DIAGRAMMES

A. C. REEVES

Towards a sketch based model of self-interpreters

Diagrammes, tome 33 (1995), p. I-178

<http://www.numdam.org/item?id=DIA_1995__33__R1_0>

DIAGRAMMES VOLUME 33 , 1995

TOWARDS A SKETCH BASED MODEL OF SELF-INTERPRETERS

A. C. Reeves

Abstract

There has been a steady stream of research into compiler generation systems since the late 1960's most of which has involved some sort of approach where the user specifies the *source* language, the *target* language and the *source* → *target* relationship. This specification of the *source* → *target* relationship is, in effect, a specification of the compiler and the correctness of the generated compiler depends on the correctness of this relationship.

In this thesis we propose an approach, based on partial evaluation, which does not involve the specification of the *source* → *target* relationship. Correctness of the generated compilers therefore depends solely on the specification of *source* and *target* languages and upon the soundness of the theory underlying the technique. The method requires the automatic derivation of both a target partial evaluator and a source interpreter, expressed as a target program. We attempt the development of a technique to calculate a self-interpreter, an $\mathcal{L}$ interpreter which is itself an $\mathcal{L}$ program, for an arbitrary language, $\mathcal{L}$, as this represents a significant step towards the goal of the automatic derivation of both partial evaluators and interpreters.

Initially we examine an algebraic model of language which allows us to specify the function which an interpreter for the language $\mathcal{L}$ computes, solely in terms of the algebraic specification of the language $\mathcal{L}$. The interpreter is described as the composition of a pair of functions, $learn : Semantics \rightarrow Syntax$ which forms part of the algebraic specification of $\mathcal{L}$, and $eval : Syntax \rightarrow Semantics$ which arises naturally from the language specification due to the properties of the category of algebras over a common signature.

Using the algebraic model of language the composition $learn \circ eval$, which is the $\mathcal{L}$ interpreter function, does not lie within the semantics of $\mathcal{L}$ and therefore cannot easily be used to construct the $\mathcal{L}$ self-interpreter.

For this reason a category theoretic model of language based on finite limit sketches is developed. This model is similar to the algebraic model above and shares many of its properties but has the advantage that *learn* is expressed as an indexed family of arrows from **SET**, the category of sets, and that *eval* is a natural transformation whose components also lie within **SET**. As a result of this the components of $learn \circ eval$ can be brought within the semantics of $\mathcal{L}$. We can then use the structure of the natural transformation *eval* to construct an implementation of $learn \circ eval$ as an $\mathcal{L}$ program.

For Charles and Isobel.
Nobody could have better friends,
I am extremely lucky to have you as parents.

Acknowledgements

Over the course of the last five years I have received help and encouragement from numerous people both inside the computing departments at Stirling and Keele Universities and outside them. This support, not always with the technical problems in this work, has been of indescribable value to me and I would like to thank you all. Several people deserve special mention.

First and foremost Charles Rattray, my friend and supervisor, whose support and encouragement, not to mention his, to my mind, irrational belief in me has not only kept me sane but has helped to make this work something it could not otherwise have become, completed.

Teodor Rus deserves special mention for taking the time to discuss both his own work in the context of mine and my work in the context of his. As do Christian Lair and Laurent Coppey, whose boundless enthusiasm and penetrating questions helped to keep alive my interest in this work.

I would like to thank David Budgen both for making it possible for me to attend "Journées esquisses et types de structures" in Paris and for his encouragement over the time I have been at Keele.

Thanks should also go to Jim Cowie, whose encouragement and comments on the initial drafts finally got me to admit that I could actually start writing this document, and to Mike Johnson who explained Kan extensions to me in a way I could understand.

The discussions I had with Samuel Kortas, Simon Jones, John Launchbury, Richard Quatrain, and Peter Sestoft helped no end in improving my understanding of the problems I was trying to solve.

Sam Nelson and Graham Cochrane deserve my thanks both for the excellent computing facilities and support and for being prepared to explain those facilities to me and listen to my explanations of this work with a patience that I will never understand.

The office cricket club: Dave Harrison, Andrew Sinclair, and Paul Williamson not only helped me to survive the trials and tribulations of life as a Ph.D. student but also provided useful feed-back via many discussions.

On a personal note I would also like to thank my friends Maria Shaw, Bill Stewart, Sarah Willis-Culpitt, Tony Douglas, Andy Poxon. Anne Tweed, the other irregulars, and Aunty Eleanor for being there when I needed them and for understanding when I was too busy.

Without you this thesis would never have reached (been battered into) submission!

Contents

Chapter 1

A Brief Introduction and Guided Tour

This dissertation describes a method for calculating self-interpreters for arbitrary programming languages, i.e. an interpreter for the language $\mathcal{L}$ which is an $\mathcal{L}$ program. On the face of it an $\mathcal{L}$ self-interpreter, *$\mathcal{L}$-self-int*, is a singularly useless program, it cannot provide an implementation of the language $\mathcal{L}$ because an implementation of $\mathcal{L}$ is required before *$\mathcal{L}$-self-int* can be run. Even then *$\mathcal{L}$-self-int* can only make $\mathcal{L}$ programs run slower by adding an extra layer of unnecessary interpretation to the implementation. So what is the purpose of an $\mathcal{L}$ self-interpreter?

The calculation of *$\mathcal{L}$-self-int* is a step towards the ability to calculate an interpreter, *int*, for $\mathcal{L}$ as a program in the arbitrary language $\mathcal{T}$. Calculation of *$\mathcal{L}$-self-int* is also a reasonable starting point if we wish to calculate the partial evaluator *mix* [Ersh82, Jone88] for the language $\mathcal{L}$. Given the ability to calculate *int* and *mix* for arbitrary languages we can construct a compiler generation system which requires no user input other than the specifications of the source and target languages.

In chapter 2 we set the scene by outlining the development of compiler-compilers and make the distinction between a *compiler specification language* which requires the user to *specify* the relationship between the source language $\mathcal{S}$ and the target language $\mathcal{T}$, and a true *compiler generation system* which requires no such specification of the $\mathcal{S} \rightarrow \mathcal{T}$ relationship. We attempt to show that a true compiler generation system can be constructed based on partial evaluation and the ability to calculate the appropriate *int* and *mix* programs for the languages $\mathcal{S}$ and $\mathcal{T}$. The main purpose of chapter 2 is to motivate the subsequent chapters which deal with the calculation of *$\mathcal{L}$-self-int.*

The algebraic model of language developed by Rus [HaRu76, Rus76, RuHe84, Rus85, Rus87, Rus90, Rus92] is discussed in chapter 3. Using this model of language it is possible to describe the function computed by *$\mathcal{L}$-self-int* in terms of the specification of $\mathcal{L}$. The function *interpreter*, computed by *$\mathcal{L}$-self-int* is described as the composition of certain functions which form part of Rus' algebraic model. There is no obvious way to turn the *interpreter* function into an $\mathcal{L}$ program because the functions used to describe it are not within the semantics of $\mathcal{L}$. In spite of this the algebraic approach is not a complete blind alley, it provides a gentle introduction to the category theoretic approach used subsequently.

In chapter 4 we describe aspects of the theory of sketches [Ehre68] which are used in chapter 5 to construct a categorical model of language which has similar properties to Rus' algebraic model of language. For reasons of space we have assumed that the reader is familiar with basic category theory, an understanding of (at least) the concepts of category, functor, natural transformation, and adjunction are required *before* reading further. Readers unfamiliar with category theory are referred to [BaWe90, Gold84, Macl71, RyBu88].

We describe a categorical model of language based on sketches in chapter 5 and discuss some of its implications for the way in which we specify certain language constructs.

Chapter 6 concerns the derivation of a self-interpreter. Using the categorical model of language the function computed by *$\mathcal{L}$-self-int* is described as the pointwise composition of a pair of indexed collections of arrows in the category of sets and functions. These collections can be calculated from the sketch specification of the language $\mathcal{L}$. This version of the *interpreter* function also lies outside the semantics of $\mathcal{L}$ but, because it is structured as a collection of arrows in **SET**, we can construct an analogue of each component arrow of the *interpreter* function which acts on a representation of the syntax of $\mathcal{L}$ and is within the semantics of $\mathcal{L}$. This allows us not only to convert the *interpreter* function into an $\mathcal{L}$ program but also, at least partially, to formalise the notion of expressive power required for $\mathcal{L}$ to express *$\mathcal{L}$-self-int.* We do not attempt to derive a representation of the syntax of $\mathcal{L}$ as an $\mathcal{L}$ datatype as this is a relatively trivial problem.

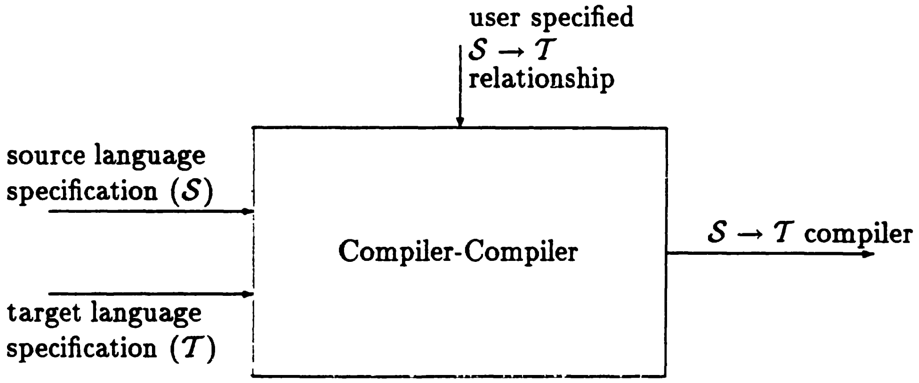
The final chapter, chapter 7, re-examines the method used to calculate the self-interpreter, points out some of its shortcomings, and suggests possible extensions to allow the calculation of partial evaluators and interpreters.

Finally appendix A contains an example of the calculation of a self-interpreter.

Chapter 2

Compiler Generation: A Science Fiction Story

The compiler generation system described in this chapter is a work of fiction but, in common with many other science fiction stories, its roots are firmly planted in science fact. Since the birth of formal language specification attempts have been made to produce systems which generate compilers from formal descriptions of the source and target languages. A conventional compiler generation system requires the user to specify the relationship between the source and target languages in addition to the source and target languages themselves, see figure 2.1.



user specified $\mathcal{S} \rightarrow \mathcal{T}$ relationship
source language specification ($\mathcal{S}$)
target language specification ($\mathcal{T}$)
Compiler-Compiler
$\mathcal{S} \rightarrow \mathcal{T}$ compiler

Figure 2.1: A conventional compiler specification system

Given this fact, conventional compiler generation systems could perhaps be more correctly described as compiler specification languages. The major drawbacks of the approach are:

1. the specification of the $\mathcal{S} \rightarrow \mathcal{T}$ relationship requires a great deal of time and effort.
2. If the user incorrectly specifies the $\mathcal{S} \rightarrow \mathcal{T}$ relationship, the compiler-compiler will usually generate an incorrect compiler. As a result if the user wishes to guarantee the correctness of the generated compiler they *must* prove the correctness of their $\mathcal{S} \rightarrow \mathcal{T}$ relationship [BuLa69, Morr73, ThWW80, Wand80, Coll86]. This proof is likely to be rather involved and just as prone to errors as the original specification of the $\mathcal{S} \rightarrow \mathcal{T}$ relationship.

The process could perhaps be improved somewhat by providing machine assistance for the correctness proof. but even then the process of compiler specification is still a long and involved task.

A true compiler generation system, in the opinion of the author, should require no user input other than the specifications of the source and target languages.

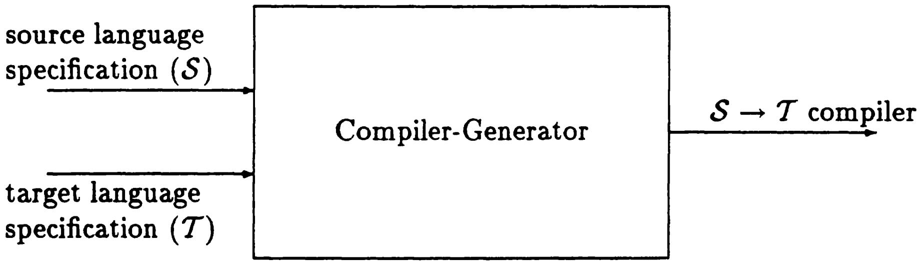

source language
specification ($\mathcal{S}$)
target language
specification ($\mathcal{T}$)
Compiler-Generator
$\mathcal{S} \rightarrow \mathcal{T}$ compiler

Figure 2.2: A True Compiler Generator

If such a system had a sound basis in mathematics it would not only provide considerable savings both in user time and effort but would also generate compilers which could be guaranteed correct by construction.

The remainder of this chapter attempts to answer the question, "How could a true compiler generation system operate?"

2.1 The factual basis of the story

Before attempting to answer the question above we should examine the main approaches to the implementation of compiler specification languages which are currently available.

2.1.1 Syntax directed compiler generators

Probably the simplest form of compiler specification system is the syntax directed compiler generator. Using this technique the source language is specified as a context free grammar. A semantic action is associated with each production rule of the source grammar and the compiler is produced by generating a parser for the source language. The parser is constructed in such a way that it executes the semantic action associated with a production rule whenever it recognises a phrase generated using that production rule. One of the first attempts to produce a syntax directed compiler generator was the STAGE2 system [Wait70]. Other examples of syntax directed compiler generators include YACC [John78], DELTA [Lorh82], and SYNTAX [Boul80]. Of these YACC is probably the most generally available as it is distributed as part of the UNIX[1] operating system. The most obvious shortcoming of the syntax directed technique is that the semantic action associated with a production rule does *not* describe either the semantics of the source language phrase, or the target language construct used to implement the source language phrase. In fact the target language of the generated compiler is not specified at all using the syntax directed approach.

What the semantic action actually specifies is the action to be taken by the generated compiler on recognising the source phrase associated with each action. This effectively obscures the $\mathcal{S} \rightarrow \mathcal{T}$ relationship by hiding it within the implementation details of the compiler, making its construction and correctness proof much harder.

Since the only objects which are formally specified are the source syntax and (in some cases) the meta-language used to express the compiler specification, the correctness proof requires a great deal of additional information: i.e. source semantics, target syntax, and target semantics. The requirement for additional information also makes the correctness proof much more difficult. It could be argued that the requirement for additional information makes the

[1] UNIX is a trademark of AT & T Bell Laboratories

syntax directed compiler specification technique a semi-formal compiler specification method rather than a formal one.

2.1.2 Semantics directed compiler generators

A second approach to compiler specification is the semantics directed compiler generation technique. Compiler writing using this approach is based on a formal description of the source language as input data, and the target language as output data. Usually the source language is described as a context free grammar where each source phrase has an associated target language construction which describes its semantics.

The details of compiler specification vary from system to system but, in general, all semantics directed compiler specification systems conform to one or other of the approaches given in [Moss76].

> "Choose a 'universal' object code with a well defined semantics. Then to generate a compiler from a given denotational semantics for some programming language. find code sequences which simulate the abstract meanings of the phrases of the language. and construct a compiler which produces these code sequences"

or

> "Take a more abstract view of compiling: instead of
>
> $$\mathit{Compiler}: \mathit{progs} \rightarrow \mathit{code}$$
>
> consider
>
> $$\mathit{Compiler}: \mathit{progs} \rightarrow \mathit{input\text{-}output\text{-}fns}.$$
>
> Thus an abstract compiler does not transform an (abstract) program text into an (abstract) sequence of instructions; rather it transforms it into the abstract *input-output-fn* represented by those instructions. The concrete version of such an abstract compiler produces denotations (i.e. representations) of *input-output-fns* from denotations of programs — it is just an implementation of a denotational semantics."

Although Mosses was referring specifically to semantics directed compiler specification systems based on denotational semantics the same thing applies to systems based on attribute grammars [Boch78] or on algebraic semantics [Desc82].

Most commonly semantics directed compiler generators are based on the denotational approach to programming language semantics [ScSt71, Stoy77, Schm86], and there is a great deal of literature dealing with this type of semantics directed compiler specification system, for example: [Ganz79, Moss79, RaTu79, JoSc80, Schm85, Wand85, Roye86, Vick86].

Of these [Roye86] is of particular interest because it attempts to derive a target semantics from a source denotational semantics in, "the most constructive way possible." The technique described by Royer is still a compiler specification technique rather than a true compiler generation system because the user has to supply the $\mathcal{S} \rightarrow \mathcal{T}$ relationship in the form of a collection of target domains which are used to implement the source domains, together with a congruence relation for these domains.

In [Schm85] Schmidt also constructs a semantics directed compiler directly from the standard denotational semantics of a programming language rather than a continuation style denotational semantics as is more usual. This has the advantage that the semantics used to specify the programming language is at a much higher level. The approach used is to transform the semantics so that operational properties of the semantics become clearer. The transformations used are focussed on the operational properties of the particular reduction strategy used to implement an interpreter for the denotational semantics definition. Using this technique the implementor has to supply rather more information about the $\mathcal{S} \rightarrow \mathcal{T}$ relationship than is the case if a continuation semantics were used. This may in fact be an advantage because the user can use implementation "tricks" to produce a much more optimal implementation, however; it also moves further away from the goal of this chapter.

The compiler specification system described in [JoSc80] consists of a back end compiler $\varphi : LAMC \rightarrow STM$ which translates a dialect of the lambda calculus, $(LAMC)$, into a language of state transition machines, (STM). The front end is defined by providing a denotational definition, Δ, of the source language, $\mathcal{S}$, using the $LAMC$ language, this defines a mapping $\overline{\Delta} : \mathcal{S} \rightarrow LAMC$. The compiler is then specified as the function $\varphi \circ \overline{\Delta}$.

In general semantics directed compilers are specified as the composition of front and back ends.

$$\begin{tikzcd}
\text{Source Language} \arrow[r, "\textit{front end}"] & \text{Intermediate Language} \arrow[r, "\textit{back end}"] & \text{Target Language}
\end{tikzcd}$$

This leads to a problem in the correctness proof because the intermediate language is *not* formally specified as part of the generation process and must therefore be specified as additional information during the correctness proof.

2.1.3 Algebraic directed compiler generators

The T.I.C.S. System developed by Rus [Rus83, RuHe84, Rus90, Rus92] is, to the best of the author's knowledge, the only working example of the third class of compiler specification system, namely Algebraic directed compiler specification systems. The system is based on the "commuting square" notion of compiler correctness [BuLa69, Morr73, ThWW80, Wand80], and depends on an algebraic model of language also developed by Rus [HaRu76, Rus76, RuHe84, Rus85, Rus86, Rus87].

This algebraic model of language is described in detail in chapter 3 but can be summarised here as follows. A programming language $\mathcal{L}$ is a triple

$$\langle Sem(\Sigma), Syn(\Sigma), learn : Sem(\Sigma) \rightarrow Syn(\Sigma)\rangle$$

where $Syn(\Sigma)$ specifies the syntax and is the word algebra generated by the signature Σ. The semantics is specified by the similar algebra $Sem(\Sigma)$ and the $Syntax \leftrightarrow Semantics$ association is specified by the function $learn : Sem(\Sigma) \rightarrow Syn(\Sigma)$ and by the initiality of $Syn(\Sigma)$, which gives rise to an homomorphism $eval : Syn(\Sigma) \rightarrow Sem(\Sigma)$. If $\mathcal{S}$ is a programming language specified over Σ_1 and $\mathcal{T}$ is specified over Σ_2 then a compiler $\mathcal{C} : \mathcal{S} \rightarrow \mathcal{T}$ is specified as a pair of homomorphisms, *compile* and *encode*, such that the equations

$$\begin{aligned} encode &= eval_2 \circ compile \circ learn_1 \\ compile &= eval_1 \circ encode \circ learn_2 \end{aligned}$$

both hold in figure 2.3.

$$\begin{tikzcd}
Syn(\Sigma_1) \arrow[r, "compile"] \arrow[d, shift left, "eval_1"] & Syn(\Sigma_2) \arrow[d, shift left, "eval_2"] \\
Sem(\Sigma_1) \arrow[r, "encode"'] \arrow[u, shift left, "learn_1"] & Sem(\Sigma_2) \arrow[u, shift left, "learn_2"]
\end{tikzcd}$$

Figure 2.3: The algebraic directed view of a compiler

In the T.I.C.S. System the $\mathcal{S} \rightarrow \mathcal{T}$ specification takes the form of a set of parameterised macro expressions, one for each operation in Σ_1. Each macro expression is the target code to be used to implement the source operation. Compilation begins by identifying patterns in the source string which correspond to the generators of $Syn(\Sigma_1)$ and replacing them by their target representations. On the subsequent passes through the source string the compiler attempts to identify source operations whose arguments have already been replaced by their target representations. When the compiler recognises such a source operation it uses the embedded target representations to parameterise the associated macro operation and replaces the source operation by the result of the macro expansion. Compilation is complete when there are no source operations left to translate.

In [Reev87] Reeves attempts to show the relationship between this approach and the semantics directed approach by using a tree of partially expanded macro expressions as the intermediate language of a semantics directed compiler specification system.

The algebraic basis of the T.I.C.S. System makes the specification of T.I.C.S. generated compilers particularly amenable to the usual methods for proving the correctness of the $\mathcal{S} \rightarrow \mathcal{T}$ relationship.

2.1.4 Compiler generation by partial evaluation

Partial evaluation [Futa71, Ersh77] or mixed computation can be described informally as the process of "doing as much evaluation as possible with, possibly, incomplete input." If p is a program whose input can be divided into two classes: S - static i.e. input which is fixed at a particular value, and D - dynamic i.e. input which is not fixed and may vary over all possible values of the correct type, the program p can be evaluated fully only if it is given both S and a particular value of D.

If only S is available the process of *partial evaluation* can be applied to p. The part of the computation of p which depends only on S is performed. The result of this process is a new specialised version of p whose input is the dynamic part of the input of p, D, and which, when it is applied to D, produces the same output as p applied to both S and D.

$$p(S,D) = p_S(D)$$

This specialised function p_S is known as a *residual program*. For example consider the function *power*

$$\begin{aligned} \mathit{power}\ x\ n &= 1, && \textbf{if } n = 0 \\ &= x * \mathit{power}\ x\ (n-1), && \textbf{otherwise} \end{aligned}$$

This function raises x to the power of n. Suppose the value of n is fixed at 3 but the value of x is dynamic. Specialisation of *power* to its static input $n = 3$ produces the residual program

$$\mathit{power}'\ x \quad = \quad x * x * x * 1$$

because all computation except multiplication by x, which is dynamic, can be performed at partial evaluation time.

Beckmann et al [BHOS76] and Futamura [Futa82] describe some of the potential applications of partial evaluation. These include:

- Automatic theorem proving. It is possible to use a partial evaluator to produce a specific theorem prover by specialising a general theorem prover to a specific set of axioms [Futa82].
- Construction of small specialised utility programs from more general routines [BHOS76].
- Construction of specific parsers from general parsing routines [Futa82]. If there is a general parsing algorithm $P : BNF\_grammar \times text \rightarrow parse\_tree$, and S is the BNF

grammar of the language $\mathcal{S}$. A specific $\mathcal{S}$ parser $P_{\mathcal{S}}$ can be produced by specialising P to $\mathcal{S}$ by partial evaluation.

- Compilation and compiler generation. This use of partial evaluation is discussed in detail below.

In general partial evaluation is a useful technique where there is a need to construct a fast, specific, algorithm to do a particular job and a slower more general, data driven, algorithm already exists.

Partial evaluation and compilation

The use of partial evaluation as a compiler construction technique is described in [Futa71, Ersh77, Ersh82, Futa82. JoSS85, JoSS89]. A brief overview of the technique is shown below.

Suppose that *mix* is a self-applicable partial evaluator for the target language $\mathcal{T}$, i.e. *mix* is a $\mathcal{T}$ program which implements a partial evaluator for the language $\mathcal{T}$. Because *mix* is a partial evaluator the following equation holds for all $\mathcal{T}$ programs t with static input i_{static} and dynamic input $i_{dynamic}$

$$t[\![i_{static}, i_{dynamic}]\!] = (mix[\![t]\!][\![i_{static}]\!])[\![i_{dynamic}]\!]$$

where $mix[\![t]\!][\![i_{static}]\!]$ is the residual program produced from t and i_{static}.

Now assume *int* is an interpreter for the programming language $\mathcal{S}$ and is written in the target language $\mathcal{L}$. If s is an $\mathcal{S}$ program which takes i as its input and produces o as its output then

$$s[\![i]\!] = o$$

represents running the program s on an $\mathcal{S}$ machine with input i. The same output can be produced using a $\mathcal{T}$ machine by running *int* and giving it s and i as its input.

$$int[\![s.i]\!] = o$$

Since *int* is a $\mathcal{T}$ program and *s* is some of its input we can use *mix* to produce a specialised version of *int* which can only interpret the program *s* by setting *s* as *static* input for *int* and *i* as *dynamic* input.

$$int_s = mix[\![int]\!][\![s]\!]$$

Now using int_s and a $\mathcal{T}$ machine we have

$$int_s[\![i]\!] = o$$

furthermore all the computation in *int* which applies only to the analysis of the program *s* is done at partial evaluation time and does not have to be done when int_s is executed. The $\mathcal{T}$ program int_s has the same input/output relation as the $\mathcal{S}$ program *s* and is, in fact, a compiled version of *s*.

Since $mix[\![int]\!][\![s]\!]$ is a compiled version of *s* and *mix* is a $\mathcal{T}$ program, an $\mathcal{S}$ to $\mathcal{T}$ compiler can be produced using *mix* and *int* by regarding *int* as static input for *mix* and leaving *s* as dynamic input.

$$\mathbf{comp} = mix[\![mix]\!][\![int]\!]$$

This is easy to verify because:

$$\begin{aligned}\mathbf{comp}[\![s]\!] &= (mix[\![mix]\!][\![int]\!])[\![s]\!] \\ &= mix[\![int]\!][\![s]\!] \\ &= int_s\end{aligned}$$

If we remember that the basic function of a partial evaluator is to eliminate redundancy from a partially bound $\mathcal{T}$ program it is easy to understand *how* these results arise. By definition an $\mathcal{S}$ interpreter, *int*, must contain the $\mathcal{T}$ expressions necessary to execute *any* $\mathcal{S}$ program with *any* input data in addition to the $\mathcal{T}$ expressions necessary to parse an $\mathcal{S}$ program and execute its static semantics. If the program argument of an interpreter is bound to a particular $\mathcal{S}$ program, *q*, it is possible to execute the parsing and static semantics components of the interpreter as they only depend on the $\mathcal{S}$ program text. The code segments of the interpreter which actually simulate the run time behaviour of q depend on the input data for q as well as the program text and therefore become part of the residual program int_q. This reasoning extends to the construction of mix_{int} in the obvious manner.

Partial evaluation systems

There is a large, and growing, body of literature on the subject of partial evaluation as a compiler generation technique.

In [MaBe85] partial evaluation is used to derive a compiler and an object interpreter from an operational semantics given using the V.D.L. specification language. However the approach used needs to place several restrictions on the style of V.D.L. specification and does not appear to generalise to the automatic generation of compiler generators in any obvious manner.

The first working version of a fully self-applicable *mix* was produced by Jones et al [JoSS85, Sest85, JoSS87, JoSS89]. This project identified the process of *binding time analysis* as critical to the effective operation of a partial evaluator.

To specialise the *power* function given above to some fixed value of n the partial evaluator need only unfold recursive calls of *power* until the value of n falls to 0. Now consider the specialisation of *power* to some fixed value of x (say 5) rather than n. The *obvious* specialisation is

$$\begin{array}{rcll} power''\ n & = & 1, & \textbf{if } n = 0 \\ & = & 5 * power''(n-1), & \textbf{otherwise} \end{array}$$

but this specialisation cannot be produced by repeated unfolding of recursive calls because the value of n is dynamic and therefore never falls to 0. Specialisation by repeated unfolding will actually cause non-termination of the partial evaluator as it attempts to produce the infinite residual program shown below.

$$
\begin{aligned}
\mathit{power}''' \; n &= 1, && \textbf{if } n = 0 \\
&= 5 * 1, && \textbf{if } n - 1 = 0 \\
&= 5 * 5 * 1, && \textbf{if } n - 1 - 1 = 0 \\
&\vdots \\
&= 5 * \cdots * 5 * 1, && \textbf{otherwise}
\end{aligned}
$$

The problem arises because the expression *power* 5 n where n is dynamic is itself dynamic and *must* not be unfolded at partial evaluation time. To overcome this problem a partial evaluation system must perform a process of binding time analysis on the program to be specialised to determine which sub-expressions in its body are static (reducible) and which are dynamic (irreducible) at partial evaluation time.

In [JoSS85] the binding time analysis is done "by hand", but in later versions the process is automated, all be it in a fairly ad hoc manner.

The treatment of binding time analysis given by Launchbury [Laun88, Laun89, Laun90] using a domain theoretic construction of dependent sums which allows aspects of binding time analysis to be expressed as domain projections is particularly interesting but for reasons of space cannot be discussed here.

Another interesting example of partial evaluation is the work of Turchin et al [Turc80, TuNT82, Turc85, Turc86] on the *supercompiler* concept. A supercompiler is a generalised from of partial evaluator. Supercompilation consists of a process called *driving* in which a $\mathcal{T}$ program, p, is run in a generalised form (with unknown values for some of the variables of p) to produce a graph of states and state transitions of the possible configurations of the computing system specified by p. To keep this driving finite the supercompiler examines the configurations of p and generalises them until a set of generalised configurations are produced which are capable of describing the whole of the computing system, p. This generalisation process replaces the binding time analysis of the more traditional partial evaluation system. Because the driving and generalisation process has access to more information than a simple partial evaluator a supercompiler can carry out transformations to the program p which are not possible using partial evaluation alone. On the other hand self-application is much harder to achieve because of the increased complexity of a supercompiler.

Other work on partial evaluation includes: compilation of pattern matching [Bond88. Jorg90] by partial evaluation. The extension of partial evaluation to lazy functional languages [Bond90b] and partial evaluation of higher order languages [Goma89, Bond90a, Cons90].

Compiler generation by partial evaluation is included as an example of a compiler specification system where the $\mathcal{S} \rightarrow \mathcal{T}$ relationship is specified as an $\mathcal{S}$ interpreter expressed in the language $\mathcal{T}$. To be a fully formal compiler specification technique we require formal specifications of the $\mathcal{S}$ and $\mathcal{T}$ languages and a formal description of the process of partial evaluation, in order to prove the correctness of *mix*. In reality the technique is much more powerful than simple compiler generation, it is probably better described as a program transformation technique but in its guise as a compiler specification system it provides the inspiration for the fictional true compiler generator described below.

2.2 A true compiler generation system

By making two, rather large, assumptions we can now take a look inside the "compiler generator" box in figure 2.2 and speculate about its internal workings, based on a partial evaluator.

Assumption 1. there is a technique which allows us to examine the specification of a computer language, $\mathcal{T}$, and from this specification, *calculate* a $\mathcal{T}$ program which implements *mix* for the language $\mathcal{T}$.

Assumption 2. given the specifications of two languages, $\mathcal{S}$, and $\mathcal{T}$, it is possible to derive an implementation of $\mathcal{S}$ in the form of an interpreter expressed as a $\mathcal{T}$ program.

By allowing assumption 1 only, we could implement a compiler specification language in the following way:

1. examine the target language specification, $\mathcal{T}$, and compute the implementation of *mix* for this language.

2. Accept a source interpreter, *int*, written as a target program and compute the value $mix[\![mix, int]\!]$.

3. Output the value $mix[\![mix, int]\!]$ as the generated compiler.

This system is still a compiler specification system rather than a compiler generation system because the user has to supply the $\mathcal{S} \rightarrow \mathcal{T}$ relationship in the form of the source interpreter, *int*.

If we also allow assumption 2 we can implement a true compiler generator system by calculating the source interpreter from the specifications of $\mathcal{S}$ and $\mathcal{T}$, rather than accepting it as input.

The proposed overall structure of the "compiler generator" box in figure 2.2 is shown in figure 2.4.

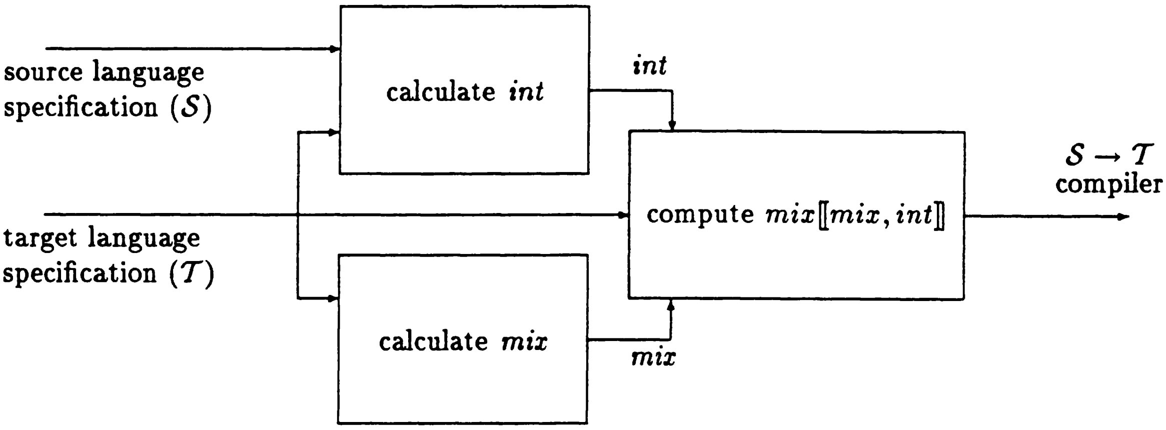

source language specification ($\mathcal{S}$)
target language specification ($\mathcal{T}$)
calculate *int*
calculate *mix*
int
mix
compute $mix[\![mix, int]\!]$
$\mathcal{S} \rightarrow \mathcal{T}$ compiler

Figure 2.4: The anatomy of a true compiler generator

The boxes labelled "calculate *mix*" and "calculate *int*" are implementations of assumptions 1 and 2 respectively. The last box, labelled "compute $mix[\![mix, int]\!]$" is a parameterised simulator which accepts the following inputs:

1. a language specification $\mathcal{T}$.
2. A $\mathcal{T}$ program which implements *mix* for the language $\mathcal{T}$.
3. A $\mathcal{T}$ program, *int*, which is a programming language interpreter.

When given the specification of the language $\mathcal{T}$ the "compute $mix[\![mix, int]\!]$" component becomes a $\mathcal{T}$ interpreter and computes the value $mix[\![mix, int]\!]$ which it constructs from its remaining inputs. This last component is relatively trivial to construct as it is basically an interpreter for the meta-language used to specify $\mathcal{T}$.

2.3 Fiction to fact: the requirements

How unreasonable are the assumptions in section 2.2? The short answer to this question is currently *very* unreasonable. Taking each assumption in turn, for assumption 1 to be reasonable we need to be able to:

1. construct a representation of the syntax of an arbitrary language $\mathcal{T}$ as a data type of the language $\mathcal{T}$. This is required because *mix* must have some way of representing the $\mathcal{T}$ programs it processes.
2. Construct a binding time analysis phase from the specification of the semantics of an arbitrary language $\mathcal{T}$.
3. Construct the function specialisation phase for an arbitrary language $\mathcal{T}$. This is probably the easiest of the three requirements necessary to justify assumption 1. The function specialisation phase of a $\mathcal{T}$ partial evaluator is very closely related to the evaluation function of the programming language $\mathcal{T}$ and is basically a $\mathcal{T}$ program which reduces other $\mathcal{T}$ programs to their canonical form with respect to the static input and binding time analysis.

The requirements necessary to justify assumption 2 are:

1. given arbitrary programming languages $\mathcal{S}$ and $\mathcal{T}$ we must be able to construct a $\mathcal{T}$ data type which represents the syntax of $\mathcal{S}$. Here again this is required because the interpreter, *int*, must be able to represent any $\mathcal{S}$ program to process it.
2. The ability to derive an $\mathcal{S}$ interpreter, *int*, as a $\mathcal{T}$ program. To derive an $\mathcal{S}$ interpreter as a $\mathcal{T}$ program we must be able to implement the $\mathcal{S}$ evaluation function as a $\mathcal{T}$ program.

We will concentrate on the development of a technique which allows us to calculate a self-interpreter for the arbitrary language $\mathcal{T}$, i.e. an interpreter for the language $\mathcal{T}$ which is itself a $\mathcal{T}$ program. The reason for this is that the calculation of a self-interpreter is a reasonable step on the road to both the calculation of an $\mathcal{S}$ interpreter as a $\mathcal{T}$ program. and toward the calculation of a function specialisation phase for the language $\mathcal{T}$.

The problem of constructing a $\mathcal{T}$ data structure to represent the $\mathcal{S}$ programs can be reduced to the problem of implementing binary trees in $\mathcal{T}$, since any tree structure can be transformed into a binary tree and any $\mathcal{S}$ program can be represented as its derivation tree. A less efficient but more straightforward representation technique could be constructed by implementing n-ary trees in $\mathcal{T}$, where n is the largest number of subtrees possible for a node in the derivation tree of an $\mathcal{S}$ program. From this point on we will assume that one or other of these techniques is used to construct a representation of $\mathcal{S}$ programs as a data type in the arbitrary language $\mathcal{T}$. This will be required for the construction of a self-interpreter, (both $\mathcal{S}$ and $\mathcal{T}$ are the same language for a self-interpreter).

Chapter 3

An Algebraic Approach to a Self-Interpreter

The algebraic model of language developed by Rus [HaRu76, Rus76, RuHe84, Rus85, Rus87, Rus90, Rus92] can be used to specify a programming language as a triple

$$\mathcal{L} = \langle Sem, Syn, learn : Sem \rightarrow Syn \rangle$$

where *Sem* and *Syn* are algebraic structures over a common signature and *learn* is function which associates an expression in *Syn* with each meaning in *Sem*. There is an associated homomorphism $eval : Syn \rightarrow Sem$ which defines the evaluation of expressions in *Syn*. The model is described in section 3.1 and in section 3.2 we show how the properties of the model can be used to construct the function computed by a self-interpreter.

3.1 An algebraic model of language

Rus describes an algebraic model of language based on two properties of many sorted algebra. Given the category of Σ algebras $\mathcal{C}(\Sigma)$:

1. the word algebra $\mathcal{W}$ is unique up to isomorphism and coincides with the initial algebra in $\mathcal{C}(\Sigma)$.

2. Any function defined on the generators of $\mathcal{W}$ returning values in the carrier of a similar algebra $\mathcal{A}$ extends to an unique homomorphism $\mathcal{E} : \mathcal{W} \rightarrow \mathcal{A}$.

The construction of the model is shown below.

3.1.1 The specification basis

The three components of the specification basis are:

1. a set of names of the abstract objects specified in the language, denoted by I.
2. A finite set of reserved words denoted by S.
3. A finite set of operation schemes Σ. The operation schemes, $\sigma \in \Sigma$ specify operations on families of sets (indexed by I) and are denoted by a triple.

$$\sigma = \langle n, s_0 s_1 \ldots s_n, i_1 \ldots i_n i \rangle$$

   The components of the triple are:

   - $n \geq 0$, the arity of the operation.
   - The operation symbol $s_0 s_1 \ldots s_n$, $s_j \in S$.
   - The operand sorts $i_1 \ldots i_n$, $i_j \in I$ and result sort of the operation $i \in I$.

A proof that every context free grammar generates a basis B and every finite basis B generates a context free grammar is given in [Rus87].

3.1.2 The semantics algebra

The semantics of a programming language is given as an algebra specified by some basis B over a family of sets $A = \{A_1, A_2, \ldots\}$. The family A represents the collection of abstract objects which are denotable within the language semantics. The algebra $Sem(B, A) = \langle Sem(I), Sem(S), Sem(\Sigma) \rangle$ is constructed as follows:

1. $Sem(i_k) = A_k.i_k \in I$. $Sem(I)$ is then a family of sets chosen from A and indexed by I, allowing $Sem(B,A)$ to be constructed as a many sorted algebra.

2. $Sem(S) = S$. The purpose of this set is to fix the symbols used to express constructions over $Sem(I)$.

3. The set of operations on $Sem(I)$ is denoted by $Sem(\Sigma)$ and $\forall \sigma \in \Sigma, \sigma = \langle n,\ s_0 s_1 \ldots s_n,\ i_1 \ldots i_n i\rangle$, $Sem(\sigma)$ is an operation

$$Sem(\sigma) : Sem(i_1) \times \ldots \times Sem(i_n) \rightarrow Sem(i)$$

The tuple $\langle s_0, s_1, \ldots, s_n\rangle$ is used as the operation symbol and for $a_k \in Sem(i_k)$, $k = 1, \ldots, n$ $Sem(\sigma)$ applied to appropriate a_k is denoted $s_0 a_1 s_1 a_2 \ldots s_{n-1} a_n s_n$ and is of sort i.

This construction of $Sem(B,A)$ as a many sorted algebra with operation symbols which distribute over their operands provides a very natural association between the semantics algebra and the phrases of a context free grammar.

3.1.3 The syntax algebra

The set $W(X,\Sigma) = \{W_i(X,\Sigma), i \in I\}$ is the family of well formed expressions freely generated from the family of finite symbol sets $X = \{X_i, i \in I\}$ by the signature Σ. Details of this construction are given in [Higg63, Rus90]. The algebra $Syn(B,W) = \langle Syn(I), Syn(S), Syn(\Sigma)\rangle$ is constructed as follows:

1. $Syn(I) = \{W_i(X,\Sigma), i \in I\}$.

2. $Syn(S) = S$.

3. The set of operations on $Syn(I)$ is denoted by $Syn(\Sigma)$ and $\forall \sigma \in \Sigma, \sigma = \langle n,\ s_0 s_1 \ldots s_n,\ i_1 \ldots i_n i\rangle$, $Syn(\sigma)$ is an operation

$$Syn(\sigma) = Syn(i_1) \times \ldots \times Syn(i_n) \rightarrow Syn(i)$$

defined by the rules for well formed expressions in $W_i(X,\Sigma)$ as

$$\forall w_j \in W_{i_j}(X,\Sigma), j = 1,\dots n, Syn(\sigma)(w_1,\dots,w_n) = s_0 w_1 s_1 \dots s_{n-1} w_n s_n \in W_i(X,\Sigma).$$

Note that for any context free grammar G, the language generated by G is the set of words $W(\emptyset, B(G))$ where $B(G)$ is the basis generated by G [HaRu76].

3.1.4 The *learn* and *eval* functions

Given a basis $B = \langle I, S, \Sigma \rangle$, a family of abstract objects $A = \{A_1,\dots,A_n\}$, and a family of symbol sets $X = \{X_i, i \in I\}$. *Syn* defines an algebra of words on $W(X,\Sigma)$ and *Sem* defines a similar algebra on A.

$$\mathcal{A} = \langle \{A_i, i \in I\}, \Sigma, Sem(\Sigma) \rangle$$
$$\mathcal{W} = \langle \{W_i(X,\Sigma), i \in I\}, \Sigma, Syn(\Sigma) \}$$

The triple $\mathcal{L} = \langle Sem(B,A), Syn(B, W(X,\Sigma)), learn : Sem(B,A) \rightarrow Syn(B, W(X,\Sigma)) \rangle$ specifies a programming language with semantics $Sem(B,A)$, and syntax $Syn(B, W(X,\Sigma))$.

The purpose of the learning function is to specify the process of sentence construction carried out by a sender communicator using the language $\mathcal{L}$. The other communication process, *understanding*, is modelled by the *eval* : $Syn(B, W(X,\Sigma)) \rightarrow Sem(B,A)$ homomorphism given by property 2 above. A construction for the *eval* homomorphism is given in [Rus92]. For the sake of clarity we shall give a simpler, and less general, construction here, by assuming *learn* to be injective[1].

1. Let $Syn_0 = \{Syn_{i_0}, i_0 \in I\}$ be the indexed family of free generators of the algebra $Syn(B, W(X,\Sigma))$. For each $w \in Syn_{i_0}$, $\sigma = \langle 0, w, i_0 \rangle \in \Sigma$ is an operation scheme and $a \in A_{i_0}$ a unique value with $learn(a) = w$. Define $eval_0 : Syn_0 \rightarrow Sem(B,A)$ as $eval_0(w) = a$. For any $\sigma' = \langle 0, w', i \rangle \neq \sigma$ such that $Sem(\sigma') = a$, set $eval_0(w') = a$.
2. Extend $eval_0$ homomorphically to $eval$: $Syn(B, W(X,\Sigma)) \rightarrow Sem(B,A)$.

When the algebras $Syn(B, W(X,\Sigma))$ and $Sem(B,A)$ are finitely generated, i.e. when X and A are finite collections, *learn* and *eval* are constructed such that $eval \circ learn = id_{Sem(B,A)}$,

[1] This assumption is not unreasonable as we would not expect more than one meaning to be expressed by any single programming language sentence.

where $\circ$ denotes function composition. In the case of a conventional programming language both A and X are finite.

3.1.5 Example: a language of numbers and addition

The algebraic model of language described above provides a formal definition of the three components of a programming language (*Syntax*, *Semantics*, and the *Syntax* $\leftrightarrow$ *Semantics* association) within the single framework of universal algebra. This section illustrates the model using a simple expression language of natural numbers with an addition operator. Expressions in the language are generated according to the BNF grammar.

$$\begin{aligned}
\langle Exp\rangle &\rightarrow \texttt{0}\\
\langle Exp\rangle &\rightarrow \texttt{succ(}\langle Exp\rangle\texttt{)}\\
\langle Exp\rangle &\rightarrow \langle Exp\rangle\texttt{+}\langle Exp\rangle
\end{aligned}$$

The semantics of this language are the expected semantics for natural numbers and addition: 0 is the syntactic expression denoting the number 0, succ denotes the function $\lambda x.x + 1$, and the symbol + is the addition operator.

Specification basis

To specify this language algebraically we must first define the basis B. The language contains only one abstract object, namely Exp, so the set $I = \{Exp\}$. There are four reserved words: '0', 'succ(', ')', and '+', together these reserved words form the set S. Each of the three BNF rules adds the operation scheme shown below to the set Σ.

BNF rule	operation scheme
$\langle Exp\rangle \rightarrow$ 0	$\langle 0,\text{'0'}, Exp\rangle$
$\langle Exp\rangle \rightarrow$ succ($\langle Exp\rangle$)	$\langle 1,\text{'succ(")'}, ExpExp\rangle$
$\langle Exp\rangle \rightarrow \langle Exp\rangle$+$\langle Exp\rangle$	$\langle 2, \epsilon\text{'+'}\epsilon, ExpExpExp\rangle$

Note that the operation symbol for the third operation scheme contains two occurrences of the empty string ϵ. The specification basis B is the triple:

$$\begin{aligned} B &= \langle I = \{Exp\}, S = \{\texttt{0,succ(,),+}\}, \\ &\quad \Sigma = \{\langle 0,\text{`}\texttt{0}\text{'}, Exp\rangle, \langle 1,\text{`}\texttt{succ("\,)}\text{'}, ExpExp\rangle, \langle 2, \epsilon\text{`}\texttt{+}\text{'}\epsilon, ExpExpExp\rangle\rangle \end{aligned}$$

Semantics algebra

To specify the semantics algebra $Sem(B, A)$ we must first define the family of abstract objects A. The only object required for the semantics is the set of natural numbers, *Nat*, constructed by the signature below:

$$\begin{aligned} zero &: \ \rightarrow Nat \\ succ &: \ Nat \rightarrow Nat \end{aligned}$$

the family A is therefore $A = \{Nat\}$. The construction of $Sem(I)$ is $Sem(I) = \{Sem(Exp)\} = \{A_{Exp}\} = \{Nat\}$, and $Sem(S)$ is constructed as $Sem(S) = \{\texttt{0,succ(,),+}\}$. We can now construct $Sem(\Sigma)$. The set of operations of the algebra $Sem(B, A)$, i.e. $\{Sem(\sigma), \sigma \in \Sigma\}$, is constructed by the assignment shown in the table below.

	operation signature	operation
$Sem(\langle 0,\text{`}\texttt{0}\text{'}, Exp\rangle)$	$\rightarrow Sem(Exp)$	$zero$
$Sem(\langle 1,\text{`}\texttt{succ("\,)}\text{'} ExpExp\rangle)$	$Sem(Exp) \rightarrow Sem(Exp)$	$succ$
$Sem(\langle 2, \epsilon\text{`}\texttt{+}\text{'}\epsilon. ExpExpExp\rangle)$	$Sem(Exp) \times Sem(Exp) \rightarrow Sem(Exp)$	f

where the operation $f : Nat \times Nat \rightarrow Nat$ is defined as addition on natural numbers.

$$\begin{aligned} f(zero, x) &= x \\ f(succ(x), y) &= succ(f(x, y)) \end{aligned}$$

Using this assignment the set $Sem(\Sigma)$ is defined as $Sem(\Sigma) = \{zero, succ, f\}$, this completes the definition of $Sem(B, A)$.

