\ S.neg\ then\ -L.val \\
 & & & & & elseL.val\ \} \\
S & \rightarrow & + & \{\, S.neg & = & false\ \} \\
S & \rightarrow & - & \{\, S.neg & = & true\ \} \\
L & \rightarrow & B & \{\, B.scale & = & L.scale \\
 & & & L.val & = & B.val\ \} \\
L_0 & \rightarrow & L_1B & \{\, L_1.scale & = & L_0.scale + 1 \\
 & & & B.scale & = & L_0.scale \\
 & & & L_0.val & = & L_1.val + B.val\ \} \\
B & \rightarrow & 0 & \{\, B.val & = & 0\ \} \\
B & \rightarrow & 1 & \{\, B.val & = & 2^{B.scale}\ \}
\end{array}$$

Figure 5.6: Attribute grammar for signed binary numerals

the generation of language based programming environments [Rep84]. Programs in this model are represented as *attributed trees*, i.e. syntax trees with attributes carrying semantic values. The goal of incremental attribute evaluation is to efficiently produce a well attributed tree after each editing operation. In what follows, editing is modelled by subtree replacement, and we only consider *non-circular* attribute grammars [Knu68].

As usual let us first describe a non-incremental algorithm, which we do by means of an example. Consider the attribute grammar in Figure 5.6, which describes the syntax for signed binary numerals, as well as the semantics: the decimal value that the numeral denotes. For example, the parse tree for -110 is shown in Figure 5.7 (we refer to this example later).

Given an attribute grammar and a parse tree, the non-incremental program for attribute evaluation is given in Figure 5.8.

Note that the representation of the parse tree is in the form of a list of its subtrees with the root being considered part of the first subtree. Each subtree is represented by the list of equations generated from the tree. For example, the tree in Figure 5.7 is represented by the following list in Haskell:

```
[  [[ "val0", "neg1", "val2" ],
```

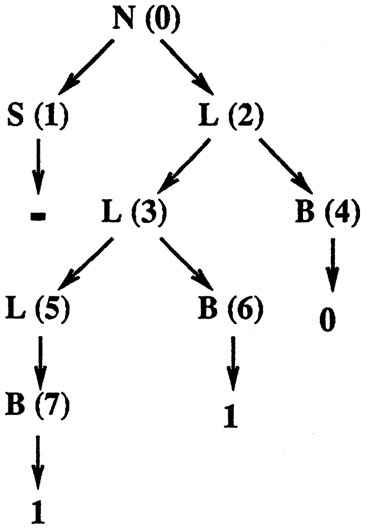

Figure 5.7: Parse tree for -110

```
 [ "neg1" ]],

[[ "scale2" ],
 [ "val2", "val3", "val4" ],
 [ "scale3", "scale2" ],
 [ "scale4", "scale2" ],
 [ "val4" ],
 [ "val3", "val5", "val6" ],
 [ "scale5", "scale3" ],
 [ "scale6", "scale3" ],
 [ "val6", "scale6" ],
 [ "val5", "val7" ],
 [ "scale7", "scale5" ],
 [ "val7", "scale7" ]]]
```

The function `solve` takes a list of equations, sorts them in topological order and evaluates all those variables whose values depend only on the subtree under consideration. Thus, it returns an environment and a list of unsatisfied equations. The function `combine` takes a list of these unevaluated equations and evaluates them in topological order using the environment obtained by merging the individual environments. It also returns any unevaluated equations if any.

Figure 5.9 shows the list of equations generated from the parse tree in Figure

```
-- Attribute grammar evaluator
type Eqn = (Var, Expr)
type Env = [(Var, Val)]

attreval :: [Eqn] -> (Env, [Eqn])
attreval subtrees = foldl combine (nullenv, nulleqns)
                          (map solve subtrees)

solve :: [Eqn] -> (Env, [Eqn])
solve tree = evallist (sched [] tree)

-- schedule a list of equations for solving by a topological sort
sched :: [Eqn] -> [Eqn] -> ([Eqn], [Eqn])
sched readylist eqns =
      case v1 of
       [] -> (readylist, v2)
       (v, e) -> sched (v:readylist) v2
      where (v1, v2) = removereadyeqn readylist eqns

-- find the next ready equation: an equation is ready if all the
-- variables it depends upon have been evaluated
removereadyeqn :: [Eqn] -> [Eqn] -> (Eqn, [Eqn])
removereadyeqn readylist eqns =
               removereadyeqn' readylist [] eqns
removereadyeqn' readylist acc [] = ([], acc)
removereadyeqn' readylist acc (eqn:eqns) =
       if (ready eqn readylist)
       then (eqn, acc ++ eqns)
       else removereadyeqn' readylist (eqn : acc) eqns

-- check if an equation is ready by checking if all the variables it
-- depends upon have been evaluated
ready :: Eqn -> [Eqn] -> Bool
ready (v, e) readylist = allready e readylist

allready :: [Var] -> [Eqn] -> Bool
allready [] readylist = True
allready (var:vars) readylist =
             if or (map (\ v -> v == var) readylist)
             then allready vars readylist
             else False
```

Figure 5.8: Attribute Grammar Evaluator.

$$\begin{aligned}
val_0 &= if\ neg_1\ then\ \ -val_2\ else\ val_2 \\
scale_2 &= 0 \\
neg_1 &= true \\
val_2 &= val_3 + val_4 \\
scale_3 &= scale_2 + 1 \\
scale_4 &= scale_2 \\
val_4 &= 0 \\
val_3 &= val_5 + val_6 \\
scale_5 &= scale_3 + 1 \\
scale_6 &= scale_3 \\
val_6 &= 2^{scale_6} \\
val_5 &= val_7 \\
scale_7 &= scale_5 \\
val_7 &= 2^{scale_7}
\end{aligned}$$