Syntax algebra

Since the expression language is generated by a context free grammar, the family $W(X, \Sigma)$ for the syntax algebra is freely generated from the family of symbol sets $X = \{\emptyset\}$ by the signature Σ. Following the rules for the construction of $W(X, \Sigma)$ in [Rus90] we obtain an unchanged set Σ (X is the family of empty sets). The set $W_{Exp}(X, \Sigma)$ is described below.

$$\begin{aligned}
W_{Exp_0}(X, \Sigma) &= \{\mathtt{0}\} \\
W_{Exp_n}(X, \Sigma) &= W_{Exp_{n-1}}(X, \Sigma) \cup \{\ \mathtt{succ}(w) : w \in W_{Exp_{n-1}}(X, \Sigma)\} \\
&\quad \cup \{\epsilon w_1 \ \mathtt{+}\ w_2 \epsilon : (w_1, w_2) \in W_{Exp_{n-1}}(X, \Sigma) \times W_{Exp_{n-1}}(X, \Sigma)\}
\end{aligned}$$

$$W_{Exp}(X, \Sigma) = \bigcup W_{Exp_n}(X, \Sigma), n \in \{0, 1, 2, \ldots\}$$

So $W(X, \Sigma) = \{W_{Exp}(X, \Sigma)\}$. The family $Syn(I)$ is defined as $\{W_{Exp}(X, \Sigma)\}$ and the set $Syn(S)$ is {0,succ(,),+}. The elements of the set $Syn(\Sigma)$ are described in the table below.

	operation signature	operation
$Syn(\langle 0, \text{‘}\mathtt{0}\text{’}, Exp\rangle)$	$\rightarrow Syn(Exp)$	0
$Syn(\langle 1, \text{‘}\mathtt{succ(“)}\text{’}, ExpExp\rangle)$	$Syn(Exp) \rightarrow Syn(Exp)$	g
$Syn(\langle 2, \epsilon\text{‘}\mathtt{+}\text{`}\epsilon, ExpExpExp\rangle)$	$Syn(Exp) \times Syn(Exp) \rightarrow Syn(Exp)$	h

The operations $g : W_{Exp}(X, \Sigma) \rightarrow W_{Exp}(X, \Sigma)$ and $h : W_{Exp}(X, \Sigma) \times W_{Exp}(X, \Sigma) \rightarrow W_{Exp}(X, \Sigma)$ are defined as:

$$\begin{aligned}
g(w) &= \mathtt{succ}(w) \\
h(w_1, w_2) &= \epsilon w_1 \mathtt{+} w_2 \epsilon
\end{aligned}$$

$Syn(\Sigma)$ is therefore defined as $Syn(\Sigma) = \{\mathtt{0}, g, h\}$ and the definition of the algebra of words $Syn(B, W(X, \Sigma))$ is complete.

The *learn* and *eval* functions

The function $learn_0 : Sem_0(B, A) \rightarrow Syn_0(B, W(X, \Sigma))$ is defined as: $learn_0(zero) = \mathbf{zero}$. This function can be extended through the signature Σ as follows:

$$\begin{aligned}
learn(zero) &= \mathbf{zero} \\
learn(succ(a)) &= \mathtt{succ}(learn(a)) \\
learn(f(a_1, a_2)) &= \epsilon learn(a_1) \ \mathtt{+}\ learn(a_2) \epsilon.
\end{aligned}$$

Since the value $f(a_1, a_2)$ is constructed by the operations *zero* and *succ* for all values $a_1, a_2 \in Sem(A)$ this definition simplifies to:

$$\begin{aligned} learn(zero) &= \texttt{zero} \\ learn(succ(a)) &= \texttt{succ}(learn(a)). \end{aligned}$$

We can now define the *eval* homomorphism as follows:

1. define $eval_0 : Syn_0(B, W(X, \Sigma)) \rightarrow Sem_0(B, A)$ as: $eval_0(\texttt{zero}) = zero.$

2. Extend $eval_0$ homomorphically to *eval.*

$$\begin{aligned} eval(\texttt{zero}) &= zero \\ eval(\texttt{succ}(w)) &= succ(eval(w)) \\ eval(\epsilon w_1 + w_2 \epsilon) &= f(eval(w_1), eval(w_2)) \end{aligned}$$

3.2 The interpreter function

The conventional definition of an $\mathcal{L}$ interpreter is a program which, when given an $\mathcal{L}$ program l and input i for the $\mathcal{L}$ program as its input, produces the same output as l produces when given input i. If the interpreter is itself an $\mathcal{L}$ program it is called an $\mathcal{L}$ self-interpreter. For the purposes of the algebraic model above this definition must be made a little more precise.

Definition: An $\mathcal{L}$ self-interpreter.
An $\mathcal{L}$ self-interpreter is a term *int* such that:

1. $eval(int)$ is a function $interpreter : Q \rightarrow Q$, where $W(X, \Sigma) \subseteq Q$.

2. For every term $w \in W(X, \Sigma)$, $eval(w) = eval(interpreter(w))$ and no further reduction of $interpreter(w)$ is possible. □

In other words an $\mathcal{L}$ self-interpreter is an $\mathcal{L}$ program which takes $\mathcal{L}$ syntactic terms as input and delivers maximally reduced $\mathcal{L}$ syntactic terms as output while preserving the meaning of these terms during the reduction process.

The algebraic model of language outlined above can be used to describe the *interpreter* function.

Proposition 3.2.1 *If $\mathcal{L}$ is a programming language:*

$$\mathcal{L} = \langle Sem(B,A), Syn(B,W(X,\Sigma)), learn : Sem(B,A) \rightarrow Syn(B,W(X,\Sigma))\rangle$$

with evaluation homomorphism:

$$eval : Syn(B,W(X,\Sigma)) \rightarrow Sem(B,A).$$

The interpreter functions for $\mathcal{L}$ is defined as

$$\text{interpreter} = \text{learn} \circ \text{eval}$$

Proof: The *eval* homomorphism defines an equivalence relation on $Syn(B,W(X,\Sigma))$.

$$\forall\, \omega_1, \omega_2 \in Syn(B,W(X,\Sigma)) : \omega_1 \equiv \omega_2 \Leftrightarrow eval(\omega_1) = eval(\omega_2)$$

This relation can be used to construct a quotient algebra $Syn(B,W(X,\Sigma))_{/\equiv}$ where each element of $Syn(B,W(X,\Sigma))_{/\equiv}$ is not an $\mathcal{L}$ program but the complete collection of all $\mathcal{L}$ programs which have a given meaning. For example, in the expression language above, the equivalence class which contains the term ϵ`succ(0) + succ(succ(0))`ϵ will also contain the term `succ(succ(succ(0)))`, and all other terms which evaluate to $succ(succ(succ(zero)))$.

This suggests a mechanism for the computing the *interpreter* function.

1. Identify the equivalence class containing the term to be specialised.

2. Select a pre-determined term from this equivalence class and use it as the result term.

There is an isomorphism between $Syn(B,W(X,\Sigma))_{/\equiv}$ and $Sem(B,A)$ so if a term $w \in Syn(B,W(X,\Sigma))$ can be uniquely identified as the preferred syntactic representation of each element of $Sem(B, A)$ the *interpreter* function can be described using the *eval* homomorphism. The *learn* function from the definition of $\mathcal{L}$ performs exactly this task and so the function computed by a self-interpreter can be described as: $interpreter = learn \circ eval$. □

The effect of proposition 3.2.1 is to define a family of functions F_σ on the syntax algebra which correspond to the operations $Sem(\sigma)$ of the semantics algebra, for each $\sigma \in \Sigma$.

Theorem 3.2.1 *For each operation scheme $\sigma = \langle n, s_0 s_1 \ldots s_n, i_1 \ldots i_n i\rangle \in \Sigma$ the function*

$$F_\sigma : Syn(i_1) \times \ldots \times Syn(i_n) \to Syn(i)$$

defined as

$$F_\sigma = interpreter \circ Syn(\sigma)$$

has the same behaviour on syntactic objects as $Sem(\sigma) : Sem(i_1) \times \ldots \times Sem(i_n) \to Sem(i)$ has on semantic objects.

Proof: F_σ is defined as:

$$\begin{aligned} F_\sigma(w_1, \ldots, w_n) &= (learn \circ eval)(Syn(\sigma)(w_1, \ldots, w_n)) \\ &= learn(eval(Syn(\sigma)(w_1, \ldots, w_n))) \\ &= learn(Sem(\sigma)(eval(w_1), \ldots, eval(w_n))) \end{aligned}$$

for each $\sigma = \langle n, s_0 s_1 \ldots s_n, i_1 \ldots i_n i\rangle \in \Sigma$, $n > 0$. F_σ is defined as:

$$F_\sigma = (learn \circ eval)(w)$$

for each $\sigma = \langle 0, w, i\rangle \in \Sigma$. □

Returning to the example from section 3.1.5 the *interpreter* function can be defined as: $interpreter = learn \circ eval$. Using theorem 3.2.1, this definition can be expanded as:

as indexed collections of arrows whose components can be brought within the semantics of the specified language.

$$\begin{aligned} interpreter(\mathtt{zero}) &= (learn \circ eval)(\mathtt{zero}) \\ &= learn(zero) \\ &= \mathtt{zero} \end{aligned}$$

$$\begin{aligned} interpreter(\mathtt{succ}(w)) &= (learn \circ eval)(\mathtt{succ}(w)) \\ &= learn(succ(eval(w)) \\ &= \mathtt{succ}((learn \circ eval)(w)) \\ &= \mathtt{succ}(interpreter(w)) \end{aligned}$$

$$\begin{aligned} interpreter(\epsilon w_1 + w_2 \epsilon) &= (learn \circ eval)(\epsilon w_1 + w_2 \epsilon) \\ &= learn(f(eval(w_1), eval(w_2))) \\ &= F_{\langle 2, \epsilon + \epsilon, ExpExpExp\rangle}(interpreter(w_1), interpreter(w_2)). \end{aligned}$$

The function $F_{\langle 2, \epsilon + \epsilon, ExpExpExp\rangle} : Syn(Exp) \times Syn(Exp) \rightarrow Syn(Exp)$ given by theorem 3.2.1 is the syntactic equivalent of the semantic operation $f : Nat \times Nat \rightarrow Nat$ and is defined in figure 3.1.

$$F(w_1, w_2) = learn(f(eval(w_1), eval(w_2)))$$

$$\begin{aligned} F(\mathtt{zero}, w) &= w \\ F(\mathtt{succ}(w_1), w_2) &= \mathtt{succ}(F(w_1, w_2)) \end{aligned}$$

Figure 3.1: The operation $F_{\langle 2, \epsilon + \epsilon, ExpExpExp\rangle}$

Although the *interpreter* function is completely described in terms of the definition of the programming language $\mathcal{L}$ it is not a description of a self-interpreter for the simple reason that it is not an $\mathcal{L}$ program. In fact the *interpreter* function is not actually an element of the algebra $Sem(B, A)$ and so there is no guarantee that an $\mathcal{L}$ program to compute the *interpreter* function actually exists. The following chapters describe a categorical model of language based on finite limit sketches [BaWe85]. The categorical model of language exploits the fact that finite limit sketches modelled in the category of sets and functions (**SET**) exceed the expressive power of many sorted algebraic theories and have all the properties used above. Using finite limit sketches we can therefore construct analogues of the *learn* and *eval* functions

Chapter 4

Sketches

The concept of a sketch originates with Ehresmann and is described in [BaEh68]. Sketches have been studied extensively by several groups worldwide, mainly in France and Canada, and a general introduction to the work can be found in [Ehre68, BaEh68, Lair75, GuLa80, CoLa84, BaWe85, Gray87, WeBa87, BaWe90]; this list of references is by no means complete. The formalism used here most closely follows that of Barr and Wells [BaWe85, BaWe90, WeBa87] as these are more widely distributed than the majority of the other references.

4.1 Definitions

Sketches provide a formal specification technique based on graphs and, "as such are the intrinsically categorical way of providing a finite specification of a possibly infinite mathematical object or class of models" [BaWe90] (pp161). The definition used in [WeBa87] is given below.

Definition: Directed Graph.
A directed graph, G, is a pair of sets G_0 — nodes, and G_1 — edges, together with two functions: $src : G_1 \rightarrow G_0$, which returns the source node of a given edge, and function $trg : G_1 \rightarrow G_0$ maps the edges to their target nodes. □

For example:

$$\begin{tikzcd} a \arrow[r, "f"] & b \arrow[r, "g"] & c \arrow[loop right, "h"] \end{tikzcd}$$

with G_0, G_1, *src*, and *trg* defined as:

$$G_0 = \{a, b, c\} \quad G_1 = \{f, g, h\}$$

$$\begin{array}{ll} src(f) = a & trg(f) = b \\ src(g) = b & trg(g) = c \\ src(h) = c & trg(h) = c. \end{array}$$

The definition of a sketch requires the definition of a diagram. To define a diagram we must first define a graph homomorphism.

Definition: Graph Homomorphism.
A graph homomorphism $H : G \rightarrow E$ is defined as a pair of functions $H_i : G_i \rightarrow E_i, i = 0, 1$ such that the following properties hold:

$$\forall e \in G_1 : H_0(src(e)) = src(H_1(e))$$
$$\forall e \in G_1 : H_0(trg(e)) = trg(H_1(e)).$$

That is to say H preserves the connectivity of the graph G. □

A diagram can now be defined.

Definition: Diagram.
If d and G are graphs, a diagram of shape d in G is defined as a graph homomorphism $D : d \rightarrow G$. □

$$d = \begin{tikzcd} w \arrow[r, "g"] \arrow[d, "f"'] & x \arrow[d, "j"] \\ y \arrow[r, "h"'] & z \end{tikzcd} \qquad G = \begin{tikzcd} p \arrow[loop above, "r"] \arrow[d, "s"] \\ q \arrow[loop below, "t"] \end{tikzcd}$$

$$D : d \rightarrow G = (D_0, D_1)$$

where D_0 and D_1 are defined as

$$D_0(w) = D_0(x) - p$$
$$D_0(y) = D_0(z) = q$$

$$D_1(f) = D_1(j) = s$$
$$D_1(g) = r$$
$$D_1(h) = t$$

A directed graph and a set of distinguished diagrams in that graph form two of the components of a sketch. The remaining two components are a set of cones and a set of cocones, defined below.

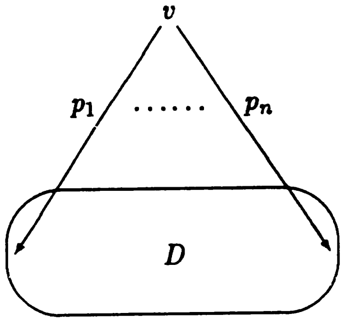
Definition: Cone.

A cone in a graph G consists of:

1. a diagram of shape d in G, $D : d \rightarrow G$. This diagram is called the *base* of the cone.
2. A node v of G, called the *vertex* of the cone.
3. A family of projection edges $p = \{p_i : v \rightarrow D(i)\}$ indexed by the nodes of d.

A cone with vertex v and base D is referred to as a cone from v to D or as cone $p : v \rightarrow D$. □

Any cone $p : v \rightarrow D$ can be indicated by a diagram of the form



In the category C, a cone $p : v \to D$ is a limit cone if it has two additional properties:

1. for every arrow $a : i \to j$ of d, $D(a) \circ p_i = p_j$ where $\circ$ is the composition operator of C. A cone with this property is called a *commutative cone.*
2. If $q : s \to D$ is a different commutative cone there is a unique arrow $u : s \to v$ such that $p_i \circ u = q_i$ for all nodes i of d. In the category of commutative cones over diagram D, the limit cone is the terminal object.

A limit cone over a discrete diagram, in any category, is called a *product cone* and its vertex is known as the *product* of the objects in its base. In **SET**, the category of sets, for example, the vertex of a limit cone over a discrete diagram is the cartesian product of the sets in its base.

A cocone is defined to be the dual of a cone.

Definition: Cocone.
A cocone in a graph G consists of:

1. a diagram of shape d in G, $D : d \to G$. This diagram is called the *base.*
2. A node v of G, called the *vertex.*
3. A family of injections $in = \{in_i : D(i) \to v\}$ indexed by the nodes of d. □

The colimit cocone over diagram D in the category C is defined as the initial object in the category of commutative cocones over diagram D. That is to say, if $j : D \to v$ is the colimit cocone over diagram $D : d \to G$ and $k : D \to s$ is another commutative cocone over D there is a unique arrow $u : v \to s$ such that $k_i = u \circ j_i$.

In any category the colimit cocone over a discrete diagram is the sum (coproduct) of the objects in its base. In **SET**, for example, the vertex of the colimit cocone is the disjoint union of the sets in its base.

These definitions of directed graph, diagram, cone, and cocone are combined to give the definition of a sketch.

Definition: Sketch.

A sketch is a 4-tuple (G, Di, C, Co) consisting of a graph G, a set Di of diagrams on G, a set C of cones on G, and a set Co of cocones on G. $\sqsubset$

Definition: FP Sketch.

A sketch is called an FP (finite product) sketch if it contains no cocones and all cones are over finite discrete diagrams. $\sqsubset$

Definition: FL Sketch.

A sketch is called an FL (finite limit) sketch if it contains no cocones and all cones are over finite diagrams. Clearly every FP sketch is also an FL sketch. $\sqsubset$

Definition: Sketch Morphism.

If $S_1 = (G_1, Di_1, C_1, Co_1)$ and $S_2 = (G_2, Di_2, C_2, Co_2)$ are sketches then a sketch morphism $F : S_1 \rightarrow S_2$ is a graph homomorphism such that:

1. for each diagram $D : d \rightarrow G_1$ in Di_1, $F \circ D : d \rightarrow G_2$ is a diagram in Di_2.
2. For each cone $p : v \rightarrow D$ in C_1, the cone $F(p) : F(v) \rightarrow F \circ D$ belongs to C_2.
3. For each cocone $j : D \rightarrow v$ of Co_1, the cocone $F(j) : F \circ D \rightarrow F(v)$ is a cocone belonging to Co_2. $\sqsubset$

That is to say that $F : S_1 \rightarrow S_2$ takes the diagrams of S_1 to diagrams of S_2, the cones of S_1 to cones of S_2, and cocones of S_1 to cocones of S_2.

Given any category $\mathcal{C}$, there is a sketch underlying $\mathcal{C}$ defined as (G, Di, C, Co) where G is the underlying graph of $\mathcal{C}$, Di is the set of all commutative diagrams of $\mathcal{C}$, C is the set of all limit cones of $\mathcal{C}$, and Co is the set of all colimit cocones. This leads to the final definition in this section.

Definition: Model of a sketch.

A model of a sketch, S, is a sketch morphism $M : S \rightarrow |\mathcal{D}|$ where $|\mathcal{D}|$ is the sketch underlying some category $\mathcal{D}$ (typically **SET**). It follows that the diagrams of S will be taken to commutative diagrams of $\mathcal{D}$, and the cones (cocones) of S will be taken to limit cones (colimit cocones) of $\mathcal{D}$. $\sqsubset$

Although $M : S \rightarrow |\mathcal{D}|$ is actually a graph homomorphism, it is sometimes convenient to regard it as a functor $M : \mathcal{S} \rightarrow \mathcal{D}$ where $\mathcal{S}$ is the free category generated by the sketch S.

The models of a sketch S in category $\mathcal{D}$, $M : S \rightarrow \mathcal{D}$ also form a category denoted $\mathbf{Mod}_{\mathcal{D}}(S)$. The objects of this category are the models M and the arrows are natural transformations. The category $\mathbf{Mod}_{\mathcal{D}}(S)$ is a full reflective subcategory of the functor category $[S, \mathcal{D}]$. The category of models of S in **SET** is denoted by $\mathbf{Mod}(S)$.

4.2 Example: lists

Currently, interest is growing in the use of sketches as a tool for the specification of abstract data types. Sketches offer a specification tool which is far more powerful than any which is currently available. Two reasons for this are:

1. the diagrams of a sketch contain no variables and become commuting diagrams (equations) when the sketch is modelled in any category, $\mathcal{D}$; equational reasoning is therefore greatly simplified for the model of a sketch.
2. The existence of a set of cocones in a sketch allows the user to specify sorts as sums, this can drastically reduce the complexity of a sketch. To quote from Wells and Barr [WeBa87].

> "Having the ability to form disjoint unions makes it easy to define operations ...which are undefined on part of the datatype. We don't need to give it some artificial value such as 'error' — we just don't define it on the embarrassing part of the datatype, and in any model it is then not defined there and thus gives no trouble."

Gray [Gray87] shows how sketches of simple datatypes may be combined to form more complex datatypes such as: **SETofNAT**, and **SETofSETofNAT**, and is currently developing a technique for implementing sketches using the computer algebra package *Mathematica* [Gray?].

A simple example, a sketch of lists of natural numbers with a distinguished error number, is included here to give a flavour of the use of sketches in the specification of abstract data types.

The sketch of the abstract data type **List** has seven operations:

$$\begin{aligned} empty &: \to List \\ cons &: Data \times List \to List \\ head &: List \to Data \\ tail &: List \to List. \end{aligned}$$

which operate on lists and

$$\begin{aligned} zero &: \to Data \\ error &: \to Data \\ succ &: Data \to Data \end{aligned}$$

The operations *empty* and *cons* are constructors, $head : List \to Data$ and $tail : List \to List$ are described by the functions below.

$$\begin{aligned} head(empty) &= error \\ head(cons(d, l)) &= d \end{aligned}$$

$$\begin{aligned} tail(empty) &= empty \\ tail(cons(d, l)) &= l \end{aligned}$$

For the sake of simplicity the *tail* function is defined so that the tail of an empty list is the empty list rather than an error. Defining *tail* in this manner is done to avoid the need include cocones in the sketch.

To force the *Data* sort to contain a unique error element we also require:

$$succ(error) = error.$$

4.2.1 The sketch of lists

The sketch **List** comprises a graph G with four nodes, and nine edges. There are two cones and five diagrams.

Graph - G

The graph of the sketch of lists contains a node for each sort and an edge corresponding to each operation mentioned in the signature above. The nodes of the graph are: *Data*, *List*, $\mathbf{T}$, $Data \times List$ and the edges are $empty : \mathbf{T} \rightarrow List$, $cons : Data \times List \rightarrow List$, $head : List \rightarrow Data$, and $tail : List \rightarrow List$. In addition to the edges above the graph of the sketch also contains edges: $error : \mathbf{T} \rightarrow Data$, $zero : \mathbf{T} \rightarrow Data$, and $succ : Data \rightarrow Data$. The complete graph is represented pictorially below.

$$\begin{tikzcd}
& Data \times List \arrow[dl, shift right, "cons"'] \arrow[dl, shift left, "pr_{List}"] \arrow[dr, "pr_{Data}"] & \\
List \arrow[rr, "head"] \arrow[loop left, "tail"] & & Data \arrow[loop right, "succ"] \\
& \mathbf{T} \arrow[ul, "empty"] \arrow[ur, shift right, "zero"] \arrow[ur, shift left, "error"'] &
\end{tikzcd}$$

The construction of the objects $\mathbf{T}$, and $Data \times List$, and arrows pr_{List} and pr_{Data} is described below.

The set of cones - C

The cones for the sketch **List** are:

$$\mathbf{T}$$

the cone over the empty diagram. For any model, M, in category C, $M(\mathbf{T})$ will be the vertex of the limit cone over the empty diagram, so $M(\mathbf{T})$ must be the terminal object of C. The second cone is used to specify the object $Data \times List$ as a product.

$$\begin{tikzcd}
& \mathit{Data} \times \mathit{List} \arrow[dl, "pr_{Data}"'] \arrow[dr, "pr_{List}"] & \\
\mathit{Data} & & \mathit{List}
\end{tikzcd}$$

Any model, M, in category C, will take this cone to the product cone

$$\begin{tikzcd}
& M(\mathit{Data} \times \mathit{List}) \arrow[dl, "M(pr_{Data})"'] \arrow[dr, "M(pr_{List})"] & \\
M(\mathit{Data}) & & M(\mathit{List})
\end{tikzcd}$$

so the presence of this cone specifies that $M(\mathit{Data} \times \mathit{List}) \cong M(\mathit{Data}) \times M(\mathit{List})$ with the arrows $M(pr_{List})$ and $M(pr_{Data})$ as the coordinate projections. It should be emphasised that the node $\mathit{Data} \times \mathit{List}$ in the graph G is *not* a product, in spite of its name, it is merely a node of the graph.

The set of diagrams - D

The sketch of lists requires five diagrams: two to specify the behaviour of $\mathit{head} : \mathit{List} \to \mathit{Data}$, and two to specify $\mathit{tail} : \mathit{List} \to \mathit{List}$.

$$\begin{tikzcd}
\mathit{List} \arrow[rr, "head"] & & \mathit{Data} \\
& \mathbf{T} \arrow[ul, "empty"] \arrow[ur, "error"'] &
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
& \mathit{Data} \times \mathit{List} \arrow[dl, "cons"'] \arrow[dr, "pr_{Data}"] & \\
\mathit{List} \arrow[rr, "head"'] & & \mathit{Data}
\end{tikzcd}$$

(b)

Together these diagrams specify the behaviour of *head* since any model, M, will force the diagrams (a) and (b) to commute. By (a) we obtain the equation $M(\mathit{head}) \circ M(\mathit{empty}) = M(\mathit{error})$ and (b) gives rise to the equation $M(\mathit{head}) \circ M(\mathit{cons}) = M(pr_{Data})$. The diagrams

$$\begin{tikzcd}
List \arrow[rr, "tail"] & & List \\
& \mathbf{T} \arrow[ul, "empty"] \arrow[ur, "empty"'] &
\end{tikzcd}$$

(c)

$$\begin{tikzcd}
& Data \times List \arrow[dl, "cons"'] \arrow[dr, "pr_{List}"] & \\
List \arrow[rr, "tail"'] & & List
\end{tikzcd}$$

(d)

specify the behaviour of *tail*. Again because any model, M, forces (c) and (d) to commute, we obtain the equations $M(tail) \circ M(empty) = M(empty)$, from (c), and $M(tail) \circ M(cons) = M(pr_{List})$, from diagram (d).

One final diagram is required to specify the behaviour of the *succ* operation:

$$\begin{tikzcd}
Data \arrow[rr, "succ"] & & Data \\
& \mathbf{T} \arrow[ul, "error"] \arrow[ur, "error"'] &
\end{tikzcd}$$

(e)

which gives rise to the equation $M(succ) \circ M(error) = M(error)$. This diagram will be used to force the *Data* sort to contain a unique error value. The sketch contains no other diagrams.

Since the sketch **List** is an FP sketch it contains no cocones and is fully described as the 4-tuple

$$\mathbf{List} = (G, D, C, \emptyset).$$

4.2.2 The semantics of List

A set valued model of an FP sketch S is called a *term model* if it is the initial object in the category $\mathbf{Mod}(S)$. FP sketches always have a term model [Barr86] as do FL sketches. To provide a semantics for the sketch **List** we take its term model. $I : \mathbf{List} \rightarrow \mathbf{SET}$. To do this we must first define a congruence relation.

Definition: Congruence relation.

A congruence relation $\sim$ is an equivalence relation on the arrows of a category C such that:

1. if $f \sim g$, then f and g have the same source and target.

2. In the diagram:

$$\begin{tikzcd} A \arrow[r, "h"] & B \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & C \arrow[r, "k"] & D \end{tikzcd}$$

   if $f \sim g$, then $f \circ h \sim g \circ h$ and $k \circ f \sim k \circ g$.

The congruence class containing the arrow f is denoted $[f]$. □

In [BaWe90] Barr and Wells give a set of rules for the construction of the term model $I : S \rightarrow \mathbf{SET}$ for the FP sketch $S = (G, D, C, \emptyset)$. The terms are constructed as congruence classes of strings of tuples of composable arrows from the graph G and the rules recursively construct terms from an *alphabet* which consists of: the arrows of G, and all finite length tuples of these arrows. For each cone $c \in C$ of the form:

$$\begin{tikzcd} & & q \arrow[dll, "pr_1"'] \arrow[d, "pr_i"] \arrow[drr, "pr_n"] & & \\ a_1 & \cdots & a_i & \cdots & a_n \end{tikzcd}$$

1. If $f : a \rightarrow b$ is an arrow of G and $[x] \in I(a)$, then $[fx] \in I(b)$ and $I(f)[x] = [fx]$.

2. If $(f_1, \ldots, f_m)$ and $(g_1, \ldots, g_n)$ are paths, $a \rightarrow^{+} b$, in a diagram $d \in D$ and $[x] \in I(a)$, then

$$(I(f_1) \circ \ldots \circ I(f_m))[x] = (I(g_1) \circ \ldots \circ I(g_n))[x]$$

   in $I(b)$.

3. If for $i = 1, \ldots, n$, $[x_i] \in I(a_i)$, then $[c(x_1, \ldots, x_n)] \in I(q)$ is a congruence class of strings consisting of the cone, c, followed by a tuple of arrows, so if $n = 0$ there is only one element $[c\langle\rangle]$ for the empty product.

4. If for $i = 1, \ldots, n$, $[x_i], [y_i] \in I(a_i)$ and $[x_i] = [y_i]$, then $c(x_1, \ldots . x_n) = c(y_1, \ldots, y_n)$.

5. For $i = 1, \ldots, n$, $[p_i c(x_1, \ldots, x_n)] = [x_i]$.

By rules 1 and 2 each $I(f) : I(a) \rightarrow I(b)$ is forced to be a function which respects the diagrams D. From rule 3 the vertex $I(q)$ of a cone is forced to contain an element corresponding to each tuple $\langle I(a_1), \ldots, I(a_n)\rangle$. Rule 5 forces $I(p_i)$, $i = 1, \ldots, n$ to be the coordinate projections and from rules 1 and 5 we obtain

$$I(p_i)[c(x_1, \ldots, x_n)] = [x_i], \forall i = 1, \ldots, n.$$

Rule 4 extends the congruence relation to cover tuples.

We can now construct the term model, I : **List** $\rightarrow$ **SET**. The alphabet is constructed as:

$$\begin{aligned} A_1 &= \{empty, zero, error, succ, cons, pr_{list}, pr_{data}, tail, head\} \\ A_n &= A_1^n = \{\langle a_1, \ldots, a_n\rangle : a_i \in A_1, i = 1, \ldots, n\} \\ \\ A &= \bigcup A_n, n \in \{1, 2, \ldots\}. \end{aligned}$$

Together rules 1, 3 and 5 define $I(Data \times List)$ as the set $I(Data) \times I(List)$, and $I(pr_{List})$, $I(pr_{Data})$ are the coordinate projections giving $I(List)$ and $I(Data)$ respectively. The functions $I(pr_{List})$ and $I(pr_{Data})$ cannot construct any elements of $I(List)$ and $I(Data)$ and will be ignored below, except where they form part of a diagram.

By rule 3, $I(\mathbf{T})$ is a singleton set, $I(empty)$ is an element of $I(List)$ which we shall name nil, while $I(zero)$ and $I(error)$ are elements of $I(Data)$ which we name 0 and err respectively. From rule 1, the set $I(Data)$ is inductively defined as:

$$\begin{aligned} I(Data)_0 &= \{0, err\} \\ I(Data)_n &= \{succ(x) : x \in I(Data)_{n-1}\} \cup \{head(x) : x \in I(List)\} \end{aligned}$$

$$I(Data) = \bigcup I(Data)_n, n \in \{0.1, 2, \ldots\}.$$

Notice that from rule 2 and diagrams (a) and (b) we obtain:

$$\begin{aligned} I(head) \circ I(empty) &= I(error) \text{ and} \\ I(head) \circ I(cons) &= I(pr_{Data}) \end{aligned}$$

so the set $\{head(x), x \in I(List)\}$ adds no new elements to $I(Data)_n$ and can be ignored. Similarly, by rule 2 and diagram (e), $I(succ) \circ I(error) = I(error)$ so $I(Data) = \{0, succ(0), succ(succ(0)), \ldots\} \cup \{err\}$, i.e the set of natural numbers with a distinguished error element.

By rule 1, the set $I(List)$ is constructed as:

$$\begin{aligned} I(List)_0 &= \{nil\} \\ I(List)_n &= I(List)_{n-1} \cup \\ &\quad \{cons(x,y) : (x,y) \in I(Data) \times I(List)_{n-1}\} \cup \{tail(x) : x \in I(List)_{n-1}\} \end{aligned}$$

$$I(List) = \bigcup I(List)_n, n \in \{0, 1, 2, \ldots\}.$$

From rule 2 and diagram (c), we obtain $I(tail) \circ I(empty) = I(empty)$, therefore $tail(nil) = nil$. Similarly rule 2 and diagram (d) produce $I(tail) \circ I(cons) = I(pr_{List})$, so the set $\{tail(x), x \in I(List)_{n-1}\}$ adds no new elements to $I(List)_n$. The description of $I(List)$ can therefore be simplified to

$$\begin{aligned} I(List)_0 &= \{nil\} \\ I(List)_n &= I(List)_{n-1} \cup \{cons(x,y) : (x,y) \in I(Data) \times I(List)_{n-1}\} \end{aligned}$$

$$I(List) = \bigcup I(List)_n, n \in \{0, 1, 2, \ldots\}$$

the set of sequences of elements of $I(Data)$ terminated by the value nil, i.e. Lists of natural numbers with a distinguished error number. The operation $I(head : List \to Data)$ is specified by the equations generated by diagrams (a) and (b) as:

$$\begin{array}{llll}(a) & I(head)(nil) & = & err \\ (b) & I(head)(I(cons)(x,y)) & = & x, \quad \forall\langle x,y\rangle \in I(Data)\times I(List)\end{array}$$

and $I(tail : List \rightarrow List)$ is specified by (c) and (d) as:

$$\begin{array}{llll}(c) & I(tail)(nil) & = & nil \\ (d) & I(tail)(I(cons)(x,y)) & = & y, \quad \forall\langle x,y\rangle \in I(Data)\times I(List).\end{array}$$

In other words, the expected *head* and *tail* operations.

4.3 Sketch morphisms and induced functors

In this section we examine some of the properties of sketches and their models. The construction of the categorical model of language in chapter 5 is based on these properties.

Property 4.3.1 *If $h : S \rightarrow T$ is a sketch morphism it induces a functor between the categories of models of* S *and* T, *$h^* : \mathbf{Mod}(T) \rightarrow \mathbf{Mod}(S)$.*

Proof: $h^* : \mathbf{Mod}(T) - \mathbf{Mod}(S)$ is defined as

$$\begin{array}{rcl}h^*(M) & = & M \circ h \\ h^*(f : M \rightarrow N) & = & fh : h^*(M) \rightarrow h^*(N)\end{array}$$

□

We can use property 4.3.1 to construct models of a sketch, S, which play the role of datatypes with hidden sorts and operations.

Proposition 4.3.1 *Let $S = (G, D, C, \emptyset)$ be an FL sketch, $T = (G', D', C', Co')$ be a sketch, and $h : S \rightarrow T$ be a sketch morphism. For each model, $M : T \rightarrow$ **SET**, of T, the datatype $h^*(M) : S \rightarrow$ **SET** has the same behaviour as M except that each object $n' \in G'_0$ which is not the image in h of some object $n \in G_0$ becomes a hidden sort of $h^*(M)$ and each edge $e' \in G'_1$ which is not the image in h of some edge $e \in G_1$ becomes a hidden operation.*

Proof: If $n \in G_0$ then $h(n) \in G'_0$ and from property 4.3.1 $h^*(M)(n) = M(h(n))$ are the same sort. If $n' \in G'_0$ and there is no $n \in G_0$ such that $h(n) = n'$ then $h^*(M)(n)$ is undefined so the sort $M(n')$ is hidden.

Similarly if $e \in G_1$ then $h(e) \in G'_1$ and from property 4.3.1 $h^*(M)(e) = M(h(e))$ are the same operation. If $e' \in G'_1$ and there is no $e \in G_1$ such that $h(e) = e'$ then $h^*(M)(e)$ is undefined and the operation $M(e')$ is hidden. □

A second property allows us to map the initial model of S to the model $h^*(M)$.

Property 4.3.2 *If S is an FL sketch then for any sketch, T, and sketch morphism $h : S \to T$ we have a unique natural transformation, $e : I_S \dot{\to} h^*(M)$, where $I_S : S \to$ **SET** is the initial model of S and $M : T \to$ **SET** is any model of T.*

Proof: $I_S : S \to$ **SET** is initial in **Mod**(S). □

For FL sketches S and T and sketch morphism $h : S \to T$ the construction in **Mod**(S) is shown in figure 4.1

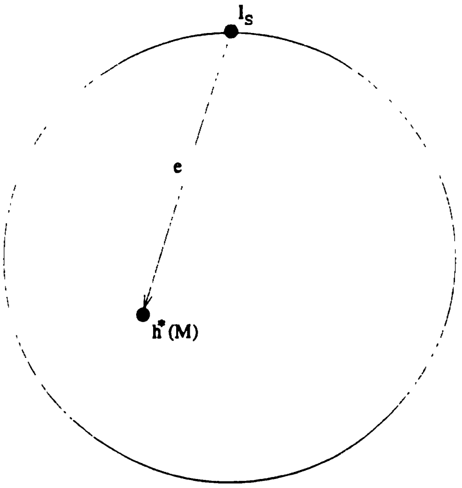


Figure 4.1: The category **Mod**(S)

where $I_S : S \to$ **SET** is the initial model of S, $M : T \to$ **SET** is a model of T, and $h^* :$ **Mod**$(T) \to$ **Mod**(S) is given by $h : S \to T$ and property 4.3.1. The natural transformation $e : I_S \dot{\to} h^*(M)$ is given by property 4.3.2. Our intention is to use **Mod**(S) to construct a model of language where I_S models language syntax, $h^*(M)$ models language semantics and $e : I_S \dot{\to} h^*(M)$ models the evaluation of programs.

To construct this model of language with properties similar to Rus' model of language we need to establish:

1. that I_S can be used to model a language syntax.
2. That $h^*(M)$ can model a language semantics and
3. the conditions under which we can construct an arrow $learn : h^*(M) \rightarrow I_S$.

We will leave 1 and 2 for the next chapter and concentrate here on 3, the construction of *learn*.

To discuss the conditions sufficient to allow the construction of *learn* we must first define what sort of object *learn* is. In order to impose as few conditions on S, T, and $h : S \rightarrow T$ as possible we define *learn* as a transformation [Copp80].

Definition: Transformation.
Let $F : \mathcal{C} \rightarrow \mathcal{D}$ and $G : \mathcal{C} \rightarrow \mathcal{D}$ be functors. A transformation $t : F \rightarrow G$ is defined as any collection of arrows $f_c : F(c) \rightarrow G(c)$ indexed by the objects, c, of $\mathcal{C}$. □

This definition is simply a much weaker form of the definition of natural transformation where the naturality condition has been completely removed. The composition of transformations we require is the horizontal composition given in [Copp80] and is defined below.

Definition: Composition of transformations.
Let $F, G, H : \mathcal{C} \rightarrow \mathcal{D}$ be functors and $s : F \rightarrow G$, $t : G \rightarrow H$ be transformations. The composition of t and s is defined as the transformation $t \circ s : F \rightarrow H$ given by the collection of arrows $t_c \circ s_c : F(c) \rightarrow H(c)$ indexed by the nodes, c, of $\mathcal{C}$. □

If $learn : h^*(M) \rightarrow I_S$ is a transformation then to construct and analogue of Rus' learning function we require that $e \circ learn = 1_{h^*(M)}$. To be able to construct *learn* we require a relationship between S and T which is illustrated in figure 4.2

In the situation where the arrow $g \in T$ is not the image in h of an arrow from S it must be the case that the arrow $M(g)$ adds no new elements to the set $M(h(y))$. Additionally we also require that the set $M(h(y))$ contains no elements which are not constructed by some arrow

$$\begin{tikzcd}
S & x \arrow[r, "f"] & y \\
T & h(x) \arrow[r, shift left, "h(f)"] \arrow[r, shift right, "g"'] & h(y)
\end{tikzcd}$$

Figure 4.2: Generalised arrows in sketches S and T.

M(p) where p is a path in T, i.e. we can ensure this by insisting the $M : T \rightarrow \mathbf{SET}$ is the initial model $I_T : T \rightarrow \mathbf{SET}$ of T.

When the sketches S and T have the relationship described above we will say that S is *learnable* from T. That is to say we can construct the sketch S by deleting parts of T without removing elements from the sets constructed by models of T from objects of T which are common to both S and T. We can now describe the construction of the transformation *learn* $: h^*(I_T) \rightarrow I_S$.

Proposition 4.3.2 *If S and T be an FL sketches and $h : S \rightarrow T$ is a sketch morphism such that S is learnable from T. We can construct a transformation* learn $: h^*(I_T) \rightarrow I_S$ *where $I_S : S \rightarrow$* **SET** *and $I_T : T \rightarrow$* **SET** *are the initial models of S and T respectively.*

The natural transformation $\epsilon : I_S \dot{\rightarrow} h^*(I_T)$ given by property 4.3.1 defines an equivalence $\equiv_s$ on the set $I_S(s)$ for each node $s \in S$.

$$\forall x, y \in I_S(s) : x \equiv_s y \Leftrightarrow e_s(x) = e_s(y)$$

We can therefore construct a quotient set $I_S(s)_{\equiv_s}$ where each element $[x] \in I_S(s)_{\equiv_s}$ is the class of terms $t \in I_S(s)$ which are equivalent under $\equiv_s$. Since S is learnable from T there is an isomorphism between the sets $I_S(s)_{\equiv_s}$ and $h^*(I_T)(s)$ so to construct an arrow $l_s : h^*(I_T)(s) \rightarrow I_S(s)$, set $l_s(x) = y$ for each $x \in h^*(I_T)(s)$ where y is a member of the equivalence class, $[y]$ such that $e_s(y) = x$. The arrows l_s form the transformation *learn* $: h^*(I_T) \rightarrow I_S$. □

Obviously the arrows l_s are not unique as we can choose an arbitrary y from $[y]$ but regardless of the choice of y we know, by construction, that $e_s \circ l_s = 1_{h^\bullet(I_T)(s)}$ and therefore $e \circ \mathit{learn} = 1_{h^\bullet(I_T)}$.

We now have all the components necessary to construct a categorical model of language with similar properties to the algebraic model of language discussed in chapter 3.

Chapter 5

A Categorical Model of Language

The categorical model of language described in this chapter is a development of the model discussed in [ReRa89], and is used to construct a category **Mod**(S), where S is an FL sketch describing the syntax of a programming language. The category, **Mod**(S), is generated by an FL sketch and, as a result, has properties similar to those of the category of Σ-algebras, $\mathcal{C}(\Sigma)$. The category of Σ-algebras is actually equivalent to the models of FP sketches, so by using the FL class of sketches (which includes all FP sketches) we can increase the power of the model of language. The model of language described therefore has similar properties to Rus' algebraic model of language discussed in chapter 3 while having a greater expressive power.

5.1 Using sketches to model language syntax

To model the syntax of a programming language, $\mathcal{L}$, we construct a sketch which describes the abstract syntax trees of $\mathcal{L}$ programs. To define the abstract syntax trees of $\mathcal{L}$ we will assume that the syntax of $\mathcal{L}$ is described by a context free grammar, CFG.

Definition: Context Free Grammar.
A context free grammar, CFG, is defined as a 4-tuple $\langle N, T, P, S\rangle$ where:

1. the set N, called the set of *nonterminal* symbols, is a set of names used to name the types of phrases of the language, $L(CFG)$, described by the context free grammar.

2. The set T, of *terminal* symbols, is the set of symbols which may appear in a sentence of the language, L. The set of strings of terminal symbols is denoted by T^+.

3. P, called the set of *production* rules, is a non-symmetric, non-transitive binary relation, $P : N \rightarrow RHS$, where RHS is the set of strings which can be constructed from the set $N \cup T$.

4. The *start* symbol $S \in N$ is a distinguished nonterminal symbol such that:

   $$\forall s \in T^+ : s \textit{ is a sentence in } L \Leftrightarrow S \rightarrow^+ s$$

   where $\rightarrow^+$ is the transitive closure of P.

The set, $L(CFG) = \{x : x \in T^+ \wedge S \rightarrow^+ x\}$, describes the language generated by the context free grammar, *CFG*. □

Using this definition of context free grammar the abstract syntax trees generated by *CFG* are defined below.

Definition: Abstract syntax tree.
An abstract syntax tree is a labelled, ordered, rooted tree such that:

Abs-1. if t is a string in T^+ and there is a production rule, $p : N \rightarrow t$, then t is an abstract syntax tree describing a phrase belonging to type N.

Abs-2. Let $p : N \rightarrow c_0 \ldots c_x$, $x > 0$, be a production rule such that, $c_i, \ldots, c_k, \ldots, c_j$, $0 \leq i \leq k \leq j \leq x$, is the sequence of nonterminals from the string $c_0 \ldots c_x$. If

t_i is the abstract syntax tree of a phrase of type c_i,
t_k is the abstract syntax tree of a phrase of type c_k, and
t_j is the abstract syntax tree of a phrase of type c_j,

an abstract syntax tree, t, rooted by p is constructed by setting $t_i, \ldots, t_k, \ldots, t_j$ in order as the children of node p. The abstract syntax tree, t, describes a phrase of type N.

Abs-3. Nothing else is an abstract syntax tree. □

Theorem 5.1.1 shows that we can construct an FP sketch, S, which describes the abstract syntax trees of the language generated by the arbitrary context free grammar, *CFG*.

Theorem 5.1.1 *For every context free grammar,* CFG, *there is an FP sketch.* S, *such that each node,* n, *which is not the vertex of a cone, is mapped to the set phrases of type $n \in N$ of $L(CFG)$ by the initial model, $I_S : S \rightarrow$ **SET**, in* $\mathbf{Mod}(S)$.

Proof: The FP sketch, S, describing the abstract syntax trees of $L(CFG)$ is constructed as:

1. set $S = (G, \emptyset, C, \emptyset)$ where G is the graph containing exactly one node, $\mathbf{T}$, and no edges, and C is the set of cones containing just the cone over the empty diagram, $e : \mathbf{T} \rightarrow G$.
2. For each nonterminal symbol, $n \in N$, add a node n to the set of nodes, G_0.
3. For each production rule, $p : n \rightarrow t$, $t \in T^+$ add an edge, $p : \mathbf{T} \rightarrow n$, to the set of edges, G_1.
4. For each production rule, $p : n \rightarrow c_0 \ldots c_x$, where $c_i, \ldots, c_k, \ldots, c_j$, $0 \leq i \leq k \leq j \leq x$ is the sequence of nonterminals from $c_0 \ldots c_x$:
   (a) if $i \neq j$, add a node named, $c_i \times \ldots \times c_k \times \ldots \times c_j$, to G_0.
   (b) If $i \neq j$, add a cone $pr : c_i \times \ldots \times c_k \times \ldots \times c_j \rightarrow D$ over the discrete diagram
   $$c_i \quad \ldots \quad c_k \quad \ldots \quad c_j$$
   to the set of cones, C.
   (c) If $i = j$, add the edge $p : c_i \rightarrow n$ to G_1.
   If $i \neq j$ add the edge, $p : c_i \times \ldots \times c_k \times \ldots \times c_j \rightarrow n$ to G_1.
5. Nothing else belongs to S.

We must now show that each set $I_S(n)$, $n \in G_0$, where n is not the vertex of a cone is the set of phrases of type $n \in N$ of $L(CFG)$.

From **Abs-1** we know that each phrase of type n, t, which has no subtrees is generated by a rule of the form $p : n \rightarrow t$. Rules 1, 2, and 3 above ensure that the sketch, S, has an edge $p : \mathbf{T} \rightarrow n$ corresponding to each production rule $p : n \rightarrow t$. Since S contains no diagrams we know that the arrow $I_S(p) : I_S(\mathbf{T}) \rightarrow I_S(n)$ uniquely identifies a term in $I_S(n)$, corresponding to the phrase, t.

From rule **Abs-2** we know that each phrase, t, with subtrees $t_i, \ldots, t_k, \ldots, t_j$ is constructed by a production rule $p : n \rightarrow c_0 \ldots c_x$ where $c_i, \ldots, c_k, \ldots, c_j$, $0 \leq i \leq k \leq j \leq x$ are nonterminal symbols such that: t_i is a phrase belonging to type c_i, t_k is a phrase belonging to type c_k, and t_j is a phrase belonging to type c_j. Rule 2 ensures that the sketch, S, contains a node, n, while rules 4a, and 4b ensure that $I_S(c_i, \ldots, c_k, \ldots, c_j)$ is the product $I_S(c_i) \times \ldots \times I_S(c_k) \times \ldots \times I_S(c_j)$. By rule 4c we obtain an edge in G, $p : c_i \times \ldots \times c_k \times \ldots \times c_j \rightarrow n$, corresponding to each production rule $p : n \rightarrow c_0 \ldots c_x$. Since, S, contains no diagrams the arrow $I_S(p) : I_S(c_i) \times \ldots \times I_S(c_k) \times \ldots \times I_S(c_j) \rightarrow I_S(n)$ constructs terms such that for each $y = (t_i, \ldots, t_k, \ldots, t_j) \in I_S(c_i) \times \ldots \times I_S(c_k) \times \ldots \times I_S(c_j)$, $I_S(p)(y)$ is uniquely identified as a term in $I_S(n)$ and has subterms, in order, $t_i, \ldots, t_k, \ldots, t_j$. The term $I_S(p)(y)$ corresponds to the phrase, t.

From rule 5 we know that each $I_S(n)$ contains no other terms. □

The simple expression language in chapter 3 with syntax:

$$\begin{aligned} \langle Exp \rangle &\rightarrow \texttt{0} \\ \langle Exp \rangle &\rightarrow \texttt{succ(}\langle Exp \rangle\texttt{)} \\ \langle Exp \rangle &\rightarrow \langle Exp \rangle\texttt{+}\langle Exp \rangle \end{aligned}$$

has the abstract syntax trees shown.

$$\begin{aligned} \textit{exp-trees}_0 &= \{\texttt{0}\} \\ \textit{exp-trees}_n &= \{\texttt{succ(}x\texttt{)} : x \in \textit{exp-trees}_{n-1}\} \cup \\ &\quad \{\texttt{+(}x, y\texttt{)} : (x, y) \in \textit{exp-trees}_{n-1} \times \textit{exp-trees}_{n-1}\} \end{aligned}$$

$$\textit{exp-trees} = \bigcup \textit{exp-trees}_n.\, n \in \{0, 1, \ldots\}$$

The sketch, *Exp*, which describes this set of abstract syntax trees has 3 nodes, **T**, *exp*, and, $exp \times exp$, and 5 edges:

$$\begin{array}{rcl} 0 & : & \mathbf{T} \rightarrow exp \\ succ & : & exp \rightarrow exp \\ pr_1 & : & exp \times exp \rightarrow exp \\ pr_2 & : & exp \times exp \rightarrow exp \\ + & : & exp \times exp \rightarrow exp \end{array}$$

The edge, $0 : \mathbf{T} \rightarrow exp$ arises from the production rule $\langle Exp \rangle \rightarrow$ 0, because of rule 3. The production rules $\langle Exp \rangle \rightarrow$ succ($\langle Exp \rangle$). and $\langle Exp \rangle \rightarrow \langle Exp \rangle$+$\langle Exp \rangle$, together with rule 4 force the existence of the edges $succ : exp \rightarrow exp$ and $+ : exp \times exp \rightarrow exp$ respectively. The edges $pr_1 : exp \times exp \rightarrow exp$ and $pr_2 : exp \times exp \rightarrow exp$ arise solely because of rule 4b, as they are the projection edges of a cone.

There are just 2 cones:

$$\mathbf{T}$$

the cone over the empty diagram, and

$$\begin{tikzcd} & exp \times exp \arrow[dl, "pr_1"'] \arrow[dr, "pr_2"] & \\ exp & & exp \end{tikzcd}$$

which is forced to exist by rules 4a and 4b and, when modelled in **SET**, forces $M(exp \times exp)$ to be the product $M(exp) \times M(exp)$. The sketch is represented pictorially below.

$$\begin{tikzcd} \mathbf{T} \arrow[r, "0"] & exp \arrow[loop above, "succ"] & exp \times exp \arrow[l, "+"'] \arrow[l, bend right, "pr_1"'] \arrow[l, bend left, "pr_2"] \end{tikzcd}$$

The sketch, *Exp* contains only one node which is not the vertex of a cone, namely *exp*. Using rules for constructing the term model, $I_{Exp} : Exp \rightarrow$ **SET**, of *Exp* given in section 4.2.2 we obtain the set, $I_{Exp}(exp)$, shown below.

$$\begin{aligned}
I_{Exp}(exp)_0 &= \{I_{Exp}(0)(\emptyset)\} \\
I_{Exp}(exp)_n &= \{I_{Exp}(succ)(x) : x \in I_{Exp}(exp)_{n-1}\} \cup \\
&\quad \{I_{Exp}(pr_1)(x) : x \in I_{Exp}(exp)_{n-1}\} \cup \\
&\quad \{I_{Exp}(pr_2)(x) : x \in I_{Exp}(exp)_{n-1}\} \cup \\
&\quad \{I_{Exp}(+)(x, y) : (x, y) \in I_{Exp}(exp)_{n-1} \times I_{Exp}(exp)_{n-1}\}
\end{aligned}$$

$$I_{Exp}(exp) = \bigcup I_{Exp}(exp)_n, n \in \{0, 1, \ldots\}$$

The sketch contains no diagrams and so $I_{Exp}(0), I_{Exp}(succ), I_{Exp}(+)$ uniquely construct terms in $I_{Exp}(exp)$. The arrows $I_{Exp}(pr_1)$ and $I_{Exp}(pr_2)$ are forced to be the coordinate projections of the product $I_{Exp}(exp \times exp)$ and as a consequence do not construct terms in $I_{Exp}(exp)$. The description of $I_{Exp}(exp)$ can therefore be simplified to:

$$\begin{aligned}
I_{Exp}(exp)_0 &= \{\texttt{0}\} \\
I_{Exp}(exp)_n &= \{\texttt{succ}(x) : x \in I_{Exp}(exp)_{n-1}\} \cup \\
&\quad \{\texttt{+}(x, y) : (x, y) \in I_{Exp}(exp)_{n-1} \times I_{Exp}(exp)_{n-1}\}
\end{aligned}$$

$$I_{Exp}(exp) = \bigcup I_{Exp}(exp)_n, n \in \{0, 1, \ldots\}$$

and so $I_{Exp}(exp) \cong$ *exp-trees.*

In section 5.4 we will show that by using FL rather than FP sketches to model language syntax we can simplify the process of language specification by capturing the static semantics of a programming language within the specification of the syntax.

5.2 Using sketches to model language semantics

In this section we show how a sketch can be used to construct the semantics of a programming language. To describe the semantics of a programming language we must actually describe the computational universe in which the language exists. To describe this universe we simply view it as a complex abstract data type and construct an FL sketch, *Sem*, which has this datatype as its initial model.

The semantics of the simple expression language is given by an FP sketch, *Nat*, which describes the natural numbers with an addition operation. This sketch has nodes, **T**, *nat*, and *nat*×*nat*, and edges:

$$\begin{aligned}
0 &: \mathbf{T} \to nat \\
dispose &: nat \to \mathbf{T} \\
succ &: nat \to nat \\
pr_1 &: nat \times nat \to nat \\
pr_2 &: nat \times nat \to nat \\
+ &: nat \times nat \to nat \\
z &: nat \to nat \\
id_{nat} &: nat \to nat \\
\langle z, id_{nat} \rangle &: nat \to nat \times nat \\
succ \times id_{nat} &: nat \times nat \to nat \times nat.
\end{aligned}$$

There are two cones:

$$\mathbf{T}$$

and

$$\begin{tikzcd}
 & nat \times nat \arrow[dl, "pr_1"'] \arrow[dr, "pr_2"] & \\
nat & & nat
\end{tikzcd}$$

These cones force the objects, $M(\mathbf{T})$, and $M(nat \times nat)$ to be the sets, $\{\emptyset\}$, and $M(nat) \times M(nat)$ respectively. The graph of the sketch is shown below.

$$\begin{tikzcd}
\mathbf{T} \arrow[r, shift left, "0"] & nat \arrow[l, shift left, "dispose"] \arrow[loop, out=120, in=80, looseness=6, "succ"'] \arrow[loop, out=240, in=200, looseness=6, "id_{nat}"] \arrow[loop, out=280, in=250, looseness=6, "z"'] \arrow[rr, bend left=60, "\langle z, id_{nat} \rangle"] & & nat \times nat \arrow[ll, "+"'] \arrow[ll, bend right=35, "pr_1"'] \arrow[ll, bend left=35, "pr_2"] \arrow[loop right, "succ \times id_{nat}"]
\end{tikzcd}$$

We require six diagrams to complete the specification of the semantics.

$$\begin{tikzcd}
nat \arrow[rr, "z"] \arrow[dr, "dispose"'] & & nat \\
& \mathbf{T} \arrow[ur, "0"'] &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
nat \arrow[loop above, "id_{nat}"]
\end{tikzcd}$$

(a) (b)

From diagram (a) we obtain the equation

$$M(z) = M(0) \circ M(dispose)$$

For the initial model of Nat, $I_{Nat} : Nat \rightarrow \mathbf{SET}$, this equation forces $I_{Nat}(z)$ to be the function, $x \rightarrow I_{Nat}(0)(I_{Nat}(\mathbf{T}))$, i.e. $x \rightarrow 0$. Diagram (b) forces $I_{Nat}(id_{nat})$ to be the function $1_{I_{Nat}(nat)}$.

$$\begin{tikzcd}
& nat \arrow[dl, "z"'] \arrow[d, "\langle z, id_{nat} \rangle"] \arrow[dr, "id_{nat}"] & \\
nat & nat \times nat \arrow[l, "pr_1"] \arrow[r, "pr_2"'] & nat
\end{tikzcd}$$

(c)

The equations:

$$\begin{aligned} M(pr_1)\circ M(\langle z, id_{nat}\rangle) &= M(z) \\ M(pr_2)\circ M(\langle z, id_{nat}\rangle) &= M(id_{nat}) \end{aligned}$$

are obtained from diagram (c). Together, these equations force $I_{Nat}(\langle z, id_{nat}\rangle)$ to be the function: $x \rightarrow \langle I_{Nat}(z)(x), I_{Nat}(id_{nat})(x)\rangle$.

$$\begin{tikzcd}
nat \arrow[d, "succ"'] & nat \times nat \arrow[l, "pr_1"'] \arrow[r, "pr_2"] \arrow[d, "succ \times id_{nat}"] & nat \arrow[d, "id_{nat}"] \\
nat & nat \times nat \arrow[l, "pr_1"] \arrow[r, "pr_2"'] & nat
\end{tikzcd}$$

(d)

From diagram (d) we obtain the equations:

$$\begin{aligned} M(pr_1)\circ M(succ \times id_{nat}) &= M(succ)\circ M(pr_1) \\ M(pr_2)\circ M(succ \times id_{nat}) &= M(id_{nat})\circ M(pr_2) \end{aligned}$$

so $I_{Nat}(succ \times id_{nat})$ is the function, $\langle x, y\rangle \rightarrow \langle I_{Nat}(succ)(x), I_{Nat}(id_{nat})(y)\rangle$.

$$\begin{tikzcd}
nat \arrow[r, "\langle z, id_{nat}\rangle"] \arrow[dr, "id_{nat}"'] & nat \arrow[d, "+"] \\
& nat
\end{tikzcd}
\qquad
\begin{tikzcd}
nat \times nat \arrow[r, "+"] \arrow[d, "succ \times id_{nat}"'] & nat \arrow[d, "succ"] \\
nat \times nat \arrow[r, "+"'] & nat
\end{tikzcd}$$

(e) (f)

The final two diagrams (e), and (f) provide the equations:

$$\begin{aligned} M(+) \circ M(\langle z, id_{nat} \rangle) &= M(id_{nat}) \\ M(+) \circ M(succ \times id_{nat}) &= M(succ) \circ M(+) \end{aligned}$$

which force $I_{Nat}(+)$ to be addition on natural numbers. The sketch, *Nat*, is discussed in greater detail in [BaWe90], chapter 7, and I_{Nat} describes the semantics of the simple expression language of natural numbers and addition. In section 5.3 below we show how to combine the sketches *Exp* and *Nat* to produce a complete description of this language.

5.3 A categorical model of language

5.3.1 A categorical specification of language

Recall that Rus' algebraic model of language specifies a language as a triple

$$\langle Sem, Syn, learn : Sem \longrightarrow Syn \rangle$$

where *Syn* is the initial algebra over signature Σ and *Sem* is a similar algebra. The function $learn : Sem \longrightarrow Syn$ is defined on the carrier sets of *Sem* and *Syn* such that, if $eval : Syn \longrightarrow Sem$ is an homomorphism given by the initiality of *Syn* then $eval \circ learn = 1_{Sem}$.

To construct a categorical model of language with properties similar to Rus' algebraic model we specify it as a 4-tuple

$$\langle Sem, Syn, E : Syn \longrightarrow Sem, learn : E^*(M_{Sem}) \longrightarrow I_{Syn} \rangle$$

where *Sem* and *Syn* are FL sketches, and $E : Syn \longrightarrow Sem$ is a sketch morphism such that *Syn* is learnable from *Sem*. The transformation $learn : E^*(I_{Sem})(Sem) \longrightarrow I_{Syn}(Syn)$ is constructed by following the procedure given in the proof of proposition 4.3.2 and is described in greater detail below.

By theorem 5.1.1 we know that for every context free grammar, *CFG*, we can construct an FL sketch (actually an FP sketch but the extra power of FL sketches can be used to describe the static semantics) whose initial model in **SET** describes the abstract syntax trees of the language, $L(CFG)$. The sketch, *Syn*, is just such a sketch and we take its initial model in **SET**, $I_{Syn} : Syn \to \mathbf{SET}$ to be the *abstract* syntax of the language, $L(CFG)$.

The semantics of $L(CFG)$ is specified by the sketch, *Sem*, but we do not regard the initial model, $I_{Sem} : Sem \to \mathbf{SET}$, as the semantics because there is no obvious way to describe the evaluation of programs if I_{Sem} is the semantics. To construct the semantics of $L(CFG)$ we use property 4.3.1. The sketch morphism $E : Syn \to Sem$ induces a functor $E^* : \mathbf{Mod}(Sem) \to \mathbf{Mod}(Syn)$, and so by proposition 4.3.1 the model $E^*(I_{Sem}) : Syn \to \mathbf{SET} \in \mathbf{Mod}(Syn)$ specifies a datatype which is equivalent to I_{Sem} with hidden sorts. We use the model $E^*(I_{Sem}) \in \mathbf{Mod}(Syn)$ as the semantics of $L(CFG)$.

The evaluation function, $eval : I_{Syn} \dot{\to} E^*(I_{Sem})$, is the natural transformation which is known to exist because of property 4.3.2.

To complete the specification of $L(CFG)$ we specify $learn : E^*(I_{Sem}) \to I_{Syn}$ as a transformation such that, $eval \circ learn = 1_{E^*(I_{Sem})}$, where $\circ$ is composition of transformations. Since *Syn*, *Sem*, and $E : Syn \to Sem$ are such that *Syn* is learnable from *Sem* proposition 4.3.2 guarantees the existence of *learn*. The 4-tuple

$$\langle Sem, Syn, E : Syn \to Sem, learn : E^*(I_{Sem}) \to I_{Syn} \rangle$$

therefore completely specifies the *syntax*, *semantics* and *syntax* ↔ *semantics* association of the language, $L(CFG)$.

5.3.2 The language of natural numbers and addition

We have already constructed sketches, *Exp*, and *Nat*, which we will use to specify the syntax and semantics of the simple expression language from section 3.1.5. To complete the specification we must:

1. construct a sketch morphism $E : Syn \to Sem$ such that *Exp* is learnable from *Nat*.

2. Construct the transformation $learn : E^*(I_{Nat}) \to I_{Exp}$.

The construction of $E : Syn \rightarrow Sem$ follows a fairly simple procedure. We simply map elements from the syntax (the sketch *Exp*) to the corresponding elements from the semantics (the sketch *Nat*) which we wish to use to express the meanings of the syntactic objects. In this way we can construct $E : Syn \rightarrow Sem$ as:

$$\begin{aligned}
E(\mathbf{T}) &= \mathbf{T} \\
E(exp) &= nat \\
E(exp \times exp) &= nat \times nat \\
\\
E(0 : \mathbf{T} \rightarrow exp) &= 0 : \mathbf{T} \rightarrow nat \\
E(succ : exp \rightarrow exp) &= succ : nat \rightarrow nat \\
E(pr_1 : exp \times exp \rightarrow exp) &= pr_1 : nat \times nat \rightarrow nat \\
E(pr_2 : exp \times exp \rightarrow exp) &= pr_2 : nat \times nat \rightarrow nat \\
E(+ : exp \times exp \rightarrow exp) &= + : nat \times nat \rightarrow nat.
\end{aligned}$$

This leaves us with the following edges of *Nat* which are not the images in E of edges of *Exp*.

$$\begin{aligned}
dispose &: nat \rightarrow \mathbf{T} \\
z &: nat \rightarrow nat \\
id_{nat} &: nat \rightarrow nat \\
\langle z, id_{nat} \rangle &: nat \rightarrow nat \times nat \\
succ \times id_{nat} &: nat \times nat \rightarrow nat \times nat
\end{aligned}$$

To show that *Exp* is learnable from *Nat* we need to show that none of these arrows construct elements of the sets $I_{Nat}(\mathbf{T})$, $I_{Nat}(nat)$, or $I_{Nat}(nat \times nat)$.

We know that except for $dispose : nat \rightarrow \mathbf{T}$, the model I_{Nat} maps each of the above edges to functions which are defined in terms of $I_{Nat}(0)$, $I_{Nat}(succ)$, and $1_{I_{Nat}(nat)}$. As a result none of these functions produce elements of $I_{Nat}(\mathbf{T})$, $I_{Nat}(nat)$, or $I_{Nat}(nat \times nat)$ which are not constructed by some combination of $I_{Nat}(0)$ and $I_{Nat}(succ)$. Since we also know that $I_{Nat}(\mathbf{T})$ is terminal we know that $I_{Nat}(dispose) : I_{Nat}(nat) \rightarrow I_{Nat}(\mathbf{T})$ can only be the function $x \rightarrow \emptyset$ so it cannot construct elements either. We therefore know that *Exp* is learnable from *Nat*. We must now construct the transformation $learn : E^*(I_{Nat}) \rightarrow I_{Exp}$.

The component arrows of *learn* are constructed so that they are right inverses of the *eval* natural transformation. Obviously to specify *learn* we must first calculate *eval*.

The *eval* natural transformation

From property 4.3.2 we know that *eval* is the unique natural transformation $eval : I_{Exp} \rightarrow E^{*}(I_{Nat})$ given by the initiality of I_{Exp}. We therefore know that the diagrams below commute.

$$\begin{tikzcd}
I_{Exp}(\mathrm{T}) \arrow[r, "I_{Exp}(0)"] \arrow[d, "eval_{\mathrm{T}}"'] & I_{Exp}(exp) \arrow[d, "eval_{exp}"] \\
E^{*}(I_{Nat})(\mathrm{T}) \arrow[r, "E^{*}(I_{Nat})(0)"'] & E^{*}(I_{Nat})(exp)
\end{tikzcd}$$

(a)

From the definitions of I_{Exp} and I_{Nat} we know that $I_{Exp}(\mathrm{T}) = E^{*}(I_{Nat})(\mathrm{T}) = \{\emptyset\}$ and so $eval_{\mathrm{T}} = 1_{\{\emptyset\}}$

From diagram (a) we obtain the equation

$$eval_{exp} \circ I_{Exp}(0) = E^{*}(I_{Nat})(0) \circ eval_{\mathrm{T}}.$$

We know from the definitions of I_{Exp}, and I_{Nat} that $I_{Exp}(0)(\emptyset) = \mathtt{0}$ where $\mathtt{0}$ is a syntactic term, and $E^{*}(I_{Nat})(0)(\emptyset) = 0$, i.e. the number 0, so from the equation above we obtain

$$eval_{exp} \circ I_{Exp}(0) = (\emptyset \mapsto \mathtt{0}) \circ 1_{\{\emptyset\}}$$
$$eval_{exp} \circ (\emptyset \mapsto \mathtt{0}) = \emptyset \mapsto 0.$$

From this final equation we can partially define $eval_{exp}$ as: $eval_{exp}(\mathtt{0}) = 0$.

$$\begin{tikzcd}
I_{Exp}(exp) \arrow[r, "I_{Exp}(succ)"] \arrow[d, "eval_{exp}"'] & I_{Exp}(exp) \arrow[d, "eval_{exp}"] \\
E^*(I_{Nat})(exp) \arrow[r, "E^*(I_{Nat})(succ)"'] & E^*(I_{Nat})(exp)
\end{tikzcd}$$

(b)

From diagram (b) we obtain

$$eval_{exp} \circ I_{Exp}(succ) = E^*(I_{Nat})(succ) \circ eval_{exp}$$

By the definitions of I_{Exp} and I_{Nat} we know that $I_{Exp}(succ)$, and $E^*(I_{Nat})(succ)$ are respectively the functions $x \rightarrow \texttt{succ}(x)$ and $x \rightarrow x + 1$. From this we obtain:

$$eval_{exp} \circ (x \rightarrow \texttt{succ}(x)) = (x \rightarrow x + 1) \circ eval_{exp}$$

which allows us to partially define $eval_{exp}$ as: $eval_{exp}(\texttt{succ}(x)) = eval_{exp}(x) + 1$.

$$\begin{tikzcd}
I_{Exp}(exp \times exp) \arrow[r, "I_{Exp}(pr_1)"] \arrow[d, "eval_{exp \times exp}"'] & I_{Exp}(exp) \arrow[d, "eval_{exp}"] \\
E^*(I_{Nat})(exp \times exp) \arrow[r, "E^*(I_{Nat})(pr_1)"'] & E^*(I_{Nat})(exp)
\end{tikzcd}$$

(c)

The commutativity of diagram (c) gives us the equation:

$$eval_{exp} \circ I_{Exp}(pr_1) = E^*(I_{Nat})(pr_1) \circ eval_{exp \times exp}$$

while commutativity of diagram (d) gives us:

$$eval_{exp} \circ I_{Exp}(pr_2) = E^*(I_{Nat})(pr_2) \circ eval_{exp\times exp}.$$

$$\begin{tikzcd}
I_{Exp}(exp \times exp) \arrow[r, "I_{Exp}(pr_2)"] \arrow[d, "eval_{exp\times exp}"'] & I_{Exp}(exp) \arrow[d, "eval_{exp}"] \\
E^*(I_{Nat})(exp \times exp) \arrow[r, "E^*(I_{Nat})(pr_2)"'] & E^*(I_{Nat})(exp)
\end{tikzcd}$$

(d)

Since $I_{Exp}(pr_1)$ and $I_{Exp}(pr_2)$ are the co-ordinate projections for $I_{Exp}(exp) \times I_{Exp}(exp)$ and $E^*(I_{Nat})(pr_1)$ and $E^*(I_{Nat})(pr_2)$ are the co-ordinate projections for $I_{Nat}(nat) \times I_{Nat}(nat)$, diagrams (c) and (d) define $eval_{exp\times exp}$: $eval_{exp\times exp}(x, y) = (eval_{exp}(x), eval_{exp}(y))$.

$$\begin{tikzcd}
I_{Exp}(exp \times exp) \arrow[r, "I_{Exp}(+)"] \arrow[d, "eval_{exp\times exp}"'] & I_{Exp}(exp) \arrow[d, "eval_{exp}"] \\
E^*(I_{Nat})(exp \times exp) \arrow[r, "E^*(I_{Nat})(+)"'] & E^*(I_{Nat})(exp)
\end{tikzcd}$$

(e)

This final diagram adds one last equation which allows us to complete the definition of $eval_{exp}$.

$$eval_{exp} \circ I_{Exp}(+) = E^*(I_{Nat})(+) \circ eval_{exp\times exp}.$$

From the definition of I_{Exp} we know that $I_{Exp}(+)$ is the function $(x, y) \rightarrow +(x,y)$ which constructs syntactic expressions involving the plus operator from pairs of expressions. Similarly we know from I_{Nat} that $E^*(I_{Nat})(+)$ is addition on natural numbers. Using diagram (e) we can derive the equation

$$eval_{exp}(\texttt{+(}x\texttt{,}y\texttt{)}) = eval_{exp}(x) + eval_{exp}(y).$$

By collecting the various parts of the definition of the *eval* natural transformation together we construct the following:

$$\begin{aligned}
eval_{\mathrm{T}} &= \emptyset \mapsto \emptyset \\
eval_{exp} &= f \;\; where \;\; \begin{aligned}[t] f(\texttt{0}) &= 0 \\ f(\texttt{succ(}x\texttt{)}) &= f(x) + 1 \\ f(\texttt{+(}x,y\texttt{)}) &= f(x) + f(y) \end{aligned} \\
eval_{exp \times exp} &= (x,y) \rightarrow (eval_{exp}(x), eval_{exp}(y))
\end{aligned}$$

Having calculated the *eval* natural transformation we can now specify $learn : E^*(I_{Nat}) \rightarrow I_{Exp}$ so that

$$\forall n \in Exp : eval_n \circ learn_n = 1_{E^*(I_{Nat})n}.$$

The components of $learn : E^*(I_{Nat}) \rightarrow I_{Exp}$ are specified as:

$$\begin{aligned}
learn_{\mathrm{T}} &= \emptyset \mapsto \emptyset \\
learn_{exp} &= l \;\; where \;\; \begin{aligned}[t] l(0) &= \texttt{0} \\ l(x+1) &= \texttt{succ(}l(x)\texttt{)} \end{aligned} \\
learn_{exp \times exp} &= (x,y) \rightarrow (learn_{exp}(x), learn_{exp}(y)).
\end{aligned}$$

The language of natural numbers and addition is therefore completely specified by the 4-tuple:

$$\langle Nat, Exp, E, learn \rangle$$

where Nat is the sketch from section 5.2, Exp is the FP sketch given in section 5.1. $E : Syn \rightarrow Sem$ and $learn : E^*(I_{Nat}) \rightarrow I_{Exp}$ are, respectively, the sketch morphism and transformation shown above.

In the remainder of this chapter we discuss some of the implications the sketch based model of language has for the way in which we specify certain language features while in chapter 6 we will discuss the process by which we arrive at a self-interpreter for the arbitrary language $\mathcal{L}$ specified using this model.

5.4 Describing language features using the model

The categorical model of language described above allows the language specifier to use limits which are not simple products in a language specification. The use of such constructs can drastically simplify the specification of certain types of language construct.

In [KoQu92] Kortas and Quatrain use the categorical model of language described above to specify a subset of the pascal programming language. The specification that they provide is interesting because it uses these features to construct a specification which is both clear and less complex than can be achieved using conventional methods.

In this section we show how the model of language can be used to construct a specification of the type scheme for a simple FP [Back78] like language.

The specification that we construct demonstrates some of the extra power which is available within the categorical model of language, and shows that while the model has a great deal in common with Rus' algebraic model it is significantly more powerful.

5.4.1 A simple type scheme

The language we describe here constructs programs as the composition of functions. The language has two basic types: *num* and *char* and two basic operations:

1. **ord** which returns the ordinal number of a given character.

2. **chr** which will return the character with the ordinal number of its argument.

We also have one structured type constructor, *list*, which allows us to construct lists of any depth. The function **map** takes a function **f** of type

$$\mathtt{f}\colon * \rightarrow **$$

and constructs a function of type

$$\mathtt{map\ f}\colon *\ \mathit{list} \rightarrow **\ \mathit{list}.$$

Functions are composed using the **o** operator. This operator is a partial operator since the composition **f o g** is only defined if the source type of the function **f** is equal to the target type of the function **g**.

We can describe this aspect of the semantics of our language using a very simple sketch whose graph contains only 4 nodes: **T**, *exp*, *type*, and *exp* × *exp*. we require 12 edges:

$$\begin{array}{rcl}
\mathtt{ord} & : & \mathbf{T} \rightarrow exp \\
\mathtt{chr} & : & \mathbf{T} \rightarrow exp \\
\mathtt{map} & : & exp \rightarrow exp \\
\mathtt{o} & : & exp \times exp \rightarrow exp \\
num & : & \mathbf{T} \rightarrow type \\
char & : & \mathbf{T} \rightarrow type \\
list & : & type \rightarrow type \\
src & : & exp \rightarrow type \\
trg & : & exp \rightarrow type \\
pr_1 & : & exp \times exp \rightarrow exp \\
pr_2 & : & exp \times exp \rightarrow type \\
pr_3 & : & exp \times exp \rightarrow exp
\end{array}$$

to produce the graph shown pictorially as:

$$\begin{tikzcd}
\mathbf{T} \arrow[r, shift left, "\mathtt{ord}"] \arrow[r, shift right, "\mathtt{chr}"'] \arrow[dr, shift right, "char"'] \arrow[dr, shift left, "num"] & exp \arrow[loop above, "\mathtt{map}"] \arrow[d, shift right, "src"'] \arrow[d, shift left, "trg"] & exp \times exp \arrow[l, "\circ"'] \arrow[l, bend right, "pr_1"'] \arrow[l, bend left, "pr_3"] \arrow[dl, "pr_2"] \\
& type \arrow[loop below, "list"'] &
\end{tikzcd}$$

We require four diagrams to describe the *src* arrow:

$$\begin{tikzcd}
\mathbf{T} \arrow[r, "\mathtt{ord}"] \arrow[dr, "char"'] & exp \arrow[d, "src"] \\
& type
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\mathbf{T} \arrow[r, "\mathtt{chr}"] \arrow[dr, "num"'] & exp \arrow[d, "src"] \\
& type
\end{tikzcd}$$

(*a*) (*b*)

which give us the equations:

$$M(src) \circ M(\mathtt{ord}) = M(char)$$
$$M(src) \circ M(\mathtt{chr}) = M(num)$$

In other words the source type of the function **ord** is *char* and the source type of the function **chr** is *num*. From diagram (*c*)

$$\begin{tikzcd}
exp \arrow[r, "\mathtt{map}"] \arrow[d, "src"'] & exp \arrow[d, "src"] \\
type \arrow[r, "list"'] & type
\end{tikzcd}$$

(*c*)

$$\begin{tikzcd}
exp \times exp \arrow[r, "\circ"] \arrow[d, "pr_3"'] & exp \arrow[d, "src"] \\
exp \arrow[r, "src"'] & type
\end{tikzcd}$$

(*d*)

we obtain the equation:

$$M(src) \circ M(\mathtt{map}) = M(list) \circ M(src).$$

This tells us that the source type of the expression `map f` is $*$ *list* where $*$ is the source type of `f`, while diagram (d) gives us the equation:

$$M(src) \circ M(\circ) = M(src) \circ M(pr_3)$$

which states that the source of the composition `f o g` is equal to the source of `g`. Four similar diagrams are needed to describe *trg*.

We now have only the partial nature of the $\circ$ operator left to describe. This is done using the cones. There are two cones. Firstly the cone over the empty diagram

$$\mathbf{T}$$

and secondly the cone

$$\begin{tikzcd}
 & exp \times exp \arrow[dl, "pr_1"'] \arrow[d, "pr_2"] \arrow[dr, "pr_3"] & \\
exp \arrow[r, "src"'] & type & exp \arrow[l, "trg"]
\end{tikzcd}$$

When this cone is modelled we obtain an equation

$$M(src) \circ M(pr_1) = M(pr_2) = M(trg) \circ M(pr_3)$$

and the cone becomes a new kind of limit known as a *pullback*. In **SET** the vertex of this cone is a restricted form of product containing only those pairs which conform to certain properties. In this case the restricting property is given by the commutativity of the cone so we know that the set $M(exp \times exp)$ is

$$\{(x, y) : (x, y) \in M(exp) \times M(exp) \wedge M(src)(x) = M(trg)(y)\}.$$

This allows us to specify $\circ$ as a total operator since the only members of the set $M(exp \times exp)$ are those pairs of functions whose types make them composable.

Contrast this with a specification given using FP sketches or using Rus' algebraic model. In the case of an FP sketch the node $exp \times exp$ would be defined by the cone

$$\begin{tikzcd}
 & exp \times exp \arrow[dl, "pr_1"'] \arrow[dr, "pr_3"] & \\
exp & & exp
\end{tikzcd}$$

and so $M(exp \times exp)$ would contain very many pairs of non-composable functions. We would need to add several diagrams to the sketch to describe the behaviour of $\circ$ when applied to these pairs, which would need to be mapped to a new expression value type-error. We would then need to add a new arrow type-error: **T** → *exp* and several more diagrams to describe the behaviour of *src*, *trg*, and map applied to this value. We would also need to add another arrow *error-type*: **T** → *type* so that we could define the source and target type of the expression type-error. The result is a sketch (or an algebraic specification) which is drastically more complicated than the one given above and in which the simplicity of the type scheme we are trying to specify is consequently obscured. The inclusion of a cone which is modelled as a pullback allows us simply to ignore incorrectly typed programs because our syntax cannot contain them.

5.4.2 The semantics

The simple language described here is a higher order language which presents us with something of a problem since we cannot use any type of sketch directly to describe a higher order construct. We can, however, still describe the semantics of our language indirectly by describing the effect of applying programs to data objects.

To explain: we construct a node, *data*, whose model $I_{Sem}(\mathit{data})$ contains all well formed (i.e. type correct) data objects which can be processed in our language. We also define an edge

$$\mathit{typ} : \mathit{data} \rightarrow \mathit{type}$$

with appropriate diagrams so that the function $I_{Sem}(\mathit{typ})$ returns the type of any element of $I_{Sem}(\mathit{data})$ to which it is applied.

Using the *data* node and the $\mathit{typ} : \mathit{data} \rightarrow \mathit{type}$ edge we can construct the cone

$$\begin{tikzcd}
 & \mathit{exp} \times \mathit{data} \arrow[dl, "pr_4"'] \arrow[d, "pr_5"'] \arrow[dr, "pr_6"] & \\
\mathit{exp} \arrow[r, "\mathit{src}"'] & \mathit{type} & \mathit{data} \arrow[l, "\mathit{typ}"]
\end{tikzcd}$$

whose vertex $I_{Sem}(\mathit{exp} \times \mathit{data})$ contains all pairs of programs and the data objects to which they can be applied.

If we now add an edge

$$\mathit{run} : \mathit{exp} \times \mathit{data} \rightarrow \mathit{data}$$

we can construct diagrams which describe the effect of applying programs to data objects and thus describe the semantics of the language. The resulting sketch is, however, rather complex and there is little to be gained by showing it here.

Chapter 6

A Categorical Approach to a Self-Interpreter

So far we have constructed a model of language based on FL sketches which while it has many properties in common with Rus' algebraic model of language is, as we demonstrated in section 5.4, significantly more powerful. In this chapter we use the properties which our sketch based model of language shares with Rus' model to construct a self-interpreter.

6.1 Construction of the self-interpreter

It was shown in section 3.2 that using Rus' algebraic model of language, the function computed by the interpreter for the programming language

$$\mathcal{L} = \langle Syn(B, W(X, \Sigma)), Sem(B, A), learn : Sem(B, A) \rightarrow Syn(B, W(X, \Sigma))\rangle$$

is defined as

$$interpreter = learn \circ eval.$$

This is also true for the categorical model of language.

Proposition 6.1.1 *If $\mathcal{L}$ is specified using sketches*

$$\mathcal{L} = \langle \mathrm{Sem}, \mathrm{Syn}, E : \mathrm{Syn} \rightarrow \mathrm{Sem}, \mathrm{learn} : E^*(I_{\mathrm{Sem}}) \rightarrow I_{\mathrm{Syn}} \rangle$$

the interpreter function is described as

$$\mathrm{interpreter} = \mathrm{learn} \circ \mathrm{eval}.$$

Proof: Since *Sem*, *Syn*, and $E : Syn \rightarrow Sem$ are such that *Syn* is learnable from *Sem* we know that for each object $S \in Syn$ there is a quotient set $I_{Syn}(S)_{\equiv_S}$ defined by $eval_S$ and that $I_{Syn}(S)_{\equiv_S} \cong E^*(I_{Sem})(S)$. From the construction of $learn_S$ we know that for each meaning $m \in E^*(I_{Sem})(S)$, $m \cong [x]$, where $eval_S(x) = m$, $learn_S(m) \in [x]$ is the preferred syntactic representation of m. The function $interpreter = learn \circ eval$ therefore maps programs to their preferred syntactic form (whilst preserving their meaning) and is an $\mathcal{L}$ interpreter. □

Proposition 6.1.1 is an exact analogue of proposition 3.2.1 stated in section 3.2. We also obtain an analogue of theorem 3.2.1 shown below

Theorem 6.1.1 *For each edge, $f : a \rightarrow b$, from the graph of the sketch* Syn, *the function.*

$$F_{f:a\rightarrow b} : I_{Syn}(a) \rightarrow I_{Syn}(b)$$

in **SET**, *defined as*

$$F_{f:a\rightarrow b} = \mathrm{interpreter} \circ I_{\mathrm{Syn}}(f)$$

has the same behaviour on syntactic objects as $E^*(I_{Sem})(f) : E^*(I_{Sem})(a) \rightarrow E^*(I_{Sem})(b)$ has on semantic objects.

Proof: $F_{f:a\rightarrow b}$ is defined as:

$$\begin{aligned} F_{f:a\rightarrow b} &= interpreter \circ I_{Syn}(f) \\ &= learn_b \circ eval_b \circ I_{Syn}(f) \\ &= learn_b \circ E^*(I_{Sem})(f) \circ eval_a \end{aligned}$$

This theorem can in fact be generalised to include all arrows $g : E(a) \rightarrow E(b)$ from the graph of Sem where $E : Syn \rightarrow Sem$ is the sketch morphism, given in the language specification, and defines E^*.

$$F_{g:E(a)\rightarrow E(b)} = learn_b \circ E^*(I_{Sem})(g) \circ eval_a$$

This generalisation allows us to construct arrows which correspond to the hidden operations defined on the objects of $E^*(I_{Sem})(Syn)$. □

In the example language given in chapter 5 the interpreter function for the language:

$$\mathcal{L} = \langle Nat, Exp, E : Exp \rightarrow Nat, learn : E^*(I_{Nat}) \rightarrow I_{Exp} \rangle$$

is therefore specified as:

$$\begin{aligned}
interpreter_{\mathrm{T}} &= learn_{\mathrm{T}} \circ eval_{\mathrm{T}} \\
&= \emptyset \mapsto \emptyset \\[1em]
interpreter_{exp} &= learn_{exp} \circ eval_{exp} \\
&= f \quad where \quad \begin{aligned}[t] f(\mathtt{0}) &= \mathtt{0} \\ f(\mathtt{succ}(x)) &= \mathtt{succ}(f(x) \\ f(\mathtt{+}(x,y)) &= F_{+:exp\times exp\rightarrow exp}(x, y) \end{aligned} \\
interpreter_{exp\times exp} &= learn_{exp\times exp} \circ eval_{exp\times exp} \\
&= learn_{exp\times exp} \circ (x, y) \rightarrow (eval_{exp}(x), eval_{exp}(y)) \\
&= (x, y) \rightarrow ((learn_{exp} \circ eval_{exp})(x), (learn_{exp} \circ eval_{exp})(y)) \\
&= (x, y) \rightarrow (interpreter_{exp}(x), interpreter_{exp}(y))
\end{aligned}$$

The function, $F_{+:exp\times exp\rightarrow exp} : I_{Syn}(exp) \times I_{Syn}(exp) \rightarrow I_{Syn}(exp)$, used in the definition of $interpreter_{exp}$ above is given by theorem 6.1.1 and is defined exactly as in figure 3.1.

$$\begin{aligned}
F_{+:exp\times exp\rightarrow exp} &= learn_{exp} \circ + \circ eval_{exp\times exp} \\[1em]
F_{+:exp\times exp\rightarrow exp}(\mathtt{0}, y) &= y \\
F_{+:exp\times exp\rightarrow exp}(\mathtt{succ}(x), y) &= \mathtt{succ}(F_{+:exp\times exp\rightarrow exp}(x, y))
\end{aligned}$$

As with the algebraic description of a self-interpreter in section 3.2 the components of the interpreter function above lie outside the semantics of the language $\mathcal{L}$. The remainder of this chapter deals with the process by which the *interpreter* transformation is converted into an $\mathcal{L}$ program.

6.2 Moving into the semantics

The first step in the process of converting *interpreter* into an $\mathcal{L}$ program is to construct a datatype which exists within the semantics of $\mathcal{L}$ and is capable of representing the abstract syntax trees of $\mathcal{L}$ programs. Although the construction of this representation is not addressed in this thesis we still need to provide a definition of such a representation since we require certain properties for the construction of the self-interpreter.

Definition: A representation of syntax.
Given a language specification

$$\mathcal{L} = \langle Sem, Syn, E : Syn - Syn, learn : E^*(I_{Sem}) \rightarrow I_{Syn} \rangle$$

a representation of the syntax of $\mathcal{L}$ is defined as a pair of transformations

$$\begin{aligned} encode &: I_{Syn} \rightarrow E^*(I_{Sem}) \\ decode &: E^*(I_{Sem}) \rightarrow I_{Syn} \end{aligned}$$

such that $decode \circ encode = 1_{I_{Syn}}$. □

Each object, $I_{Syn}(t)$, of the syntax must be taken to an object, rep_t, of the semantics. The objects rep_s and rep_t need not be distinct if they represent different objects from the syntax. The only restriction required is that where a semantics object represents more than one syntax object the representations form disjoint subsets. In this way the syntax of $\mathcal{L}$ is represented in the semantics of $\mathcal{L}$ with no loss of information. The choice of these arrows is dependent on the exact representation chosen for the syntax of $\mathcal{L}$ and is outside the scope of this discussion. An example representation may be found in appendix A.4.

Once the representation of the syntax has been constructed the *interpreter* function can be moved within the semantics of $\mathcal{L}$ by composing it with the *encode* and *decode* transformations.

$$rep\_interpreter = encode \circ interpreter \circ decode$$

For each function $F_{f:a \to b} : I_{Syn}(a) \to I_{Syn}(b)$ we can also construct the function $\text{rep\_}F_{f:a \to b}$ by composition with *encode* and *decode*.

$$\text{rep\_}F_{f:a \to b} = encode_b \circ F_{f:a \to b} \circ decode_a$$

We can now formalise a notion of a language with sufficient power to express its self interpreter. The language $\mathcal{L}$ is powerful enough to express its self-interpreter if the following holds:

$$\forall t \in G_{Syn}0 : rep\_interpreter_t \in E^*(I_{Sem})(Syn).$$

Less formally, the language $\mathcal{L}$ can express its self-interpreter if its semantics contains the arrow $rep\_interpreter_t$ for every node, t, from the graph of *Syn*.

We should note that *rep_interpreter* is defined in terms of the $rep\_F_{f:a \to b}$ functions so if the semantics of $\mathcal{L}$ contains the arrows $rep\_interpreter_t$ for each node t it will also contain the arrow $\text{rep\_}F_{f:a \to b}$ for every arrow $f : a \to b$ of *Syn*.

The predicate above is really too abstract to tell us much about the nature of the language $\mathcal{L}$ because it does not relate to any language features. We would, however, expect $\mathcal{L}$ to provide methods of constructing:

Binary trees: Which are necessary to represent $\mathcal{L}$ programs as data objects.

Conditional: We require some form of conditional in order to be able to select the correct code segments to simulate a particular syntactic construct in an $\mathcal{L}$ program.

Recursive functions: These are necessary to enable us to form the code segments necessary to simulate the behaviour of the syntactic constructs of the language $\mathcal{L}$.

These requirements may be met directly by $\mathcal{L}$, as is the case for the *Toy* language defined in appendix A, or indirectly as would be the case if $\mathcal{L}$ were, say, an assembler language. In this second case $\mathcal{L}$ does not provide either binary trees or recursive functions but is sufficiently powerful to be used to construct implementations of both.

6.3 The self-interpreter

For a suitable language, $\mathcal{L} = \langle Sem, Syn, E : Syn \rightarrow Sem, learn : E^*(I_{Sem}) \rightarrow I_{Syn} \rangle$, the category $E^*(I_{Sem})(Syn)$ contains the arrows, $\{rep\_interpreter_t : t \in G_{Sem0}\}$, which are the functions the $\mathcal{L}$ self-interpreter, *$\mathcal{L}$-self-int*, computes. To complete the construction of *$\mathcal{L}$-self-int* we must convert these functions into an $\mathcal{L}$ program.

To perform this conversion we require an algorithm which generates an $\mathcal{L}$ program, *$\mathcal{L}$-self-int*, such that $eval(\mathcal{L}\text{-}self\text{-}int) = rep\_interpreter_s$, where s is the node of *Syn* which denotes complete $\mathcal{L}$ programs. Provided we make the, not unreasonable, assumption that the notation used to represent the rep_*interpreter* function has a fixed syntax and semantics, this algorithm is, in fact, a parameterised compiler. The extra pieces of information which we must supply as parameters of the compiler are:

1. the syntactic construct which $\mathcal{L}$ uses to define functions.
2. The $\mathcal{L}$ syntactic construct corresponding to a function call.
3. The form of conditional used by $\mathcal{L}$.

These parameters are required because the *rep_interpreter* function is structured as a set of functions which perform re-writes of the encoding of the syntax of $\mathcal{L}$. We therefore need to be able to:

1. define functions in *$\mathcal{L}$-self-int* which perform these re-writes.
2. Generate calls of these re-writing functions.
3. Generate conditionals to decide which re-writing function to call in a given situation.

In appendix A we define *Toy*, a simple, typeless, first order functional language. Functions in *Toy* can only be defined at the top level and have one *implicit* argument, named **arg** in the body of the function. The only data objects in *Toy* are natural numbers and binary trees. We can test natural numbers for equality using the = operator and construct and destruct binary trees using the (_,_) and fst, snd operators respectively.