Figure 5.9: Attribute equations for the example parse tree

5.7. Figure 5.10 shows the result of topologically sorting the variables in these equations.

Suppose p_i is the projection reflecting the fact that only the descendants of node i are known. Then $\overline{p_i}$ is the projection where every node *except* the descendants of i are known. A candidate partition is thus:

$$\mathrm{P} = \{p_i\} \cup \{\overline{p_i}\}$$

Recall that the **Setup** phase computes the residual functions corresponding to each projection in the partition. Using the above partition for Figure 5.7, let us see what $r_{\overline{p_5}}$ looks like. The partial evaluator has knowledge of all nodes except 7. This corresponds to the modified dependency graph shown in Figure 5.11, where the dotted portion is unknown. It is not difficult to see that the partial evaluation of the batch program on a tree with these subtrees unknown results in the evaluation of $scale_2$, $scale_3$, $scale_4$, $scale_5$, $scale_6$, val_4, val_6, and neg_1 (but not val_0, val_2, $val3$, val_5, val_7 and $scale_7$).

The **Reestablish** and **Combine** phases work as follows. If we are given a new subtree at (say) node 5, we compute r_{p_5} and take the lub of $r_{\overline{p_5}}$ and r_{p_5}

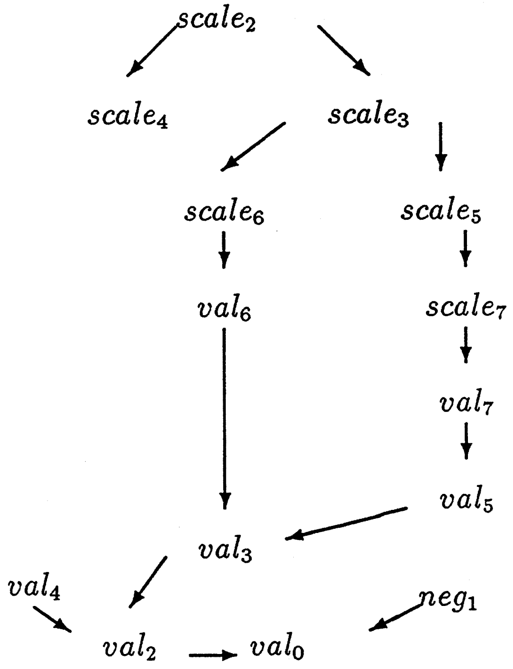

Figure 5.10: Dependency graph

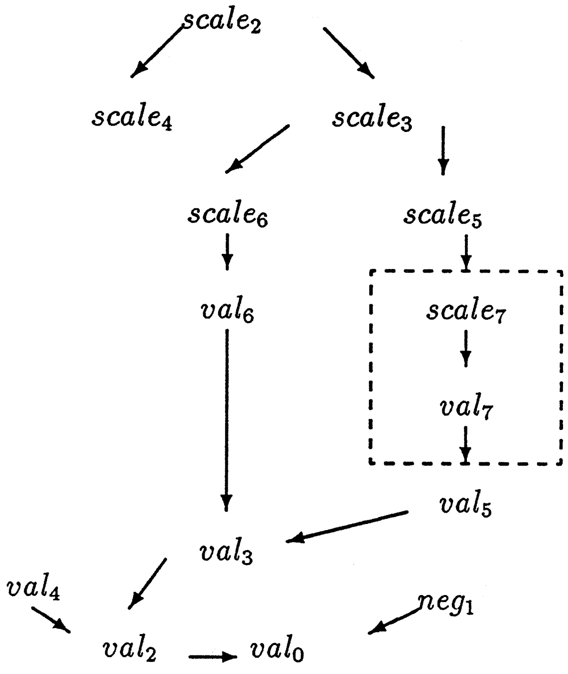

Figure 5.11: Partial dependency graph

to obtain the answer. But now the **Reestablish** phase can be unacceptably expensive. Changing a subtree rooted at node 5 causes residual functions far away from node 5 to be altered. This means that every time a subtree is changed, unacceptably many residual functions have to be recomputed.

This problem leads us to seek a new partition. Consulting the literature, we find in [RTD83] an incremental algorithm for the same problem which reduces the amount of work done while updating the stored information. We can use exactly the same method, described below in our framework.

The key idea is to make use of a *restricted editing model*, in which *cursor* position is maintained:

- The cursor is at any given moment at one node in the parse tree, and can be moved in one of two ways: to the parent node, or to one of the children.
- The only edit operation permitted is subtree replacement.

Thus at any given moment the nodes can be partitioned into three disjoint sets: The first is the singleton set R consisting of the cursor position r. The second is the set S of nodes on the path from the root to the cursor position, including the root. The third is the set T of remaining nodes, which includes nodes below the cursor.

Using this knowledge, the new input partition is as follows:

$$\mathrm{P} = \{p_i, \overline{p_i} \mid i \in R\} \cup \{p_i \mid i \in T\} \cup \{\overline{p_i} \mid i \in S\}$$

The **Setup** phase is similar to that described for the previous partition. The **Reestablish** phase, however, is more interesting. Consider a node a with three children b, c and d. There can now be three kinds of changes.

- **Move To Parent.** The set T gets a new member (the old cursor position); but since the old R had both r_p and $r_{\overline{p}}$ computed, there is no work to do. The set S loses a member, so again their is no work to do. The set R gets a new member for which we need to compute r_p. Moving from child b (say) to parent a, the operation needed to **Reestablish** the invariant is (note that `combine` is the function which forms part of the non incremental program):

$$(r_{p_a} \downarrow 1, r_{p_a} \downarrow 2) = \mathit{combine}\,(r_{p_b} \downarrow 1, r_{p_b} \downarrow 2)\,(r_{p_c} \downarrow 1, r_{p_c} \downarrow 2)\,(r_{p_d} \downarrow 1, r_{p_d} \downarrow 2)$$

We use $r_p \downarrow 1$ to represent the environment produced by evaluating the subtree corresponding to projection p. $r_p \downarrow 2$ represent the list of unsatisfied equations in the subtree represented by projection p.

- **Move to Child.** By similar reasoning, the operation to be performed for a move from parent a to child c is:

$$(r_{\overline{p_c}} \downarrow 1, r_{\overline{p_c}} \downarrow 2) = \mathit{combine}\,(r_{\overline{p_a}} \downarrow 1, r_{\overline{p_a}} \downarrow 2)\,(r_{p_b} \downarrow 1, r_{p_b} \downarrow 2)\,(r_{p_d} \downarrow 1, r_{p_d} \downarrow 2)$$

- **Replace Subtree.** Here we simply need to recompute r_p for the current cursor position. By virtue of the partition, nothing else needs to be changed (cf. the previous partition).

When a subtree is replaced at node p, the new result is obtained by computing $r_p \sqcup r_{\bar{p}}$. The first observation we make concerns the computation of $r_{\bar{p}}$: any attribute which does not depend on projection p of the tree gets a value during the computation of $r_{\bar{p}}$. During the **Reestablish** stage, the work done is in computing those attributes in the modified subtree which do not depend on the rest of the tree. During the **Combine** phase the work done is in computing those attributes in the modified subtree which depend on the rest of the tree. Thus the total work done is exactly in recomputing those attributes which are potentially affected by the subtree modification – only those attributes which depend on the changed subtree get reevaluated. Since we are working within the context of a functional language the time complexity of the topological sort is quadratic in the number of attributes. This means that the time taken to update the parse tree is actually proportional to the square of the attributes potentially affected.

5.5.1 Comparison with known algorithms

The algorithm for incremental attribute evaluation in [RTD83] achieves a better time complexity, namely that an attribute is re-evaluated only if the values of any of the attributes it depends upon change. The algorithm we outlined re-evaluates all attributes which depend on the changed subtree. This algorithm thus achieves an asymptotically linear time complexity in the number of attributes affected. Our algorithm on the other hand is worst case quadratic in the number of attributes potentially affected.

The reason for the quadratic factor is our use of a functional language – linear topological sort cannot be expressed in a garden variety functional language. [4] The other inefficiency relates to our algorithm reevaluating all attributes which may be *potentially* affected, not those which are affected. This is because the algorithm we construct does not take into account a recomputation of an attribute which results in the old value – this does not need to be propagated. This can be seen as a "run-time check", while our algorithm attempts only "compile-time checks".

During cursor movement, the algorithm in [RTD83] achieves unit cost per move. The algorithm we have outlined takes time proportional to the number of attributes whose values are resolved as a result of the `combine` operation.

5.5.2 Performance

Figure 5.12 shows the performance results of running the incremental program on a tree generated by the attribute grammar $L \rightarrow L\ L\ L$, $L \rightarrow a$. The size of the subtree modified is approximately a third of the whole tree. The speedups are less than the "ideal" speedup of three, due to the overhead of the **Combine** step. A similar experiment for a larger tree is shown in 5.13. Again this shows less than the ideal speedup of five.

[4] Recent research [GH90], [SRI91], [Wad90] points to efficient ways of achieving side-effects in a functional language. We have not used these methods.

Batch	*Reestablish*	*Combine*	*Total*	*Speedup*
31.63	6.03	8.71	14.74	2.15
30.43	6.29	7.51	13.80	2.20
30.17	5.92	8.20	14.12	2.14
32.79	4.85	7.53	12.38	2.65

Figure 5.12: Attribute Evaluation: $L \rightarrow L\ L\ L$, $L \rightarrow a$. A 100 node tree with changes made to a subtree of approximately a third the size.

Batch	*Reestablish*	*Combine*	*Total*	*Speedup*
128.84	30.85	0.41	31.26	4.12
128.52	30.91	0.43	31.34	4.10
128.45	30.91	0.41	31.32	4.10
126.94	31.20	0.47	31.67	4.00
127.43	31.20	0.42	31.62	4.03

Figure 5.13: Attribute Evaluation: $L \rightarrow L\ L\ L$, $L \rightarrow a$. A 150 node tree with changes made to a subtree of approximately a fifth the size.

5.6 Strictness Analysis

Strictness analysis is a well known optimization technique in compilers for non-strict functional languages. The purpose of the analysis is to determine which arguments to a function are guaranteed to be evaluated (i.e. in which arguments the function is strict), so that expensive closures do not have to be built for these arguments. We examine the problem of strictness analysis of first order functional programs. First let us examine a non incremental version of the program. Given a set of (possibly recursive) function definitions, an abstract interpretation can give us the desired strictness information [PJ87]. Here is an example:

```
f 0 y = g x y
f x y = f (x - 1) y
```

The abstract version of `f` is given by (abstract functions will be written in a functional style):

```
f# x y = x AND ((g# x y) OR (f# x y))
```

The abstract version is over the domain of monotone boolean values where 0 $\sqsubseteq$ 1 and 0 stands for non-termination and 1 for "may terminate". The solution to this equation can be found by a least fix point construction using the ascending Kleene chain of approximations. In this sense the example resembles the data flow analysis example, the major difference being that the recursive equations in that example involved set valued variables while here the equations involve boolean valued functions. The solution to the above equation is (assuming that `g# x y = y`):

```
f# x y = 1 if x = y = 1,
       = 0 otherwise
```