The function which our *Toy* self-interpreter computes is shown in appendix A.5.2. Below we outline the final stage of the construction of the self-interpreter. Note that for the sake of clarity we have used arabic numerals and meaningful identifiers rather than the *Toy* syntactic constructs.

The node of the sketch Toy_{Syn} which corresponds to complete *Toy* programs is named *prg*. We can use this information to index the component of *rep_int* corresponding to the top level of the self-interpreter.

$$rep\_int_{prg} = \texttt{(10,(9,(}e\texttt{,}d\texttt{)))} \rightarrow rep\_F_{where:exp \times decs \rightarrow prg}(e,d)$$

This allows us to construct the top level of the self-interpreter as a call to the function ***where**.

```
*where(arg) where
```

We now use the definition given by theorem 6.1.1 of $rep\_F_{where:exp\times decs\rightarrow prg}$ shown below

$$rep\_F_{where:exp\times decs\rightarrow prg}(e,d) \quad = \quad \texttt{(10,(9,(}rep\_F_{apply:exp\times decs\rightarrow exp}(x,y)\texttt{,(3,(0,0))))}$$

to add the definition of ***where** to the self-interpreter.

```
*where(arg) where *where = (10,(9,(*apply(arg),(3,(0,0))));
```

Since $rep\_F_{where:exp\times decs\rightarrow prg}$ is defined in terms of $rep\_F_{apply:exp\times decs\rightarrow prg}$, we must use its definition

$$
\begin{aligned}
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(0,0)).}d) &= \texttt{(4,(0,0))}\\
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(0,1)),}d) &= \texttt{(4,(0,1))}\\
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(1,}n\texttt{)),}d) &= \texttt{(4,(1,}n\texttt{))}\\
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(2,}e\texttt{)),}d) &= \mathit{rep\_F}_{fst:exp\rightarrow exp}(\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(e,d))\\
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(3,}e\texttt{)),}d) &= \mathit{rep\_F}_{snd:exp\rightarrow exp}(\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(e,d))
\end{aligned}
$$

$$\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(4,(5,(}x\texttt{,}y\texttt{)))),}d)$$

$$= \texttt{(4,(4,(}\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(x,d)\texttt{,}\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(y,d)\texttt{)))}$$

$$\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(5,(5,(}x\texttt{,}y\texttt{)))),}d)$$

$$= \mathit{rep\_F}_{=:exp\times exp\rightarrow exp}(\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(x,d), \mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(y,d))$$

$$\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(6,(6,(}x\texttt{,(}y\texttt{,}z\texttt{))))),}d)$$

$$= \mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\mathit{rep\_F}_{if:exp\times exp\times exp\rightarrow exp}(\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(x,d),y,z),d)$$

$$\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(7,(}i\texttt{,}e\texttt{))),}d)$$

$$= \mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\mathit{rep\_F}_{replace:exp\times exp\rightarrow exp}(\mathit{body}, \mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(e,d)),d)$$

$$\text{where}\quad \mathit{body} \;=\; \mathit{rep\_F}_{fetch:ident\times decs\rightarrow exp}(i,d)$$

to construct the definition of *apply which we then add to the self-interpreter.

```
*apply = if fst(fst(arg)) = 4 then
          if fst(snd(fst(arg))) = 0 then fst(arg)
          else if fst(snd(fst(arg))) = 1 then fst(arg)
          else if fst(snd(fst(arg))) = 2 then
                   *fst(*apply((snd(fst(arg)),snd(arg)))
          else if fst(snd(fst(arg))) = 3 then
                   *snd(*apply((snd(fst(arg)),snd(arg)))
          else if fst(snd(fst(arg))) = 4 then
                   (4,(4,(*apply((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                          *apply((snd(snd(snd(snd(fst(arg))))),snd(arg))))))
          else if fst(snd(fst(arg))) = 5 then
                   *=(*apply((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                         *apply((snd(snd(snd(snd(fst(arg))))),snd(arg))))
          else if fst(snd(fst(arg))) = 6 then
               *apply((*if((*apply((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                          (fst(snd(snd(snd(snd(fst(arg)))))),
                    snd(snd(snd(snd(snd(fst(arg)))))))),
               snd(arg)))
      else if fst(snd(fst(arg))) = 7 then
          *apply((*replace((*fetch(fst(snd(snd(fst(arg))))),snd(arg))),
                  *apply((snd(snd(snd(fst(arg)))),snd(arg)))))


      else error
    else error;
```

Each clause of the definition of $rep\_F_{apply:exp \times decs \rightarrow exp}$ adds a conditional branch to `*apply` and to complete the definition of this function we must construct implementations of the functions: $rep\_F_{fst:exp \rightarrow exp}$, $rep\_F_{snd:exp \rightarrow exp}$, $rep\_F_{=:exp \times exp \rightarrow exp}$, $rep\_F_{if:exp \times exp \times exp \rightarrow exp}$, $rep\_F_{replace:exp \times exp \rightarrow exp}$, $rep\_F_{fetch:ident \times decs \rightarrow exp}$. These functions implement the language operations: *fst*, *snd*, =, *if*, and *call*, where *call* is actually implemented using *apply*, *replace*, and *fetch*.

$$
\begin{aligned}
rep\_F_{fst:exp \rightarrow exp}(x) &= \texttt{(4,(0,1))}, & x &= \texttt{(4,(0,1))} \\
&= x, & x &= \texttt{(4,(1,}y\texttt{))} \\
&= a, & x &= \texttt{(4,(4,(5,(}a\texttt{,}b\texttt{))))} \\
&= \texttt{(4,(2,}x\texttt{))}, & &\text{otherwise}
\end{aligned}
$$

This function gives us the definition of `*fst`.

```
*fst = if fst(arg) = 4 then
            if fst(snd(arg)) = 0 then
                if snd(snd(arg)) = 1 then (4,(0,1))
                else error
            else if fst(snd(arg)) = 1 then arg
            else if fst(snd(arg)) = 4 then fst(snd(snd(snd(arg))))
            else (4,(2,arg))
        else (4,(2,arg));
```

$$\begin{array}{rcll} rep\_F_{snd:exp\rightarrow exp}(x) & = & \mathtt{(4,(0,1))}, & x = \mathtt{(4,(0,1))} \\ & = & x, & x = \mathtt{(4,(1,}y\mathtt{))} \\ & = & b, & x = \mathtt{(4,(4,(5,(}a\mathtt{,}b\mathtt{))))} \\ & = & \mathtt{(4,(3,}x\mathtt{))}, & \textbf{otherwise} \end{array}$$

The function rep_F$_{snd:exp\rightarrow exp}$ provides the implementation of *snd shown.

```
*snd = if fst(arg) = 4 then
          if fst(snd(arg)) = 0 then
               if snd(snd(arg)) = 1 then (4,(0,1))
               else error
          else if fst(snd(arg)) = 1 then arg
          else if fst(snd(arg)) = 4 then snd(snd(snd(snd(arg))))
          else (4,(3,arg))
       else (4,(3,arg));
```

The function *rep_F*$_{=:exp\times exp\rightarrow exp}$ is defined as

$$\begin{array}{lcll} rep\_F_{=:exp\times exp\rightarrow exp}(x,y) & & & \\ & = & \mathtt{(4,(1,}rep\_equal(a,b)\mathtt{))}, & x = \mathtt{(4,(1,}a\mathtt{))} \wedge y = \mathtt{(4,(1,}b\mathtt{))} \\ & = & \mathtt{(4,(0,1))}, & x = \mathtt{(4,(4,}a\mathtt{))} \vee y = \mathtt{(4,(4,}b\mathtt{))} \vee \\ & & & x = \mathtt{(4,(0,1))} \vee y = \mathtt{(4,(0,1))} \\ & = & \mathtt{(4,(5,(}x\mathtt{,}y\mathtt{)))}, & \textbf{otherwise} \end{array}$$

$$\begin{array}{llcl} \text{where} & rep\_equal(\mathtt{(2,0)},\mathtt{(2,0)}) & = & \mathtt{(2,0)} \\ & rep\_equal(\mathtt{(2,}x+1\mathtt{)},\mathtt{(2,0)}) & = & \mathtt{(2,1)} \\ & rep\_equal(\mathtt{(2,0)},\mathtt{(2,}x+1\mathtt{)}) & = & \mathtt{(2,1)} \\ & rep\_equal(\mathtt{(2,}x+1\mathtt{)},\mathtt{(2,}y+1\mathtt{)}) & = & rep\_equal(x,y) \end{array}$$

and adds *= to the self-interpreter.

```
*= = if *and((fst(fst(arg))=4,fst(snd(arg))=4)) then
        if *and((fst(snd(fst(arg)))=1,fst(snd(snd(arg)))=1)) then
            (4,(1,*rep_equal((snd(snd(fst(arg))),snd(snd(snd(arg)))))))
        else if *or((*and((fst(snd(fst(arg)))=4,fst(snd(snd(arg)))=4)),
                     *and((*and((fst(snd(fst(arg)))=0,
                                 snd(snd(fst(arg)))=1)),
                           *and((fst(snd(snd(arg)))=0,
                                 snd(snd(snd(arg)))=1))))) then
             (4,(0,1))
        else (4,(5,arg))
     else error;
```

Since the definition of $rep\_F_{=:exp\times exp\to exp}$ is given in terms of the function *rep_equal* we must also construct `*rep-equal`. In addition, for the sake of readability, we have defined the functions `*and` and `*or`.

```
*rep_equal = if *and((fst(fst(arg))=2,fst(snd(arg))=2)) then
                (2,snd(fst(arg))=snd(snd(arg)))
             else error;

*and = if fst(arg) then snd(arg) else fst(arg);

*or  = if fst(arg) then fst(arg) else snd(arg);
```

The definition of $rep\_F_{if:exp\times exp\times exp\to exp}$ shown below requires no auxiliary functions.

$$\begin{array}{lll} rep\_F_{if:exp\times exp\times exp\to exp}(x,y,z) & & \\ \quad = & (4,(0,1)), & x = (4,(0,1)) \\ \quad = & y, & x = (4,(1,0)) \\ \quad = & z, & x = (4,(1,y+1)) \vee x = (4,(4,a)) \\ \quad = & (4,(6,(6,(x,(y,z))))). & \textbf{otherwise} \end{array}$$

We therefore only need to add `*if` to the self-interpreter to implement it.

```
*if = if fst(fst(arg)) = 4 then
         if *and((fst(snd(fst(arg)))=0,snd(snd(fst(arg)))=1)) then
            (4,(0,1))
         else if *and((fst(snd(fst(arg)))=1,snd(snd(fst(arg)))=0)) then
```

```
      fst(snd(arg))
   else if *or((fst(snd(fst(arg)))=1,fst(snd(fst(arg)))=4)) then
      snd(snd(arg))
   else (4,(6,(6,(fst(arg),(fst(snd(arg)),snd(snd(arg)))))))
else (4,(6,(6,(fst(arg),(fst(snd(arg)),snd(snd(arg)))))));
```

The operation $rep\_F_{replace:exp\times exp\rightarrow exp}$ also becomes a single *Toy* function in the implementation.

$$
\begin{array}{lcl}
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(0,0)),r) & = & r \\
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(0,1)),r) & = & (4,(0,1)) \\
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(1,n)),r) & = & (4,(1,n)) \\
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(2,e)),r) & = & \\
\multicolumn{3}{r}{rep\_F_{fst:exp\rightarrow exp}(rep\_F_{replace:exp\times exp\rightarrow exp}(e,r))} \\
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(3,e)),r) & = & \\
\multicolumn{3}{r}{rep\_F_{snd:exp\times exp\rightarrow exp}(rep\_F_{replace:exp\times exp\rightarrow exp}(e,r))}
\end{array}
$$

$$
\begin{array}{l}
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(5,(5,(x,y)))),r) \\
\qquad = rep\_F_{=:exp\times exp\rightarrow exp}(rep\_F_{replace:exp\times exp\rightarrow exp}(x,r), F_{replace:exp\times exp\rightarrow exp}(y,r)) \\
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(4,(5,(x,y)))),r) \\
\qquad\qquad = (rep\_F_{replace:exp\times exp\rightarrow exp}(x,r), rep\_F_{replace:exp\times exp\rightarrow exp}(y,r)) \\
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(6,(6,(x,(y,z))))),r) \\
\qquad\qquad\qquad = \quad rep\_F_{if:exp\times exp\times exp\rightarrow exp}(x',y',z')
\end{array}
$$

$$
\begin{array}{lrcl}
\text{where} & x' & = & rep\_F_{replace:exp\times exp\rightarrow exp}(x,r) \\
 & y' & = & rep\_F_{replace:exp\times exp\rightarrow exp}(y,r) \\
 & z' & = & rep\_F_{replace:exp\times exp\rightarrow exp}(z,r)
\end{array}
$$

$$
\begin{array}{l}
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(7,(7,(i,e)))),r) \ = \\
\qquad\qquad\qquad (4,(7,(7,(i,rep\_F_{replace:exp\times exp\rightarrow exp}(e,r)))))
\end{array}
$$

```
*replace = if fst(fst(arg)) = 4 then
              if fst(snd(fst(arg))) = 0 then
                 if snd(snd(fst(arg))) = 0 then snd(arg)
                 else (4,(0,1))
              else if fst(snd(fst(arg))) = 1 then fst(arg)
```

```
        else if fst(snd(fst(arg))) = 2 then
                *fst(*replace((snd(snd(fst(arg))),snd(arg))))
        else if fst(snd(fst(arg))) = 3 then
                *snd(*replace((snd(snd(fst(arg))),snd(arg))))
        else if fst(snd(fst(arg))) = 5 then
                *=((*replace((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                    *replace((snd(snd(snd(snd(fst(arg))))),snd(arg)))))
        else if fst(snd(fst(arg))) = 4 then
                (*replace((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                 *replace((snd(snd(snd(snd(fst(arg))))),snd(arg))))
        else if fst(snd(fst(arg))) = 6 then
                *if((*replace((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                    (*replace((fst(snd(snd(snd(snd(fst(arg)))))),
                               snd(arg))),
                     *replace((snd(snd(snd(snd(snd(fst(arg)))))),
                               snd(arg)))))
        else if fst(snd(fst(arg))) = 7 then
                (4,(7,(7,(fst(snd(snd(snd(fst(arg))))),
                    *replace((snd(snd(snd(snd(fst(arg))))),
                              snd(arg)))))));
```

In the definition of $rep\_F_{replace:exp\times exp\rightarrow exp}$ we only encounter one function for which we do not already have a *Toy* implementation, namely $rep\_F_{fetch:ident\times decs\rightarrow exp}$

$$rep\_F_{fetch:ident\times decs\rightarrow exp}(x,(3,(0,0))) \quad = \quad (4,(0,1))$$

$$rep\_F_{fetch:ident\times decs\rightarrow exp}(x,(3,(1,(8,(y,(e,d))))))$$
$$= rep\_F_{get:ident\times num\times exp\times decs\rightarrow exp}(x, rep\_F_{same:ident\times ident\rightarrow num}(x,y), e, d)$$

We implement this function as `*fetch`.

```
*fetch = if fst(snd(arg)) = 3 then
            if fst(snd(snd(arg))) = 0 then (4,(0,1))
            else if fst(snd(snd(arg))) = 1 then
                 *get((fst(arg),
                      (*same((fst(arg),fst(snd(snd(snd(snd(arg)))))))),
                      (fst(snd(snd(snd(snd(snd(arg)))))),
                       snd(snd(snd(snd(snd(snd(arg)))))))))
            else error;
         else error;
```

The $rep\_F_{fetch:ident\times decs\rightarrow exp}$ function is defined in terms of two new functions

$$
\begin{aligned}
rep\_F_{get:ident\times num\times exp\times decs\rightarrow exp}(x,\texttt{(1,0)},e,d) &= e\\
rep\_F_{get:ident\times num\times exp\times decs\rightarrow exp}(x,\texttt{(1,}n+1\texttt{)},e,d) &= rep\_F_{fetch:ident\times decs\rightarrow exp}(x,d)
\end{aligned}
$$

and

$$
\begin{aligned}
rep\_F_{same:ident\times ident\rightarrow num}(\texttt{(1,0),(1,0)}) &= \texttt{(1,0)}\\
rep\_F_{same:ident\times ident\rightarrow num}(\texttt{(1,}x+1\texttt{),(1,0)}) &= \texttt{(1,1)}\\
rep\_F_{same:ident\times ident\rightarrow num}(\texttt{(1,0),(1,}x+1\texttt{)}) &= \texttt{(1,1)}\\
rep\_F_{same:ident\times ident\rightarrow num}(\texttt{(1,}x+1\texttt{),(1,}y+1\texttt{)}) &= rep\_F_{same:ident\times ident\rightarrow num}(x,y).
\end{aligned}
$$

These definitions provide us with the last two functions definitions we need to add to the *Toy* self-interpreter to complete it.

```
*get = if fst(fst(snd(arg))) = 1 then
          if snd(fst(snd(srg))) = 0 then fst(snd(snd(arg)))
          else *fetch((fst(arg),snd(snd(snd(arg)))))
       else error;

*same = if *and((fst(fst(arg))=1,fst(snd(arg))=1)) then
           (1,snd(fst(arg))=snd(snd(arg)))
        else error;
```

The complete text of the *Toy* self-interpreter is shown in appendix A.6.

For this process to be completely automatic we need to be able to recognise the *Toy* syntactic forms of function definition, function application, and conditional. The algorithm to perform this analysis must be sufficiently general to recognise these forms however they manifest themselves. The reader should bear in mind that if *Toy* were, for example, an assembler language then the features we would need to recognise could in fact be: labels, and conditional goto. At this time we are unable to construct such a recognition algorithm and as a result the precess described here still requires considerable input from the user. We will return to this problem in section 7.1.2.

Chapter 7

Concluding Discussion

In the preceding six chapters we propose and construct an alternative framework for considering the *source* → *target* relationship in the specification of compilers. The approach is centred upon partial evaluation and the framework is categorical in nature and based on the theory of sketches. Inevitably, this work reveals a number of new lines of research and identifies a variety of interesting open problems as well as providing useful experience, and experimental data, on the theoretical and practical usefulness of sketch theory in a large and important area of computing science. In section 7.1 we review the model of language and self-interpreter construction method developed in this thesis. Section 7.2 deals with some of the extensions necessary to convert the construction technique so that it becomes a true compiler construction method. In section 7.2 we also address some of the questions of practicality, both of the true compiler construction technique outlined in chapter 2, and of fully automatic compiler construction in general. In the final section, 7.3, we speculate about some methods of removing the more obvious shortcomings of the sketch based model of language used here.

7.1 Summary

The work in this thesis falls roughly into two interrelated sections. The bulk of the work deals with the construction of a categorical model of language based on sketches. This model of language is, however, not the main aim of the thesis as its construction is motivated by the desire to develop a technique for the calculation of a self-interpreter. Accordingly this section is split into two parts: in section 7.1.1 we critically appraise the categorical model of language while in section 7.1.2 the discussion focusses on the self-interpreter construction technique.

7.1.1 The model of language

The categorical model of language describes a language, $\mathcal{L}$, as a 4-tuple

$$\mathcal{L} = \langle Sem, Syn, E : Syn \to Sem, learn : E^*(I_{Sem}) \to I_{Syn} \rangle.$$

Sem and *Syn* are FL sketches, and $E : Syn \to Sem$ is a sketch morphism such that *Syn* is learnable from *Sem*. The syntax and semantics of $\mathcal{L}$ are both constructed as models of *Syn*, where the syntax is constructed as the initial model: $I_{Syn} : Syn \to \mathbf{SET}$. We use property 4.3.1 to construct the semantics of $\mathcal{L}$ as the model

$$E^*(I_{Sem}) : Syn \to \mathbf{SET}$$

where $I_{Sem} : Sem \to \mathbf{SET}$ is the initial model of *Sem* in **SET**.

The construction of *eval* as the natural transformation

$$eval : I_{Syn} \stackrel{.}{\to} E^*(I_{Sem})$$

arises from property 4.3.2. Since *Sem*, *Syn*, and $E : Syn \to Sem$ are such that *Syn* is learnable from *Sem* we can use proposition 4.3.2 to construct the components of the *learn* transformation

$$learn : E^*(I_{Sem}) \to I_{Syn}$$

such that $eval \circ learn = 1_{E^*(I_{Sem})}$ and the language $\mathcal{L}$ is fully specified.

The extremely abstract nature of the language specification enables very general properties of languages and classes of languages to be studied and understood. Since the category

Mod(Syn) is a full reflexive sub-category of $[Syn, \mathbf{SET}]$ we can obtain a great deal of information about any class of languages without ever having to consider the details of an individual language, simply by studying what is known about the category $[Syn, \mathbf{SET}]$.

An example language specification using the sketch based model of language is given in appendix A. The sketch describing the semantics of *Toy* is an extremely large and complex object, much more so than, say, a domain theoretic semantics for the same language. The complexity arises in a number of ways. Firstly it is due to the fact that every object used in a sketch must be described *explicitly* from a few basic constants and operations, we cannot for example simply assume the existence of a particular product, we must construct that product. To paraphrase Barr and Wells [BaWe90] pp 172, "when all the girders and braces are exposed the true complexity of an object is revealed." This may be no bad thing as it forces the language specifier to consider exactly what properties he or she requires of each "girder", and certainly it is what allows us to define models of a sketch in an arbitrary category. Secondly the model of language based on sketches is essentially context free and programming languages are not. To explain, the meaning of the expression $x + 1$, where x is an identifier, depends on exactly what value x is bound to when it is evaluated. In other words, in different contexts or environments, $x + 1$ will have different meanings and so the language has a context sensitive[1] aspect.

Since the categorical model of language describes the evaluation function of a programming language as a natural transformation we know that the diagram below commutes.

$$\begin{tikzcd}
I_{Syn}(A) \arrow[r, "I_{Syn}(f)"] \arrow[d, "eval_A"'] & I_{Syn}(B) \arrow[d, "eval_B"] \\
E^*(I_{Sem})(A) \arrow[r, "E^*(I_{Sem})(f)"'] & E^*(I_{Sem})(B)
\end{tikzcd}$$

The fact that $eval_A$ is a function, combined with the commutativity of the diagram above forces $eval_A(x)$, where $x \in I_{Syn}(A)$, to have a constant value even when the term x is

[1] This use of the term *context sensitive* refers to the semantics of the language and should not be confused with the term context sensitive as used to describe a language whose *grammar* falls into type 1 of the Chomsky hierarchy.

embedded in the larger term $I_{Syn}(f)(x)$. This forces x to have the same meaning *regardless* of its context and thus forces the model to be essentially context free. To overcome this problem we have to introduce more cones and auxiliary operations to put terms like x into a correct context and then determine the value of this context, rather than just the value of x. This can add a great deal of complexity to the sketch as illustrated by the semantics of *Toy* in appendix section A.2. We shall return to this problem in section 7.3.

As it stands at the moment the model of language is unrealistic as it does not constrain the evaluation order of the language. In appendix A, for example, the semantics of *Toy* does not state whether *Toy* uses *call-by-value* or *call-by-need* semantics. This is a serious deficiency in any language specification method, but is potentially disastrous if the specification method is used to specify the semantics of a functional language like *Toy*. This information is missing because we use **SET** to model the sketch specifying the semantics of a programming language. Since we can construct our categorical model of language in any category we could rectify this omission by constructing a sketch of the semantics of *Toy* to be modelled in **DOM**, the category of domains and continuous functions. In moving to **DOM** we would, however, add to the complexity of the sketch without gaining any new insight into the technique for the construction of the self-interpreter. For this reason the work in this thesis has centred on models in **SET** only.

A third criticism of the model of language concerns the nature of the *learn* transformation. Since *learn* lacks any form of naturality condition its usefulness is strictly limited. There have been a number of attempts to weaken the naturality condition from the definition of natural transformation. Several such weaker conditions are contained in [Copp80] and these should certainly be explored.

In spite of these disadvantages the model of language we have constructed is not without merit. As we illustrated in section 5.4 we can include limits which are more complex than simple products. These limits can be included in both the syntax and the semantics, allowing us to capture the static semantics of a language in the specification of its syntax if we wish. This is a considerable enhancement over the algebraic model developed by Rus. At the cost of moving away from initial models of the semantics we could also include colimits in the models of programming language semantics and further simplify the specification of language semantics as Kortas and Quatrain demonstrate in [KoQu92].

Finally, in [BaWe90], *pp 171*, Barr and Wells state:

> "A deeper difference is that there are no distinguished nodes or operations in a sketch. The graph of the sketch for semigroups, for example, has three nodes, no one singled out, whereas in the usual definition of semigroup, the underlying set $S \ldots$ is singled out and other things are defined in terms of it. Similarly in the graph there are various arrows; c is just one of them."

In other words, "all the objects and operations specified by a sketch have equal status." This can cause a problem if we have something complex to specify. Just as it can be useful to have hidden sorts and operations in an algebraic specification it can also be useful to conceal parts of the inner structure of a sketch, either to specify the interface to a data sort or to formally draw attention to specific parts of a specification.

Although Barr and Wells are quite correct when they state that there are no distinguished nodes (or operations) in a single sketch, proposition 4.3.1 which is used here to specify the semantics of a programming language, in effect, provide a mechanism by which any nodes or edges of a sketch can be distinguished from the remaining parts of the sketch.

To explain, when we construct the sketch morphism $E : Syn \to Sem$ what we are actually doing is picking out some of the nodes and edges of *Sem* as being of special interest. Formally, because $E : Syn \to Sem$ allows us to use property 4.3.1 to construct the functor $E^* : \mathbf{Mod}(Sem) \to \mathbf{Mod}(Syn)$ we can use E^* to construct a model of *Syn*. From proposition 4.3.1 we know that this model of *Syn*, $E^*(I_{Sem}) : Syn \to \mathbf{SET}$ has the properties of the given model of *Sem*, $I_{Sem} : Sem \to \mathbf{SET}$, and can be used to specify the external interface of an abstract data type. We can therefore use proposition 4.3.1 to provide a mechanism to distinguish elements of a sketch as being of special interest.

7.1.2 The self-interpreter construction technique

Suppose we have a language

$$\mathcal{L} = \langle Sem, Syn, E : Syn \to Sem, learn : E^*(I_{Sem}) \to I_{Syn} \rangle$$

specified using the categorical model of language above. We can describe the function which an interpreter for this language computes using the expression

$$interpreter = learn \circ eval$$

where *eval* is the evaluation natural transformation

$$eval : I_{Syn} \overset{\cdot}{\rightarrow} E^{*}(I_{Sem})$$

used in the construction of *learn*.

If we then take a pair of transformations

$$\begin{aligned} encode &: I_{Syn} \rightarrow E^{*}(I_{Sem}) \\ decode &: E^{*}(I_{Sem}) \rightarrow I_{Syn} \end{aligned}$$

which describe an encoding of the syntax of $\mathcal{L}$ within the semantics of $\mathcal{L}$ we can construct a family of arrows

$$rep\_interpreter = encode \circ learn \circ eval \circ decode$$

which may also lie inside the semantics of $\mathcal{L}$, if $\mathcal{L}$ is powerful enough to describe its own interpreter. It is then a fairly simple matter to use the structure of *rep_interpreter* and theorem 6.1.1 to construct an $\mathcal{L}$ program, *$\mathcal{L}$-self-int*, which computes the function $rep\_interpreter_S$, where S is the start symbol of the grammar of $\mathcal{L}$. The program *$\mathcal{L}$-self-int* is the self-interpreter for the language $\mathcal{L}$.

This is the case because the specification of a programming language, however given, *must* in some sense be the description of an interpreter for that language. In the case of our categorical model of language this description is relatively clear as it exists in the form of the *eval* and *learn* transformations. As a result of this we have a fairly straightforward, if time consuming, process of symbol manipulation by which we can produce the function *rep_interpreter*. To transform the description of *rep_interpreter* into a self-interpreter is still, unfortunately, something of an art form. We rely on a programmer's intuition for the last step in the derivation of the self-interpreter.

As shown in section 6.3 the programmer is needed to supply the $\mathcal{L}$ syntactic forms of function definition. function application, and conditional. This is because there are simply too many different ways of providing these constructs in a programming language. For example, conditional can be realised as: if ...then, if ...then ...else, case, pattern matching, computed goto, etc.

These constructs all generate significantly different structures within the semantics of a programming language. To make matters worse there are very many different mechanisms that the language specifier may use within *Sem* to specify the semantics of any single one of these constructs, particularly if we allow the specifier to work with models of *Sem* in categories other than **SET**. This makes the construction of a general analysis procedure to recognise the structure of conditional at least extremely difficult. It may even make it impossible.

This begs the question: " of what value is the self-interpreter construction technique described here?" In my view it is not likely to lead to a completely automatic process, but it still has value because it does produce a complete description of the $\mathcal{L}$ self-interpreter as a function within the semantics of $\mathcal{L}$. Even if we cannot use the self-interpreter derivation process to construct the actual program code we can still use it to construct a design of this code which is so highly detailed that any programmer who knows how to define and call functions and express conditionals in $\mathcal{L}$ can write the code for *$\mathcal{L}$-self-int* with little need for extra intellectual effort.

7.2 Partial evaluators and interpreters

The true compiler generator system discussed in section 2.2 depends on a pair of assumptions, re-stated below.

Assumption 1. there is a technique which allows us to examine the specification of a computer language, $\mathcal{T}$, and from this specification, *calculate* a $\mathcal{T}$ program which implements *mix* for the language $\mathcal{T}$.

Assumption 2. given the specifications of two languages, $\mathcal{S}$, and $\mathcal{T}$, it is possible to derive an implementation of $\mathcal{S}$ in the form of an interpreter expressed as a $\mathcal{T}$ program.

The self-interpreter calculation technique was originally proposed as a useful *first step* towards justifying these assumptions. In sections 7.2.1 and 7.2.2 we discuss exactly how large this first step is.

7.2.1 Partial evaluators

There are two distinct problems which need to be solved before a technique for deriving a self-interpreter can be converted into a technique for deriving a partial evaluator.

Firstly, we must develop a method which allows us to derive the binding time analysis phase of the partial evaluator. The work of Launchbury [Laun90] using dependent sums to factorise domains into their static and dynamic values offers a promising starting point as it is a significant step towards the formalisation of the process of binding time analysis. It is, however, not at all clear how to incorporate this work into the categorical method developed here.

The second problem is the transformation of a self-interpreter into the function specialisation phase of a partial evaluator. In principle it should be possible to modify both the syntax and semantics of $\mathcal{L}$ by adding elements to represent dynamic values. Since dynamic values are not reduced by the function specialisation phase of a partial evaluator we would not need to alter the diagrams in the sketch describing the semantics of $\mathcal{L}$. The process used to derive the self-interpreter with the original semantics should now construct a function specialiser when applied to the altered semantics. This is because a function specialiser behaves *exactly* like an interpreter when it is working with static values, and suspends evaluation when it encounters a dynamic value. The original diagrams of the sketch of the semantics are therefore sufficient to deal with static values, and since dynamic values are not reduced, no new diagrams are required to describe their reduction. Unfortunately, without a complete description of the binding time analysis phase, we cannot begin to solve this second problem because we would have no clear idea of where a dynamic value could occur and so do not know where we need to add new values to the syntax and semantics of $\mathcal{L}$.

Clearly, there is still a very long way to go before the automatic derivation of a partial evaluator is a practical proposition. This is not the case with the second assumption as we explain below.

7.2.2 Interpreters

The fact that Assumption 2 fairs rather better than Assumption 1 is almost certainly due to the fact that the generalisation from self-interpreter to interpreter is much smaller than that from self-interpreter to partial evaluator.

Because of the close relationship between an interpreter and a self-interpreter the techniques used here to calculate a self-interpreter for language $\mathcal{S}$ can also be used to calculate an $\mathcal{S}$ interpreter in language $\mathcal{T}$ given sketch specifications of both $\mathcal{S}$ and $\mathcal{T}$.

The extra generality of the technique arises because the composition of *learn* and *eval* specifies the *function* to be computed by an $\mathcal{S}$ interpreter, not the $\mathcal{S}$ interpreter itself. To get from the function to the actual interpreter we need to construct a representation of the syntax of $\mathcal{S}$ as a data object within the semantics of $\mathcal{S}$. The interpreter is then produced by implementing the function *rep_interpreter*.

To recap: to represent the syntax of $\mathcal{S}$ within the semantics of $\mathcal{S}$ we require a pair of transformations

$$\begin{aligned} \mathit{encode} &: I^{\mathcal{S}}_{Syn} \rightarrow E^*(I^{\mathcal{S}}_{Sem}) \\ \mathit{decode} &: E^*(I^{\mathcal{S}}_{Sem}) \rightarrow I^{\mathcal{S}}_{Syn} \end{aligned}$$

with the property that $\mathit{decode} \circ \mathit{encode} = 1_{I^{\mathcal{S}}_{Syn}}$. The interpreter function is then embedded within this representation as

$$\mathit{interpret} = \mathit{encode} \circ \mathit{learn} \circ \mathit{eval} \circ \mathit{decode}$$

to move it within the semantics of $\mathcal{S}$.

To construct an $\mathcal{S}$ interpreter in the programming language $\mathcal{T}$ we can replace the representation functions *encode* and *decode* by a pair of transformations which represent the syntax of $\mathcal{S}$ as a datatype within the semantics of the arbitrary language $\mathcal{T}$:

$$encode_{\mathcal{T}} : I^{\mathcal{S}}_{Syn} \rightarrow E^{*}(I^{\mathcal{T}}_{Sem})$$
$$decode_{\mathcal{T}} : E^{*}(I^{\mathcal{T}}_{Sem}) \rightarrow I^{\mathcal{S}}_{Syn}.$$

This allows us to move the $\mathcal{S}$ interpreter function into the semantics of $\mathcal{T}$:

$$interpret = encode_{\mathcal{T}} \circ learn \circ eval \circ decode_{\mathcal{T}}$$

provided that $\mathcal{T}$ is sufficiently powerful to express the interpreter for the language $\mathcal{S}$.

The remainder of the interpreter calculation process then proceeds *exactly* as before. The only extra requirement necessary to use the technique for the calculation of a general $\mathcal{S}$ interpreter is that we can construct a representation of the $\mathcal{S}$ syntax within the semantics of $\mathcal{T}$.

7.2.3 The true compiler generator

The true compiler generator system discussed in section 2.2 is it seems still a long way off. We are still unable to derive the actual program code of *mix* for the target language and we cannot derive the code for *int*, the source interpreter, either. So have we actually advanced towards this goal at all? The answer to this question is, I believe, yes!

While we cannot, as yet, derive a $\mathcal{L}$ self-interpreter we can at least derive a definition of the function which an $\mathcal{L}$ self-interpreter computes. In section 7.2.2 above we indicated that we can even generalise this derivation process to derive the interpreter function for an $\mathcal{S}$ interpreter as a $\mathcal{T}$ program. To construct a true compiler generator of sorts we need only accomplish one more task.

We need to be able to derive the binding time analysis for the language $\mathcal{T}$. If we can achieve this we can construct a true compiler generator because we can at least derive the two functions below.

1. *rep_mix* the function which a $\mathcal{T}$ partial evaluator computes.
2. *rep_int* the function which a $\mathcal{T}$ implementation of an $\mathcal{S}$ interpreter computes.

Provided we use a standard language, $\mathcal{R}$, to describe these functions we can construct the compiler generator shown in figure 7.1.

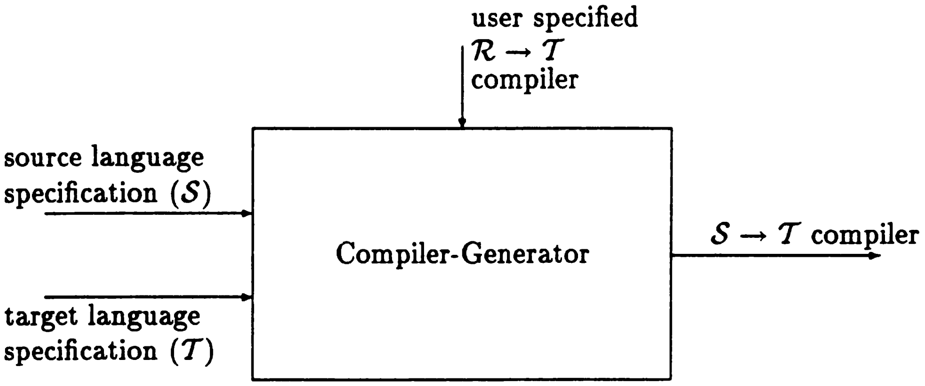

user specified
$\mathcal{R} \to \mathcal{T}$
compiler
source language
specification ($\mathcal{S}$)
target language
specification ($\mathcal{T}$)
Compiler-Generator
$\mathcal{S} \to \mathcal{T}$ compiler

Figure 7.1: A different true compiler generator system

This system accepts as input: the specification of the source language $\mathcal{S}$, the specification of the target language $\mathcal{T}$, and a compiler which translates from the internal representation $\mathcal{R}$ to the target language $\mathcal{T}$. The process discussed in the preceding chapters can then be used to derive the $\mathcal{R}$ representations of *rep_mix* and *rep_int*. The given $\mathcal{R} \to \mathcal{T}$ compiler is used to generate *mix* and *int* as $\mathcal{T}$ programs. We can then realise the $\mathcal{S} \to \mathcal{T}$ compiler as $mix[\![mix, int]\!]$.

While this is not the compiler generation system envisaged in chapter 2 is is still an improvement over the current situation because we do not need to specify the $\mathcal{S} \to \mathcal{T}$ relationship. The burden of proof on the compiler writer is therefore reduced since they only need to prove the $\mathcal{R} \to \mathcal{T}$ compiler correct rather than having to prove a different $\mathcal{S} \to \mathcal{T}$ relationship for each language $\mathcal{S}$.

Even without the ability to calculate the binding time analysis for the language $\mathcal{T}$ we can construct a compiler generator (shown in figure 7.2) which does not require the user to define the $\mathcal{S} \to \mathcal{T}$ relationship.

The operation of this system is similar to the system shown in figure 7.1 except that *mix* is supplied by the user rather than calculated as part of the generation process. With this last system the compiler writer's proof obligations are again increased as they must now prove *mix* correct in addition to the $\mathcal{R} \to \mathcal{T}$ compiler, but once again these proofs need only be

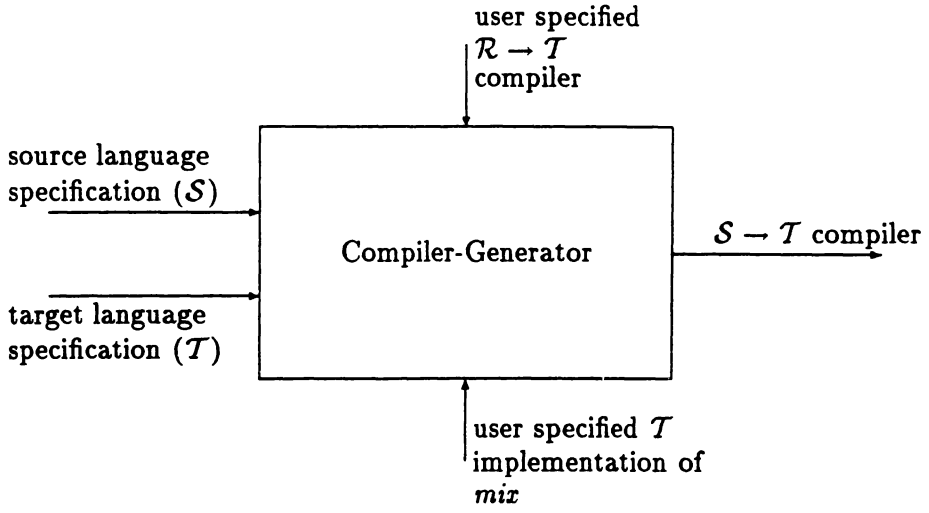
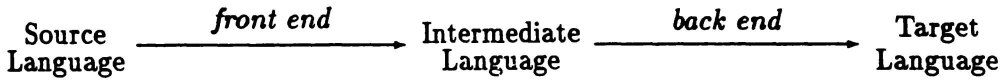

user specified
$\mathcal{R} \rightarrow \mathcal{T}$
compiler
source language
specification ($\mathcal{S}$)
target language
specification ($\mathcal{T}$)
Compiler-Generator
$\mathcal{S} \rightarrow \mathcal{T}$ compiler
user specified $\mathcal{T}$
implementation of
mix

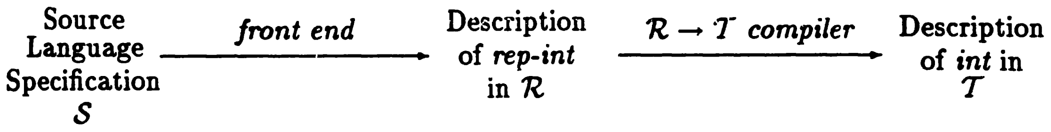
Figure 7.2: A final true compiler generator

carried out once for each target language $\mathcal{T}$.

There is an interesting parallel between the approach that both compiler generation systems above use to construct the $\mathcal{S} \rightarrow \mathcal{T}$ relationship (encoded in the program *int*) and the usual construction of a semantics directed compiler. Typically a semantics directed compiler is factorised into a front end which translates source language sentences into some universal intermediate language and a back end which translates from the intermediate language to the target language.



With the compiler generation systems outlined in this section we factorise the construction of the $\mathcal{S} \rightarrow \mathcal{T}$ relationship into a front and back end. The back end of this process is the $\mathcal{R} \rightarrow \mathcal{T}$ compiler provided by the user. We use a universal intermediate language $\mathcal{R}$ to describe the $\mathcal{S} \rightarrow \mathcal{T}$ relationship. The front end is the process for calculating the *rep-int* function described in this thesis.



In spite of these similarities there are two striking differences between the approach we propose and that of a semantics directed compiler.

Firstly a semantics directed compiler factorises the actual process of translating individual sentences from $\mathcal{S}$ into $\mathcal{T}$. Our approach operates as a higher level and factorises the construction of the $\mathcal{S} \rightarrow \mathcal{T}$ relationship.

Secondly, with a semantics directed compiler the front end is specific to a particular source programming language $\mathcal{S}$. In our approach the front end is universal.

7.2.4 Open questions

Here we examine some open questions about the true compiler generation technique. With the exception of question 4, all the questions below are related to the single question, "Do we really want compilers which are produced without human intervention?"

1. How much static computation is there in $mix(mix, int)$ when both mix and int are machine generated?

   This is a very important question since the power of partial evaluation depends on the ability to eliminate static computation. Consider the function

   $$f(x,y) \quad = \quad x+1+y$$

   Partial evaluation of the expression $f(4,y)$, where y is dynamic, produces the function

   $$f_4(y) \quad = \quad 5+y$$

   because the expression $x+1$ is static if x is static. If, on the other hand, f is expressed as

   $$f(x,y) \quad = \quad x+y+1$$

   then there is *no* static computation because $x+y$ is dynamic, unless both x and y are static, and so partial evaluation gives no improvement in the cost of computing f_4. In other words, the improvement gained by partial evaluation of a program depends on the style in which that program is written. We currently do not know whether the style of a machine generated *int* and *mix* will be "partial evaluation friendly" or not. The problem of transforming "partial evaluation unfriendly" functions into "friendly" ones is addressed in [HoHu90] where the technique of obtaining "free theorems" from a functions polymorphic type, developed by Wadler [Wadl89], is used to derive transformation rules. This technique is still in its infancy but seems like a good starting point for the related problem of synthesising "partial evaluation friendly" implementations of *int* and *mix* from language specifications expressed as sketches.

2. Can we ensure that a machine generated compiler will generate good quality object code?

   In some respects this question is related to the previous one. The more static computation contained in the expression $mix(mix, int)$ the better the object code generated by the compiler is likely to be. This, however, is not the only issue in the efficiency of the generated target code, for example, a compiler generated from an interpreter written using labels and "goto" to express its control flow is likely to generate better target code than the code generated if the interpreter uses recursion exclusively. This issue will need to be examined in detail before the proposed true compiler generation technique becomes viable for the generation of "industrial quality" compilers.

3. Is the technique applicable to imperative languages?

   The assumption that we are dealing with functional languages has been implicit throughout the preceding chapters. Although the categorical model of language is capable of specifying an imperative language in theory, this has not been done yet. In [KoQu92] Kortas and Quatrain give a specification of a subset of the pascal language, however the subset that they use avoids having to specify the store. It is in the specification of the store (and of assignment) that the most serious problems are to be encountered so whilst this work provides a useful insight into this use of sketches it leaves several questions unanswered. Any problems encountered in the synthesis of *mix* and *int* for a functional language are likely to be at least an order of magnitude worse for an imperative language. This question cannot be answered without a great deal more work.

4. Can sketches be implemented on a computer?

By using the techniques for constructing implementations of categorical constructs given in [RyBu88] we can certainly construct implementations of graph, diagram, cone, cocone, and model (functor) which would allow us to implement a specific sketch. This, however, is not what we mean by implementation of sketches on a computer because for each sketch we would need to construct its implementation by hand. When we ask "can sketches be implemented on a computer" we mean: can we construct an computer program which will, given an arbitrary sketch, automatically generate an implementation of the datatype specified by that sketch?

The answer to this question has to be a qualified *no.* Work has been undertaken in this area, for example Gray's work using *Mathematica* [Gray?], and the work of Yusop [Yuso91] using prolog. There are some quite serious problems with implementing sketches because, as with algebraic specifications, it is possible to use sketches to specify objects which are simply unimplementable on a computer, or to write sketches in a style which is non-constructive therefore not *directly* implementable on a computer. The problems above are fairly general problems with the implementation of formal specifications. Of the problems specific to the implementation of sketches the most obvious ones are caused by the fact that sketches can be modelled in an arbitrary category and do not always have an initial model. This problem even arises in **SET**. It is true that every FL sketch has a term model but this is not the case for every class of sketch. These problems will have to be addressed before a useful technique for the implementation of sketches can be developed. As a starting point we suggest the technique of *dynamic evaluation* developed by Duval and Raynaud [DuRa91] which provides an interesting and promising approach to this problem.

5. Do we really want compilers which are produced without human intervention?

This question is basically impossible to answer. In [Schm85] Schmidt briefly argues that a compiler generation system which requires more decisions from the implementor than is normal can be an advantage as the extra freedom of choice allows the implementor to orient the implementation towards the specific hardware and software available. It is indubitably true that when the user has to provide the implementation relationship such orientation is possible; what is less clear is that such orientation is not possible when the implementation relationship is automatically produced from the specifications of the source and target languages. Sketches would seem to be an ideal method of representing the source and target languages in this context since, by their very nature, every detail in a language specification must be stated explicitly and is therefore readily available to any software for calculating an implementation relationship. Due to the categorical nature of sketches this software could also have some extremely powerful analytical tools available to it. To the best of my knowledge nobody has examined these issues in any detail.

7.3 More science fiction: a better model of language?

There is one fundamental problem with the categorical model of language discussed above. The language specifications developed using this technique are far too large and unwieldy. The result of this is that a language specification using sketches is almost impossible to work with. The reasons for this complexity are illustrated in section 5.4.2, explained in section 7.1.1 and can be summarised in one sentence.

Sketches cannot be used to specify functions as objects.

If we could describe higher order objects using sketches, or some related tool, we could drastically reduce the complexity of the sketch describing the semantics of a programming language. A function is an extremely natural way to describe context sensitivity within a formal system. The context sensitive object becomes a function and its context can be passed into it as its argument. It is using this technique that a denotational semantics typically handles context sensitivity, the typical evaluation function for an expression looking something like

$$\mathcal{E} : \textit{Expression} \rightarrow \textit{Environment} \rightarrow \textit{Value}.$$

So the meaning of an expression, *exp*, is the function

$$\mathcal{E}(\textit{exp}) : \textit{Environment} \rightarrow \textit{Value}$$

which expects its context (the environment) and will only produce an actual value when given this context.