Given an abstract function, the non-incremental program in Haskell is shown in Figure 5.6 (again we assume that the input equations have been factored into their strongly connected components, where the underlying graph is the function dependency graph).

```
-- abstract expressions
data Expression = Const
                | Var Int
                | If Expression Expression Expression
                | App Fname [Expression]
                | Prim [Expression]
                | Eapp Fname [Expression]
-- abstract values
data Value = Const1 Bool
           | And [Value]
           | Or [Value]
           | Eapp1 Fname [Value]
type Prog = [ Comp ]
type Comp = [ Eqn ]
type Eqn = (Fname, Arity, Expression)
type Env = [ (Fname, [( [Abstractvalue], Abstractvalue )]) ]
type Fname = String

sa :: Prog -> Env -> Env
sa eqns env0 = foldl solve nullenv eqns
-- fix point finder
solve :: Env -> Comp -> Env
solve env scc = if (env == env') then env else solve env' scc
                where env' = nextapprox scc env
nextapprox :: Comp -> Env -> Env
nextapprox prog env =
  map (\ (fname, arity, exp)->
       (fname, map (\ args-> (args, eval exp args env)) (gen arity))) prog
-- evaluate an abstract expression
eval :: Expression -> [ Abstractvalue ] -> Env -> Abstractvalue
eval Const args env = Const1 True
eval (Var n) args env = Const1 (args !! n)
eval (If e1 e2 e3) args env =
   (or' (and' (eval e1 args env) (eval e2 args env)) (eval e3 args env))
eval (App fname exps) args env =
   (lookup env fname (map (\ e-> (con (eval e args env))) exps))
eval (Prim exps) args env = foldl and' (Const1 True)
                                  (map (\ e -> eval e args env) exps)
eval (Eapp fname exps) args env = Eapp1 fname (map(\e-> eval e args env) exps)
```

Figure 5.14: Strictness Analysis.

Again solve is a function which computes the solution to a set of mutually recursive abstract function definitions. The method used is the standard ascending Kleene chain method. The program is represented by an abstract data type with entries for each syntactic class. To complete the program we use the same partition as in the data flow analysis example (this partition enables fix point computation in each element of the partition to proceed to completion locally): $P = \{p_i\}$ such that:

$$\begin{aligned} p_0\,[] &= [], p_0\,(x : xs) &= \bot \\ p_1\,[] &= \bot, p_1\,(x : xs) &= (x : \bot) \\ p_i\,[] &= \bot, p_i\,(x : xs) &= (\bot : (p_{i-1}\ xs)) \end{aligned}$$

Similar to the data flow analysis example, we first transform the strictness analysis code into a form amenable to partial evaluation by performing *arity raising.*

The second transformation is also similar to the data flow analysis case. This transformation splits the environment into static and dynamic parts. Using the filter mechanism, Schism is informed about when to unfold and when to specialize. It differs from the data flow analysis example in that when the evaluation of the fixpoint is complete the static environment is evaluated in the context of the dynamic environment. The reason for this is to evaluate the place holders for the dependencies from outside the connected component.

```
solve scc env_s env_d =
      if (env_s == env_s')
      then (enveval env_s env_d)
      else solve scc env_s' env_d
```

The filter corresponding to the function solve is:

```
(filter (if (and (stat? scc) (stat? env_s)) UNFOLD SPECIALIZE)
        (list scc env_s env_d))
```

As in the data flow analysis example, the solve function needs to be duplicated to achieve the effect of polyvariant binding time analysis. The result of

Batch	*Reestablish*	*Combine*	*Total*	*Speedup*
64.98	39.25	8.07	47.32	1.37
67.44	39.16	9.18	48.34	1.39
66.02	38.54	8.01	46.55	1.42
65.71	39.13	8.30	47.43	1.39
64.18	40.81	8.16	48.97	1.31

Figure 5.15: Strictness Analysis. Program partitioned into two components.

specializing `dfa` with `scc2` and `env0` as `STATIC` is shown below. The value used for `scc2` is the function `f` described in the beginning of this section and the value used for `env0` is the empty environment. Note the unresolved dependence on `g#`:

```
(lambda (scc1 scc3)
  (solve3 scc3 (map31 (car scc3))
    (env-eval2 '(f . ((0 0 0) (0 1 0)
                      (1 0 (g 1 0)) (1 1 (g 1 1))))
               (solve1 scc1 (map11 (car scc1)) null-env))))
```

This describes the details of the **Setup** and the **Reestablish** phases of the interpreter. The **Combine** stage works exactly as in the data flow analysis case, sharing the same deficiencies: non structural changes are not supported, and the **Combine** phase propagates change information throughout the program, rather than affecting only those components dependent on the change. We know of no other attempt to construct an incremental strictness analyzer.

5.6.1 Performance

Tables 5.15, 5.16 show the performance figures for the incremental strictness analyzer. They show the performance for two cases: a partition of size two and a partition of size five.

Batch	*Reestablish*	*Combine*	*Total*	*Speedup*
260.54	46.53	9.98	56.51	4.61
255.40	46.05	8.17	54.22	4.71
259.94	48.29	8.25	56.54	4.59
260.68	48.31	8.35	56.66	4.60
261.01	47.89	8.19	56.08	4.65

Figure 5.16: Strictness Analysis. Program partitioned into five components.

5.7 Type Inference

Hindley-Milner type inference for functional programs concerns itself with finding a most general polymorphic type for each expression in a program (or reporting a type error). This type discipline was first described by Milner [Mil78]. For example, the most general type that can be inferred for the `map` function is: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$.

We start with a version of the Hindley Milner typing algorithm which takes as its input a lambda expression, and outputs two items: a most general polymorphic type and a substitution. The main parts of the non-incremental program are shown in Figure 5.7, 5.18 (the input to the type inference algorithm is assumed to be a list of the strongly connected components of the equations based on the function dependency graph):

The program is a modified version of the type inference program in [PJ87]. `tc` is a function which computes the most general type of a set of mutually recursive abstract function definitions. The method used is the standard Hindley Milner type inference algorithm. The `combine` function simply combines the type environments by type checking those functions which have dependencies outside of their partition. To complete the program we use the same partition as in the dataflow analysis example: $P = \{p_i\}$ such that:

```
-- concrete syntax
data Exp = Var Integer
         | App Exp Exp
         | Lam Integer Exp
-- type expressions
data Texp = Tvar Integer
          | Arrow Texp Texp
          | Error
-- substitutions
data Subst = Scomp sub1 sub2
           | Base [(Integer, Texp)]
           | Error1
type Prog = [Comp]
type Comp = [Exp]
type Tenv = [(Tvar, Texp)]
type Tvar = Integer

typecheck :: Prog -> Subst
typecheck exps = foldl combine nullsubst
                        (map2 (\ exp -> (\ names ->
                               tc exp nulltenv idsubst names)
                             exps namelist))

tc :: Exp -> Tenv -> Subst -> [Tvar] -> (Texp, Subst)
tc (Var v) tenv subst cache = (lookup tenv v, subst)

tc (App e1 e2) tenv subst (name:names) =
   (Tvar alpha, subst3)
   where (te1, s1) = tc e1 tenv subst names
         (te2, s2) = tc e2 tenv s1 names
         alpha = name
         subst3 = unify te1 (Arrowy te2 (Tvar alpha)) s2

tc (Lam v e) tenv subst (name:names) =
   (Arrow (Tvar alpha) te, s)
   where alpha = name
         tenv' = (v,Tvar alpha) : tenv
         (te, s)= tc e tenv' subst names
```

Figure 5.17: Type Inference: part 1

```
-- unify two type expressions to produce a new substitution
unify :: Texp -> Texp -> Subst -> Subst
unify (Tvar v) te2 phi =
      if (tvar (sapp phi v))
      then if (v == (var (sapp phi v)))
           then extend phi v (subtype phi te2)
           else unify (sapp phi v) (subtype phi te2) phi
      else unify (sapp phi v) (subtype phi te2) phi
unify (Arrow te11 te12) (Tvar te21) phi =
      unify (Tvar te21) (Arrow te11 te12) phi
unify (Arrow te11 te12) (Arrow te21 te22) phi =
      phi2
      where phi2 = unify te12 te22 phi1
            phi1 = unify te11 te21 phi

-- perform substitution phi on all variables in type expression
subtype :: Subst -> Texp -> Texp
subtype phi (Tvar v) = sapp phi v
subtype phi (Arrow te1 te2) =
        Arrow (subtype phi te1) (subtype phi te2)
idsubst = Base []

-- Apply a substitution
sapp :: Subst -> Tvar -> Texp
sapp (Scomp s1 s2) tv = subtype s2 (sapp s1 tv)
sapp (Base s) tv =
 if (Error x) then Tvar tv else x
 where x = lookup s tv

-- Extend a substitution
extend :: Subst -> Tvar -> Texp -> Subst
extend phi tvn (Tvar v) =
 if (v == tvn) then phi else Scomp [(tvn, Tvar v)] phi
extend phi tvn texp =
 if (mem tvn (tvars texp)) then Error1  else Scomp [(tvn, texp)] phi

-- collect the type variables in a type expression
tvars :: Texp -> [Tvar]
tvars (Tvar v) = [v]
tvars (Arrow te1 te2) = (tvars te1) ++ (tvars te2)
```

Figure 5.18: Type Inference: part 2

$$
\begin{aligned}
p_0\, [] &= [], p_0\, (x : xs) &&= \bot \\
p_1\, [] &= \bot, p_1\, (x : xs) &&= (x : \bot) \\
p_i\, [] &= \bot, p_i\, (x : xs) &&= (\bot : (p_{i-1}\ xs))
\end{aligned}
$$

This means that each strongly connected component is in a separate element of the partition. This means that all type checks which do not depend on function definitions from outside the component. Similar to the data flow analysis example, we first transform the code into a form amenable to partial evaluation: to make sure that the binding times for arguments to the main function are not partially static, we carry out a process similar to *arity raising.* We perform the usual duplication transformation to achieve polyvariant specialization.

When a residual function is computed, the corresponding `tc` function is completely unrolled. The **Combine** stage works exactly as in the data flow analysis case, sharing the same deficiencies: non structural changes are not supported, and the **Combine** phase propagates change information throughout the program, rather than affecting only those components dependent on the change.

5.7.1 Performance

Tables 5.19, 5.20 show the performance figures for the incremental strictness analyzer. They show the performance for two cases: a partition of size two and a partition of size four.

5.8 Conclusions

In this section we have presented the development of incremental programs for a variety of tasks. Typically the non incremental program needs to be transformed in certain ways to satisfy the partial evaluator – to enable the partial evaluator to specialize the program as intended. Some common features emerge:

- Typically partitions are defined as a function of the data structure under consideration. For example, we considered partitioning a graph into

Batch	*Reestablish*	*Combine*	*Total*	*Speedup*
54.82	33.78	0.09	33.87	1.62
53.17	37.61	0.12	37.73	1.41
52.83	39.72	0.11	39.83	1.33
53.01	47.63	0.11	47.74	1.11
52.98	46.13	0.13	46.26	1.15

Figure 5.19: Type Inference. Program partitioned into two components.