There are a number of extensions to the concept of a sketch which could possibly be used to solve this problem. The first of these extensions is the *form* which is described by Wells in [Well90]. A *form* is essentially a sketch but we have the additional ability to require diagrams to become instances of any essentially algebraic categorical construction when modelled. Although there is insufficient space to describe the approach in any detail here the basic idea is to provide a uniform method for defining the primitive types and operations on which the constructors specified within a sketch can operate. This allows the introduction of objects other than limits and colimits within the model of a sketch, in particular function objects can be specified for *forms* modelled in a cartesian closed category. Using this technique Wells hopes to be able to specify functional programming languages using sketches. A second extension to the concept of sketch which may allow function objects to be introduced is the ≪*trame*≫ described by Lair in [Lair87b].

The model of language developed in this thesis exists within an extremely general framework and is, as a result, easy to extend. The model is not tied to any specific procedure for the calculation of models of *Sem* so we can easily incorporate developments like dynamic evaluation [DuRa91] to increase the power of the model by allowing the sketch *Sem* to be modelled in new ways. We can even replace the sketch *Sem*, describing the semantics of a programming language by any graph theoretic structure, $\mathcal{X}$, containing cones and diagrams. This is because the key components on which the model of language is based are property 4.3.1 and the notion of learnability. Provided we can define a graph homomorphism $E : Syn \rightarrow \mathcal{X}$ which preserves diagrams and cones we know both that property 4.3.1 holds and that the notion of learnability is still applicable. Extension of the model of language to use either forms or ≪*trames*≫ is as a result not likely to present too many problems.

Finally, sketches themselves should not be dismissed. We have been able to construct a model of language which exceeds the power of Rus' algebraic model of language, which is itself a powerful language specification tool with an impressive compiler technology of its own. This technology is now available for study in a categorical universe. There are likely to be many useful discoveries still to be made. This dissertation has only scratched the surface.

Bibliography

[Back78] Backus J.; *Can programming be liberated from the von Neuman Style? A functional style and its algebra of programs*, Communications of the A.C.M., Vol 20, No 8, 1978, pp 613–641.

[Barr86] Barr M.; *Models of sketches*, Cahiers de Topologie Géomértie Différentielle Catégorique 27, 1986. pp 93–107.

[BaWe85] Barr M., Wells C.; *Toposes, triples and theories*, Springer-Verlag, 1985.

[BaWe90] Barr M., Wells C.; *Category theory for computing science*, Prentice Hall International Series in Computer Science, Ed C. A. R. Hoare, 1990.

[BaEh68] Bastiani A., Ehresmann C.; *Categories of sketched structures*, Cahiers de Topologie Géomértie Différentielle 10, pp 104–213, 1968.

[BHOS76] Beckman L., Haraldson A., Oskarsson, Ö., Sandewall E.; *A partial evaluator, and its use as a programming tool*, Artificial Intelligence, Vol 7, No 4, pp 319-357, 1976.

[BjEJ88] Bjørner D., Ershov A.P., Jones N.D. (Eds); *Partial evaluation and mixed computation*, Proceedings IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987, North-Holland, 1988.

[Boch78] Bochmann G. V.; *Compiler writing system for attribute grammars*, Computing Journal Vol 21, No 2, pp 144–148, 1978.

[Bond88] Bondorf A.; *Pattern matching in a self-applicable partial evaluator*, Unpublished draft, DIKU, University of Copenhagen, 1988.

[Bond90a] Bondorf A.; *Automatic autoprojection of higher order recursive equations*, ESOP '90, Copenhagen, Denmark, LNCS 432, pp 70–87, Jones (Ed), Springer-Verlag.

[Bond90b] Bondorf A.; *Compiling laziness by partial evaluation*, Functional Programming: Proceedings of the 1990 Glasgow Workshop, pp 11–21. 13–15th August 1990, Ullapool, Scotland, Peyton Jones S.L., Hutton G., Holst C.K. (Eds.), Springer Workshops in Computing, Springer-Verlag.

[BoDa80] Borceaux F., Day B.; *Universal algebra in closed categories*, Journal of pure and applied algebra 16, 1980, pp 133–147.

[Boul80] Boullier P.; *Generation automatique d'analysers syntaxiques avec rattrapage d'errors*, Journees Francophone sur la Production Assistee de Logiciel, Geneva, 1980.

[Burr70] Burroni A.; *Esquisse des catégories à limites et des quasi-topologies*, Esquisse Math. 5, 1970.

[BuLa69] Burstall R. M., Landin P. J.; *Programs and their proofs: an algebraic approach*, Machine Intelligence 4, 1969.

[Coll86] Collier P. D.; *Simple compiler correctness - a tutorial on the algebraic approach*, The Australian Computer Journal, Vol 18, No 3, pp128–135, 1986.

[Cons90] Consel C.; *Binding time analysis for higher order untyped functional languages*, ACM Conference on Lisp and Functional Languages, Nice, France, pp 264–272, June 1990.

[Copp80] Coppey L.; *Quelques problemes typiques concernant les graphes multiplicatifs*, Diagrammes, Vol 3, pp C1–C46.

[CoLa84] Coppey C., Lair C.; *Leçons de théorie des esquisses (I)*, Diagrammes, Vol 12, 1984.

[CoLa85] Coppey C., Lair C.; *Algébricité, monadicité, esquissabilité et non algébricité*, Diagrammes 13, 1985.

[Desc82] Deschamp Ph.; *PERLUETTE: a compiler producing system using abstract data types*, Proceedings of International Symposium on Programming, Turin, April 1982.

[DuRa91] Duval D., Raynaud J-C.; *Sketches and computation*, Rapport De Recherche RR 871-I-IMAG-123 LIFIA, LIFIA-IMAG, Institut National Polytechnique de Grenoble, 1991.

[Ehre67] Ehresmann C.; *Sur les structures algébriques*, CRAS, tome 264, pp 840–843, 1967.

[Ehre68] Ehresmann C. *Esquisses et types de structures algébriques*, Bull. Instit. Polit. **XIV**, pp 1–14, 1968.

[Ersh77] Ershov A.P.; *On the partial computation principle*, Information processing letters, Vol 6, No 2, 1977, pp 38-41.

[Ersh82] Ershov A.P.; *Mixed computation: potential applications and problems for study*, Theoretical computer science, Vol 18, No 1, 1982, pp 41–67.

[Folt69] Foltz F.; *Sur la catégorie des foncteurs dominés*, Cahiers de Topologie Géomértie Diff-érentielle 9, 2, 1969

[Futa71] Futamura Y.; *Partial evaluation of computation process - an approach to a compiler-compiler*, Systems, Computers, Controls, Vol 2, No 5, 1971, pp 45–50.

[Futa82] Futamura Y.; *Partial computation of programs*, Proc: RIMS Symposia on software science and engineering, Kyoto 1982, LNCS 147, pp 1–34, Springer-Verlag.

[Ganz79] Ganzinger H.; *Some principles for the development of compiler descriptions from denotational language definitions*, Technical University of Munich, Technical Report, 1979.

[Gold84] Goldblatt R.; *Topoi, the categorical analysis of logic*, Revised edition, Studies in logic and the foundations of mathematics, Vol 98, Eds: J. Barwise, D. Kaplan, H.J. Keisler, P. Suppes, A.S. Troelstra, 1984, North-Holland.

[Goma89] Gomard C.K.; *Higher order partial evaluation - HOPE for the lambda calculus*, Masters Thesis, DIKU, Department of Computer Science, University of Copenhagen, 1989.

[Gray87] Gray J.W.; *Categorical aspects of data type constructors*, Theoretical computer science, Vol 50, No 2, 1987, pp 103–135.

[Gray?] Gray J.W.; *Executable specifications for data-type constructors*, in preparation.

[GuLa80] Guitart R., Lair C.; *Calcul syntaxique des modèles et calcul des formules internes*, Diagrammes 4, (1980).

[HaRu76] Hatcher W. S., Rus T.; *Context-free algebras*, Journal of Cybernetics 6:2–3, 1976, pp 65–77.

[Higg63] Higgins P. J.; *Algebra with a scheme of operations*, Mathematische Nachrichten 27, 1963/64, pp 115–132.

[HoHu90] Holst C.K., Hughes J.; *Towards a binding-time analysis improvement for free*, Functional Programming: Proceedings of the 1990 Glasgow Workshop, pp 11–21, 13–15th August 1990, Ullapool, Scotland, Peyton Jones S.L., Hutton G., Holst C.K. (Eds.), Springer Workshops in Computing, Springer-Verlag.

[John78] Johnson S.C.; *Yacc: Yet another compiler compiler*, in the UNIX programmer's manual, Vol 2B, 1978.

[Jone88] Jones N.D.; *Automatic program specialization: A re-examination from basic principles*, In [BjEJ88], pp 225–282, 1988.

[JoSc80] Jones N.D., Schmidt D.A.; *Compiler generation from denotational semantics*, Semantics Directed Compiler Generation, Proceedings of a workshop Aarhus, January 1980, LNCS 94, pp 70–93, Springer-Verlag.

[JoSS85] Jones N.D., Sestoft P., Søndergaard H.; *An experiment in partial evaluation: the generation of a compiler generator*, Rewriting techniques and applications, J.P. Jouannaud (Ed), 1985, LNCS 202, pp 124–140, Springer-Verlag.

[JoSS87] Jones N.D., Sestoft P., Søndergaard H.; *Mix: A self-applicable partial evaluator for experiments in compiler generation*, Mathematical foundations of programming language semantics, Proceedings: 3rd workshop, Tulane University, New Orleans, Louisiana, 1987, LNCS 298, pp 386–413, Springer-Verlag.

[JoSS89] Jones N.D.. Sestoft P., Søndergaard H.; *Mix: A self-applicable partial evaluator for experiments in compiler generation*, Lisp and Symbolic Computation, Vol 2, No 1, 1989.

[Jorg90] Jørgensen J.; *Generating a pattern matching compiler by partial evaluation*, Functional Programming: Proceedings of the 1990 Glasgow Workshop, pp 11–21, 13–15th August 1990, Ullapool, Scotland, Peyton Jones S.L., Hutton G., Holst C.K. (Eds.), Springer Workshops in Computing, Springer-Verlag.

[KoQu92] Kortas S., Quatrain R.; *Modélisation de la syntaxe et la sémantique d'un language informatique par les esquisses*, Rapport du Séminaire d'Initiation a la Recherche (1991/1992), Ecols Centrale de Paris, Laboratoire de Mathématiques Appliquées, Multigraphie, Paris 1992.

[Lair75] Lair C.; *Etude générale de la Catégorie des esquisses*, Esquisses Mathematiques 23, pp 1–62, 1975.

[Lair87] Lair C.; *Categorrie qualifiables et catégories esquissables*, Diagrammes 17, 1987.

[Lair87b] Lair C.; *Trames et sémantiques catégoriques des systèmes trames*, Diagrammes 18, 1987.

[Laun88] Launchbury J.; *Projections for specialisation*, University of Glasgow Department of Computing Science, Technical Report 88/R8, 1988.

[Laun89] Launchbury J.; *Dependent sums express separation of binding times*, Functional Programming: Proceedings of the 1989 Glasgow Workshop, pp 238–253, 21–23 August 1989, Fraserburgh, Scotland, Davis K, Hughes J (Eds), Springer Workshops in Computing, Springer-Verlag.

[Laun90] Launchbury J.; *Projection factorisations in partial evaluation*, Ph.D. Thesis, Department of Computing Science, University of Glasgow, CSC 90/R2, 1990.

[Lorh82] Lorho B.; *The system DELTA and its derivatives*, Tools and Notions for Program Construction, D. Neel (Ed), Cambridge University Press, pp 306–317, 1982.

[Macl71] Mac Lane S.; *Categories for the working mathematician*, 1971, Springer-Verlag.

[MaBe85] Mazaher S., Berry D.M.; *Deriving a compiler from an operational semantics written in V.D.L.*, Computer Languages, Vol 10, No 2, pp 147–164, 1985.

[Morr73] Morris F.L; *Advice on structuring compilers and proving them correct*, Proceedings ACM Symposium on Principles of Programming Languages, Boston, 1973, pp 144–152.

[Moss76] Mosses P.D.; *Compiler generation using denotational semantics*, Mathematical foundations of Computer Science, 1976, LNCS 45, pp 463–441, Springer-Verlag.

[Moss79] Mosses P.D.; *A constructive approach to compiler correctness*, DAIMI IR-16, University of Aarhus, 1979.

[RaTu79] Raskovsky M., Turner R.; *Compiler generation and denotational semantics*, Fundamentals of Computation Theory, 1979.

[Reev87] Reeves A.C.; *An algebraic directed compiler generator using denotational semantics*, University of Stirling Department of Computing Science, Honours Project, May 1987.

[ReRa89] Reeves A.C., Rattray C.; *Sketching a constructive definition of 'mix'*, Functional Programming: Proceedings of the 1989 Glasgow Workshop, pp 118–132, 21–23 August 1989, Fraserburgh, Scotland, Davis K, Hughes J (Eds), Springer Workshops in Computing, Springer-Verlag.

[Roye86] Royer V.; *Transformations of denotational semantics in semantics directed compiler generation*, Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, USA, In: SIGPLAN Notices (USA), Vol 12, No 7, pp 68–73, 1986.

[Rus76] Rus T.; *Context-free algebra: a mathematical device for compiler specification*, Mathematical foundations of Computer Science, 1976, LNCS 45, pp 488–494, Springer-Verlag.

[Rus83] Rus T.; *T.I.C.S. System: a compiler generator*, University of Iowa Department of Computer Science technical report 83–08, 1983.

[RuHe84] Rus T., Herr F.B.; *An algebraic directed compiler generator*, University of Iowa Department of Computer Science, Technical Report 84–02, 1984.

[Rus85] Rus T.; *An inductive approach for program evaluation*, University of Iowa Department of Computer Science, Technical Report 85–02, 1985.

[Rus86] Rus T.; *An alternative to C.F. grammar for language specification*, Proceedings of IEEE conference on computer languages, Oct 27–30, 1986, Miami beach, Florida.

[Rus87] Rus T.; *An algebraic model for programming languages*, Computer Languages, Vol 12, No 3/4, pp173–195, 1987.

[Rus90] Rus T.; *Algebraic construction of a compiler*, University of Iowa Department of Computing Science, Technical Report 90–01, 1990.

[Rus92] Rus T.; *Algebraic construction of compilers*, The Unified Computation Laboratory, Rattray C.M.I. Clark R.G. (Eds), The Institute of Mathematics and its Applications, Oxford University Press, 1992.

[RyBu88] Rydeheard D.E., Burstall R.M.; *Computational category theory*, Prentice Hall international series in computer science, Ed: C.A.R. Hoare, 1988.

[ScSt71] Scott D.S., Strachey C.; *Toward a mathematical semantics for computer languages*, Oxford University Computing Laboratory, Programming Research Group, Technical Monograph PRG-6, 1971.

[Sest85] Sestoft P.; *The structure of a self-applicable partial evaluator*, Programs as data objects, Proceedings of a workshop, Denmark, 1985, LNCS 217, pp 236–256, Springer-Verlag.

[Stoy77] Stoy J.E.; *Denotational semantics*, MIT press, Cambridge, Mass., 1977.

[Schm85] Schmidt D.A.; *An implementation from a direct semantics definition*, Programs as data objects, Proceedings of a workshop, Denmark, 1985, LNCS 217, pp 222–235, Springer-Verlag.

[Schm86] Schmidt D.A.; *Denotational semantics - A methodology for language development*, Allyn and Bacon, 1986.

[ThWW80] Thatcher J.W., Wagner E.G., Wright J.B.; *More advice on structuring compilers and proving them correct*, Semantics Directed Compiler Generation, Proceedings of a workshop Aarhus, January 1980, LNCS 94, pp 165–188, Springer-Verlag.

[Turc80] Turchin V.F.; *Semantic definitions in REFAL and automatic production of compilers*, Semantics Directed Compiler Generation, Proceedings of a workshop Aarhus, January 1980, LNCS 94, pp 441–474, Springer-Verlag.

[TuNT82] Turchin V.F., Nirenberg R.M., Turchin D.V.; *Experiments with a supercompiler*, ACM Symposium on Lisp and Functional Programming, Pittsburgh Pennsylvania, pp 47–55, 1982.

[Turc85] Turchin V.F.; *Program transformation by supercompilation*. Programs as data objects, Proceedings of a workshop, Denmark, 1985, LNCS 217, pp 257–281, Springer-Verlag.

[Turc86] Turchin V.F.; *The concept of a supercompiler*, ACM-TOPLAS, Vol 8, No 3, 1986, pp 292–325.

[Vick86] Vickers T.N.; *Quokka: A translator generator using denotational semantics*, The Australian Computer Journal, Vol 18, No 1, pp 9–17, 1986.

[Wadl89] Wadler P.; *Theorems for free!*, Proceedings FPCA' 89, Fourth International Conference on Functional Programming Languages and Computer Architecture, London, September 1989, pp 347–359, Addison Wesley.

[Wait70] Waite W. M.; *The mobile programming system: STAGE2*, Communications of the A.C.M., Vol 13, No 7, pp 415–421, 1970.

[Wand80] Wand M.; *Different advice on structuring compilers and proving them correct*, Indiana University Department of Computer Science, Technical Report No 95, 1980.

[Wand85] Wand M.; *From interpreter to compiler: a representational derivation*, Programs as data objects, Proceedings of a workshop, Denmark, 1985, LNCS 217, pp 306–324, Springer-Verlag.

[WeBa87] Wells C., Barr M.; *The formal description of data types using sketches*, Mathematical foundations of programming language semantics, Proceedings: 3rd workshop, Tulane University, New Orleans, Louisiana. 1987, LNCS 298, pp 386–413, Springer-Verlag.

[Well90] Wells C.; *A generalisation of the concept of sketch.* Theoretical Computer Science, Vol 20, No 1, 1990, pp 159–178.

[Yuso91] Yusop N.I.; *Generating executable sketches in prolog*, MSc IT dissertation, Department of Computing Science, University of Stirling, 1991.

Appendix A

Example: A *Toy* Self-Interpreter

Toy is a typeless first order functional language. Functions can only be declared at the top level and have one implicit argument, named **arg** within the body of the function. Function names are therefore the only type of identifier which can exist in a *Toy* program. The only data objects which can be processed by a *Toy* program are natural numbers and binary trees. Natural numbers can be tested for equality using the = operator. Binary trees may be constructed using the (_,_) constructor and dismantled using the fst and snd operators. This restricted language is specified because it is amongst the simplest languages which are capable of expressing a self-interpreter.

A.1 The syntax of *Toy*

Using conventional methods the syntax of the *Toy* programming language is described by the following set of production rules:

$$\begin{aligned} \langle \mathit{ident} \rangle &\rightarrow \mathtt{x} \\ \langle \mathit{ident} \rangle &\rightarrow \mathtt{x}\langle \mathit{ident} \rangle \end{aligned}$$

$$\begin{aligned} \langle \mathit{num} \rangle &\rightarrow \mathtt{0} \\ \langle \mathit{num} \rangle &\rightarrow \mathtt{succ}(\langle \mathit{num} \rangle) \end{aligned}$$

$$\begin{array}{lcl}
\langle decs \rangle & \rightarrow & \varepsilon \\
\langle decs \rangle & \rightarrow & \langle ident \rangle \; \texttt{=} \; \langle exp \rangle \; \texttt{;} \; \langle decs \rangle
\end{array}$$

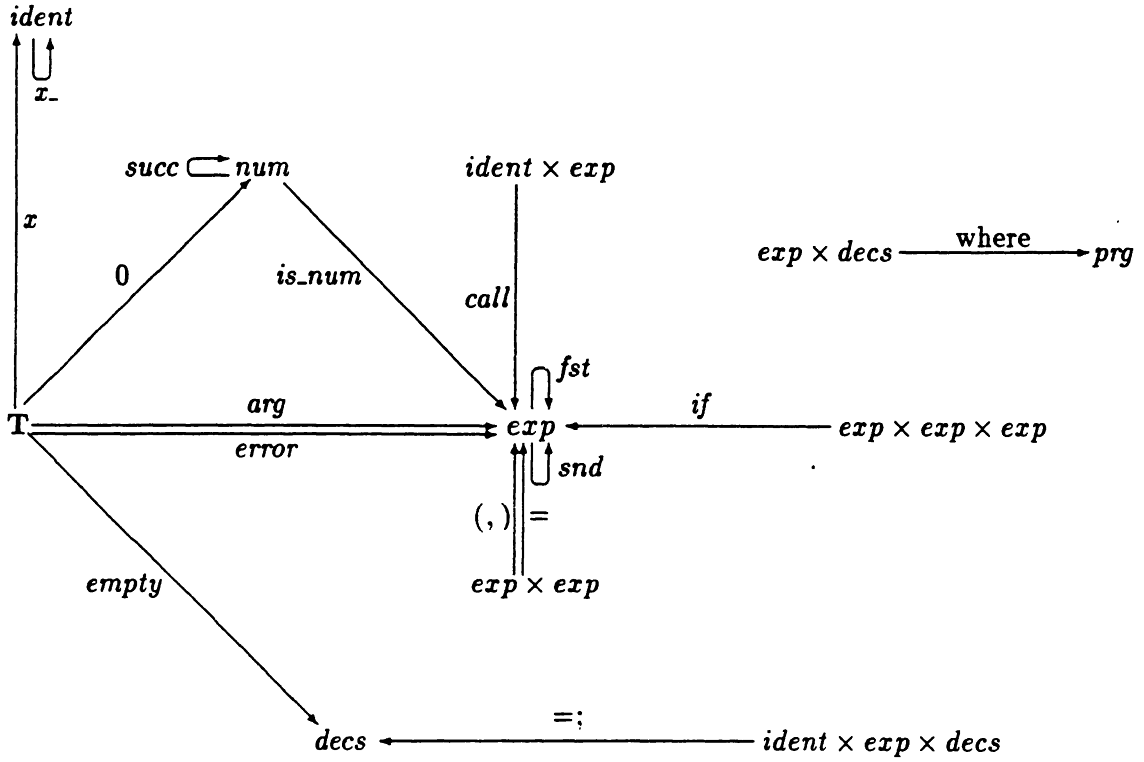
$$\begin{array}{lcl}
\langle exp \rangle & \rightarrow & \texttt{arg} \\
\langle exp \rangle & \rightarrow & \texttt{error} \\
\langle exp \rangle & \rightarrow & \langle num \rangle \\
\langle exp \rangle & \rightarrow & \texttt{fst(}\langle exp \rangle\texttt{)} \\
\langle exp \rangle & \rightarrow & \texttt{snd(}\langle exp \rangle\texttt{)} \\
\langle exp \rangle & \rightarrow & \texttt{(}\langle exp \rangle\texttt{,}\langle exp \rangle\texttt{)} \\
\langle exp \rangle & \rightarrow & \langle exp \rangle \; \texttt{=} \; \langle exp \rangle \\
\langle exp \rangle & \rightarrow & \texttt{if} \; \langle exp \rangle \; \texttt{then} \; \langle exp \rangle \; \texttt{else} \; \langle exp \rangle \\
\langle exp \rangle & \rightarrow & \texttt{call} \; \langle ident \rangle \langle exp \rangle
\end{array}$$

$$\langle prg \rangle \;\rightarrow\; \langle exp \rangle \; \texttt{where} \; \langle decs \rangle$$

The corresponding sketch Toy_{Syn}, which describes the phrases of the *Toy* language is shown below.

A.1.1 A sketch of the *Toy* syntax — Toy_{Syn}

Graph — G_{Syn}



Note: projection arrows omitted for clarity.

Cones — C_{Syn}

The six cones shown below are required in the sketch Toy_{Syn}. These cones are required to construct the production rules:

$$\begin{aligned} \langle ident \rangle &\rightarrow \mathtt{x} \\ \langle num \rangle &\rightarrow \mathtt{0} \\ \langle decs \rangle &\rightarrow \varepsilon \end{aligned}$$

$\langle exp\rangle \rightarrow \texttt{arg}$

$\langle exp\rangle \rightarrow \texttt{error}$

$\langle exp\rangle \rightarrow \texttt{(}\langle exp\rangle\texttt{,}\langle exp\rangle\texttt{)}$

$\langle exp\rangle \rightarrow \langle exp\rangle \texttt{ = } \langle exp\rangle$

$\langle exp\rangle \rightarrow \texttt{if } \langle exp\rangle \texttt{ then } \langle exp\rangle \texttt{ else } \langle exp\rangle$

$\langle exp\rangle \rightarrow \texttt{call } \langle ident\rangle\langle exp\rangle$

$\langle decs\rangle \rightarrow \langle ident\rangle \texttt{ = } \langle exp\rangle \texttt{ ; } \langle decs\rangle$

$\langle prg\rangle \rightarrow \langle exp\rangle \texttt{ where } \langle decs\rangle$

T

$$\begin{tikzcd}
 & exp \times exp \arrow[dl, "pr_{11}"'] \arrow[dr, "pr_{12}"] & \\
exp & & exp
\end{tikzcd}$$

$$\begin{tikzcd}
 & exp \times exp \times exp \arrow[dl, "pr_{13}"'] \arrow[d, "pr_{14}"] \arrow[dr, "pr_{15}"] & \\
exp & exp & exp
\end{tikzcd}$$

$$\begin{tikzcd}
 & ident \times exp \times decs \arrow[dl, "pr_{27}"'] \arrow[d, "pr_{28}"] \arrow[dr, "pr_{29}"] & \\
ident & exp & decs
\end{tikzcd}$$

$$\begin{tikzcd}
 & exp \times decs \arrow[dl, "pr_{47}"'] \arrow[dr, "pr_{48}"] & \\
exp & & decs
\end{tikzcd}$$

$$\begin{tikzcd}
 & ident \times exp \arrow[dl, "pr_{3}"'] \arrow[dr, "pr_{4}"] & \\
ident & & exp
\end{tikzcd}$$

Diagrams — D_{Syn}

Since the sketch Toy_{Syn} describes the syntax of a programming language and does not attempt to capture its static semantics the set of diagrams, D_{Syn}, is empty. $D_{Syn} = \emptyset$.

A.1.2 The initial model $I_{Syn} : Toy_{Syn} \to \mathbf{SET}$

$$
\begin{array}{lll}
I_{Syn}(\mathrm{T}) & = & \{\emptyset\} \\
I_{Syn}(num) & = & \bigcup I_{Syn}(num)_n, n \in \{0,1,2,\ldots\} \\
\quad \text{where } I_{Syn}(num)_0 & = & \{0\} \\
\quad\quad I_{Syn}(num)_n & = & \{\mathtt{succ}(x)\text{: } x \in I_{Syn}(num)_{n-1}\} \\
I_{Syn}(ident) & = & \bigcup I_{Syn}(ident)_n, n \in \{0,1,2,\ldots\} \\
\quad \text{where } I_{Syn}(ident)_0 & = & \{\mathtt{x}\} \\
\quad\quad I_{Syn}(ident)_n & = & \{\mathtt{x}y : y \in I_{Syn}(ident)_{n-1}\} \\
I_{Syn}(exp) & = & \bigcup I_{Syn}(exp)_n, n \in \{0,1,2,\ldots\} \\
\quad \text{where } I_{Syn}(exp)_0 & = & \{\mathtt{arg},\mathtt{error}\} \cup \{\mathtt{is\_num}(x)\text{: } x \in I_{Syn}(num)\} \\
\quad\quad I_{Syn}(exp)_n & = &
\end{array}
$$

$$
\begin{array}{l}
\{\mathtt{fst}(x)\text{: } x \in I_{Syn}(exp)_{n-1}\} \cup \{\mathtt{snd}(x)\text{: } x \in I_{Syn}(exp)_{n-1}\} \cup \\
\{(x,y)\text{: } (x,y) \in I_{Syn}(exp)_{n-1} \times I_{Syn}(exp)_{n-1}\} \cup \\
\{\mathtt{=}(x,y)\text{: } (x,y) \in I_{Syn}(exp)_{n-1} \times I_{Syn}(exp)_{n-1}\} \cup \\
\{\mathtt{if}(x,y,z)\text{: } (x,y,z) \in I_{Syn}(exp)_{n-1} \times I_{Syn}(exp)_{n-1} \times I_{Syn}(exp)_{n-1}\} \cup \\
\{\mathtt{call}(x,y)\text{: } x \in I_{Syn}(ident), y \in I_{Syn}(exp)_{n-1}\}
\end{array}
$$

$$
\begin{array}{lll}
I_{Syn}(ident \times exp) & = & I_{Syn}(ident) \times I_{Syn}(exp) \\
I_{Syn}(exp \times exp) & = & I_{Syn}(exp) \times I_{Syn}(exp) \\
I_{Syn}(exp \times exp \times exp) & = & I_{Syn}(exp) \times I_{Syn}(exp) \times I_{Syn}(exp) \\
I_{Syn}(decs) & = & \bigcup I_{Syn}(decs)_n, n \in \{0,1,2,\ldots\} \\
\quad \text{where } I_{Syn}(decs)_0 & = & \{\mathtt{empty}\} \\
\quad\quad I_{Syn}(decs)_n & = &
\end{array}
$$

$$
\{\mathtt{bind}(x,y,z)\text{: } x \in I_{Syn}(ident), y \in I_{Syn}(exp), z \in I_{Syn}(decs)_{n-1}\}
$$

$$
\begin{array}{lll}
I_{Syn}(exp \times decs) & = & I_{Syn}(exp) \times I_{Syn}(decs) \\
I_{Syn}(ident \times exp \times decs) & = & I_{Syn}(ident) \times I_{Syn}(exp) \times I_{Syn}(decs) \\
I_{Syn}(prg) & = & \{\mathtt{where}(x,y)\text{: } (x,y) \in I_{Syn}(exp) \times I_{Syn}(decs)\}
\end{array}
$$

$$
\begin{array}{lll}
I_{Syn}(0 : \mathrm{T} \to num) & = & \emptyset \mapsto 0 \\
I_{Syn}(succ : num \to num) & = & x \to \mathtt{succ}(x)
\end{array}
$$

$$\begin{aligned}
I_{Syn}(x : \mathbf{T} \to ident) &= \emptyset \mapsto \mathtt{x} \\
I_{Syn}(x\_ : ident \to ident) &= y \to \mathtt{x}y
\end{aligned}$$

$$\begin{aligned}
I_{Syn}(arg : \mathbf{T} \to exp) &= \emptyset \mapsto \mathtt{arg} \\
I_{Syn}(error : \mathbf{T} \to exp) &= \emptyset \mapsto \mathtt{error} \\
I_{Syn}(is\_num : num \to exp) &= x \to \mathtt{is\_num}(x) \\
I_{Syn}(call : ident \times exp \to exp) &= (x, y) \to \mathtt{call}(x\mathtt{,}y) \\
I_{Syn}(fst : exp \to exp) &= x \to \mathtt{fst}(x) \\
I_{Syn}(snd : exp \to exp) &= x \to \mathtt{snd}(x) \\
I_{Syn}(if : exp \times exp \times exp \to exp) &= (x, y, z) \to \mathtt{if}(x\mathtt{,}y\mathtt{,}z) \\
I_{Syn}(=: exp \times exp \to exp) &= (x, y) \to \mathtt{=}(x\mathtt{,}y) \\
I_{Syn}((,) : exp \times exp \to exp) &= (x, y) \to (x\mathtt{,}y)
\end{aligned}$$

$$\begin{aligned}
I_{Syn}(pr_{11} : exp \times exp \to exp) &= (x, y) \to x \\
I_{Syn}(pr_{12} : exp \times exp \to exp) &= (x, y) \to y
\end{aligned}$$

$$\begin{aligned}
I_{Syn}(pr_{13} : exp \times exp \times exp \to exp) &= (x, y, z) \to x \\
I_{Syn}(pr_{14} : exp \times exp \times exp \to exp) &= (x, y, z) \to y \\
I_{Syn}(pr_{15} : exp \times exp \times exp \to exp) &= (x, y, z) \to z
\end{aligned}$$

$$\begin{aligned}
I_{Syn}(empty : \mathbf{T} \to decs) &= \emptyset \mapsto \mathtt{empty} \\
I_{Syn}(=;: ident \times exp \times decs \to decs) &= (x, y, z) \to \mathtt{=;}(x\mathtt{,}y\mathtt{,}z)
\end{aligned}$$

$$\begin{aligned}
I_{Syn}(pr_{27} : ident \times exp \times decs \to ident) &= (x, y, z) \to x \\
I_{Syn}(pr_{28} : ident \times exp \times decs \to exp) &= (x, y, z) \to y \\
I_{Syn}(pr_{29} : ident \times exp \times decs \to decs) &= (x, y, z) \to z
\end{aligned}$$

$$\begin{aligned}
I_{Syn}(pr_{47} : exp \times decs \to exp) &= (x, y) \to x \\
I_{Syn}(pr_{48} : exp \times decs \to decs) &= (x, y) \to y
\end{aligned}$$

$$\begin{aligned}
I_{Syn}(pr_{3} : ident \times exp \to ident) &= (x, y) \to x \\
I_{Syn}(pr_{4} : ident \times exp \to exp) &= (x, y) \to y
\end{aligned}$$

$$I_{Syn}(where : exp \times decs \rightarrow prg) \quad = \quad (x, y) \rightarrow \texttt{where(}x\texttt{,}y\texttt{)}$$

A.2 The semantics of *Toy*

A.2.1 The sketch Toy_{Sem}

Graph — G_{Sem}

The graph G_{Sem} needed to describe the semantics of *Toy* is an extremely large and complex object. A pictorial representation of this graph is not practical so the graph is represented in tabular from below.

Nodes

$\mathbf{T}$

$decs$

$num \times num$

$ident \times \mathbf{T}$

$exp \times exp \times exp$

$num \times exp$

$num \times exp \times exp$

$\mathbf{T} \times decs$

$exp \times exp \times decs$

$ident$

$ident \times exp \times decs$

exp

$exp \times exp$

$exp \times \mathbf{T}$

$ident \times exp \times exp$

$ident \times num \times exp \times decs$

$num \times decs$

$exp \times exp \times exp \times decs$

num

$ident \times ident$

$ident \times decs$

$ident \times exp$

$\mathbf{T} \times exp$

$exp \times exp \times exp \times exp$

$ident \times ident \times exp \times decs$

$exp \times decs$

prg

Edges

$x : \mathbf{T} \rightarrow ident$

$x_{-} : ident \rightarrow ident$

$x_0 : ident \rightarrow ident$

$id_{ident} : ident \rightarrow ident$

$dispose_{ident} : ident \rightarrow \mathbf{T}$

$\mathit{empty} : \mathbf{T} \to \mathit{decs}$

$=;\, : \mathit{ident} \times \mathit{exp} \times \mathit{decs} \to \mathit{decs}$

$\mathit{id}_{\mathit{decs}} : \mathit{decs} \to \mathit{decs}$

$\mathit{pr}_{27} : \mathit{ident} \times \mathit{exp} \times \mathit{decs} \to \mathit{ident}$

$\mathit{pr}_{28} : \mathit{ident} \times \mathit{exp} \times \mathit{decs} \to \mathit{exp}$

$\mathit{pr}_{29} : \mathit{ident} \times \mathit{exp} \times \mathit{decs} \to \mathit{decs}$

$\langle \mathit{call} \circ \langle \mathit{pr}_{27}, \mathit{pr}_{28} \rangle, \mathit{pr}_{29} \rangle : \mathit{ident} \times \mathit{exp} \times \mathit{decs} \to \mathit{exp} \times \mathit{decs}$

$\langle \mathit{fetch} \circ \langle \mathit{pr}_{27}, \mathit{pr}_{29} \rangle, \mathit{apply} \circ \langle \mathit{pr}_{28}, \mathit{pr}_{29} \rangle, \mathit{pr}_{29} \rangle : \mathit{ident} \times \mathit{exp} \times \mathit{decs} \to \mathit{exp} \times \mathit{exp} \times \mathit{decs}$

$\langle \mathit{pr}_{27}, \mathit{pr}_{28} \rangle : \mathit{ident} \times \mathit{exp} \times \mathit{decs} \to \mathit{ident} \times \mathit{exp}$

$\langle \mathit{pr}_{27}, \mathit{pr}_{29} \rangle : \mathit{ident} \times \mathit{exp} \times \mathit{decs} \to \mathit{ident} \times \mathit{decs}$

$\langle \mathit{pr}_{28}, \mathit{pr}_{29} \rangle : \mathit{ident} \times \mathit{exp} \times \mathit{decs} \to \mathit{exp} \times \mathit{decs}$

$\mathit{pr}_1 : \mathit{ident} \times \mathit{ident} \to \mathit{ident}$

$\mathit{pr}_2 : \mathit{ident} \times \mathit{ident} \to \mathit{ident}$

$\langle x, x \rangle : \mathbf{T} \to \mathit{ident} \times \mathit{ident}$

$x_- \times x_0 : \mathit{ident} \times \mathit{ident} \to \mathit{ident} \times \mathit{ident}$

$x_0 \times x_- : \mathit{ident} \times \mathit{ident} \to \mathit{ident} \times \mathit{ident}$

$x_- \times x_- : \mathit{ident} \times \mathit{ident} \to \mathit{ident} \times \mathit{ident}$

$\mathit{dispose}_{\mathit{ident} \times \mathit{ident}} : \mathit{ident} \times \mathit{ident} \to \mathbf{T}$

$\mathit{same} : \mathit{ident} \times \mathit{ident} \to \mathit{num}$

$0 : \mathbf{T} \to \mathit{num}$

$\mathit{succ} : \mathit{num} \to \mathit{num}$

$\mathit{zero} : \mathit{num} \to \mathit{num}$

$\mathit{id}_{\mathit{num}} : \mathit{num} \to \mathit{num}$

$\mathit{dispose}_{\mathit{num}} : \mathit{num} \to \mathbf{T}$

$pr_9 : num \times num \to num$

$pr_{10} : num \times num \to num$

$\langle 0, 0\rangle : \mathbf{T} \to num \times num$

$succ \times zero : num \times num \to num \times num$

$zero \times succ : num \times num \to num \times num$

$succ \times succ : num \times num \to num \times num$

$dispose_{num \times num} : num \times num \to \mathbf{T}$

$equal : num \times num \to num$

$arg : \mathbf{T} \to exp$

$undef : \mathbf{T} \to exp$

$is\_num : num \to exp$

$fst : exp \to exp$

$snd : exp \to exp$

$id_{exp} : exp \to exp$

$dispose_{exp} : exp \to \mathbf{T}$

$(,) : exp \times exp \to exp$

$=: exp \times exp \to exp$

$if : exp \times exp \times exp \to exp$

$call : ident \times exp \to exp$

$is\_num \times is\_num : num \times num \to exp \times exp$

$is : exp \to prg$

$pr_7 : ident \times decs \to ident$

$pr_8 : ident \times decs \to decs$

$pr_5 : ident \times \mathbf{T} \to ident$

$pr_6 : ident \times \mathbf{T} \to \mathbf{T}$

$id_{exp} \times empty : ident \times \mathbf{T} \to ident \times decs$

$dispose_{ident \times \mathbf{T}} : ident \times \mathbf{T} \to \mathbf{T}$

$pr_{11} : exp \times exp \to exp$

$pr_{12} : exp \times exp \to exp$

$fst \times id_{exp} : exp \times exp \to exp \times exp$

$snd \times id_{exp} : exp \times exp \to exp \times exp$

$replace : exp \times exp \to exp$

$pr_3 : ident \times exp \to ident$

$pr_4 : ident \times exp \to exp$

$pr_{13} : exp \times exp \times exp \to exp$

$pr_{14} : exp \times exp \times exp \to exp$

$pr_{15} : exp \times exp \times exp \to exp$

$dispose_{exp \times exp \times exp} : exp \times exp \times exp \to \mathbf{T}$

$\langle (,) \circ \langle pr_{13}, pr_{14} \rangle, pr_{15} \rangle : exp \times exp \times \mathrm{exp} \to exp \times exp$

$\langle pr_{13}, (,) \circ \langle pr_{14}, pr_{15} \rangle \rangle : exp \times exp \times \mathrm{exp} \to exp \times exp$

$\langle = \circ \langle pr_{13}, pr_{14} \rangle, pr_{15} \rangle : exp \times exp \times \mathrm{exp} \to exp \times exp$

$U \times id_{exp} \times id_{exp} : exp \times exp \times exp \to exp \times exp \times exp$

$\langle pr_{13}, pr_{14} \rangle : exp \times exp \times exp \to exp \times exp$

$\langle pr_{13}, pr_{15} \rangle : exp \times exp \times exp \to exp \times exp$

$\langle pr_{14}, pr_{15} \rangle : exp \times exp \times exp \to exp \times exp$

$\langle replace \circ \langle pr_{13}, pr_{15} \rangle, replace \circ \langle pr_{14}, pr_{15} \rangle \rangle : exp \times exp \times exp \to exp \times exp$

$pr_{41} : exp \times \mathbf{T} \to exp$

$pr_{42} : exp \times \mathbf{T} \to \mathbf{T}$

$id_{exp} \times undef : exp \times \mathbf{T} \to exp \times exp$

$dispose_{exp \times \mathbf{T}} : exp \times \mathbf{T} \to \mathbf{T}$

$pr_{34} : \mathbf{T} \times exp \to \mathbf{T}$

$pr_{35} : \mathbf{T} \times exp \to exp$

$dispose_{\mathbf{T} \times exp} : \mathbf{T} \times exp \to \mathbf{T}$

$arg \times id_{exp} : \mathbf{T} \times exp \to exp \times exp$

$undef \times id_{exp} : \mathbf{T} \times exp \to exp \times exp$

$pr_{36} : num \times exp \rightarrow num$

$pr_{37} : num \times exp \rightarrow exp$

$is\_num \times id_{exp} : num \times exp \rightarrow exp \times exp$

$pr_{38} : ident \times exp \times exp \rightarrow ident$

$pr_{39} : ident \times exp \times exp \rightarrow exp$

$pr_{40} : ident \times exp \times exp \rightarrow exp$

$\langle call \circ \langle pr_{38}, pr_{39} \rangle, pr_{40} \rangle : ident \times exp \times exp \rightarrow exp \times exp$

$\langle pr_{38}, pr_{39} \rangle : ident \times exp \times exp \rightarrow ident \times exp$

$\langle pr_{39}, pr_{40} \rangle : ident \times exp \times exp \rightarrow ident \times exp$

$\langle pr_{38}, replace \circ \langle pr_{39}, pr_{40} \rangle \rangle : ident \times exp \times exp \rightarrow ident \times exp$

$pr_{19} : exp \times exp \times exp \times exp \rightarrow exp$

$pr_{20} : exp \times exp \times exp \times exp \rightarrow exp$

$pr_{21} : exp \times exp \times exp \times exp \rightarrow exp$

$pr_{22} : exp \times exp \times exp \times exp \rightarrow exp$

$\langle if \circ \langle pr_{19}, pr_{20}, pr_{21} \rangle, pr_{22} \rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp$

$\langle (,) \circ \langle pr_{19}, pr_{20} \rangle, pr_{21}, pr_{22} \rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp$

$\langle pr_{19}, pr_{20} \rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp$

$\langle pr_{19}, pr_{22} \rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp$

$\langle pr_{20}, pr_{22} \rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp$

$\langle pr_{21}, pr_{22} \rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp$

$\langle pr_{19}, pr_{20}, pr_{21} \rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp \times exp$

$$\langle replace \circ \langle pr_{19}, pr_{22} \rangle, replace \circ \langle pr_{20}, pr_{22} \rangle, replace \circ \langle pr_{21}, pr_{22} \rangle \rangle : exp \times exp \times exp \times exp \\ \rightarrow exp \times exp \times exp$$

$pr_{16} : num \times exp \times exp \rightarrow num$

$pr_{17} : num \times exp \times exp \rightarrow exp$

$pr_{18} : num \times exp \times exp \rightarrow exp$

$is\_num \circ zero \times id_{exp} \times id_{exp} : num \times exp \times exp \rightarrow exp \times exp \times exp$

$is\_num \circ succ \times id_{exp} \times id_{exp} : num \times exp \times exp \rightarrow exp \times exp \times exp$

$pr_{30} : ident \times num \times exp \times decs \rightarrow ident$

$pr_{31} : ident \times num \times exp \times decs \rightarrow num$

$pr_{32} : ident \times num \times exp \times decs \rightarrow exp$

$pr_{33} : ident \times num \times exp \times decs \rightarrow decs$

$id_{ident} \times zero \times id_{exp} \times id_{decs} : ident \times num \times exp \times decs \rightarrow ident \times num \times exp \times decs$

$id_{ident} \times succ \times id_{exp} \times id_{decs} : ident \times num \times exp \times decs \rightarrow ident \times num \times exp \times decs$

$\langle pr_{30}, pr_{33} \rangle : ident \times num \times exp \times decs \rightarrow ident \times decs$

$fetch : ident \times decs \rightarrow exp$

$get : ident \times num \times exp \times decs \rightarrow exp$

$pr_{23} : ident \times ident \times exp \times decs \rightarrow ident$

$pr_{24} : ident \times ident \times exp \times decs \rightarrow ident$

$pr_{25} : ident \times ident \times exp \times decs \rightarrow exp$

$pr_{26} : ident \times ident \times exp \times decs \rightarrow decs$

$\langle pr_{24}, pr_{25}, pr_{26} \rangle : ident \times ident \times exp \times decs \rightarrow ident \times exp \times decs$

$\langle pr_{23}, =; \circ \langle pr_{24}, pr_{25}, pr_{26} \rangle \rangle : ident \times ident \times exp \times decs \rightarrow ident \times exp \times decs$

$\langle pr_{23}, pr_{24} \rangle : ident \times ident \times exp \times decs \rightarrow ident \times ident$

$same \circ \langle pr_{23}, pr_{24} \rangle : ident \times ident \times exp \times decs \rightarrow num$

$$\langle pr_{23}, same \circ \langle pr_{23}, pr_{24} \rangle, pr_{25}, pr_{26} \rangle : ident \times ident \times exp \times decs$$
$$\rightarrow ident \times num \times exp \times decs$$

$pr_{43} : \mathbf{T} \times decs \rightarrow \mathbf{T}$

$pr_{44} : \mathbf{T} \times decs \rightarrow decs$

$dispose_{\mathbf{T} \times decs} : \mathbf{T} \times decs \rightarrow \mathbf{T}$

$arg \times id_{decs} : \mathbf{T} \times decs \rightarrow exp \times decs$

$undef \times id_{decs} : \mathbf{T} \times decs \rightarrow exp \times decs$

$pr_{45} : num \times decs \rightarrow num$

$pr_{46} : num \times decs \rightarrow decs$

$is\_num \times id_{decs} : num \times decs \rightarrow exp \times decs$

$pr_{47} : exp \times decs \rightarrow exp$

$pr_{48} : exp \times decs \rightarrow decs$

$fst \times id_{decs} : num \times decs \rightarrow exp \times decs$

$snd \times id_{decs} : num \times decs \rightarrow exp \times decs$

$apply : exp \times decs \rightarrow exp$

$pr_{49} : exp \times exp \times decs \rightarrow exp$

$pr_{50} : exp \times exp \times decs \rightarrow exp$

$pr_{51} : exp \times exp \times decs \rightarrow decs$

$\langle (,) \circ \langle pr_{49}, pr_{50} \rangle, pr_{51} \rangle : exp \times exp \times exp \rightarrow exp \times decs$

$\langle = \circ \langle pr_{49}, pr_{50} \rangle, pr_{51} \rangle : exp \times exp \times exp \rightarrow exp \times decs$

$\langle apply \circ \langle pr_{49}, pr_{51} \rangle, apply \circ \langle pr_{50}, pr_{51} \rangle \rangle : exp \times exp \times decs \rightarrow exp \times exp$

$\langle replace \circ \langle pr_{49}, pr_{50} \rangle, pr_{51} \rangle : exp \times exp \times decs \rightarrow exp \times decs$

$\langle pr_{49}, pr_{50} \rangle : exp \times exp \times decs \rightarrow exp \times exp$

$\langle pr_{49}, pr_{51} \rangle : exp \times exp \times decs \rightarrow exp \times decs$

$\langle pr_{50}, pr_{51} \rangle : exp \times exp \times decs \rightarrow exp \times decs$

$\langle apply \circ \langle pr_{49}, pr_{51} \rangle, apply \circ \langle pr_{50}, pr_{51} \rangle \rangle : exp \times exp \times decs \rightarrow exp \times exp$

$pr_{52} : exp \times exp \times exp \times decs \rightarrow exp$

$pr_{53} : exp \times exp \times exp \times decs \rightarrow exp$

$pr_{54} : exp \times exp \times exp \times decs \rightarrow exp$

$pr_{55} : exp \times exp \times exp \times decs \rightarrow decs$

$\langle if \circ \langle pr_{52}, pr_{53}, pr_{54} \rangle, pr_{55} \rangle : exp \times exp \times decs \rightarrow exp \times decs$

$\langle apply \circ \langle pr_{52}, pr_{55} \rangle, pr_{53}, pr_{54}, pr_{55} \rangle : exp \times exp \times exp \times decs \rightarrow exp \times exp \times exp \times decs$

$\langle pr_{52}, pr_{53}, pr_{54} \rangle : exp \times exp \times exp \times decs \rightarrow exp \times exp \times exp$

$\langle pr_{52}, pr_{55} \rangle : exp \times exp \times exp \times decs \rightarrow exp \times decs$

$\langle apply \circ \langle pr_{52}, pr_{55} \rangle, pr_{53}, pr_{54}, pr_{55} \rangle : exp \times exp \times exp \times decs \rightarrow exp \times exp \times exp \times decs$

$where : exp \times decs \rightarrow prg$

$is^{-1} : prg \rightarrow exp$

Cones — C_{Sem}

The twenty two cones required in the definition of the semantics of *Toy* are shown below.

$$\mathbf{T}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathit{ident} \arrow[dl, "pr_1"'] \arrow[dr, "pr_2"] & \\
\mathit{ident} & & \mathit{ident}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathit{exp} \arrow[dl, "pr_3"'] \arrow[dr, "pr_4"] & \\
\mathit{ident} & & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathbf{T} \arrow[dl, "pr_5"'] \arrow[dr, "pr_6"] & \\
\mathit{ident} & & \mathbf{T}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathit{decs} \arrow[dl, "pr_7"'] \arrow[dr, "pr_8"] & \\
\mathit{ident} & & \mathit{decs}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{num} \times \mathit{num} \arrow[dl, "pr_9"'] \arrow[dr, "pr_{10}"] & \\
\mathit{num} & & \mathit{num}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{11}"'] \arrow[dr, "pr_{12}"] & \\
\mathit{exp} & & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{13}"'] \arrow[d, "pr_{14}"] \arrow[dr, "pr_{15}"] & \\
\mathit{exp} & \mathit{exp} & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{num} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{16}"'] \arrow[d, "pr_{17}"] \arrow[dr, "pr_{18}"] & \\
\mathit{num} & \mathit{exp} & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& & & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dlll, "pr_{19}"'] \arrow[dl, "pr_{20}"] \arrow[dr, "pr_{21}"'] \arrow[drrr, "pr_{22}"] & & & \\
\mathit{exp} & & \mathit{exp} & & \mathit{exp} & & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& & & \mathit{ident} \times \mathit{ident} \times \mathit{exp} \times \mathit{decs} \arrow[dlll, "pr_{23}"'] \arrow[dl, "pr_{24}"] \arrow[dr, "pr_{25}"'] \arrow[drrr, "pr_{26}"] & & & \\
\mathit{ident} & & \mathit{ident} & & \mathit{exp} & & \mathit{decs}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathit{exp} \times \mathit{decs} \arrow[dl, "pr_{27}"'] \arrow[d, "pr_{28}"] \arrow[dr, "pr_{29}"] & \\
\mathit{ident} & \mathit{exp} & \mathit{decs}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathit{num} \times \mathit{exp} \times \mathit{decs} \arrow[dl, "pr_{30}"'] \arrow[d, "pr_{31}"] \arrow[dr, "pr_{32}"] \arrow[drr, "pr_{33}"] & & \\
\mathit{ident} & \mathit{num} & \mathit{exp} & \mathit{decs}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathbf{T} \times \mathit{exp} \arrow[dl, "pr_{34}"'] \arrow[dr, "pr_{35}"] & \\
\mathbf{T} & & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{num} \times \mathit{exp} \arrow[dl, "pr_{36}"'] \arrow[dr, "pr_{37}"] & \\
\mathit{num} & & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{38}"'] \arrow[d, "pr_{39}"] \arrow[dr, "pr_{40}"] & \\
\mathit{ident} & \mathit{exp} & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathbf{T} \arrow[dl, "pr_{41}"'] \arrow[dr, "pr_{42}"] & \\
\mathit{exp} & & \mathbf{T}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathbf{T} \times \mathit{decs} \arrow[dl, "pr_{43}"'] \arrow[dr, "pr_{44}"] & \\
\mathbf{T} & & \mathit{decs}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{num} \times \mathit{decs} \arrow[dl, "pr_{45}"'] \arrow[dr, "pr_{46}"] & \\
\mathit{num} & & \mathit{decs}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{decs} \arrow[dl, "pr_{47}"'] \arrow[dr, "pr_{48}"] & \\
\mathit{exp} & & \mathit{decs}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{exp} \times \mathit{decs} \arrow[dl, "pr_{49}"'] \arrow[d, "pr_{50}"] \arrow[dr, "pr_{51}"] & \\
\mathit{exp} & \mathit{exp} & \mathit{decs}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{decs} \arrow[dl, "pr_{52}"'] \arrow[d, "pr_{53}"] \arrow[dr, "pr_{54}"] \arrow[drr, "pr_{55}"] & & \\
\mathit{exp} & \mathit{exp} & \mathit{exp} & \mathit{decs}
\end{tikzcd}$$

Diagrams — D_{Sem}

The 10 diagrams below are used to describe the operation *same* : *ident* × *ident* → *num*. This operation is used to specify equality of identifiers.

$$\begin{tikzcd}
ident \arrow[loop below, "id_{ident}"]
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathbf{T} \arrow[dl, "x"'] \arrow[d, "\langle x, x\rangle"] \arrow[dr, "x"] & \\
ident & ident \times ident \arrow[l, "pr_1"] \arrow[r, "pr_2"'] & ident
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{T} \arrow[dr, "x"'] & ident \arrow[l, "dispose_{ident}"'] \arrow[d, "x_0"] \\
 & ident
\end{tikzcd}$$

$$\begin{tikzcd}
ident \arrow[d, "x_0"'] & ident \times ident \arrow[l, "pr_1"'] \arrow[r, "pr_2"] \arrow[d, "x_0 \times x_-"] & ident \arrow[d, "x_-"] \\
ident & ident \times ident \arrow[l, "pr_1"] \arrow[r, "pr_2"'] & ident
\end{tikzcd}$$

$$\begin{tikzcd}
ident \arrow[d, "x_-"'] & ident \times ident \arrow[l, "pr_1"'] \arrow[r, "pr_2"] \arrow[d, "x_- \times x_0"] & ident \arrow[d, "x_0"] \\
ident & ident \times ident \arrow[l, "pr_1"] \arrow[r, "pr_2"'] & ident
\end{tikzcd}$$

$$\begin{tikzcd}
ident \arrow[d, "x_-"'] & ident \times ident \arrow[l, "pr_1"'] \arrow[r, "pr_2"] \arrow[d, "x_- \times x_-"] & ident \arrow[d, "x_-"] \\
ident & ident \times ident \arrow[l, "pr_1"] \arrow[r, "pr_2"'] & ident
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{T} \arrow[r, "\langle x, x\rangle"] \arrow[dr, "0"'] & ident \times ident \arrow[d, "same"] \\
 & num
\end{tikzcd}$$

$$\begin{tikzcd}
ident \times ident \arrow[rr, "x \times x_-"] \arrow[d, "dispose_{ident \times ident}"] & & ident \times ident \arrow[d, "same"] \\
\mathbf{T} \arrow[r, "0"'] & num \arrow[r, "succ"'] & num
\end{tikzcd}$$

$$\begin{tikzcd}
ident \times ident \arrow[rr, "x_- \times x"] \arrow[d, "dispose_{ident \times ident}"] & & ident \times ident \arrow[d, "same"] \\
\mathbf{T} \arrow[r, "0"'] & num \arrow[r, "succ"'] & num
\end{tikzcd}$$

$$\begin{tikzcd}
ident \times ident \arrow[rr, "x_- \times x_-"] \arrow[dr, "same"'] & & ident \times ident \arrow[dl, "same"] \\
 & num &
\end{tikzcd}$$

Since the sets $I_{Sem}(ident)$ and $I_{Sem}(num)$ are, to all intents and purposes the same we require a similar set of 10 diagrams to describe the equality operation on numbers, $equal : num \times num \rightarrow num$.

$$\begin{tikzcd}
\mathit{num} \arrow[loop above, "\mathit{id}_{\mathit{num}}"]
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathbf{T} \arrow[dl, "0"'] \arrow[d, "\langle 0,0\rangle"] \arrow[dr, "0"] & \\
\mathit{num} & \mathit{num} \times \mathit{num} \arrow[l, "\mathit{pr}_9"] \arrow[r, "\mathit{pr}_{10}"'] & \mathit{num}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{T} \arrow[dr, "0"'] & \mathit{num} \arrow[l, "\mathit{dispose}_{\mathit{num}}"'] \arrow[d, "\mathit{zero}"] \\
& \mathit{num}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \arrow[d, "\mathit{zero}"'] & \mathit{num} \times \mathit{num} \arrow[l, "\mathit{pr}_9"'] \arrow[r, "\mathit{pr}_{10}"] \arrow[d, "\mathit{zero} \times \mathit{succ}"] & \mathit{num} \arrow[d, "\mathit{succ}"] \\
\mathit{num} & \mathit{num} \times \mathit{num} \arrow[l, "\mathit{pr}_9"] \arrow[r, "\mathit{pr}_{10}"'] & \mathit{num}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \arrow[d, "\mathit{succ}"'] & \mathit{num} \times \mathit{num} \arrow[l, "\mathit{pr}_9"'] \arrow[r, "\mathit{pr}_{10}"] \arrow[d, "\mathit{succ} \times \mathit{zero}"] & \mathit{num} \arrow[d, "\mathit{zero}"] \\
\mathit{num} & \mathit{num} \times \mathit{num} \arrow[l, "\mathit{pr}_9"] \arrow[r, "\mathit{pr}_{10}"'] & \mathit{num}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \arrow[d, "\mathit{succ}"'] & \mathit{num} \times \mathit{num} \arrow[l, "\mathit{pr}_9"'] \arrow[r, "\mathit{pr}_{10}"] \arrow[d, "\mathit{succ} \times \mathit{succ}"] & \mathit{num} \arrow[d, "\mathit{succ}"] \\
\mathit{num} & \mathit{num} \times \mathit{num} \arrow[l, "\mathit{pr}_9"] \arrow[r, "\mathit{pr}_{10}"'] & \mathit{num}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{T} \arrow[r, "\langle 0,0\rangle"] \arrow[dr, "0"'] & \mathit{num} \times \mathit{num} \arrow[d, "\mathit{equal}"] \\
& \mathit{num}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \times \mathit{num} \arrow[rr, "\mathit{zero} \times \mathit{succ}"] \arrow[d, "\mathit{dispose}_{\mathit{num} \times \mathit{num}}"] & & \mathit{num} \times \mathit{num} \arrow[d, "\mathit{equal}"] \\
\mathbf{T} \arrow[r, "0"'] & \mathit{num} \arrow[r, "\mathit{succ}"'] & \mathit{num}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \times \mathit{num} \arrow[rr, "\mathit{succ} \times \mathit{zero}"] \arrow[d, "\mathit{dispose}_{\mathit{num} \times \mathit{num}}"] & & \mathit{num} \times \mathit{num} \arrow[d, "\mathit{equal}"] \\
\mathbf{T} \arrow[r, "0"'] & \mathit{num} \arrow[r, "\mathit{succ}"'] & \mathit{num}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \times \mathit{num} \arrow[rr, "\mathit{succ} \times \mathit{succ}"] \arrow[dr, "\mathit{equal}"'] & & \mathit{num} \times \mathit{num} \arrow[dl, "\mathit{equal}"] \\
& \mathit{num} &
\end{tikzcd}$$

The next collection of diagrams describe the operation =: *exp* × *exp* → *exp*. This is the *Toy* language equality operator. Notice that 0 is the *True* value and that 1 is the *False* value.

$$\begin{tikzcd}
\mathit{num} \times \mathit{num} \arrow[r, "\mathit{equal}"] \arrow[d, "\mathit{is\_num} \times \mathit{is\_num}"'] & \mathit{num} \arrow[d, "\mathit{is\_num}"] \\
\mathit{exp} \times \mathit{exp} \arrow[r, "="'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \arrow[d, "\mathit{is\_num}"'] & \mathit{num} \times \mathit{num} \arrow[l, "\mathit{pr}_9"'] \arrow[r, "\mathit{pr}_{10}"] \arrow[d, "\mathit{is\_num} \times \mathit{is\_num}"] & \mathit{num} \arrow[d, "\mathit{is\_num}"] \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "\mathit{pr}_{11}"] \arrow[r, "\mathit{pr}_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \arrow[d, "{(\,,)}"'] & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[l, "{\langle \mathit{pr}_{13}, \mathit{pr}_{14} \rangle}"'] \arrow[d, "{\langle (\,,) \circ \langle \mathit{pr}_{13}, \mathit{pr}_{14} \rangle, \mathit{pr}_{15} \rangle}"'] \arrow[dr, "\mathit{pr}_{15}"] & \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "\mathit{pr}_{11}"] \arrow[r, "\mathit{pr}_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[r, "{\langle \mathit{pr}_{14}, \mathit{pr}_{15} \rangle}"] \arrow[dl, "\mathit{pr}_{13}"'] \arrow[d, "{\langle \mathit{pr}_{13}, (\,,) \circ \langle \mathit{pr}_{14}, \mathit{pr}_{15} \rangle \rangle}"] & \mathit{exp} \times \mathit{exp} \arrow[d, "{(\,,)}"] \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "\mathit{pr}_{11}"] \arrow[r, "\mathit{pr}_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "\mathit{pr}_{13}"'] \arrow[d, "{\langle \mathit{pr}_{13}, \mathit{pr}_{14} \rangle}"] \arrow[dr, "\mathit{pr}_{14}"] & \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "\mathit{pr}_{11}"] \arrow[r, "\mathit{pr}_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "\mathit{pr}_{14}"'] \arrow[d, "{\langle \mathit{pr}_{14}, \mathit{pr}_{15} \rangle}"] \arrow[dr, "\mathit{pr}_{15}"] & \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "\mathit{pr}_{11}"] \arrow[r, "\mathit{pr}_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[r, "{\langle (\,,) \circ \langle \mathit{pr}_{13}, \mathit{pr}_{14} \rangle, \mathit{pr}_{15} \rangle}"] \arrow[d, "\mathit{dispose}_{\mathit{exp} \times \mathit{exp} \times \mathit{exp}}"] & \mathit{exp} \times \mathit{exp} \arrow[d, "="] \\
\mathbf{T} \arrow[r, "\mathit{undef}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[r, "{\langle \mathit{pr}_{13}, (\,,) \circ \langle \mathit{pr}_{14}, \mathit{pr}_{15} \rangle \rangle}"] \arrow[d, "\mathit{dispose}_{\mathit{exp} \times \mathit{exp} \times \mathit{exp}}"] & \mathit{exp} \times \mathit{exp} \arrow[d, "="] \\
\mathbf{T} \arrow[r, "\mathit{undef}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathbf{T} \arrow[r, "\mathit{id}_{\mathit{exp}} \times \mathit{undef}"] \arrow[d, "\mathit{dispose}_{\mathit{exp} \times \mathbf{T}}"] & \mathit{exp} \times \mathit{exp} \arrow[d, "="] \\
\mathbf{T} \arrow[r, "\mathit{undef}"'] & \mathit{exp}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbf{T} \times \mathit{exp} \arrow[r, "\mathit{undef} \times \mathit{id}_{\mathit{exp}}"] \arrow[d, "\mathit{dispose}_{\mathbf{T} \times \mathit{exp}}"] & \mathit{exp} \times \mathit{exp} \arrow[d, "="] \\
\mathbf{T} \arrow[r, "\mathit{undef}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathbf{T} \arrow[dl, "pr_{41}"'] \arrow[r, "pr_{42}"] \arrow[dd, "\mathit{id}_{\mathit{exp}} \times \mathit{undef}"] & \mathbf{T} \arrow[dd, "\mathit{undef}"] \\
\mathit{exp} & & \\
& \mathit{exp} \times \mathit{exp} \arrow[ul, "pr_{11}"] \arrow[r, "pr_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathbf{T} \times \mathit{exp} \arrow[dl, "pr_{35}"'] \arrow[r, "pr_{34}"] \arrow[dd, "\mathit{undef} \times \mathit{id}_{\mathit{exp}}"] & \mathbf{T} \arrow[dd, "\mathit{undef}"] \\
\mathit{exp} & & \\
& \mathit{exp} \times \mathit{exp} \arrow[ul, "pr_{12}"] \arrow[r, "pr_{11}"'] & \mathit{exp}
\end{tikzcd}$$

In total 6 diagrams describe the behaviour of the *Toy* operations $\mathit{fst} : \mathit{exp} \rightarrow \mathit{exp}$ and $\mathit{snd} : \mathit{exp} \rightarrow \mathit{exp}$. Note that both *fst* and *snd* behave as identity when applied to a number.

$$\begin{tikzcd}
\mathbf{T} \arrow[r, "\mathit{undef}"] \arrow[dr, "\mathit{undef}"'] & \mathit{exp} \arrow[d, "\mathit{fst}"] \\
& \mathit{exp}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbf{T} \arrow[r, "\mathit{undef}"] \arrow[dr, "\mathit{undef}"'] & \mathit{exp} \arrow[d, "\mathit{snd}"] \\
& \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \arrow[r, "\mathit{is\_num}"] \arrow[dr, "\mathit{is\_num}"'] & \mathit{exp} \arrow[d, "\mathit{fst}"] \\
& \mathit{exp}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathit{num} \arrow[r, "\mathit{is\_num}"] \arrow[dr, "\mathit{is\_num}"'] & \mathit{exp} \arrow[d, "\mathit{snd}"] \\
& \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \arrow[r, "(\,,)"] \arrow[dr, "pr_{11}"'] & exp \arrow[d, "fst"] \\
& exp
\end{tikzcd}
\qquad
\begin{tikzcd}
exp \times exp \arrow[r, "(\,,)"] \arrow[dr, "pr_{12}"'] & exp \arrow[d, "snd"] \\
& exp
\end{tikzcd}$$

We require 9 diagrams to specify the behaviour of *if.* these diagrams are shown below. Note that the *True* value is 0 and the *false* value is any non zero value including a pair constructed by (,).

$$\begin{tikzcd}
exp \times exp \times exp \arrow[r, "U \times id_{exp} \times id_{exp}"] \arrow[d, "dispose_{exp \times exp \times exp}"] & exp \times exp \times exp \arrow[d, "if"] \\
\mathbf{T} \arrow[r, "undef"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \times exp \arrow[r, "is\_num \circ zero \times id_{exp} \times id_{exp}"] \arrow[dr, "pr_{17}"'] & exp \times exp \times exp \arrow[d, "if"] \\
& exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \times exp \arrow[r, "is\_num \circ succ \times id_{exp} \times id_{exp}"] \arrow[dr, "pr_{18}"'] & exp \times exp \times exp \arrow[d, "if"] \\
& exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \times exp \times exp \arrow[r, "\langle (\,,) \circ \langle pr_{19}, pr_{20} \rangle, pr_{21}, pr_{22} \rangle"] \arrow[dr, "pr_{22}"'] & exp \times exp \times exp \arrow[d, "if"] \\
& exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \arrow[d, "dispose_{exp}"'] & exp \times exp \times exp \arrow[l, "pr_{13}"'] \arrow[dd, "U \times id_{exp} \times id_{exp}"] \arrow[dr, "pr_{14}"] \arrow[drr, "pr_{15}"] & & \\
\mathbf{T} \arrow[d, "undef"'] & & exp & exp \\
exp & exp \times exp \times exp \arrow[l, "pr_{13}"] \arrow[ur, "pr_{14}"'] \arrow[urr, "pr_{15}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
num \arrow[d, "zero"'] & num \times exp \times exp \arrow[l, "pr_{16}"'] \arrow[dd, "is\_num \circ zero \times id_{exp} \times id_{exp}"] \arrow[dr, "pr_{17}"] \arrow[drr, "pr_{18}"] & & \\
num \arrow[d, "is\_num"'] & & exp & exp \\
exp & exp \times exp \times exp \arrow[l, "pr_{13}"] \arrow[ur, "pr_{14}"'] \arrow[urr, "pr_{15}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
num \arrow[d, "succ"'] & num \times exp \times exp \arrow[l, "pr_{16}"'] \arrow[dd, "is\_num \circ succ \times id_{exp} \times id_{exp}"] \arrow[dr, "pr_{17}"] \arrow[drr, "pr_{18}"] & & \\
num \arrow[d, "is\_num"'] & & exp & exp \\
exp & exp \times exp \times exp \arrow[l, "pr_{13}"] \arrow[ur, "pr_{14}"'] \arrow[urr, "pr_{15}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & exp \times exp \times exp \times exp \arrow[dl, "pr_{19}"'] \arrow[d, "\langle pr_{19}, pr_{20} \rangle"] \arrow[dr, "pr_{20}"] & \\
exp & exp \times exp \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \arrow[dd, "(\,,\,)"'] & exp \times exp \times exp \times exp \arrow[l, "\langle pr_{19}, pr_{20} \rangle"'] \arrow[dd, "\langle (\,,\,) \circ \langle pr_{19}, pr_{20} \rangle, pr_{21}, pr_{22} \rangle"] \arrow[dr, "pr_{21}"] \arrow[drr, "pr_{22}"] & & \\
 & & exp & exp \\
exp & exp \times exp \times exp \arrow[l, "pr_{13}"] \arrow[ur, "pr_{14}"'] \arrow[urr, "pr_{15}"'] & &
\end{tikzcd}$$

Below are the diagrams necessary to describe the operation *apply* : *exp* × *decs* → *exp* and its auxiliary functions. The purpose of this operation is to specify function application. Its operation is as follows.

1. When a function call exists as a sub-expression it is replaced by the function body bound to the function in the *decs* part of its argument. This is the purpose of the *fetch* operation described below.

2. All occurrences of the sub-expression *arg* within the body of the function found at 1 above are replaced by the function argument from the call of the function in 1 above.

3. Together steps 1 and 2 produce an expression whose evaluation is described by the remaining diagrams of the sketch.

$$\begin{tikzcd}
\mathbf{T} \times \mathit{decs} \arrow[r, "\mathit{arg} \times \mathit{id}_{\mathit{decs}}"] \arrow[d, "\mathit{dispose}_{\mathbf{T} \times \mathit{decs}}"] & \mathit{exp} \times \mathit{decs} \arrow[d, "\mathit{apply}"] \\
\mathbf{T} \arrow[r, "\mathit{arg}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{T} \times \mathit{decs} \arrow[r, "\mathit{undef} \times \mathit{id}_{\mathit{decs}}"] \arrow[d, "\mathit{dispose}_{\mathbf{T} \times \mathit{decs}}"] & \mathit{exp} \times \mathit{decs} \arrow[d, "\mathit{apply}"] \\
\mathbf{T} \arrow[r, "\mathit{undef}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \times \mathit{decs} \arrow[r, "\mathit{is\_num} \times \mathit{id}_{\mathit{decs}}"] \arrow[d, "\mathit{pr}_{45}"'] & \mathit{exp} \times \mathit{decs} \arrow[d, "\mathit{apply}"] \\
\mathit{num} \arrow[r, "\mathit{is\_num}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{decs} \arrow[r, "\mathit{fst} \times \mathit{id}_{\mathit{decs}}"] \arrow[d, "\mathit{apply}"'] & \mathit{exp} \times \mathit{decs} \arrow[d, "\mathit{apply}"] \\
\mathit{exp} \arrow[r, "\mathit{fst}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{decs} \arrow[r, "\mathit{snd} \times \mathit{id}_{\mathit{decs}}"] \arrow[d, "\mathit{apply}"'] & \mathit{exp} \times \mathit{decs} \arrow[d, "\mathit{apply}"] \\
\mathit{exp} \arrow[r, "\mathit{snd}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \times \mathit{decs} \arrow[r, "\langle = \circ \langle \mathit{pr}_{49}, \mathit{pr}_{50} \rangle, \mathit{pr}_{51} \rangle"] \arrow[d, "{\begin{array}{l} \langle \mathit{apply} \circ \langle \mathit{pr}_{49}, \mathit{pr}_{51} \rangle, \\ \mathit{apply} \circ \langle \mathit{pr}_{50}, \mathit{pr}_{51} \rangle \rangle \end{array}}"] & \mathit{exp} \times \mathit{decs} \arrow[d, "\mathit{apply}"] \\
\mathit{exp} \times \mathit{exp} \arrow[r, "="'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \times \mathit{decs} \arrow[r, "\langle (,) \circ \langle pr_{49}, pr_{50} \rangle, pr_{51} \rangle"] \arrow[d, "{\begin{array}{l} \langle \mathit{apply} \circ \langle pr_{49}, pr_{51} \rangle, \\ \mathit{apply} \circ \langle pr_{50}, pr_{51} \rangle \rangle \end{array}}"'] & \mathit{exp} \times \mathit{decs} \arrow[d, "\mathit{apply}"] \\
\mathit{exp} \times \mathit{exp} \arrow[r, "(,)"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{decs} \arrow[rr, "\langle \mathit{if} \circ \langle pr_{52}, pr_{53}, pr_{54} \rangle, pr_{55} \rangle"] \arrow[d, "\langle \mathit{apply} \circ \langle pr_{52}, pr_{55} \rangle, pr_{53}, pr_{54}, pr_{55} \rangle"'] & & \mathit{exp} \times \mathit{decs} \arrow[d, "\mathit{apply}"] \\
\mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[r, "\langle \mathit{if} \circ \langle pr_{52}, pr_{53}, pr_{54} \rangle, pr_{55} \rangle"] & \mathit{exp} \times \mathit{decs} \arrow[r, "\mathit{apply}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{ident} \times \mathit{exp} \times \mathit{decs} \arrow[r, "\langle \mathit{call} \circ \langle pr_{27}, pr_{28} \rangle, pr_{29} \rangle"] \arrow[d, "{\begin{array}{l} \langle \mathit{fetch} \circ \langle pr_{27}, pr_{29} \rangle, \\ \mathit{apply} \circ \langle pr_{28}, pr_{29} \rangle, \\ pr_{29} \rangle \end{array}}"'] & \mathit{exp} \times \mathit{decs} \arrow[dd, "\mathit{apply}"] \\
\mathit{exp} \times \mathit{exp} \times \mathit{decs} \arrow[d, "\langle \mathit{replace} \circ \langle pr_{49}, pr_{50} \rangle, pr_{51} \rangle"'] & \\
\mathit{exp} \times \mathit{decs} \arrow[r, "\mathit{apply}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{T} \arrow[dd, "\mathit{arg}"'] & \mathbf{T} \times \mathit{decs} \arrow[l, "pr_{43}"'] \arrow[dd, "\mathit{arg} \times \mathit{id}_{\mathit{decs}}"'] \arrow[dr, "pr_{44}"] & \\
& & \mathit{decs} \\
\mathit{exp} & \mathit{exp} \times \mathit{decs} \arrow[l, "pr_{47}"] \arrow[ur, "pr_{48}"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbf{T} \arrow[dd, "\mathit{undef}"'] & \mathbf{T} \times \mathit{decs} \arrow[l, "pr_{43}"'] \arrow[dd, "\mathit{undef} \times \mathit{id}_{\mathit{decs}}"'] \arrow[dr, "pr_{44}"] & \\
& & \mathit{decs} \\
\mathit{exp} & \mathit{exp} \times \mathit{decs} \arrow[l, "pr_{47}"] \arrow[ur, "pr_{48}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \arrow[dd, "\mathit{is\_num}"'] & \mathit{num} \times \mathit{decs} \arrow[l, "pr_{45}"'] \arrow[dd, "\mathit{is\_num} \times \mathit{id}_{\mathit{decs}}"'] \arrow[dr, "pr_{46}"] & \\
& & \mathit{decs} \\
\mathit{exp} & \mathit{exp} \times \mathit{decs} \arrow[l, "pr_{47}"] \arrow[ur, "pr_{48}"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathit{exp} \arrow[dd, "\mathit{fst}"'] & \mathit{exp} \times \mathit{decs} \arrow[l, "pr_{47}"'] \arrow[dd, "\mathit{fst} \times \mathit{id}_{\mathit{decs}}"'] \arrow[dr, "pr_{48}"] & \\
& & \mathit{decs} \\
\mathit{exp} & \mathit{exp} \times \mathit{decs} \arrow[l, "pr_{47}"] \arrow[ur, "pr_{48}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \arrow[dd, "\mathit{snd}"'] & \mathit{exp} \times \mathit{decs} \arrow[l, "pr_{47}"'] \arrow[dd, "\mathit{snd} \times \mathit{id}_{\mathit{decs}}"'] \arrow[dr, "pr_{48}"] & \\
& & \mathit{decs} \\
\mathit{exp} & \mathit{exp} \times \mathit{decs} \arrow[l, "pr_{47}"] \arrow[ur, "pr_{48}"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \arrow[dd, "="'] & \mathit{exp} \times \mathit{exp} \times \mathit{decs} \arrow[l, "\langle pr_{49}, pr_{50} \rangle"'] \arrow[dd, "\langle = \circ \langle pr_{49}, pr_{50} \rangle, pr_{51} \rangle"'] \arrow[dr, "pr_{51}"] & \\
& & \mathit{decs} \\
\mathit{exp} & \mathit{exp} \times \mathit{decs} \arrow[l, "pr_{47}"] \arrow[ur, "pr_{48}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \arrow[dd, "{(\,,\,)}"'] & exp \times exp \times decs \arrow[l, "{\langle pr_{49}, pr_{50}\rangle}"'] \arrow[dd, "{\langle (\,,\,) \circ \langle pr_{49}, pr_{50}\rangle, pr_{51}\rangle}"'] \arrow[dr, "pr_{51}"] & \\
 & & decs \\
exp & exp \times decs \arrow[l, "pr_{47}"] \arrow[ur, "pr_{48}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & exp \times exp \times decs \arrow[dl, "pr_{49}"'] \arrow[d, "{\langle pr_{49}, pr_{50}\rangle}"] \arrow[dr, "pr_{50}"] & \\
exp & exp \times exp \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
 & exp \times exp \times exp \times decs \arrow[dl, "pr_{52}"'] \arrow[dd, "{\langle pr_{52}. pr_{53}, pr_{54}\rangle}"'] \arrow[dr, "pr_{53}"'] \arrow[drr, "pr_{54}"] & & \\
exp & & exp & exp \\
 & exp \times exp \times exp \arrow[ul, "pr_{13}"] \arrow[ur, "pr_{14}"] \arrow[urr, "pr_{15}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \times exp \arrow[dd, "if"'] & exp \times exp \times exp \times decs \arrow[l, "{\langle pr_{52}, pr_{53}, pr_{54}\rangle}"'] \arrow[dd, "{\langle if \circ \langle pr_{52}, pr_{53}, pr_{54}\rangle, pr_{55}\rangle}"'] \arrow[dr, "pr_{55}"] & \\
 & & decs \\
exp & exp \times decs \arrow[l, "pr_{47}"] \arrow[ur, "pr_{48}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & exp \times exp \times exp \times decs \arrow[dl, "pr_{52}"'] \arrow[d, "{\langle pr_{52}, pr_{55}\rangle}"] \arrow[dr, "pr_{55}"] & \\
exp & exp \times decs \arrow[l, "pr_{47}"] \arrow[r, "pr_{48}"'] & decs
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \arrow[dd, "apply"'] & exp \times exp \times exp \times decs \arrow[l, "{\langle pr_{52}, pr_{55}\rangle}"'] \arrow[dd, "{\langle apply \circ \langle pr_{52}, pr_{55}\rangle, pr_{53}, pr_{54}, pr_{55}\rangle}"'] \arrow[dr, "pr_{53}"] \arrow[drr, "pr_{54}"] \arrow[drrr, "pr_{55}"] & & & \\
 & & exp & exp & decs \\
exp & exp \times exp \times exp \times decs \arrow[l, "pr_{52}"] \arrow[ur, "pr_{53}"'] \arrow[urr, "pr_{54}"'] \arrow[urrr, "pr_{55}"'] & & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & ident \times exp \times decs \arrow[dl, "pr_{27}"'] \arrow[d, "\langle pr_{27}, pr_{28} \rangle"] \arrow[dr, "pr_{29}"] & \\
ident & ident \times decs \arrow[l, "pr_7"] \arrow[r, "pr_8"'] & decs
\end{tikzcd}
\qquad
\begin{tikzcd}
 & ident \times exp \times decs \arrow[dl, "pr_{28}"'] \arrow[d, "\langle pr_{28}, pr_{29} \rangle"] \arrow[dr, "pr_{29}"] & \\
exp & exp \times decs \arrow[l, "pr_{47}"] \arrow[r, "pr_{48}"'] & decs
\end{tikzcd}$$

$$\begin{tikzcd}
 & ident \times exp \times decs \arrow[dl, "pr_{27}"'] \arrow[d, "\langle pr_{28}, pr_{29} \rangle"] \arrow[dr, "pr_{28}"] & \\
ident & ident \times exp \arrow[l, "pr_3"] \arrow[r, "pr_4"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
ident \times exp \arrow[dd, "call"'] & ident \times exp \times decs \arrow[l, "\langle pr_{27}, pr_{28} \rangle"'] \arrow[dd, "\langle call \circ \langle pr_{27}, pr_{28} \rangle, pr_{29} \rangle"'] \arrow[dr, "pr_{29}"] & \\
 & & decs \\
exp & exp \times decs \arrow[l, "pr_{47}"] \arrow[ur, "pr_{48}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & exp \times exp \times decs \arrow[dl, "pr_{49}"'] \arrow[d, "\langle pr_{49}, pr_{51} \rangle"] \arrow[dr, "pr_{51}"] & \\
exp & exp \times decs \arrow[l, "pr_{47}"] \arrow[r, "pr_{48}"'] & decs
\end{tikzcd}
\qquad
\begin{tikzcd}
 & exp \times exp \times decs \arrow[dl, "pr_{50}"'] \arrow[d, "\langle pr_{50}, pr_{51} \rangle"] \arrow[dr, "pr_{51}"] & \\
exp & exp \times decs \arrow[l, "pr_{47}"] \arrow[r, "pr_{48}"'] & decs
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times decs \arrow[d, "apply"'] & exp \times exp \times decs \arrow[l, "\langle pr_{49}, pr_{51} \rangle"'] \arrow[r, "\langle pr_{50}, pr_{51} \rangle"] \arrow[d, "\langle apply \circ \langle pr_{49}, pr_{51} \rangle, apply \circ \langle pr_{50}, pr_{51} \rangle \rangle"'] & exp \times decs \arrow[d, "apply"] \\
exp & exp \times exp \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
ident \times decs \arrow[dd, "fetch"'] & ident \times exp \times decs \arrow[l, "\langle pr_{27}, pr_{29} \rangle"'] \arrow[rr, "\langle pr_{28}, pr_{29} \rangle"] \arrow[dd, "{\langle fetch \circ \langle pr_{27}, pr_{29} \rangle, apply \circ \langle pr_{28}, pr_{29} \rangle, pr_{29} \rangle}"'] \arrow[dr, "pr_{29}"] & & exp \times decs \arrow[dd, "apply"] \\
 & & decs & \\
exp & exp \times exp \times decs \arrow[l, "pr_{49}"] \arrow[rr, "pr_{50}"'] \arrow[ur, "pr_{51}"'] & & exp
\end{tikzcd}$$

$$\begin{tikzcd}
 & exp \times exp \times decs \arrow[dl, "pr_{49}"'] \arrow[d, "\langle pr_{49}, pr_{50} \rangle"] \arrow[dr, "pr_{50}"] & \\
exp & exp \times exp \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \arrow[dd, "replace"'] & exp \times exp \times decs \arrow[l, "\langle pr_{49}, pr_{50} \rangle"'] \arrow[dd, "\langle replace \circ \langle pr_{49}, pr_{50} \rangle, pr_{51} \rangle"'] \arrow[dr, "pr_{51}"] & \\
 & & decs \\
exp & exp \times decs \arrow[l, "pr_{47}"] \arrow[ur, "pr_{48}"'] &
\end{tikzcd}$$

These diagrams describe the behaviour of *fetch* : *ident* × *decs* → *exp* whose purpose is described above.

$$\begin{tikzcd}
ident \arrow[d, "id_{ident}"'] & ident \times \mathbf{T} \arrow[l, "pr_5"'] \arrow[r, "pr_6"] \arrow[d, "id_{ident} \times empty"] & \mathbf{T} \arrow[d, "empty"] \\
ident & ident \times decs \arrow[l, "pr_7"] \arrow[r, "pr_8"'] & decs
\end{tikzcd}$$

$$\begin{tikzcd}
ident \times \mathbf{T} \arrow[r, "id_{ident} \times empty"] \arrow[d, "dispose_{ident \times \mathbf{T}}"'] & ident \times decs \arrow[d, "fetch"] \\
\mathbf{T} \arrow[r, "undef"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
ident \times ident \times exp \times decs \arrow[r, "\langle pr_{23}, =; \circ \langle pr_{24}, pr_{25}, pr_{26} \rangle \rangle"] \arrow[d, "\langle pr_{23}, same \circ \langle pr_{23}, pr_{24} \rangle, pr_{25}, pr_{26} \rangle"'] & ident \times decs \arrow[d, "fetch"] \\
ident \times num \times exp \times decs \arrow[r, "get"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
 & ident \times ident \times exp \times decs \arrow[dl, "pr_{24}"'] \arrow[dd, "\langle pr_{24}, pr_{25}, pr_{26} \rangle"'] \arrow[dr, "pr_{25}"'] \arrow[drr, "pr_{26}"] & & \\
ident & & exp & decs \\
 & ident \times exp \times decs \arrow[ul, "pr_{27}"] \arrow[ur, "pr_{28}"] \arrow[urr, "pr_{29}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathit{ident} \times \mathit{exp} \times \mathit{decs} \arrow[dl, "pr_{23}"'] \arrow[dd, "{\langle pr_{23}, =; \circ \langle pr_{24}, pr_{25}, pr_{26} \rangle \rangle}"] \arrow[rr, "{\langle pr_{24}, pr_{25}, pr_{26} \rangle}"] & & \mathit{ident} \times \mathit{exp} \times \mathit{decs} \arrow[dd, "=;"] \\
\mathit{ident} & & & \\
& \mathit{ident} \times \mathit{decs} \arrow[ul, "pr_7"] \arrow[rr, "pr_8"'] & & \mathit{decs}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathit{ident} \times \mathit{exp} \times \mathit{decs} \arrow[dl, "pr_{23}"'] \arrow[d, "{\langle pr_{23}, pr_{24} \rangle}"] \arrow[dr, "pr_{24}"] & \\
\mathit{ident} & \mathit{ident} \times \mathit{ident} \arrow[l, "pr_1"] \arrow[r, "pr_2"'] & \mathit{ident}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{ident} \times \mathit{ident} \times \mathit{exp} \times \mathit{decs} \arrow[r, "{\langle pr_{23}, pr_{24} \rangle}"] \arrow[dr, "{\mathit{same} \circ \langle pr_{23}, pr_{24} \rangle}"'] & \mathit{ident} \times \mathit{ident} \arrow[d, "\mathit{same}"] \\
& \mathit{num}
\end{tikzcd}$$

$$\begin{tikzcd}
& & \mathit{ident} \times \mathit{ident} \times \mathit{exp} \times \mathit{exp} \arrow[dll, "pr_{23}"'] \arrow[dl, "pr_{25}"] \arrow[dd, "{\langle pr_{23}, \mathit{same} \circ \langle pr_{23}, pr_{24} \rangle, pr_{25}, pr_{26} \rangle}"] \arrow[dr, "{\mathit{same} \circ \langle pr_{23}, pr_{24} \rangle}"] \arrow[drr, "pr_{26}"] & & \\
\mathit{ident} & \mathit{exp} & & \mathit{num} & \mathit{exp} \\
& & \mathit{ident} \times \mathit{num} \times \mathit{exp} \times \mathit{exp} \arrow[ull, "pr_{30}"] \arrow[ul, "pr_{32}"'] \arrow[ur, "pr_{31}"] \arrow[urr, "pr_{33}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{num} \arrow[dd, "\mathit{zero}"'] & & \mathit{ident} \times \mathit{num} \times \mathit{exp} \times \mathit{decs} \arrow[ll, "pr_{31}"'] \arrow[dl, "pr_{30}"'] \arrow[dd, "\mathit{id}_{\mathit{ident}} \times \mathit{zero} \times \mathit{id}_{\mathit{exp}} \times \mathit{id}_{\mathit{decs}}"] \arrow[dr, "pr_{32}"'] \arrow[drr, "pr_{33}"] & & \\
& \mathit{ident} & & \mathit{exp} & \mathit{decs} \\
\mathit{num} & & \mathit{ident} \times \mathit{num} \times \mathit{exp} \times \mathit{decs} \arrow[ll, "pr_{31}"] \arrow[ul, "pr_{30}"] \arrow[ur, "pr_{32}"] \arrow[urr, "pr_{33}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{ident} \times \mathit{num} \times \mathit{exp} \times \mathit{decs} \arrow[r, "pr_{32}"] \arrow[d, "\mathit{id}_{\mathit{ident}} \times \mathit{zero} \times \mathit{id}_{\mathit{exp}} \times \mathit{id}_{\mathit{decs}}"'] & \mathit{exp} \\
\mathit{ident} \times \mathit{num} \times \mathit{exp} \times \mathit{decs} \arrow[ur, "\mathit{get}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
num \arrow[dd, "succ"'] & & ident \times num \times exp \times decs \arrow[ll, "pr_{31}"'] \arrow[dl, "pr_{30}"'] \arrow[dd, "id_{ident} \times succ \times id_{exp} \times id_{decs}"] \arrow[dr, "pr_{32}"] \arrow[drr, "pr_{33}"] & & \\
& ident & & exp & decs \\
num & & ident \times num \times exp \times decs \arrow[ll, "pr_{31}"] \arrow[ul, "pr_{30}"] \arrow[ur, "pr_{32}"'] \arrow[urr, "pr_{33}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
& ident \times num \times exp \times decs \arrow[dl, "pr_{30}"'] \arrow[d, "\langle pr_{30}, pr_{33} \rangle"] \arrow[dr, "pr_{33}"] & \\
ident & ident \times decs \arrow[l, "pr_7"] \arrow[r, "pr_8"'] & decs
\end{tikzcd}$$

$$\begin{tikzcd}
ident \times num \times exp \times decs \arrow[r, "\langle pr_{30}, pr_{33} \rangle"] \arrow[d, "id_{ident} \times succ \times id_{exp} \times id_{decs}"] & ident \times decs \arrow[d, "fetch"] \\
ident \times num \times exp \times decs \arrow[r, "get"'] & exp
\end{tikzcd}$$

This collection of diagrams are used to describe the *replace* : *exp* × *exp* → *exp* operation.