Batch	*Reestablish*	*Combine*	*Total*	*Speedup*
102.41	31.59	0.13	31.72	3.23
105.29	29.43	0.12	29.55	3.56
105.83	32.71	0.09	32.80	3.23
103.59	34.02	0.11	34.13	3.03
101.70	29.58	0.11	29.69	3.42

Figure 5.20: Type Inference. Program partitioned into four components.

strongly connected components. Present day partial evaluators are typically capable of handling only structural kinds of information. In other words, the known/unknown dichotomy needs to be expressed in a structural manner. In our case this entails a transformation of the source program into a form where this is the case. For example, we transformed the data flow analyzer to take as argument a list of the strongly connected components of the input data flow graph. There is work being done to extend partial evaluation to take advantage of arbitrary properties of the inputs (not simply known/unknown) to a program [CK91].

- Frequently it is necessary to separate the static and dynamic computations in a function so that it is possible to explicate the following: termination of unfolding depends only on the static component. This can be seen as a form of staging [JS86] – a technique where *precomputation* or *frequency reduction* is used to move computation among a collection of stages so that work is done as early as possible (so that less time is required in later steps) and as infrequently as possible (to reduce overall time).

- The **Combine** step affects all elements of the partition and does not use the dependency ordering among elements of the partition to eliminate unnecessary computations.

We comment further on these points in the next chapter.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We have presented a framework for constructing incremental programs from their non-incremental counterparts. We have used the notion of partial evaluation to provide a basis for our framework. This is particularly appropriate since binding-time analysis (an important phase of partial evaluation) is concerned with analyzing dependencies in a program based on the known/unknown signature of the input. This approach offers a degree of automation in the construction of incremental programs. A summary of the results in this thesis are as follows:

- We proposed a connection between incremental computation and partial evaluation. The connection was formalized by proposing a methodology which used partial evaluation as a tool to build incremental programs. An incremental interpreter was presented, which used a cache of residual function in order to obtain incrementality.

- In order to answer theoretical questions raised by the methodology we investigated the algebraic properties of residual functions over certain projection domains. In doing so, we established the existence of a least upper bound, greatest lower bound and a difference operator – both for the domain of projections and the homomorphic domain of residual functions. We identified the "combining" operator on residual functions with the least upper bound operation. Thus we were assured of its existence and uniqueness. We also obtained an insight into its algebraic properties.

- We presented an algorithm to compute the least upper bound of two residual functions. This algorithm relies on the binding time information belonging to each residual function. The binding time information is used to construct the least upper bound by choosing the "more evaluated" parts of the two residual functions. We presented a detailed proof of correctness of the algorithm. The algorithm is also shown not to reperform any computation already done in either one of the two residual functions.
- We applied the methodology to a number of problems and obtained performance results from an implementation using the Schism partial evaluator [Con90a, Con90b, Con88]. The descriptions include the non incremental program, the partition used, details of the specialization phase and the combine phase.

6.2 Future Work

Binding Time Analysis Binding time analysis is essential for good performance of the incremental programs we have described. Since the partition is known before the actual data elements are available, binding time analysis can be carried out off-line. This means that the self-application described above can avoid the overhead of interpretation. Binding time analysis is usually achieved by an abstract interpretation of the program. To ensure termination, finite domains are usually used. But as we have seen our domain of projections is infinite (although the number of projections used in any given session is finite). One way to overcome this problem for implementation purposes is to place a fixed upper limit on the size of the input data. We are investigating other solutions. We have not discussed the cost of maintaining the input partition. In some cases (the attribute grammar example) the cost of maintaining the partition is not very high. In the case of the data flow analysis we must employ a method to maintain the strongly connected components of the flow graph.

Non-structural Changes The methods we have discussed do not work well for non-structural changes – changes which may affect the structure of the partition. The naive way of handling non structural changes in our framework induces an unacceptably large amount of recomputation.

Sharing of Residual Functions Sharing residual functions across different computations may not be possible. If two data flow graphs share a strongly connected component, it should be possible to use the residual function from one incremental session in the other session. But this may not be possible since the two strongly connected components, although they are the same, may occupy different positions in the list of strongly connected components.

How can we overcome the restriction mentioned above? Here are some preliminary thoughts: If instead of building larger residual functions with respect to their projection component, consider building larger residual functions based on the argument component. i.e. instead of

$$r_{p,a} \sqcup r_{q,a} = r_{p \sqcup q,a}$$

consider:

$$r_{p,a} \sqcup r_{p,b} = r_{p,a \sqcup b}$$

The advantage of caching residual functions of this form is that it enables the residual for one element of the partition to be independent of a residual for another. This means that residuals can be freely reused between different graphs. This may also point to a way of handling non structural changes. What does the new combination operator look like? Let us look at a specific example: say data flow analysis. Here if the non-incremental program takes as input a strongly connected component graph and an environment containing bindings for arcs flowing into the component, then the projection p above is simply $LEFT$. The combination operator simply involves pipelining the residuals (function composition) in topological order. An important point of departure from the present scheme of things is that the topological order of elements in the partition plays an important role. This can be exploited to provide better change propagation strategies.

Space Utilization As mentioned in Chapter 5, storing the cache in a naive manner leads to excessive space consumption. There is considerable overlap in the bodies of the residual functions and there are opportunities for code sharing here.

Bibliography

[AHR+90] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and K. Zadeck. Incremental evaluation of computational circuits. In *ACM-SIAM Symposium on discrete algorithms*, pages 32–42, January 1990.

[Ase87] P.J. Asente. *Editing graphical objects using procedural representations.* PhD thesis, Stanford University, 1987.

[BHOS76] L. Beckman, A. Haraldson, Ö. Oskarsson, and E. Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[BM75] R. Boyer and J.S. Moore. Proving theorems about lisp functions. In *International Joint Conference on Artificial Intelligence*, 1975.

[Bro88] K.P. Brooks. A two-view document editor with user-definable document structure. Technical Report 33, DEC Systems Research Center, November 1988.

[CD89] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, 1989.

[CD90a] C. Consel and O. Danvy. From interpreting to compiling binding times. In *Proceedings of the 3rd European Symposium on Programming, Lecture Notes in Computer Science, Vol. 432.* Springer-Verlag, May 1990.