$$\begin{tikzcd}
\mathbf{T} \arrow[dd, "arg"'] & \mathbf{T} \times exp \arrow[l, "pr_{34}"'] \arrow[dd, "arg \times id_{exp}"'] \arrow[dr, "pr_{35}"] & \\
& & exp \\
exp & exp \times exp \arrow[l, "pr_{11}"] \arrow[ur, "pr_{12}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{T} \arrow[dd, "undef"'] & \mathbf{T} \times exp \arrow[l, "pr_{34}"'] \arrow[dd, "undef \times id_{exp}"'] \arrow[dr, "pr_{35}"] & \\
& & exp \\
exp & exp \times exp \arrow[l, "pr_{11}"] \arrow[ur, "pr_{12}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
num \arrow[dd, "is\_num"'] & num \times exp \arrow[l, "pr_{36}"'] \arrow[dd, "is\_num \times id_{exp}"'] \arrow[dr, "pr_{37}"] & \\
& & exp \\
exp & exp \times exp \arrow[l, "pr_{11}"] \arrow[ur, "pr_{12}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
exp \arrow[dd, "fst"'] & exp \times exp \arrow[l, "pr_{11}"'] \arrow[dd, "fst \times id_{exp}"'] \arrow[dr, "pr_{12}"] & \\
& & exp \\
exp & exp \times exp \arrow[l, "pr_{11}"] \arrow[ur, "pr_{12}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
exp \arrow[dd, "snd"'] & exp \times exp \arrow[l, "pr_{11}"'] \arrow[dd, "snd \times id_{exp}"'] \arrow[dr, "pr_{12}"] & \\
& & exp \\
exp & exp \times exp \arrow[l, "pr_{11}"] \arrow[ur, "pr_{12}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \arrow[dd, "="'] & exp \times exp \times exp \arrow[l, "\langle pr_{13}, pr_{14} \rangle"'] \arrow[dd, "\langle = \circ \langle pr_{13}, pr_{14} \rangle, pr_{15} \rangle"'] \arrow[dr, "pr_{15}"] & \\
& & exp \\
exp & exp \times exp \arrow[l, "pr_{11}"] \arrow[ur, "pr_{12}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \arrow[dd, "{(\,,\,)}"'] & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[l, "{\langle pr_{13}, pr_{14}\rangle}"'] \arrow[dd, "{\langle (\,,\,) \circ \langle pr_{13}, pr_{14}\rangle, pr_{15}\rangle}"'] \arrow[dr, "pr_{15}"] & \\
 & & \mathit{exp} \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{11}"] \arrow[ur, "pr_{12}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{19}"'] \arrow[dd, "{\langle pr_{19}, pr_{20}, pr_{21}\rangle}"'] \arrow[dr, "pr_{20}"'] \arrow[drr, "pr_{21}"] & & \\
\mathit{exp} & & \mathit{exp} & \mathit{exp} \\
 & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[ul, "pr_{13}"] \arrow[ur, "pr_{14}"] \arrow[urr, "pr_{15}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dd, "\mathit{if}"'] & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[l, "{\langle pr_{19}, pr_{20}, pr_{21}\rangle}"'] \arrow[dd, "{\langle \mathit{if} \circ \langle pr_{19}, pr_{20}, pr_{21}\rangle, pr_{22}\rangle}"'] \arrow[dr, "pr_{22}"] & \\
 & & \mathit{exp} \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{11}"] \arrow[ur, "pr_{12}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathit{ident} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{38}"'] \arrow[d, "{\langle pr_{38}, pr_{39}\rangle}"] \arrow[dr, "pr_{39}"] & \\
\mathit{ident} & \mathit{ident} \times \mathit{exp} \arrow[l, "pr_3"] \arrow[r, "pr_4"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{ident} \times \mathit{exp} \arrow[dd, "\mathit{call}"'] & \mathit{ident} \times \mathit{exp} \times \mathit{exp} \arrow[l, "{\langle pr_{38}, pr_{39}\rangle}"'] \arrow[dd, "{\langle \mathit{call} \circ \langle pr_{38}, pr_{39}\rangle, pr_{40}\rangle}"'] \arrow[dr, "pr_{40}"] & \\
 & & \mathit{exp} \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{11}"] \arrow[ur, "pr_{12}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{13}"'] \arrow[d, "{\langle pr_{13}, pr_{15}\rangle}"] \arrow[dr, "pr_{15}"] & \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{14}"'] \arrow[d, "{\langle pr_{14}, pr_{15}\rangle}"] \arrow[dr, "pr_{15}"] & \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \arrow[d, "\mathit{replace}"'] & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[l, "\langle pr_{13}, pr_{15} \rangle"'] \arrow[r, "\langle pr_{14}, pr_{15} \rangle"] \arrow[d, "\langle \mathit{replace} \circ \langle pr_{13}, pr_{15} \rangle, \mathit{replace} \langle pr_{14}, pr_{15} \rangle \rangle"] & \mathit{exp} \times \mathit{exp} \arrow[d, "\mathit{replace}"] \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{19}"'] \arrow[d, "\langle pr_{19}, pr_{22} \rangle"] \arrow[dr, "pr_{22}"] & \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{20}"'] \arrow[d, "\langle pr_{20}, pr_{22} \rangle"] \arrow[dr, "pr_{22}"] & \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{21}"'] \arrow[d, "\langle pr_{21}, pr_{22} \rangle"] \arrow[dr, "pr_{22}"] & \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathit{exp} \times \mathit{exp} \arrow[dddd, "\mathit{replace}"'] & & & \\
& & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[ull, "\langle pr_{19}, pr_{22} \rangle"'] \arrow[dl, "\langle pr_{20}, pr_{22} \rangle"'] \arrow[r, "\langle pr_{21}, pr_{22} \rangle"] \arrow[dd, "{\begin{array}{l} \langle \mathit{replace} \circ \langle pr_{19}, pr_{22} \rangle, \\ \mathit{replace} \circ \langle pr_{20}, pr_{22} \rangle, \\ \mathit{replace} \circ \langle pr_{21}, pr_{22} \rangle \rangle \end{array}}"] & \mathit{exp} \times \mathit{exp} \arrow[dd, "\mathit{replace}"] \\
& \mathit{exp} \times \mathit{exp} \arrow[d, "\mathit{replace}"'] & & \\
& \mathit{exp} & \mathit{exp} \times \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{14}"'] \arrow[dll, "pr_{13}"] \arrow[r, "pr_{15}"'] & \mathit{exp} \\
\mathit{exp} & & &
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{39}"'] \arrow[d, "\langle pr_{39}, pr_{40} \rangle"] \arrow[dr, "pr_{40}"] & \\
\mathit{exp} & \mathit{exp} \times \mathit{exp} \arrow[l, "pr_{11}"] \arrow[r, "pr_{12}"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathit{ident} \times \mathit{exp} \times \mathit{exp} \arrow[dl, "pr_{38}"'] \arrow[r, "\langle pr_{39}, pr_{40} \rangle"] \arrow[dd, "\langle pr_{38}, \mathit{replace} \circ \langle pr_{39}, pr_{40} \rangle \rangle"] & \mathit{exp} \times \mathit{exp} \arrow[dd, "\mathit{replace}"] \\
\mathit{ident} & & \\
& \mathit{ident} \times \mathit{exp} \arrow[ul, "pr_3"] \arrow[r, "pr_4"'] & \mathit{exp}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{T} \times exp \arrow[r, "arg \times id_{exp}"] \arrow[dr, "pr_{35}"'] & exp \times exp \arrow[d, "replace"] \\
 & exp
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{T} \times exp \arrow[r, "undef \times id_{exp}"] \arrow[d, "dispose_{\mathbf{T} \times exp}"] & exp \times exp \arrow[d, "replace"] \\
\mathbf{T} \arrow[r, "undef"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
num \times exp \arrow[r, "is\_num \times id_{exp}"] \arrow[d, "pr_{36}"'] & exp \times exp \arrow[d, "replace"] \\
num \arrow[r, "is\_num"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \arrow[r, "fst \times id_{exp}"] \arrow[d, "replace"'] & exp \times exp \arrow[d, "replace"] \\
exp \arrow[r, "fst"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \arrow[r, "snd \times id_{exp}"] \arrow[d, "replace"'] & exp \times exp \arrow[d, "replace"] \\
exp \arrow[r, "snd"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
ident \times exp \times exp \arrow[r, "\langle call \circ \langle pr_{38}, pr_{39} \rangle, pr_{40} \rangle"] \arrow[d, "\langle pr_{38}, replace \circ \langle pr_{39}, pr_{40} \rangle \rangle"] & exp \times exp \arrow[d, "replace"] \\
ident \times exp \arrow[r, "call"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \times exp \arrow[r, "\langle = \circ \langle pr_{13}, pr_{14}\rangle, pr_{15}\rangle"] \arrow[d, "{\begin{array}{l}\langle replace \circ \langle pr_{13}, pr_{15}\rangle,\\ replace \circ \langle pr_{14}, pr_{15}\rangle\rangle\end{array}}"'] & exp \times exp \arrow[d, "replace"] \\
exp \times exp \arrow[r, "="'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \times exp \arrow[r, "\langle (,) \circ \langle pr_{13}, pr_{14}\rangle, pr_{15}\rangle"] \arrow[d, "{\begin{array}{l}\langle replace \circ \langle pr_{13}, pr_{15}\rangle,\\ replace \circ \langle pr_{14}, pr_{15}\rangle\rangle\end{array}}"'] & exp \times exp \arrow[d, "replace"] \\
exp \times exp \arrow[r, "(,)"'] & exp
\end{tikzcd}$$

$$\begin{tikzcd}
exp \times exp \times exp \times exp \arrow[r, "\langle if \circ \langle pr_{19}, pr_{20}, pr_{21}\rangle, pr_{22}\rangle"] \arrow[d, "{\begin{array}{l}\langle replace \circ \langle pr_{19}, pr_{22}\rangle,\\ replace \circ \langle pr_{20}, pr_{22}\rangle\rangle,\\ replace \circ \langle pr_{21}, pr_{22}\rangle\rangle\end{array}}"'] & exp \times exp \arrow[d, "replace"] \\
exp \times exp \times exp \arrow[r, "if"'] & exp
\end{tikzcd}$$

Finally we describe the operation *where* : $exp \times decs \rightarrow prg$. This operation is used to place an expression within a context, (i.e. an environment) and thus allow the evaluation of function calls. The operations *is* : $exp \rightarrow prg$ and $is^{-1} : prg \rightarrow exp$ are used to force an isomorphism between the sets $I_{Sem}(exp)$ and $I_{Sem}(prg)$.

$$\begin{tikzcd}
exp \arrow[r, shift left, "is"] & prg \arrow[l, shift left, "is^{-1}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
exp \times decs \arrow[r, "where"] \arrow[dr, "apply"'] & prg \\
 & exp \arrow[u, "is"']
\end{tikzcd}$$

A.2.2 The initial model $I_{Sem} : Toy_{Sem} \rightarrow \mathbf{SET}$

$$
\begin{array}{lcl}
I_{Sem}(\mathrm{T}) & = & \{\emptyset\} \\
I_{Sem}(ident) & = & \{0,1,2,\ldots\} \\
I_{Sem}(num) & = & \{0,1,2,\ldots\} \\
I_{Sem}(decs) & = & \bigcup I_{Sem}(decs)_n, n \in \{0,1,2,\ldots\} \\
\quad \text{where } I_{Sem}(decs)_0 & = & \{\texttt{empty}\} \\
\qquad\quad\; I_{Sem}(decs)_n & = & I_{Sem}(decs)_{n-1} \cup \{\texttt{=;(}x\texttt{,}y\texttt{,}z\texttt{)}: (x,y,z) \in \\
 & & \qquad I_{Sem}(ident) \times I_{Sem}(exp) \times I_{Sem}(decs)_{n-1}\} \\
I_{Sem}(ident \times exp \times decs) & = & I_{Sem}(ident) \times I_{Sem}(exp) \times I_{Sem}(decs) \\
I_{Sem}(ident \times ident) & = & I_{Sem}(ident) \times I_{Sem}(ident) \\
I_{Sem}(num \times num) & = & I_{Sem}(num) \times I_{Sem}(num) \\
I_{Sem}(exp) & = & \bigcup I_{Sem}(exp)_n, n \in \{0,1,2,\ldots\} \\
\quad \text{where } I_{Sem}(exp)_0 & = & \{\texttt{undef,arg}\} \cup \{\ \texttt{is\_num(}x\texttt{)} : x \in I_{Sem}(num)\} \\
\qquad\quad\; I_{Sem}(exp)_n & = & I_{Sem}(exp)_{n-1} \cup \\
 & & \{\ \texttt{fst(}x\texttt{)} : x \in \{y : y \in I_{Sem}(exp)_{n-1} \wedge \neg p(y)\}\} \cup \\
 & & \{\ \texttt{snd(}x\texttt{)} : x \in \{y : y \in I_{Sem}(exp)_{n-1} \wedge \neg p(y)\}\} \cup \\
 & & \{\ \texttt{(}x\texttt{,}y\texttt{)} : (x,y) \in I_{Sem}(exp)_{n-1} \times I_{Sem}(exp)_{n-1}\} \cup \\
 & & \{\ \texttt{=(}x\texttt{,}y\texttt{)}, \texttt{=(}y\texttt{,}x\texttt{)} : (x,y) \in \{z : z \in I_{Sem}(exp)_{n-1} \wedge \neg p(z)\} \times \\
 & & \qquad\qquad\qquad\qquad I_{Sem}(exp)_{n-1}\} \cup \\
 & & \{\ \texttt{if(}x\texttt{,}y\texttt{,}x\texttt{)} : (x,y,z) \in \{a : a \in I_{Sem}(exp)_{n-1} \wedge \neg p(a)\} \times \\
 & & \qquad\qquad I_{Sem}(exp)_{n-1} \times I_{Sem}(exp)_{n-1}\} \cup \\
 & & \{\ \texttt{call(}i\texttt{,}e\texttt{)} : (i,e) \in I_{Sem}(ident) \times I_{Sem}(exp)_{n-1}\}
\end{array}
$$

$$
\begin{array}{lcl@{\qquad}lcl}
p(\texttt{arg}) & = & \mathit{False} & p(\texttt{snd(}x\texttt{)}) & = & \mathit{False} \\
p(\texttt{undef}) & = & \mathit{True} & p(\texttt{call(}i\texttt{,}e\texttt{)}) & = & \mathit{False} \\
p(\texttt{is\_num(}x\texttt{)}) & = & \mathit{True} & p(\texttt{=(}x\texttt{,}y\texttt{)}) & = & \mathit{False} \\
p(\texttt{(}x\texttt{,}y\texttt{)}) & = & \mathit{True} & p(\texttt{if(}x\texttt{,}y\texttt{,}z\texttt{)}) & = & \mathit{False} \\
p(\texttt{fst(}x\texttt{)}) & = & \mathit{False} & & &
\end{array}
$$

$$\begin{aligned}
I_{Sem}(\mathit{ident} \times \mathit{decs}) &= I_{Sem}(\mathit{ident}) \times I_{Sem}(\mathit{decs}) \\
I_{Sem}(\mathit{ident} \times \mathbf{T}) &= I_{Sem}(\mathit{ident}) \times I_{Sem}(\mathbf{T}) \\
I_{Sem}(\mathit{exp} \times \mathit{exp}) &= I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \\
I_{Sem}(\mathit{ident} \times \mathit{exp}) &= I_{Sem}(\mathit{ident}) \times I_{Sem}(\mathit{exp}) \\
I_{Sem}(\mathit{exp} \times \mathit{exp} \times \mathit{exp}) &= I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \\
I_{Sem}(\mathit{exp} \times \mathbf{T}) &= I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathbf{T}) \\
I_{Sem}(\mathbf{T} \times \mathit{exp}) &= I_{Sem}(\mathbf{T}) \times I_{Sem}(\mathit{exp}) \\
I_{Sem}(\mathit{num} \times \mathit{exp}) &= I_{Sem}(\mathit{num}) \times I_{Sem}(\mathit{exp}) \\
I_{Sem}(\mathit{ident} \times \mathit{exp} \times \mathit{exp}) &= I_{Sem}(\mathit{ident}) \times I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \\
I_{Sem}(\mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{exp}) &= I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \\
I_{Sem}(\mathit{num} \times \mathit{exp} \times \mathit{exp}) &= I_{Sem}(\mathit{num}) \times I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \\
I_{Sem}(\mathit{ident} \times \mathit{num} \times \mathit{exp} \times \mathit{decs}) & \\
&= I_{Sem}(\mathit{ident}) \times I_{Sem}(\mathit{num}) \times I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{decs}) \\
I_{Sem}(\mathit{ident} \times \mathit{ident} \times \mathit{exp} \times \mathit{decs}) & \\
&= I_{Sem}(\mathit{ident}) \times I_{Sem}(\mathit{ident}) \times I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{decs}) \\
I_{Sem}(\mathbf{T} \times \mathit{decs}) &= I_{Sem}(\mathbf{T}) \times I_{Sem}(\mathit{decs}) \\
I_{Sem}(\mathit{num} \times \mathit{decs}) &= I_{Sem}(\mathit{num}) \times I_{Sem}(\mathit{decs}) \\
I_{Sem}(\mathit{exp} \times \mathit{decs}) &= I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{decs}) \\
I_{Sem}(\mathit{exp} \times \mathit{exp} \times \mathit{decs}) &= I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{decs}) \\
I_{Sem}(\mathit{exp} \times \mathit{exp} \times \mathit{exp} \times \mathit{decs}) &= I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{exp}) \times I_{Sem}(\mathit{decs}) \\
I_{Sem}(\mathit{prg}) &= I_{Sem}(\mathit{exp})
\end{aligned}$$

$$\begin{aligned}
I_{Sem}(x : \mathbf{T} \to \mathit{ident}) &= \emptyset \mapsto 0 \\
I_{Sem}(x_{-} : \mathit{ident} \to \mathit{ident}) &= x \to x + 1 \\
I_{Sem}(x_0 : \mathit{ident} \to \mathit{ident}) &= x \to 0 \\
I_{Sem}(\mathit{id}_{\mathit{ident}} : \mathit{ident} \to \mathit{ident}) &= x \to x \\
I_{Sem}(\mathit{dispose}_{\mathit{ident}} : \mathit{ident} \to \mathbf{T}) &= x \to \emptyset
\end{aligned}$$

$$\begin{aligned}
I_{Sem}(\mathit{empty} : \mathbf{T} \to \mathit{decs}) &= \emptyset \mapsto \texttt{empty} \\
I_{Sem}(=; : \mathit{ident} \times \mathit{exp} \times \mathit{decs} \to \mathit{decs}) &= (x, y, z) \to \texttt{=;}(x, y, x) \\
I_{Sem}(\mathit{id}_{\mathit{decs}} : \mathit{decs} \to \mathit{decs}) &= x \to x
\end{aligned}$$

$$
\begin{aligned}
I_{Sem}(pr_{27} : ident \times exp \times decs \to ident) &= (x, y, z) \to x\\
I_{Sem}(pr_{28} : ident \times exp \times decs \to exp) &= (x, y, z) \to y\\
I_{Sem}(pr_{29} : ident \times exp \times decs \to decs) &= (x, y, z) \to z\\
I_{Sem}(\langle call \circ \langle pr_{27}, pr_{28}\rangle, pr_{29}\rangle : ident \times exp \times decs \to exp \times decs) &\\
&= (x, y, z) \to (I_{Sem}(call)(x, y), z)\\
I_{Sem}(\langle fetch \circ \langle pr_{27}, pr_{29}\rangle, apply \circ \langle pr_{28}, pr_{29}\rangle, pr_{29}\rangle : ident \times exp \times decs &\\
\to exp \times exp \times decs) &\\
&= (x, y, z) \to (I_{Sem}(fetch)(x, y), I_{Sem}(apply)(y, z), z)\\
I_{Sem}(\langle pr_{27}, pr_{28}\rangle : ident \times exp \times decs \to ident \times exp) &= (x, y, z) \to (x, y)\\
I_{Sem}(\langle pr_{27}, pr_{29}\rangle : ident \times exp \times decs \to ident \times decs) &= (x, y, z) \to (x, z)\\
I_{Sem}(\langle pr_{28}, pr_{29}\rangle : ident \times exp \times decs \to exp \times decs) &= (x, y, z) \to (y, z)
\end{aligned}
$$

$$
\begin{aligned}
I_{Sem}(pr_1 : ident \times ident \to ident) &= (x, y) \to x\\
I_{Sem}(pr_2 : ident \times ident \to ident) &= (x, y) \to y\\
I_{Sem}(\langle x, x\rangle : \mathbf{T} \to ident \times ident) &= \emptyset \mapsto (0, 0)\\
I_{Sem}(x_- \times x_0 : ident \times ident \to ident \times ident) &= (x, y) \to (x + 1, 0)\\
I_{Sem}(x_0 \times x_- : ident \times ident \to ident \times ident) &= (x, y) \to (0, y + 1)\\
I_{Sem}(x_- \times x_- : ident \times ident \to ident \times ident) &= (x, y) \to (x + 1, y + 1)\\
I_{Sem}(dispose_{ident \times ident} : ident \times ident \to \mathbf{T}) &= (x, y) \to \emptyset\\
I_{Sem}(same : ident \times ident \to num) &= f
\end{aligned}
$$

$$
\text{where}\quad
\begin{aligned}
f(0, 0) &= 0\\
f(x + 1, 0) &= 1\\
f(0, x + 1) &= 1\\
f(x + 1, y + 1) &= f(x, y)
\end{aligned}
$$

$$
\begin{aligned}
I_{Sem}(0 : \mathbf{T} \to num) &= \emptyset \mapsto 0\\
I_{Sem}(succ : num \to num) &= x \to x + 1\\
I_{Sem}(zero : num \to num) &= x \to 0\\
I_{Sem}(id_{num} : num \to num) &= x \to x\\
I_{Sem}(dispose_{num} : num \to \mathbf{T}) &= x \to \emptyset
\end{aligned}
$$

$$
\begin{array}{lcl}
I_{Sem}(pr_9 : num \times num \rightarrow num) & = & (x, y) \rightarrow x \\
I_{Sem}(pr_{10} : num \times num \rightarrow num) & = & (x, y) \rightarrow y \\
I_{Sem}(\langle 0, 0 \rangle : \mathbf{T} \rightarrow num \times num) & = & \emptyset \rightarrow (0, 0) \\
I_{Sem}(succ \times zero : num \times num \rightarrow num \times num) & = & (x, y) \rightarrow (x + 1, 0) \\
I_{Sem}(zero \times succ : num \times num \rightarrow num \times num) & = & (x, y) \rightarrow (0, y + 1) \\
I_{Sem}(succ \times succ : num \times num \rightarrow num \times num) & = & (x, y) \rightarrow (x + 1, y + 1) \\
I_{Sem}(dispose_{num \times num} : num \times num \rightarrow \mathbf{T}) & = & (x, y) \rightarrow \emptyset \\
I_{Sem}(equal : num \times num \rightarrow num) & = & f
\end{array}
$$

$$
\begin{array}{llcl}
\text{where} & f(0, 0) & = & 0 \\
& f(x + 1, 0) & = & 1 \\
& f(0, x + 1) & = & 1 \\
& f(x + 1, y + 1) & = & f(x, y)
\end{array}
$$

$$I_{Sem}(arg : \mathbf{T} \rightarrow exp) = \emptyset \mapsto \mathtt{arg}$$

$$I_{Sem}(undef : \mathbf{T} \rightarrow exp) = \emptyset \mapsto \mathtt{undef}$$

$$I_{Sem}(is\_num : num \rightarrow exp) = x \rightarrow \mathtt{is\_num}(x)$$

$$I_{Sem}(fst : exp \rightarrow exp) = f$$

$$\begin{array}{lllll} \text{where} & f(x) & = & \mathtt{undef}, & x = \mathtt{undef} \\ & & = & x, & x = \mathtt{is\_num}(y) \\ & & = & a, & x = (a,b) \\ & & = & \mathtt{fst}(x), & \textbf{otherwise} \end{array}$$

$$I_{Sem}(snd : exp \rightarrow exp) = f$$

$$\begin{array}{lllll} \text{where} & f(x) & = & \mathtt{undef}, & x = \mathtt{undef} \\ & & = & x, & x = \mathtt{is\_num}(y) \\ & & = & b, & x = (a,b) \\ & & = & \mathtt{snd}(x), & \textbf{otherwise} \end{array}$$

$$I_{Sem}(id_{exp} : exp \rightarrow exp) = x \rightarrow x$$

$$I_{Sem}(dispose_{exp} : exp \rightarrow \mathbf{T}) = x \rightarrow \emptyset$$

$$I_{Sem}((,) : exp \times exp \rightarrow exp) = (x,y) \rightarrow (x,y)$$

$$I_{Sem}(=: exp \times exp \rightarrow exp) = f$$

$$\begin{array}{lllll} \text{where} & f(x,y) & = & \mathtt{is\_num}(I_{Sem}(equal)(a,b)), & x = \mathtt{is\_num}(a) \wedge y = \mathtt{is\_num}(b) \\ & & = & \mathtt{undef}, & x = (a,b) \vee y = (c,d) \vee \\ & & & & x = \mathtt{undef} \vee y = \mathtt{undef} \\ & & = & \mathtt{=}(x,y), & \textbf{otherwise} \end{array}$$

$$I_{Sem}(if : exp \times exp \times exp \rightarrow exp) = f$$

$$\begin{array}{lllll} \text{where} & f(x,y,z) & = & \mathtt{undef}, & x = \mathtt{undef} \\ & & = & y, & x = \mathtt{is\_num}(0) \\ & & = & z, & x = \mathtt{is\_num}(n+1) \vee x = (a,b) \\ & & & & x = \mathtt{is\_num}(x+1) \vee x = (a,b) \\ & & = & \mathtt{if}(x,y,z), & \textbf{otherwise} \end{array}$$

$$I_{Sem}(call : ident \times exp \rightarrow exp) = (i,x) \rightarrow \mathtt{call}(i,x)$$

$$I_{Sem}(is\_num \times is\_num : num \times num \rightarrow exp \times exp)$$

$$= (x,y) \rightarrow (\mathtt{is\_num}(x),\mathtt{is\_num}(y))$$

$$I_{Sem}(is : exp \rightarrow prg) = x \rightarrow x$$

$$\begin{aligned}
I_{Sem}(pr_7 : ident \times decs \rightarrow ident) &= (x, y) \rightarrow x \\
I_{Sem}(pr_8 : ident \times decs \rightarrow decs) &= (x, y) \rightarrow y
\end{aligned}$$

$$\begin{aligned}
I_{Sem}(pr_5 : ident \times \mathbf{T} \rightarrow ident) &= (x, y) \rightarrow x \\
I_{Sem}(pr_6 : ident \times \mathbf{T} \rightarrow \mathbf{T}) &= (x, y) \rightarrow y \\
I_{Sem}(dispose_{ident \times \mathbf{T}} : ident \times \mathbf{T} \rightarrow \mathbf{T}) &= (x, y) \rightarrow \emptyset \\
I_{Sem}(id_{exp} \times empty : ident \times \mathbf{T} \rightarrow ident \times decs) &= (x, y) \rightarrow (x, \mathtt{empty})
\end{aligned}$$

$$\begin{aligned}
I_{Sem}(pr_{11} : exp \times exp \rightarrow exp) &= (x, y) \rightarrow x \\
I_{Sem}(pr_{12} : exp \times exp \rightarrow exp) &= (x, y) \rightarrow y \\
I_{Sem}(fst \times id_{exp} : exp \times exp \rightarrow exp \times exp) &= (x, y) \rightarrow (I_{Sem}(fst)(x), y) \\
I_{Sem}(snd \times id_{exp} : exp \times exp \rightarrow exp \times exp) &= (x, y) \rightarrow (I_{Sem}(snd)(x), y) \\
I_{Sem}(replace : exp \times exp \rightarrow exp) &= f
\end{aligned}$$

$$\begin{aligned}
\text{where} \quad f(\mathtt{arg}, r) &= r \\
f(\mathtt{undef}, r) &= \mathtt{undef} \\
f(\mathtt{is\_num}(n), r) &= \mathtt{is\_num}(n) \\
f(\mathtt{fst}(e), r) &= I_{Sem}(fst)(f(e, r)) \\
f(\mathtt{snd}(e), r) &= I_{Sem}(snd)(f(e, r)) \\
f(\mathtt{=}(x, y), r) &= I_{Sem}(=)(f(x, r), f(y, r)) \\
f((x, y), r) &= I_{Sem}((,))(f(x, r), f(y, r)) \\
f(\mathtt{if}(x, y, z), r) &= I_{Sem}(if)(f(x, r), f(y, r), f(z, r)) \\
f(\mathtt{call}(i, e), r) &= I_{Sem}(call)(i, f(e, r))
\end{aligned}$$

$$\begin{aligned}
I_{Sem}(pr_3 : ident \times exp \rightarrow ident) &= (x, y) \rightarrow x \\
I_{Sem}(pr_4 : ident \times exp \rightarrow exp) &= (x, y) \rightarrow y
\end{aligned}$$

$$I_{Sem}(pr_{13} : exp \times exp \times exp \to exp) = (x,y,z) \to x$$

$$I_{Sem}(pr_{14} : exp \times exp \times exp \to exp) = (x,y,z) \to y$$

$$I_{Sem}(pr_{15} : exp \times exp \times exp \to exp) = (x,y,z) \to z$$

$$I_{Sem}(dispose_{exp \times exp \times exp} : exp \times exp \times exp \to \mathbf{T}) = (x,y,z) \to \emptyset$$

$$I_{Sem}(\langle (\,,\,) \circ \langle pr_{13}, pr_{14} \rangle, pr_{15} \rangle : exp \times exp \times \mathrm{exp} \to exp \times exp) = (x,y,z) \to (I_{Sem}((\,,\,))(x,y), z)$$

$$I_{Sem}(\langle pr_{13}, (\,,\,) \circ \langle pr_{14}, pr_{15} \rangle \rangle : exp \times exp \times \mathrm{exp} \to exp \times exp) = (x,y,z) \to (x, I_{Sem}((\,,\,))(y,z))$$

$$I_{Sem}(\langle = \circ \langle pr_{13}, pr_{14} \rangle, pr_{15} \rangle : exp \times exp \times \mathrm{exp} \to exp \times exp) = (x,y,z) \to (I_{Sem}(=)(x,y), z)$$

$$I_{Sem}(U \times id_{exp} \times id_{exp} : exp \times exp \times exp \to exp \times exp \times exp) = (x,y,z) \to (\mathtt{undef}, y, z)$$

$$I_{Sem}(\langle pr_{13}, pr_{14} \rangle : exp \times exp \times exp \to exp \times exp) = (a,b,c) \to (a,b)$$

$$I_{Sem}(\langle pr_{13}, pr_{15} \rangle : exp \times exp \times exp \to exp \times exp) = (a,b,c) \to (a,c)$$

$$I_{Sem}(\langle pr_{14}, pr_{15} \rangle : exp \times exp \times exp \to exp \times exp) = (a,b,c) \to (b,c)$$

$$I_{Sem}(\langle replace \circ \langle pr_{13}, pr_{15} \rangle, replace \circ \langle pr_{14}, pr_{15} \rangle \rangle : exp \times exp \times exp \to exp \times exp) = (x,y,z) \to (I_{Sem}(replace)(x,z), I_{Sem}(replace)(y,z))$$

$$I_{Sem}(pr_{41} : exp \times \mathbf{T} \to exp) = (x,y) \to x$$

$$I_{Sem}(pr_{42} : exp \times \mathbf{T} \to \mathbf{T}) = (x,y) \to y$$

$$I_{Sem}(dispose_{exp \times \mathbf{T}} : exp \times \mathbf{T} \to \mathbf{T}) = (x,y) \to \emptyset$$

$$I_{Sem}(id_{exp} \times undef : exp \times \mathbf{T} \to exp \times exp) = (x,y) \to (x,\mathtt{undef})$$

$$I_{Sem}(pr_{34} : \mathbf{T} \times exp \to \mathbf{T}) = (x,y) \to x$$

$$I_{Sem}(pr_{35} : \mathbf{T} \times exp \to exp) = (x,y) \to y$$

$$i_{Sem}(dispose_{\mathbf{T} \times exp} : \mathbf{T} \times exp \to \mathbf{T}) = (x,y) \to \emptyset$$

$$I_{Sem}(arg \times id_{exp} : \mathbf{T} \times exp \to exp \times exp) = (x,y) \to (\mathtt{arg}, y)$$

$$I_{Sem}(undef \times id_{exp} : \mathbf{T} \times exp \to exp \times exp) = (x,y) \to (\mathtt{undef}, y)$$

$$I_{Sem}(pr_{36} : num \times exp \to num) = (x,y) \to x$$

$$I_{Sem}(pr_{37} : num \times exp \to exp) = (x,y) \to y$$

$$I_{Sem}(is\_num \times id_{exp} : num \times exp \to exp \times exp) = (x,y) \to (\mathtt{is\_num}(x), y)$$

$$I_{Sem}(pr_{38} : ident \times exp \times exp \rightarrow ident) \quad = \quad (x, y, z) \rightarrow x$$

$$I_{Sem}(pr_{39} : ident \times exp \times exp \rightarrow exp) \quad = \quad (x, y, z) \rightarrow y$$

$$I_{Sem}(pr_{40} : ident \times exp \times exp \rightarrow exp) \quad = \quad (x, y, z) \rightarrow z$$

$$I_{Sem}(\langle call \circ \langle pr_{38}, pr_{39}\rangle, pr_{40}\rangle : ident \times exp \times exp \rightarrow exp \times exp) \quad = \quad (x, y, z) \rightarrow (I_{Sem}(call)(x, y), z)$$

$$I_{Sem}(\langle pr_{38}, pr_{39}\rangle : ident \times exp \times exp \rightarrow ident \times exp) \quad = \quad (x, y, z) \rightarrow (x, y)$$

$$I_{Sem}(\langle pr_{39}, pr_{40}\rangle : ident \times exp \times exp \rightarrow ident \times exp) \quad = \quad (x, y, z) \rightarrow (y, z)$$

$$I_{Sem}(\langle pr_{38}, replace \circ \langle pr_{39}, pr_{40}\rangle\rangle : ident \times exp \times exp \rightarrow ident \times exp) \quad = \quad (x, y, z) \rightarrow (x, I_{Sem}(replace)(y, z))$$

$$I_{Sem}(pr_{19} : exp \times exp \times exp \times exp \rightarrow exp) \quad = \quad (a, b, c, d) \rightarrow a$$

$$I_{Sem}(pr_{20} : exp \times exp \times exp \times exp \rightarrow exp) \quad = \quad (a, b, c, d) \rightarrow b$$

$$I_{Sem}(pr_{21} : exp \times exp \times exp \times exp \rightarrow exp) \quad = \quad (a, b, c, d) \rightarrow c$$

$$I_{Sem}(pr_{22} : exp \times exp \times exp \times exp \rightarrow exp) \quad = \quad (a, b, c, d) \rightarrow d$$

$$I_{Sem}(\langle if \circ \langle pr_{19}, pr_{20}, pr_{21}\rangle, pr_{22}\rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp) \quad = \quad (a, b, c, d) \rightarrow (I_{Sem}(if)(a, b, c), d)$$

$$I_{Sem}(\langle (,) \circ \langle pr_{19}, pr_{20}\rangle, pr_{21}, pr_{22}\rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp) \quad = \quad (a, b, c, d) \rightarrow (I_{Sem}((,))(a, b), c, d)$$

$$I_{Sem}(\langle pr_{19}, pr_{20}\rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp) \quad = \quad (a, b, c, d) \rightarrow (a, b)$$

$$I_{Sem}(\langle pr_{19}, pr_{22}\rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp) \quad = \quad (a, b, c, d) \rightarrow (a, d)$$

$$I_{Sem}(\langle pr_{20}, pr_{22}\rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp) \quad = \quad (a, b, c, d) \rightarrow (b, d)$$

$$I_{Sem}(\langle pr_{21}, pr_{22}\rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp) \quad = \quad (a, b, c, d) \rightarrow (c, d)$$

$$I_{Sem}(\langle pr_{19}, pr_{20}, pr_{21}\rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp \times exp) \quad = \quad (a, b, c, d) \rightarrow (a, b, c)$$

$$I_{Sem}(\langle replace \circ \langle pr_{19}, pr_{22}\rangle, replace \circ \langle pr_{20}, pr_{22}\rangle, replace \circ \langle pr_{21}, pr_{22}\rangle\rangle : exp \times exp \times exp \times exp \rightarrow exp \times exp \times exp)$$
$$= \quad (a, b, c, d) \rightarrow (I_{Sem}(replace)(a, d), I_{Sem}(replace)(b, d), I_{Sem}(replace)(c, d))$$

$$\begin{aligned}
I_{Sem}(pr_{16} : num \times exp \times exp \rightarrow num) &= (x,y,z) \rightarrow x \\
I_{Sem}(pr_{17} : num \times exp \times exp \rightarrow exp) &= (x,y,z) \rightarrow y \\
I_{Sem}(pr_{18} : num \times exp \times exp \rightarrow exp) &= (x,y,z) \rightarrow z
\end{aligned}$$

$$\begin{aligned}
&I_{Sem}(is\_num \circ zero \times id_{exp} \times id_{exp} : num \times exp \times exp \rightarrow exp \times exp \times exp) \\
&\qquad = (x,y,z) \rightarrow (I_{Sem}(is\_num)(I_{Sem}(zero)(x)), y, z) \\
&I_{Sem}(is\_num \circ succ \times id_{exp} \times id_{exp} : num \times exp \times exp \rightarrow exp \times exp \times exp) \\
&\qquad = (x,y,z) \rightarrow (I_{Sem}(is\_num)(I_{Sem}(succ)(x)), y, z)
\end{aligned}$$

$$\begin{aligned}
I_{Sem}(pr_{30} : ident \times num \times exp \times decs \rightarrow ident) &= (a,b,c,d) \rightarrow a \\
I_{Sem}(pr_{31} : ident \times num \times exp \times decs \rightarrow num) &= (a,b,c,d) \rightarrow b \\
I_{Sem}(pr_{32} : ident \times num \times exp \times decs \rightarrow exp) &= (a,b,c,d) \rightarrow c \\
I_{Sem}(pr_{33} : ident \times num \times exp \times decs \rightarrow decs) &= (a,b,c,d) \rightarrow d \\
I_{Sem}(id_{ident} \times zero \times id_{exp} \times id_{decs} : ident \times num \times exp \times decs & \\
\rightarrow ident \times num \times exp \times decs) & \\
&= (a,b,c,d) \rightarrow (a,0,c,d) \\
I_{Sem}(id_{ident} \times succ \times id_{exp} \times id_{decs} : ident \times num \times exp \times decs & \\
\rightarrow ident \times num \times exp \times decs) & \\
&= (a,b,c,d) \rightarrow (a,b+1,c,d) \\
I_{Sem}(\langle pr_{30}, pr_{33} \rangle : ident \times num \times exp \times decs \rightarrow ident \times decs) & \\
&= (a,b,c,d) \rightarrow (a,d) \\
I_{Sem}(fetch : ident \times decs \rightarrow exp) &= f
\end{aligned}$$

$$\begin{aligned}
\text{where}\quad f(x,\texttt{empty}) &= \textbf{undef} \\
f(x,\texttt{=;(}y\texttt{,}e\texttt{,}d\texttt{))} &= I_{Sem}(get)(x, I_{Sem}(same)(x,y), e, d)
\end{aligned}$$

$$I_{Sem}(get : ident \times num \times exp \times decs \rightarrow exp) = f$$

$$\begin{aligned}
\text{where}\quad f(x,0,e,d) &= e \\
f(x,x+1,e,d) &= I_{Sem}(fetch)(x,d)
\end{aligned}$$

$$\begin{aligned}
I_{Sem}(pr_{23} : ident \times ident \times exp \times decs \rightarrow ident) &= (a,b,c,d) \rightarrow a \\
I_{Sem}(pr_{24} : ident \times ident \times exp \times decs \rightarrow ident) &= (a,b,c,d) \rightarrow b \\
I_{Sem}(pr_{25} : ident \times ident \times exp \times decs \rightarrow exp) &= (a,b,c,d) \rightarrow c \\
I_{Sem}(pr_{26} : ident \times ident \times exp \times decs \rightarrow decs) &= (a,b,c,d) \rightarrow d \\
I_{Sem}(\langle pr_{24}, pr_{25}, pr_{26}\rangle : ident \times ident \times exp \times decs \rightarrow ident \times exp \times decs) & \\
&= (a,b,c,d) \rightarrow (b,c,d) \\
I_{Sem}(\langle pr_{23}, =; \circ \langle pr_{24}, pr_{25}, pr_{26}\rangle\rangle : ident \times ident \times exp \times decs \rightarrow ident \times exp \times decs) & \\
&= (a,b,c,d) \rightarrow (a,\texttt{=;}\,\texttt{(}b,c,d\texttt{)}) \\
I_{Sem}(\langle pr_{23}, pr_{24}\rangle : ident \times ident \times exp \times decs \rightarrow ident \times ident) &= (a,b,c,d) \rightarrow (a,b) \\
I_{Sem}(same \circ \langle pr_{23}, pr_{24}\rangle : ident \times ident \times exp \times decs \rightarrow num) & \\
&= (a,b,c,d) \rightarrow I_{Sem}(same)(a,b) \\
I_{Sem}(\langle pr_{23}, same \circ \langle pr_{23}, pr_{24}\rangle, pr_{25}, pr_{26}\rangle : ident \times ident \times exp \times decs & \\
\rightarrow ident \times num \times exp \times decs) &= (a,b,c,d) \rightarrow (a, I_{Sem}(same)(a,b), c, d)
\end{aligned}$$

$$\begin{aligned}
I_{Sem}(pr_{43} : \mathbf{T} \times decs \rightarrow \mathbf{T}) &= (x,y) \rightarrow x \\
I_{Sem}(pr_{44} : \mathbf{T} \times decs \rightarrow decs) &= (x,y) \rightarrow y \\
I_{Sem}(dispose_{\mathbf{T} \times decs} : \mathbf{T} \times decs \rightarrow \mathbf{T}) &= (x,y) \rightarrow \emptyset \\
I_{Sem}(arg \times id_{decs} : \mathbf{T} \times decs \rightarrow exp \times decs) &= (x,y) \rightarrow (\mathtt{arg}, y) \\
I_{Sem}(undef \times id_{decs} : \mathbf{T} \times decs \rightarrow exp \times decs) &= (x,y) \rightarrow (\mathtt{undef}, y)
\end{aligned}$$

$$\begin{aligned}
I_{Sem}(pr_{45} : num \times decs \rightarrow num) &= (x,y) \rightarrow x \\
I_{Sem}(pr_{46} : num \times decs \rightarrow decs) &= (x,y) \rightarrow y \\
I_{Sem}(is\_num \times id_{decs} : num \times decs \rightarrow exp \times decs) &= (x,y) \rightarrow (\mathtt{is\_num(}x\mathtt{)}, y)
\end{aligned}$$

$$\begin{array}{lcl}
I_{Sem}(pr_{47} : exp \times decs \rightarrow exp) & = & (x, y) \rightarrow x \\
I_{Sem}(pr_{48} : exp \times decs \rightarrow decs) & = & (x, y) \rightarrow y \\
I_{Sem}(fst \times id_{decs} : num \times decs \rightarrow exp \times decs) & = & (x, y) \rightarrow (I_{Sem}(fst)(x), y) \\
I_{Sem}(snd \times id_{decs} : num \times decs \rightarrow exp \times decs) & = & (x, y) \rightarrow (I_{Sem}(snd)(x), y) \\
I_{Sem}(apply : exp \times decs \rightarrow exp) & = & f
\end{array}$$

$$\begin{array}{llcl}
\text{where} & f(\texttt{arg}.d) & = & \texttt{arg} \\
 & f(\texttt{undef}, d) & = & \texttt{undef} \\
 & f(\texttt{is\_num(}x\texttt{)}, d) & = & \texttt{is\_num(}x\texttt{)} \\
 & f(\texttt{fst(}x\texttt{)}, d) & = & I_{Sem}(fst)(f(x, d)) \\
 & f(\texttt{snd(}x\texttt{)}, d) & = & I_{Sem}(snd)(f(x, d)) \\
 & f(\texttt{(}x\texttt{,}y\texttt{)}, d) & = & I_{Sem}((,))(f(x, d), f(y, d)) \\
 & f(\texttt{=(}x\texttt{,}y\texttt{)}, d) & = & I_{Sem}(=)(f(x, d), f(y, d)) \\
 & f(\texttt{if(}x\texttt{,}y\texttt{,}z\texttt{)}, d) & = & f(I_{Sem}(if)(f(x, d), y, z).d) \\
 & f(\texttt{call(}i\texttt{,}e\texttt{)}, d) & = & f(I_{Sem}(replace)(I_{Sem}(fetch)(i, d), f(e, d)), d)
\end{array}$$

$$\begin{array}{lcl}
I_{Sem}(pr_{49} : exp \times exp \times decs \rightarrow exp) & = & (x. y. z) \rightarrow x \\
I_{Sem}(pr_{50} : exp \times exp \times decs \rightarrow exp) & = & (x. y. z) \rightarrow y \\
I_{Sem}(pr_{51} : exp \times exp \times decs \rightarrow decs) & = & (x, y. z) \rightarrow z
\end{array}$$

$$I_{Sem}(\langle (,) \circ \langle pr_{49}, pr_{50}\rangle, pr_{51}\rangle : exp \times exp \times exp \rightarrow exp \times decs)$$

$$= \quad (x. y. z) \rightarrow (I_{Sem}((,))(x, y), z)$$

$$I_{Sem}(\langle = \circ \langle pr_{49}, pr_{50}\rangle, pr_{51}\rangle : exp \times exp \times exp \rightarrow exp \times decs)$$

$$= \quad (x, y, z) \rightarrow (I_{Sem}(=)(x, y), z)$$

$$I_{Sem}(\langle apply \circ \langle pr_{49}. pr_{51}\rangle, apply \circ \langle pr_{50}, pr_{51}\rangle\rangle : exp \times exp \times decs \rightarrow exp \times exp)$$

$$= \quad (x, y, z) \rightarrow (I_{Sem}(apply)(x, z), I_{Sem}(apply)(y, z))$$

$$I_{Sem}(\langle replace \circ \langle pr_{49}, pr_{50}\rangle, pr_{51}\rangle : exp \times exp \times decs \rightarrow exp \times decs)$$

$$= \quad (x, y, z) \rightarrow (I_{Sem}(replace)(x, y), z)$$

$$\begin{array}{lcl}
I_{Sem}(\langle pr_{49}, pr_{50}\rangle : exp \times exp \times decs \rightarrow exp \times exp) & = & (x, y, z) \rightarrow (x, y) \\
I_{Sem}(\langle pr_{49}, pr_{51}\rangle : exp \times exp \times decs \rightarrow exp \times decs) & = & (x, y, z) \rightarrow (x, z) \\
I_{Sem}(\langle pr_{50}, pr_{51}\rangle : exp \times exp \times decs \rightarrow exp \times decs) & = & (x, y, z) \rightarrow (y, z)
\end{array}$$

$$I_{Sem}(\langle apply \circ \langle pr_{49}, pr_{51}\rangle, apply \circ \langle pr_{50}, pr_{51}\rangle\rangle : exp \times exp \times decs \rightarrow exp \times exp)$$

$$= \quad (x, y, z) \rightarrow (I_{Sem}(apply)(x, z), I_{Sem}(apply)(y, z))$$

$$
\begin{aligned}
I_{Sem}(pr_{52} : exp \times exp \times exp \times decs \to exp) &= (a.b.c,d) \to a \\
I_{Sem}(pr_{53} : exp \times exp \times exp \times decs \to exp) &= (a.b.c,d) \to b \\
I_{Sem}(pr_{54} : exp \times exp \times exp \times decs \to exp) &= (a.b.c,d) \to c \\
I_{Sem}(pr_{55} : exp \times exp \times exp \times decs \to decs) &= (a,b.c,d) \to d
\end{aligned}
$$

$$
\begin{aligned}
&I_{Sem}(\langle if \circ \langle pr_{52}, pr_{53}, pr_{54}\rangle, pr_{55}\rangle : exp \times exp \times decs \to exp \times decs) \\
&\qquad = (a,b,c,d) - (I_{Sem}(if)(a,b,c),d)
\end{aligned}
$$

$$
\begin{aligned}
&I_{Sem}(\langle apply \circ \langle pr_{52}, pr_{55}\rangle, pr_{53}, pr_{54}, pr_{55}\rangle : exp \times exp \times exp \times decs \\
&\qquad \to exp \times exp \times exp \times decs) \\
&\qquad = (a,b,c,d) \to (I_{Sem}(apply)(a,d),b,c,d)
\end{aligned}
$$

$$
\begin{aligned}
&I_{Sem}(\langle pr_{52}, pr_{53}, pr_{54}\rangle : exp \times exp \times exp \times decs \to exp \times exp \times exp) \\
&\qquad = (a,b,c,d) \to (a,b,c) \\
&I_{Sem}(\langle pr_{52}, pr_{55}\rangle : exp \times exp \times exp \times decs \to exp \times decs) = (a,b.c,d) \to (a,d)
\end{aligned}
$$

$$
\begin{aligned}
&I_{Sem}(\langle apply \circ \langle pr_{52}, pr_{55}\rangle, pr_{53}, pr_{54}, pr_{55}\rangle : exp \times exp \times exp \times decs \\
&\qquad \to exp \times exp \times exp \times decs) \\
&\qquad = (a,b,c,d) \to (I_{Sem}(apply)(a,d),b,c,d)
\end{aligned}
$$

$$
\begin{aligned}
I_{Sem}(where : exp \times decs \to prg) &= I_{Sem}(is)(I_{Sem}(apply)) \\
I_{Sem}(is^{-1} : prg \to exp) &= x \to x
\end{aligned}
$$

A.3 The *eval* and *learn* transformations

We begin by specifying the sketch morphism $E : Toy_{Syn} \to Toy_{Sem}$ which allows us to specify the functor $E^{*} : \mathbf{Mod}(Toy_{Sem}) \to \mathbf{Mod}(Toy_{Syn})$. Using this functor we are able to define the model $E^{*}(I_{Sem}) : Toy_{Syn} \to \mathbf{SET}$ and the transformations $eval : I_{Syn} \dot{\to} E^{*}(I_{Sem})$ and $learn : E^{*}(I_{Sem}) \to I_{Syn}$.