[CD90b] C. Consel and O. Danvy. Partial evaluation in parallel. Research Report 820, Yale University, New Haven, Connecticut, USA, 1990.

[CD91] C. Consel and O. Danvy. For a better support of static data flow. In John Hughes, editor, *Proceedings Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA*, pages 496–519. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

[CK91] C. Consel and S.C. Khoo. Parameterized partial evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Imple mentation*, 1991.

[Con88] C. Consel. New insights into partial evaluation: the schism experiment. In *ESOP'88, 2nd European Symposium on Programming, Nancy, France*, volume 300 of *Lecture Notes in Computer Science.* Springer-Verlag, March 1988.

[Con90a] C. Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Languages*, June 1990.

[Con90b] C. Consel. *The Schism Manual.* Yale University, Department of Computer Science, November 1990.

[EH80] P. Emanuelson and A. Haraldsson. On compiling embedded languages in lisp. In *Proceedings of the 1980 Lisp Conference*, pages 208–215, 1980.

[FT90] J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[Fut71] Y. Futamura. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5), 1971.

[GH90] J. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *IEEE Symposium on Logic in Computer Science*, pages 333–343, June 1990. Philadelphia, Pennsylvania.

[HKR89] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. In *Proceedings of the SIGPLAN '89 conference on programming language design and implementation*, pages 179–191, 1989.

[HT86a] R. Hoover and T. Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. In *Proceedings of the SIGPLAN '86 symposium on compiler construction*, pages 39–50, July 1986.

[HT86b] S. Horwitz and T. Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[Hug88] R.J.M. Hughes. Backwards analysis of functional programs. In A.P.Ershov D.Bjørner and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[HW87] R.J.M. Hughes and P. Wadler. Projections for strictness analysis. In *FPCA*, 1987.

[HW88] P. Henderson and M. Weiser. The visiprog environment. In *Prooceedings of the 8th international conference on software engineering*, 1988.

[HWe90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell. Technical Report YALEU/DCS/RR-777, Yale University, Department of Computer Science, April 1990.

[JGB+90] N. D. Jones, C. K. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *IEEE International Conference on Computer Languages*, pages 49–58, 1990.

[JS86] U. Jørring and W.L. Scherlis. Compilers and staging transformations. In *Proceedings of the Thirteenth principles of programming languages conference*, pages 86–96, 1986.

[JSS89] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1), 1989.

[Knu68] D. Knuth. Semantics of context-free languages. *Math Systems Theory*, 2(2):127–145, February 1968.

[Lau88] J. Launchbury. Projections for specialisation. In A.P.Ershov D.Bjørner and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[Lau90] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, University of Glasgow, January 1990.

[Lel88] W. Leler. *Constraint Programming Languages.* Addison-Wesley, 1988.

[LR64] L. A. Lombardi and B. Raphael. Lisp as the language for an incremental computer. In *The Programming Language LISP: Its Operation and Applications*, pages 204–219. Information International Inc., The MIT Press, 1964.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[Mil84] R. Milner. A proposal for standard ml. In *Proceedings of the 1984 Conference on LISP and Functional Programming*, pages 184–197, August 1984.

[Mog86] T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, 1986.

[MR90] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

[MT46] J. C. C. McKinsey and A. Tarski. On closed elements in closure algebras. *Annals of Mathematics*, 47(1), January 1946.

[Nel85] G. Nelson. Juno, a constraint-based graphics system. *Computer Graphics*, 19(3):235–243, 1985.

[Nie89] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69:117–242, 1989.

[Pai86] R. Paige. Programming with invariants. *IEEE Software*, 3(1):59–69, January 1986.

[PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[PK82] R. Paige and S. Koenig. Finite differencing computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[Pug88] W. W. Pugh, Jr. *Incremental Computation and the Incremental Evaluation of Functional Programs.* PhD thesis, Cornell University, August 1988.

[RC86] B.G. Ryder and M. Carroll. An incremental algorithm for software analysis. In *proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments.* ACM, 1986.

[Rep84] T. W. Reps. *Generating Language-Based Environments.* The MIT Press, 1984.

[Rep90] T. Reps. Algebraic properties of program integration. In *Proceedings of the 3rd European Symposium on Programming, Lecture Notes in Computer Science, Vol. 432*. Springer-Verlag, May 1990.

[Rom88] S.A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In A.P.Ershov D.Bjørner and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[RP86] B.G. Ryder and M.C. Paull. elimination algorithms for data flow analysis. *ACM computing surveys*, 18(3):277–316, 1986.

[RTD83] T. Reps, T. Teitelbaum, and A. Demers. Incremental context dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[Ryd82] B.G. Ryder. Incremental data flow analysis. In *conference record of the ninth annual ACM symposium on principles of programming languages*. ACM, 1982.

[Ses86] P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and N.D.Jones, editors, *Programs as data objects*. Springer Verlag, LNCS 217, 1986.

[SRI91] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In John Hughes, editor, *Proceedings Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA*, pages 192–214. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

[Tur81] D.A. Turner. The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 85–92, 1981.

[Tur82] D.A. Turner. Recursion equations as a programming language. In *Functional Programming and its Applications: an advanced course*, pages 1–28. Cambridge University Press, 1982.

[Wad90] P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, 1990.

[Wik88] A. Wikstrom. *Standard ML.* Prentice-Hall, 1988.

[YS91] D. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2), April 1991.

Appendix A

Code for Algorithm LUB

The code for algorithm **LUB** under the Schism partial evaluation system is shown below. r1 and r2 are the residual expressions, c1 and c2 are the source expressions, at1 and at2 are the action trees corresponding to each source expression, cenv1 and cenv2 are environments which contain the results of the evaluation of conditionals, ana1 and ana2 are the results of the binding time analysis of each of the programs.

```
(define (inc:lub r1 c1 at1 env1 cenv1 ana1 r2 c2 at2 env2
                 cenv2 ana2)
  (cond
    ((inc:application? c1)
     (cond
       ((eval? at1) r1)
       ((eval? at2) r2)
       ((id? at1) r2)
       ((id? at2) r1)

       ((and (reduce? at1) (reduce? at2))
        (let ((v1 (sel ana1 (fn-name c1)))
              (v2 (sel ana2 (fn-name c2))))

        (inc:lub r1 (extract-code v1) (extract-at v1)
             (update-env env1 (arg-names v1)
                              (dup (inc:length (arg-names v1)) '?)
                              (arg-bodies c1)
```

```
                          (arg-ats at1))
         cenv1 ana1
         r2 (extract-code v2) (extract-at v2)
         (update-env env2 (arg-names v2)
                          (dup (inc:length (arg-names v2)) '?)
                          (arg-bodies c2)
                          (arg-ats at2))
         cenv2 ana2)))

   ((and (reduce? at1) (rebuild? at2))
    (let ((v1 (sel ana1 (fn-name c1)))
          (v2 (sel ana2 (fn-name c2))))

    (inc:lub r1 (extract-code v1) (extract-at v1)
         (update-env env1 (arg-names v1)
                          (dup (inc:length (arg-names v1)) '?)
                          (arg-bodies c1)
                          (arg-ats at1))
         cenv1 ana1
         (extract-code v2) (extract-code v2)
         (extract-at v2)
         (update-env env2 (arg-names v2)
                          (arg-bodies r2)
                          (arg-bodies c2)
                          (arg-ats at2))
         cenv2 ana2)))

   ((and (rebuild? at1) (reduce? at2))
    (... ANALOGOUS TO THE LAST CASE ...))

   ((and (rebuild? at1) (rebuild? at2))
    (mk-application (fn-name c1)
                    (map6 (lambda (x y z u v w)
                            (inc:lub x y z env1 cenv1 ana1 u v w env2
                                     cenv2 ana2))
                            (arg-bodies r1) (arg-bodies c1)
                            (arg-ats at1)
                            (arg-bodies r2) (arg-bodies c2)
                            (arg-ats at2))))))

((inc:variable? c1)
 (cond
   ((eval? at1) r1)
   ((eval? at2) r2)

   ((and (rebuild? at1) (fail? (inc:lookup-env
                                  env1 (var-name c1))))
    r2)

   ((and (rebuild? at2) (fail? (inc:lookup-env
                                  env2 (var-name c2))))
    r1)

   ((and (reduce? at1) (reduce? at2))
    (let ((v1 (inc:lookup-env env1 (var-name c1)))
          (v2 (inc:lookup-env env2 (var-name c2))))
    (inc:lub r1 (snd v1) (trd v1) (frt v1) cenv1 ana1
         r2 (snd v2) (trd v2) (frt v2) cenv2 ana2)
    ))

   ((and (rebuild? at1) (rebuild? at2))
    (let ((v1 (inc:lookup-env env1 (var-name c1)))
          (v2 (inc:lookup-env env2 (var-name c2))))
         (inc:lub (fst v1) (snd v1) (trd v1) (frt v1) cenv1 ana1
              (fst v2) (snd v2) (trd v2) (frt v2) cenv2 ana2)))

   ((and (reduce? at1) (rebuild? at2))
    (let ((v1 (inc:lookup-env env1 (var-name c1)))
          (v2 (inc:lookup-env env2 (var-name c2))))
         (inc:lub r1       (snd v1) (trd v1) (frt v1) cenv1 ana1
              (fst v2) (snd v2) (trd v2) (frt v2) cenv2 ana2)))


   ((and (rebuild? at1) (reduce? at2))
    (... ANALOGOUS TO THE LAST CASE ...))))

((inc:conditional? c1)
 (cond
   ((eval? at1) r1)
   ((eval? at2) r2)
   ((id? at1) r2)
   ((id? at2) r1)

   ((and (rebuild? at1) (rebuild? at2))
 (mk-if
   (inc:lub (test-body r1) (test-body c1)
            (test-at at1) env1 cenv1 ana1
            (test-body r2) (test-body c2)
            (test-at at2) env2 cenv2 ana2)
   (inc:lub (cons-body r1) (cons-body c1)
            (cons-at at1) env1 cenv1 ana1
            (cons-body r2) (cons-body c2)
            (cons-at at2) env2 cenv2 ana2)
   (inc:lub (alt-body r1) (alt-body c1) (alt-at at1) env1
            cenv1 ana1
            (alt-body r2) (alt-body c2) (alt-at at2) env2
            cenv2 ana2
)))

((and (rebuild? at1) (reduce? at2))
 (let ((v (lookup-cenv cenv2 (cond-name c2))))
      (inc:lub (choose-b (car v) r1) (choose-b (car v) c1)
           (choose-a (car v) at1) env1 cenv1 ana1
           r2 (choose-b (car v) c2)
           (choose-a (car v) at2) env2 (cdr v) ana2)))

((and (reduce? at1) (rebuild? at2))
 (... ANALOGOUS TO THE LAST CASE ...))

((and (reduce? at1) (reduce? at2))
 (let  ((v1 (lookup-cenv cenv1 (cond-name c1)))
        (v2 (lookup-cenv cenv2 (cond-name c2))))
       (inc:lub r1 (choose-b (car v1) c1)
                (choose-a (car v1) at1)
                env1 (cdr v1) ana1
                r2 (choose-b (car v2) c2)
                (choose-a (car v2) at2)
                env2 (cdr v2) ana2)))))))
```