A.3.1 The sketch morphism $E : Toy_{Syn} \rightarrow Sem$

$$\begin{aligned}
E(\mathbf{T}) &= \mathbf{T}\\
E(num) &= num\\
E(ident) &= ident\\
E(exp) &= exp\\
E(ident \times exp) &= ident \times exp\\
E(exp \times exp) &= exp \times exp\\
E(exp \times exp \times exp) &= exp \times exp \times exp\\
E(decs) &= decs\\
E(exp \times decs) &= exp \times decs\\
E(exp \times ident \times decs) &= ident \times exp \times decs\\
E(prg) &= prg
\end{aligned}$$

$$\begin{aligned}
E(0 : \mathbf{T} \rightarrow num) &= 0\\
E(succ : num \rightarrow num) &= succ
\end{aligned}$$

$$\begin{aligned}
E(x : \mathbf{T} \rightarrow ident) &= x\\
E(x\_ : ident \rightarrow ident) &= x\_
\end{aligned}$$

$$\begin{aligned}
E(arg : \mathbf{T} \rightarrow exp) &= arg\\
E(error : \mathbf{T} \rightarrow exp) &= undef\\
E(is\_num : num \rightarrow exp) &= is\_num\\
E(call : ident \times exp \rightarrow exp) &= call\\
E(fst : exp \rightarrow exp) &= fst\\
E(snd : exp \rightarrow exp) &= snd\\
E(if : exp \times exp \times exp \rightarrow exp) &= if\\
E(=: exp \times exp \rightarrow exp) &= \; =\\
E((,) : exp \times exp \rightarrow exp) &= (,)
\end{aligned}$$

$$\begin{aligned}
E(pr_{11} : exp \times exp \rightarrow exp) &= pr_{11}\\
E(pr_{12} : exp \times exp \rightarrow exp) &= pr_{12}
\end{aligned}$$

$$\begin{aligned}
E(pr_{13} : exp \times exp \times exp \rightarrow exp) &= pr_{13} \\
E(pr_{14} : exp \times exp \times exp \rightarrow exp) &= pr_{14} \\
E(pr_{15} : exp \times exp \times exp \rightarrow exp) &= pr_{15}
\end{aligned}$$

$$\begin{aligned}
E(empty : \mathbf{T} \rightarrow decs) &= empty \\
E(=;: ident \times exp \times decs \rightarrow decs) &= =;
\end{aligned}$$

$$\begin{aligned}
E(pr_{27} : ident \times exp \times decs \rightarrow ident) &= pr_{27} \\
E(pr_{28} : ident \times exp \times decs \rightarrow ident) &= pr_{28} \\
E(pr_{29} : ident \times exp \times decs \rightarrow ident) &= pr_{29}
\end{aligned}$$

$$\begin{aligned}
E(pr_{47} : exp \times decs \rightarrow exp) &= pr_{47} \\
E(pr_{48} : exp \times decs \rightarrow exp) &= pr_{48}
\end{aligned}$$

$$\begin{aligned}
E(pr_{3} : ident \times exp \rightarrow ident) &= pr_{3} \\
E(pr_{4} : ident \times exp \rightarrow ident) &= pr_{4}
\end{aligned}$$

$$E(where : exp \times decs \rightarrow prg) = where$$

A.3.2 The model $E^*(I_{Sem})$

The functor $E^* : \mathbf{Mod}(Toy_{Sem}) \rightarrow \mathbf{Mod}(Toy_{Syn})$ is defined below.

$$\begin{aligned}
E^*(M : Toy_{Sem} \rightarrow \mathbf{SET}) &= M \circ E \\
E^*(f : M \dot{\rightarrow} N) &= f : E^*(M) \dot{\rightarrow} E^*(N)
\end{aligned}$$

From this we obtain the following definition of $E^*(I_{Sem}) : Toy_{Syn} \rightarrow \mathbf{SET}$.

$$
\begin{array}{lcl}
E^*(I_{Sem})(\mathbf{T}) & = & \{\emptyset\} \\
E^*(I_{Sem})(ident) & = & \{0,1,2,\ldots\} \\
E^*(I_{Sem})(num) & = & \{0,1,2,\ldots\} \\
E^*(I_{Sem})(decs) & = & \bigcup I_{Sem}(decs)_n,\, n \in \{0,1,2,\ldots\}
\end{array}
$$

$$
\begin{array}{lcl}
\text{where } I_{Sem}(decs)_0 & = & \{\texttt{empty}\} \\
I_{Sem}(decs)_n & = & I_{Sem}(decs)_{n-1} \cup \{\texttt{=;(}x\texttt{,}y\texttt{,}z\texttt{)}: (x,y,z) \in \\
 & & \qquad I_{Sem}(ident) \times I_{Sem}(exp) \times I_{Sem}(decs)_{n-1}\}
\end{array}
$$

$$
\begin{array}{lcl}
E^*(I_{Sem})(ident \times exp \times decs) & = & I_{Sem}(ident) \times I_{Sem}(exp) \times I_{Sem}(decs) \\
E^*(I_{Sem})(exp) & = & \bigcup I_{Sem}(exp)_n,\, n \in \{0,1,2,\ldots\}
\end{array}
$$

$$
\begin{array}{lcl}
\text{where } I_{Sem}(exp)_0 & = & \{\texttt{undef},\texttt{arg}\} \cup \{\ \texttt{is\_num(}x\texttt{)} : x \in I_{Sem}(num)\} \\
I_{Sem}(exp)_n & = & I_{Sem}(exp)_{n-1} \cup \\
 & & \{\ \texttt{fst(}x\texttt{)} : x \in \{y : y \in I_{Sem}(exp)_{n-1} \wedge \neg p(y)\}\} \cup \\
 & & \{\ \texttt{snd(}x\texttt{)} : x \in \{y : y \in I_{Sem}(exp)_{n-1} \wedge \neg p(y)\}\} \cup \\
 & & \{\ \texttt{(}x\texttt{,}y\texttt{)} : (x,y) \in I_{Sem}(exp)_{n-1} \times I_{Sem}(exp)_{n-1}\} \cup \\
 & & \{\ \texttt{=(}x\texttt{,}y\texttt{)},\ \texttt{=(}y\texttt{,}x\texttt{)} : (x,y) \in \{z : z \in I_{Sem}(exp)_{n-1} \wedge \neg p(z)\} \times \\
 & & \qquad\qquad I_{Sem}(exp)_{n-1}\} \cup \\
 & & \{\ \texttt{if(}x\texttt{,}y\texttt{,}x\texttt{)} : (x,y,z) \in \{a : a \in I_{Sem}(exp)_{n-1} \wedge \neg p(a)\} \times \\
 & & \qquad\qquad I_{Sem}(exp)_{n-1} \times I_{Sem}(exp)_{n-1}\} \cup \\
 & & \{\ \texttt{call(}i\texttt{,}e\texttt{)} : (i,e) \in I_{Sem}(ident) \times I_{Sem}(exp)_{n-1}\}
\end{array}
$$

$$
\begin{array}{lcl@{\qquad}lcl}
p(\texttt{arg}) & = & \mathit{False} & p(\texttt{snd(}x\texttt{)}) & = & \mathit{False} \\
p(\texttt{undef}) & = & \mathit{True} & p(\texttt{call(}i\texttt{,}e\texttt{)}) & = & \mathit{False} \\
p(\texttt{is\_num(}x\texttt{)}) & = & \mathit{True} & p(\texttt{=(}x\texttt{,}y\texttt{)}) & = & \mathit{False} \\
p(\texttt{(}x\texttt{,}y\texttt{)}) & = & \mathit{True} & p(\texttt{if(}x\texttt{,}y\texttt{,}z\texttt{)}) & = & \mathit{False} \\
p(\texttt{fst(}x\texttt{)}) & = & \mathit{False} & & &
\end{array}
$$

$$
\begin{array}{lcl}
E^*(I_{Sem})(exp \times exp) & = & I_{Sem}(exp) \times I_{Sem}(exp) \\
E^*(I_{Sem})(ident \times exp) & = & I_{Sem}(ident) \times I_{Sem}(exp) \\
E^*(I_{Sem})(exp \times exp \times exp) & = & I_{Sem}(exp) \times I_{Sem}(exp) \times I_{Sem}(exp) \\
E^*(I_{Sem})(exp \times decs) & = & I_{Sem}(exp) \times I_{Sem}(decs) \\
E^*(I_{Sem})(prg) & = & I_{Sem}(exp)
\end{array}
$$

$$
\begin{array}{lcl}
E^*(I_{Sem})(x : \mathbf{T} \rightarrow ident) & = & \emptyset \mapsto 0 \\
E^*(I_{Sem})(x\_ : ident \rightarrow ident) & = & x \rightarrow x + 1
\end{array}
$$

$$\begin{aligned}
E^*(I_{Sem})(empty : \mathbf{T} \rightarrow decs) &= \emptyset \mapsto \texttt{empty} \\
E^*(I_{Sem})(=;: ident \times exp \times decs \rightarrow decs) &= (x, y, z) \rightarrow \texttt{=;(}x\texttt{,}y\texttt{,}x\texttt{)}
\end{aligned}$$

$$\begin{aligned}
E^*(I_{Sem})(pr_{27} : ident \times exp \times decs \rightarrow ident) &= (x, y, z) \rightarrow x \\
E^*(I_{Sem})(pr_{28} : ident \times exp \times decs \rightarrow exp) &= (x, y, z) \rightarrow y \\
E^*(I_{Sem})(pr_{29} : ident \times exp \times decs \rightarrow decs) &= (x, y, z) \rightarrow z
\end{aligned}$$

$$\begin{aligned}
E^*(I_{Sem})(0 : \mathbf{T} \rightarrow num) &= \emptyset \mapsto 0 \\
E^*(I_{Sem})(succ : num \rightarrow num) &= x \rightarrow x + 1
\end{aligned}$$

$$
\begin{array}{lcl}
E^*(I_{Sem})(arg : \mathrm{T} \rightarrow exp) & = & \emptyset \rightarrow \mathtt{arg} \\
E^*(I_{Sem})(error : \mathrm{T} \rightarrow exp) & = & \emptyset \rightarrow \mathtt{undef} \\
E^*(I_{Sem})(is\_num : num \rightarrow exp) & = & x \rightarrow \mathtt{is\_num}(x) \\
E^*(I_{Sem})(fst : exp \rightarrow exp) & = & f
\end{array}
$$

$$
\begin{array}{llcll}
\text{where} & f(x) & = & \mathtt{undef}, & x = \mathtt{undef} \\
 & & = & x, & x = \mathtt{is\_num}(y) \\
 & & = & a, & x = (a,b) \\
 & & = & \mathtt{fst}(x), & \textbf{otherwise}
\end{array}
$$

$$
E^*(I_{Sem})(snd : exp \rightarrow exp) \quad = \quad f
$$

$$
\begin{array}{llcll}
\text{where} & f(x) & = & \mathtt{undef}, & x = \mathtt{undef} \\
 & & = & x, & x = \mathtt{is\_num}(y) \\
 & & = & b, & x = (a,b) \\
 & & = & \mathtt{snd}(x), & \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
E^*(I_{Sem})((,) : exp \times exp \rightarrow exp) & = & (x,y) \rightarrow (x,y) \\
E^*(I_{Sem})(=: exp \times exp \rightarrow exp) & = & f
\end{array}
$$

$$
\begin{array}{llcll}
\text{where} & f(x,y) & = & \mathtt{is\_num}(I_{Sem}(equal)(a,b)), & x = \mathtt{is\_num}(a) \wedge y = \mathtt{is\_num}(b) \\
 & & = & \mathtt{undef}, & x = (a,b) \vee y = (c,d) \vee \\
 & & & & x = \mathtt{undef} \vee y = \mathtt{undef} \\
 & & = & \mathtt{=}(x,y), & \textbf{otherwise}
\end{array}
$$

$$
E^*(I_{Sem})(if : exp \times exp \times exp \rightarrow exp) \quad = \quad f
$$

$$
\begin{array}{llcll}
\text{where} & f(x,y,z) & = & \mathtt{undef}, & x = \mathtt{undef} \\
 & & = & y, & x = \mathtt{is\_num}(0) \\
 & & = & z, & x = \mathtt{is\_num}(n+1) \vee x = (a,b) \\
 & & & & x = \mathtt{is\_num}(x+1) \vee x = (a,b) \\
 & & = & \mathtt{if}(x,y,z), & \textbf{otherwise}
\end{array}
$$

$$
E^*(I_{Sem})(call : ident \times exp \rightarrow exp) \quad = \quad (i,x) \rightarrow \mathtt{call}(i,x)
$$

$$
\begin{array}{lcl}
E^*(I_{Sem})(pr_{11} : exp \times exp \rightarrow exp) & = & (x,y) \rightarrow x \\
E^*(I_{Sem})(pr_{12} : exp \times exp \rightarrow exp) & = & (x,y) \rightarrow y
\end{array}
$$

$$
\begin{array}{lcl}
E^*(I_{Sem})(pr_3 : ident \times exp \rightarrow ident) & = & (x,y) \rightarrow x \\
E^*(I_{Sem})(pr_4 : ident \times exp \rightarrow exp) & = & (x,y) \rightarrow y
\end{array}
$$

$$\begin{aligned}
E^*(I_{Sem})(pr_{13} : exp \times exp \times exp \rightarrow exp) &= (x, y, z) \rightarrow x \\
E^*(I_{Sem})(pr_{14} : exp \times exp \times exp \rightarrow exp) &= (x, y, z) \rightarrow y \\
E^*(I_{Sem})(pr_{15} : exp \times exp \times exp \rightarrow exp) &= (x, y, z) \rightarrow z
\end{aligned}$$

$$\begin{aligned}
E^*(I_{Sem})(pr_{47} : exp \times decs \rightarrow exp) &= (x, y) \rightarrow x \\
E^*(I_{Sem})(pr_{48} : exp \times decs \rightarrow decs) &= (x, y) \rightarrow y
\end{aligned}$$

$$E^*(I_{Sem})(where : exp \times decs \rightarrow prg) = I_{Sem}(is)(I_{Sem}(apply))$$

A.3.3 The *eval* natural transformation

$$
\begin{aligned}
eval_{\mathrm{T}} &= 1_{\emptyset} \\
eval_{ident} &= f \text{ where } f(\mathtt{x}) = 0 \\
&\phantom{= f \text{ where }} f(\mathtt{x}y) = 1 + f(y) \\
eval_{num} &= f \text{ where } f(\mathtt{0}) = 0 \\
&\phantom{= f \text{ where }} f(\mathtt{succ(}y\mathtt{)}) = 1 + f(y) \\
eval_{exp} &= f \text{ where } f(\mathtt{arg}) = \mathtt{arg} \\
&\phantom{= f \text{ where }} f(\mathtt{error}) = \mathtt{undef} \\
&\phantom{= f \text{ where }} f(\mathtt{is\_num(}n\mathtt{)}) = \mathtt{is\_num(}eval_{num}(n)\mathtt{)} \\
&\phantom{= f \text{ where }} f(\mathtt{fst(}x\mathtt{)}) = I_{Sem}(fst)(f(x)) \\
&\phantom{= f \text{ where }} f(\mathtt{snd(}x\mathtt{)}) = I_{Sem}(snd)(f(x)) \\
&\phantom{= f \text{ where }} f(\mathtt{(}x\mathtt{,}y\mathtt{)}) = \mathtt{(}f(x)\mathtt{,}f(y)\mathtt{)} \\
&\phantom{= f \text{ where }} f(\mathtt{=(}x\mathtt{,}y\mathtt{)}) = I_{Sem}(=)(f(x), f(y)) \\
&\phantom{= f \text{ where }} f(\mathtt{if(}x\mathtt{,}y\mathtt{,}z\mathtt{)}) = I_{Sem}(if)(f(x), f(y), f(z)) \\
&\phantom{= f \text{ where }} f(\mathtt{call(}i\mathtt{,}e\mathtt{)}) = \mathtt{call(}eval_{ident}(i)\mathtt{,}f(e)\mathtt{)} \\
eval_{exp \times exp} &= (x, y) \rightarrow (eval_{exp}(x), eval_{exp}(y)) \\
eval_{exp \times exp \times exp} &= (x, y, z) \rightarrow (eval_{exp}(x), eval_{exp}(y), eval_{exp}(z)) \\
eval_{ident \times exp} &= (x, y) \rightarrow (eval_{ident}(x), eval_{exp}(y)) \\
eval_{ident \times exp \times decs} &= (x, y, z) \rightarrow (eval_{ident}(x), eval_{exp}(y), eval_{decs}(z)) \\
eval_{decs} &= f \\
&\text{where } f(\mathtt{empty}) = \mathtt{empty} \\
&\phantom{\text{where }} f(\mathtt{=;(}x\mathtt{,}y\mathtt{,}z\mathtt{)}) = \mathtt{=;(}eval_{ident}(x)\mathtt{,}eval_{exp}(y)\mathtt{,}eval_{decs}(z)\mathtt{)} \\
eval_{exp \times decs} &= (x, y) \rightarrow (eval_{exp}(x), eval_{exp}(y)) \\
eval_{prg} &= f \\
&\text{where } f(\mathtt{where(}x\mathtt{,}y\mathtt{)}) = I_{Sem}(apply)(eval_{exp}(x), eval_{decs}(y))
\end{aligned}
$$

A.3.4 The *learn* transformation

$$
\begin{array}{lll}
\mathit{learn}_{\mathrm{T}} & = & 1_{\emptyset} \\
\mathit{learn}_{ident} & = & f \text{ where } \begin{array}{lcl} f(0) & = & \mathtt{x} \\ f(1+y) & = & \mathtt{x}f(y) \end{array} \\
\mathit{learn}_{num} & = & f \text{ where } \begin{array}{lcl} f(0) & = & \mathtt{0} \\ f(1+y) & = & \mathtt{succ(}f(y)\mathtt{)} \end{array} \\
\mathit{learn}_{exp} & = & f \text{ where } \begin{array}{lcl} f(\mathtt{arg}) & = & \mathtt{arg} \\ f(\mathtt{undef}) & = & \mathtt{error} \\ f(\mathtt{is\_num(}n\mathtt{)}) & = & \mathtt{is\_num(}\mathit{learn}_{num}(n)\mathtt{)} \\ f(\mathtt{fst(}x\mathtt{)}) & = & \mathtt{fst(}f(x)\mathtt{)} \\ f(\mathtt{snd(}x\mathtt{)}) & = & \mathtt{snd(}f(x)\mathtt{)} \\ f(\mathtt{(}x\mathtt{,}y\mathtt{)}) & = & \mathtt{(}f(x)\mathtt{,}f(y)\mathtt{)} \\ f(\mathtt{=(}x\mathtt{,}y\mathtt{)}) & = & \mathtt{=(}f(x)\mathtt{,}\mathrm{f(y)}\mathtt{)} \\ f(\mathtt{if(}x\mathtt{,}y\mathtt{,}z\mathtt{)}) & = & \mathtt{if(}f(x)\mathtt{,}f(y)\mathtt{,}f(z)\mathtt{)} \\ f(\mathtt{call(}i\mathtt{,}e\mathtt{)}) & = & \mathtt{call(}\mathit{learn}_{ident}(i)\mathtt{,}f(e)\mathtt{)} \end{array} \\
\mathit{learn}_{exp \times exp} & = & (x,y) \to (\mathit{learn}_{exp}(x), \mathit{learn}_{exp}(y)) \\
\mathit{learn}_{exp \times exp \times exp} & = & (x,y,z) \to (\mathit{learn}_{exp}(x), \mathit{learn}_{exp}(y), \mathit{learn}_{exp}(z)) \\
\mathit{learn}_{ident \times exp} & = & (x,y) \to (\mathit{learn}_{ident}(x), \mathit{learn}_{exp}(y)) \\
\mathit{learn}_{ident \times exp \times decs} & = & (x,y,z) \to (\mathit{learn}_{ident}(x), \mathit{learn}_{exp}(y), \mathit{learn}_{decs}(z)) \\
\mathit{learn}_{decs} & = & f
\end{array}
$$

$$
\text{where } \begin{array}{lcl} f(\mathtt{empty}) & = & \mathtt{empty} \\ f(\mathtt{=;(}x\mathtt{,}y\mathtt{,}z\mathtt{)}) & = & \mathtt{=;(}\mathit{learn}_{ident}(x)\mathtt{,}\ \mathit{learn}_{exp}(y)\mathtt{,}\ \mathit{learn}_{decs}(z)\mathtt{)} \end{array}
$$

$$
\begin{array}{lll}
\mathit{learn}_{exp \times decs} & = & (x,y) \to (\mathit{learn}_{exp}(x), \mathit{learn}_{exp}(y)) \\
\mathit{learn}_{prg} & = & f \text{ where } f(x) = \mathtt{where(}\mathit{learn}_{exp}(x)\mathtt{,empty)}
\end{array}
$$

A.4 A *Toy* datatype to represent *Toy* programs

The datatype is described as a pair of transformations:

$$
\begin{aligned}
\mathit{encode} &: I_{Syn} \to E^{*}(I_{Sem}) \\
\mathit{decode} &: E^{*}(I_{Sem}) \to I_{Syn}
\end{aligned}
$$

such that $\mathit{decode} \circ \mathit{encode} = 1_{ISyn}$.

$$
\begin{aligned}
&\mathit{encode}_{\mathrm{T}} && = \emptyset \mapsto \texttt{(0,0)} \\
&\mathit{encode}_{ident} && = y \rightarrow \texttt{(1,}f(y)\texttt{)} \\
&\quad \text{where } f(\texttt{x}) && = \texttt{0} \\
&\quad\phantom{\text{where }} f(\texttt{x}y) && = 1 + f(y) \\
&\mathit{encode}_{num} && = y \rightarrow \texttt{(2,}f(y)\texttt{)} \\
&\quad \text{where } f(\texttt{0}) && = \texttt{0} \\
&\quad\phantom{\text{where }} f(\texttt{succ(}y\texttt{)}) && = 1 + f(y) \\
&\mathit{encode}_{decs} && = d \rightarrow \texttt{(3,}f(d)\texttt{)} \\
&\quad \text{where } f(\texttt{empty}) && = \texttt{(0,0)} \\
&\quad\phantom{\text{where }} f(\texttt{=;(}i\texttt{,}e\texttt{,}d\texttt{)}) && = \texttt{(1,}\mathit{encode}_{ident \times exp \times decs}(i, e, d)\texttt{)} \\
&\mathit{encode}_{exp} && = x \rightarrow \texttt{(4,}f(x)\texttt{)} \\
&\quad \text{where } f(\texttt{arg}) && = \texttt{(0,0)} \\
&\quad\phantom{\text{where }} f(\texttt{error}) && = \texttt{(0,1)} \\
&\quad\phantom{\text{where }} f(\texttt{is\_num(}n\texttt{)}) && = \texttt{(1,}\mathit{encode}_{num}(n)\texttt{)} \\
&\quad\phantom{\text{where }} f(\texttt{fst(}x\texttt{)}) && = \texttt{(2,}\mathit{encode}_{exp}(x)\texttt{)} \\
&\quad\phantom{\text{where }} f(\texttt{snd(}x\texttt{)}) && = \texttt{(3,}\mathit{encode}_{exp}(x)\texttt{)} \\
&\quad\phantom{\text{where }} f(\texttt{(}x\texttt{,}y\texttt{)}) && = \texttt{(4,}\mathit{encode}_{exp \times exp}(x, y)\texttt{)} \\
&\quad\phantom{\text{where }} f(\texttt{=(}x\texttt{,}y\texttt{)}) && = \texttt{(5,}\mathit{encode}_{exp \times exp}(x, y)\texttt{)} \\
&\quad\phantom{\text{where }} f(\texttt{if(}x\texttt{,}y\texttt{,}z\texttt{)}) && = \texttt{(6,}\mathit{encode}_{exp \times exp \times exp}(x, y, z)\texttt{)} \\
&\quad\phantom{\text{where }} f(\texttt{call(}i\texttt{,}e\texttt{)}) && = \texttt{(7,}\mathit{encode}_{ident \times exp}(i, e)\texttt{)} \\
&\mathit{encode}_{exp \times exp} && = (x, y) \rightarrow \texttt{(5,(}\mathit{encode}_{exp}(x),\ \mathit{encode}_{exp}(y)\texttt{))} \\
&\mathit{encode}_{exp \times exp \times exp} && = (x, y, z) \rightarrow \texttt{(6,(}\mathit{encode}_{exp}(x),\ \texttt{(}\mathit{encode}_{exp}(y),\ \mathit{encode}_{exp}(z)\texttt{)))} \\
&\mathit{encode}_{ident \times exp} && = (x, y) \rightarrow \texttt{(7,(}\mathit{encode}_{ident}(x),\ \mathit{encode}_{exp}(y)\texttt{))} \\
&\mathit{encode}_{ident \times exp \times decs} && = \\
& && \quad (x, y, z) \rightarrow \texttt{(8,(}\mathit{encode}_{ident}(x),\ \texttt{(}\mathit{encode}_{exp}(y),\ \mathit{encode}_{decs}(z)\texttt{)))} \\
&\mathit{encode}_{exp \times decs} && = (x, y) \rightarrow \texttt{(9,(}\mathit{encode}_{exp}(x),\ \mathit{encode}_{decs}(y)\texttt{))} \\
&\mathit{encode}_{prg} && = (\texttt{where(}e\texttt{,}d\texttt{)}) \rightarrow \texttt{(10,}\mathit{encode}_{exp \times decs}(e, d)\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{decode}_{\mathrm{T}} && = \ \texttt{((0,0))} \mapsto \emptyset \\
&\mathit{decode}_{ident} && = \ \texttt{((1,}y\texttt{))} \rightarrow f(y) \\
&\quad \text{where} \quad f(0) && = \ \texttt{x} \\
&\qquad\qquad f(1+y) && = \ \texttt{x}f(y) \\
&\mathit{decode}_{num} && = \ \texttt{((2,}y\texttt{))} \rightarrow f(y) \\
&\quad \text{where} \quad f(0) && = \ \texttt{0} \\
&\qquad\qquad f(1+y) && = \ \texttt{succ(}f(y)\texttt{)} \\
&\mathit{decode}_{decs} && = \ \texttt{((3,}d\texttt{))} \rightarrow f(d) \\
&\quad \text{where} \quad f(\texttt{(0,0)}) && = \ \texttt{empty} \\
&\qquad\qquad f(\texttt{(1,}x\texttt{)}) && = \ \texttt{=;(}i\texttt{,}e\texttt{,}d\texttt{)} \\
&\qquad\qquad\qquad \text{where} \quad (i,e,d) \ = \ \mathit{decode}_{ident \times exp \times decs}(x) \\
&\mathit{decode}_{exp} && = \ \texttt{(4,}x\texttt{)} \rightarrow f(x) \\
&\quad \text{where} \quad f(\texttt{(0,0)}) && = \ \texttt{arg} \\
&\qquad\qquad f(\texttt{(0,1)}) && = \ \texttt{error} \\
&\qquad\qquad f(\texttt{(1,}n\texttt{)}) && = \ \texttt{is\_num(}\mathit{decode}_{num}(n)\texttt{)} \\
&\qquad\qquad f(\texttt{(2,}x\texttt{)}) && = \ \texttt{fst(}\mathit{decode}_{exp}(x)\texttt{)} \\
&\qquad\qquad f(\texttt{(3,}x\texttt{)}) && = \ \texttt{snd(}\mathit{decode}_{exp}(x)\texttt{)} \\
&\qquad\qquad f(\texttt{(4,}x\texttt{)}) && = \ \texttt{(}a\texttt{,}b\texttt{)} \\
&\qquad\qquad\qquad\qquad \text{where} \quad (a,b) \ = \ \mathit{decode}_{exp \times exp}(x) \\
&\qquad\qquad f(\texttt{(5,}x\texttt{)}) && = \ \texttt{=(}a\texttt{,}b\texttt{)} \\
&\qquad\qquad\qquad\qquad \text{where} \quad (a,b) \ = \ \mathit{decode}_{exp \times exp}(x) \\
&\qquad\qquad f(\texttt{(6,}x\texttt{)}) && = \ \texttt{if(}a\texttt{,}b\texttt{,}c\texttt{)} \\
&\qquad\qquad\qquad\qquad \text{where} \quad (a,b) \ = \ \mathit{decode}_{exp \times exp \times exp}(x) \\
&\qquad\qquad f(\texttt{(7,}x\texttt{)} && = \ \texttt{call(}i\texttt{,}e\texttt{)} \\
&\qquad\qquad\qquad\qquad \text{where} \quad (i,e) \ = \ \mathit{decode}_{ident \times exp}(x) \\
&\mathit{decode}_{exp \times exp} && = \ \texttt{((5,(}x\texttt{,}y\texttt{)))} \rightarrow (\mathit{decode}_{exp}(x), \mathit{decode}_{exp}(y)) \\
&\mathit{decode}_{exp \times exp \times exp} && = \ \texttt{((6,(}x\texttt{,}y\texttt{,}z\texttt{)))} \rightarrow (\mathit{decode}_{exp}(x), \mathit{decode}_{exp}(y), \mathit{decode}_{exp}(z)) \\
&\mathit{decode}_{ident \times exp} && = \ \texttt{((7,(}x\texttt{,}y\texttt{)))} \rightarrow (\mathit{decode}_{ident}(x), \mathit{decode}_{exp}(y)) \\
&\mathit{decode}_{ident \times exp \times decs} && = \ \texttt{((8,(}x\texttt{,(}y\texttt{,}z\texttt{))))} \rightarrow (\mathit{decode}_{ident}(x), \mathit{decode}_{exp}(y), \mathit{decode}_{decs}(z)) \\
&\mathit{decode}_{exp \times decs} && = \ \texttt{((9,(}x\texttt{,}y\texttt{)))} \rightarrow (\mathit{decode}_{exp}(x), \mathit{decode}_{decs}(y)) \\
&\mathit{decode}_{prg} && = \ \texttt{((10,}x\texttt{))} \rightarrow \texttt{where(}e\texttt{,}d\texttt{)} \\
&\quad \text{where} \quad (e,d) \ = \ \mathit{decode}_{exp \times decs}(x)
\end{aligned}
$$

A.5 The *Toy* self-interpreter

A.5.1 The interpreter function

This function *interpreter* : $I_{Syn} \rightarrow I_{Syn}$ is defined as *interpreter* = *learn* ∘ *eval*. This definition expands to the one shown below.

$$
\begin{aligned}
&\mathit{interpreter}_{\mathbf{T}} && = && 1_{\emptyset} \\
&\mathit{interpreter}_{ident} && = && 1_{I_{Syn}(ident)} \\
&\mathit{interpreter}_{num} && = && 1_{I_{Syn}(num)} \\
&\mathit{interpreter}_{exp} && = && f \\
&\quad \text{where} \quad f(\texttt{arg}) && = && \texttt{arg} \\
&\qquad\qquad f(\texttt{error}) && = && \texttt{error} \\
&\qquad\qquad f(\texttt{is\_num(}n\texttt{)}) && = && \texttt{is\_num(}n\texttt{)} \\
&\qquad\qquad f(\texttt{fst(}e\texttt{)}) && = && F_{fst:exp\rightarrow exp}(f(e)) \\
&\qquad\qquad f(\texttt{snd(}e\texttt{)}) && = && F_{snd:exp\rightarrow exp}(f(e)) \\
&\qquad\qquad f(\texttt{(}x\texttt{,}y\texttt{)}) && = && \texttt{(}f(x)\texttt{,}f(y)\texttt{)} \\
&\qquad\qquad f(\texttt{=(}x\texttt{,}y\texttt{)}) && = && F_{=:exp\times exp\rightarrow exp}(f(x), f(y)) \\
&\qquad\qquad f(\texttt{if(}x\texttt{,}y\texttt{,}z\texttt{)}) && = && F_{if:exp\times exp\times exp\rightarrow exp}(f(x), f(y), f(z)) \\
&\qquad\qquad f(\texttt{call(}i\texttt{,}e\texttt{)}) && = && \texttt{call(}i\texttt{,}f(e)\texttt{)} \\
&\mathit{interpreter}_{exp\times exp} && = && (x, y) \rightarrow (\mathit{interpreter}_{exp}(x), \mathit{interpreter}_{exp}(y)) \\
&\mathit{interpreter}_{exp\times exp\times exp} && = && \\
&\qquad\qquad (x, y, z) \rightarrow (\mathit{interpreter}_{exp}(x), \mathit{interpreter}_{exp}(y), \mathit{interpreter}_{exp}(z)) \\
&\mathit{interpreter}_{ident\times exp} && = && (x, y) \rightarrow (\mathit{interpreter}_{ident}(x), \mathit{interpreter}_{exp}(y)) \\
&\mathit{interpreter}_{ident\times exp\times decs} && = && \\
&\qquad\qquad (x, y, z) \rightarrow (\mathit{interpreter}_{ident}(x), \mathit{interpreter}_{exp}(y), \mathit{interpreter}_{decs}(z)) \\
&\mathit{interpreter}_{decs} && = && f \\
&\quad \text{where} \quad f(\texttt{empty}) && = && F_{empty:\mathbf{T}\rightarrow decs}(\texttt{empty}) \\
&\qquad\qquad f(\texttt{=;(}i\texttt{,}e\texttt{,}d\texttt{)}) && = && F_{=;:ident\times exp\times decs\rightarrow decs}(i, \mathit{interpreter}_{exp}(e), f(d)) \\
&\mathit{interpreter}_{exp\times decs} && = && (x, y) \rightarrow (\mathit{interpreter}_{exp}(x), \mathit{interpreter}_{decs}(x)) \\
&\mathit{interpreter}_{prg} && = && (\texttt{where(}e\texttt{,}d\texttt{)}) \rightarrow F_{where:exp\times decs\rightarrow prg}(e, d)
\end{aligned}
$$

The functions used in the definition above are defined by theorem 6.1.1 and are specified below.

$$\begin{array}{lcl}
F_{empty:\mathrm{T}\rightarrow decs} & = & \emptyset \mapsto \mathtt{empty} \\
F_{=;:ident\times exp\times decs\rightarrow decs} & = & (x,y,z)\rightarrow x\ \mathtt{=}\ y\ ;\ z
\end{array}$$

$$\begin{array}{lcll}
F_{fst:exp\rightarrow exp}(x) & = & \mathtt{error}, & x=\mathtt{error} \\
 & = & x, & x=\mathtt{is\_num}(y) \\
 & = & a, & x=(a\mathtt{,}b) \\
 & = & \mathtt{fst}(x), & \textbf{otherwise}
\end{array}$$

$$\begin{array}{lcll}
F_{snd:exp\rightarrow exp}(x) & = & \mathtt{error}, & x=\mathtt{error} \\
 & = & x, & x=\mathtt{is\_num}(y) \\
 & = & b, & x=(a\mathtt{,}b) \\
 & = & \mathtt{fst}(x), & \textbf{otherwise}
\end{array}$$

$$\begin{array}{lcll}
F_{=:exp\times exp\rightarrow exp}(x,y) & = & \mathtt{is\_num}(equal(a,b)), & x=\mathtt{is\_num}(a)\wedge\mathtt{is\_num}(b) \\
 & = & \mathtt{error}, & x=(a\mathtt{,}b)\vee y=(c\mathtt{,}d)\vee \\
 & & & x=\mathtt{error}\vee y=\mathtt{error} \\
 & = & \mathtt{=}(x\mathtt{,}y), & \textbf{otherwise}
\end{array}$$

$$\begin{array}{llcl}
\text{where} & equal(\mathtt{0},\mathtt{0}) & = & \mathtt{0} \\
 & equal(\mathtt{succ}(x),\mathtt{0}) & = & \mathtt{succ}(\mathtt{0}) \\
 & equal(\mathtt{0},\mathtt{succ}(x)) & = & \mathtt{succ}(\mathtt{0}) \\
 & equal(\mathtt{succ}(x),\mathtt{succ}(y)) & = & equal(x,y)
\end{array}$$

$$\begin{array}{lcll}
F_{if:exp\times exp\times exp\rightarrow exp}(x,y,z) & = & \mathtt{error}, & x=\mathtt{error} \\
 & = & y, & x=\mathtt{is\_num}(\mathtt{0}) \\
 & = & z, & x=\mathtt{is\_num}(\mathtt{succ}(y))\vee x=(a\mathtt{,}b) \\
 & = & \mathtt{if}(x\mathtt{,}y\mathtt{,}z), & \textbf{otherwise}
\end{array}$$

$$F_{where:exp\times decs\rightarrow prg}(x\,;y) \;=\; \mathtt{where}(F_{apply:exp\times decs\rightarrow exp}(x,y)\mathtt{,}\ \mathtt{empty})$$

$$\begin{aligned}
F_{apply:exp\times decs\rightarrow exp}(\texttt{arg}, d) &= \texttt{arg}\\
F_{apply:exp\times decs\rightarrow exp}(\texttt{error}, d) &= \texttt{error}\\
F_{apply:exp\times decs\rightarrow exp}(\texttt{is\_num}(n), d) &\\
&= \texttt{is\_num}(n)\\
F_{apply:exp\times decs\rightarrow exp}(\texttt{fst}(e), d) &= F_{fst:exp\rightarrow exp}(F_{apply:exp\times decs\rightarrow exp}(e, d))\\
F_{apply:exp\times decs\rightarrow exp}(\texttt{snd}(e), d) &= F_{snd:exp\rightarrow exp}(F_{apply:exp\times decs\rightarrow exp}(e, d))\\
F_{apply:exp\times decs\rightarrow exp}((x, y), d) &= (F_{apply:exp\times decs\rightarrow exp}(x, d), F_{apply:exp\times decs\rightarrow exp}(y, d))\\
F_{apply:exp\times decs\rightarrow exp}(\texttt{=}(x, y), d) &\\
= F_{=:exp\times exp\rightarrow exp}(F_{apply:exp\times decs\rightarrow exp}(x, d), &\, F_{apply:exp\times decs\rightarrow exp}(y, d))\\
F_{apply:exp\times decs\rightarrow exp}(\texttt{if}(x, y, z), d) &\\
= F_{apply:exp\times decs\rightarrow exp}(F_{if:exp\times exp\times exp\rightarrow exp}(F_{apply:exp\times decs\rightarrow exp}(x, d), &\, y, z), d)\\
F_{apply:exp\times decs\rightarrow exp}(\texttt{call}(i, e), d) &\\
= F_{apply:exp\times decs\rightarrow exp}(F_{replace:exp\times exp\rightarrow exp}(body, F_{apply:exp\times decs\rightarrow exp}(e, d)), &\, d)\\
\text{where}\quad body &= F_{fetch:ident\times decs\rightarrow exp}(i, d)
\end{aligned}$$

$$\begin{aligned}
F_{replace:exp\times exp\rightarrow exp}(\texttt{arg}, r) &= r\\
F_{replace:exp\times exp\rightarrow exp}(\texttt{error}, r) &= \texttt{error}\\
F_{replace:exp\times exp\rightarrow exp}(\texttt{is\_num}(n), r) &= \texttt{is\_num}(n)\\
F_{replace:exp\times exp\rightarrow exp}(\texttt{fst}(e), r) &= F_{fst:exp\rightarrow exp}(F_{replace:exp\times exp\rightarrow exp}(e, r))\\
F_{replace:exp\times exp\rightarrow exp}(\texttt{snd}(e), r) &= F_{snd:exp\times exp\rightarrow exp}(F_{replace:exp\times exp\rightarrow exp}(e, r))\\
F_{replace:exp\times exp\rightarrow exp}(\texttt{=}(x, y), r) &\\
= F_{=:exp\times exp\rightarrow exp}(F_{replace:exp\times exp\rightarrow exp}(x, r), &\, F_{replace:exp\times exp\rightarrow exp}(y, r))\\
F_{replace:exp\times exp\rightarrow exp}((x, y), r) &\\
&= (F_{replace:exp\times exp\rightarrow exp}(x, r), F_{replace:exp\times exp\rightarrow exp}(y, r))\\
F_{replace:exp\times exp\rightarrow exp}(\texttt{if}(x, y, z), r) &= F_{if=:exp\times exp\times exp\rightarrow exp}(x', y', z')\\
\text{where}\quad x' &= F_{replace:exp\times exp\rightarrow exp}(x, r)\\
y' &= F_{replace:exp\times exp\rightarrow exp}(y, r)\\
z' &= F_{replace:exp\times exp\rightarrow exp}(z, r)\\
F_{replace:exp\times exp\rightarrow exp}(\texttt{call}(i, e), r) &= \texttt{call}(i, F_{replace:exp\times exp\rightarrow exp}(e, r))
\end{aligned}$$

$$
\begin{aligned}
&F_{fetch:ident\times decs\rightarrow exp}(x,\texttt{empty}) \;=\; \texttt{error}\\
&F_{fetch:ident\times decs\rightarrow exp}(x,\texttt{=;}\,(y,e,d))\\
&\qquad = \; F_{get:ident\times num\times exp\times decs\rightarrow exp}(x, F_{same:ident\times ident\rightarrow num}(x,y), e, d)
\end{aligned}
$$

$$
\begin{aligned}
F_{get:ident\times num\times exp\times decs\rightarrow exp}(x,0,e,d) &= e\\
F_{get:ident\times num\times exp\times decs\rightarrow exp}(x,\texttt{succ}(n),e,d) &= F_{fetch:ident\times decs\rightarrow exp}(x,d)
\end{aligned}
$$

$$
\begin{aligned}
F_{same:ident\times ident\rightarrow num}(0,0) &= 0\\
F_{same:ident\times ident\rightarrow num}(\texttt{succ}(x),0) &= \texttt{succ}(0)\\
F_{same:ident\times ident\rightarrow num}(0,\texttt{succ}(x)) &= \texttt{succ}(0)\\
F_{same:ident\times ident\rightarrow num}(\texttt{succ}(x),\texttt{succ}(y)) &= F_{same:ident\times ident\rightarrow num}(x,y)
\end{aligned}
$$

A.5.2 The *rep_interpreter* function

We now define the function *rep_int* : $E^*(I_{Sem}) \rightarrow E^*(I_{Sem})$. This function is defined using the functions: *interpreter*, *decode*, and *encode* as *rep..int* = *encode* $\circ$ *interpreter* $\circ$ *decode*. This definition expands to the one shown below.

$$
\begin{aligned}
\mathit{rep\_int}_{\mathrm{T}} &= (0,0) \mapsto (0,0) \\
\mathit{rep\_int}_{ident} &= (1,x) \rightarrow (1,x) \\
\mathit{rep\_int}_{num} &= (2,x) \rightarrow (2,x) \\
\mathit{rep\_int}_{exp} &= (4,x) \rightarrow f(x) \\
\text{where } \quad f((0,0)) &= (0,0) \\
f((0,1)) &= (0,1) \\
f((1,n)) &= (1,n) \\
f((2,x)) &= \mathit{rep\_F}_{fst:exp\rightarrow exp}(\mathit{rep\_int}_{exp}(x)) \\
f((3,x)) &= \mathit{rep\_F}_{snd:exp\rightarrow exp}(\mathit{rep\_int}_{exp}(x)) \\
f((4,(5,(x,y)))) &= (4,(5,(\mathit{rep\_int}_{exp}(x),\mathit{rep\_int}_{exp}(y)))) \\
f((5,(5,(x,y)))) &= \mathit{rep\_F}_{=:exp\times exp\rightarrow exp}(x,y) \\
f((6,(6,(x,(y,z))))) &= \mathit{rep\_F}_{if:exp\times exp\times exp\rightarrow exp}(x,y,z) \\
f((7,(7,(i,e)))) &= (7,(7,(i,\mathit{rep\_int}_{exp}(e)))) \\
\mathit{rep\_int}_{exp\times exp} &= ((5,(x,y))) \rightarrow (5,(\mathit{rep\_int}_{exp}(x),\ \mathit{rep\_int}_{exp}(y))) \\
\mathit{rep\_int}_{exp\times exp\times exp} &= \\
&((6,(x,(y,z)))) \rightarrow (6,(\mathit{rep\_int}_{exp}(x),(\mathit{rep\_int}_{exp}(y),\mathit{rep\_int}_{exp}(z)))) \\
\mathit{rep\_int}_{ident\times exp} &= ((7,(x,y))) \rightarrow (7,(\mathit{rep\_int}_{ident}(x),\mathit{rep\_int}_{exp}(y))) \\
\mathit{rep\_int}_{ident\times exp\times decs} &= \\
&(8,(x,(y,z))) \rightarrow (8,(\mathit{rep\_int}_{ident}(x),(\mathit{rep\_int}_{exp}(y),\mathit{rep\_int}_{decs}(z)))) \\
\mathit{rep\_int}_{decs} &= (3,x) \rightarrow f(x) \\
\text{where } \quad f((0,0)) &= \mathit{rep\_F}_{empty:\mathrm{T}\rightarrow decs}((0,0)) \\
f((1,(8,(i,(e,d))))) &= \\
&\mathit{rep\_F}_{=;::ident\times exp\times decs\rightarrow decs}(i,\mathit{rep\_int}(e),\mathit{rep\_int}_{decs}(d)) \\
\mathit{rep\_int}_{exp\times decs} &= (9,(x,y)) \rightarrow (9,(\mathit{rep\_int}_{exp}(x),\mathit{rep\_int}_{decs}(y)) \\
\mathit{rep\_int}_{prg} &= (10,(9,(e,d))) \rightarrow \mathit{rep\_F}_{where:exp\times decs\rightarrow prg}(e,d)
\end{aligned}
$$

The functions used in the definition above are the transformed versions of those defined by theorem 6.1.1 and are specified below.

$$
\begin{aligned}
\mathit{rep\_F}_{empty:\mathrm{T}\rightarrow decs} &= (0,0) \mapsto (3,(0,0)) \\
\mathit{rep\_F}_{=;::ident\times exp\times decs\rightarrow decs} &= (i,e,d) \rightarrow (3,(1,(8,(i,(e,d)))))
\end{aligned}
$$

$$\begin{aligned} \mathit{rep\_F}_{fst:exp\rightarrow exp}(x) &= (4,(0,1)), & x &= (4,(0,1)) \\ &= x, & x &= (4,(1,y)) \\ &= a, & x &= (4,(4,(5,(a,b)))) \\ &= (4,(2,x)), & &\textbf{otherwise} \end{aligned}$$

$$\begin{aligned} \mathit{rep\_F}_{snd:exp\rightarrow exp}(x) &= (4,(0,1)), & x &= (4,(0,1)) \\ &= x, & x &= (4,(1,y)) \\ &= b, & x &= (4,(4,(5,(a,b)))) \\ &= (4,(3,x)), & &\textbf{otherwise} \end{aligned}$$

$$\begin{aligned} &\mathit{rep\_F}_{=:exp\times exp\rightarrow exp}(x,y) \\ &\qquad = (4,(1,\mathit{rep\_equal}(a,b))), && x = (4,(1,a)) \wedge y = (4,(1,b)) \\ &\qquad = (4,(0,1)), && x = (4,(4,a)) \vee y = (4,(4,b)) \vee \\ &&& x = (4,(0,1)) \vee y = (4,(0,1)) \\ &\qquad = (4,(5,(x,y))), && \textbf{otherwise} \end{aligned}$$

$$\begin{aligned} \text{where}\quad \mathit{rep\_equal}((2,0),(2,0)) &= (2,0) \\ \mathit{rep\_equal}((2,x+1),(2,0)) &= (2,1) \\ \mathit{rep\_equal}((2,0),(2,x+1)) &= (2,1) \\ \mathit{rep\_equal}((2,x+1),(2,y+1)) &= \mathit{rep\_equal}(x,y) \end{aligned}$$

$$\begin{aligned} &\mathit{rep\_F}_{if:exp\times exp\times exp\rightarrow exp}(x,y,z) \\ &\quad = (4,(0,1)), && x = (4,(0,1)) \\ &\quad = y, && x = (4,(1,0)) \\ &\quad = z, && x = (4,(1,y+1)) \vee x = (4,(4,a)) \\ &\quad = (4,(6,(6,(x,(y,z))))), && \textbf{otherwise} \end{aligned}$$

$$\mathit{rep\_F}_{where:exp\times decs\rightarrow prg}(e,d) = (10,(9,(\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(x,y),(3,(0,0)))))$$

$$
\begin{aligned}
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(0,0))},d) &= \texttt{(4,(0,0))}\\
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(0,1))},d) &= \texttt{(4,(0,1))}\\
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(1,}n\texttt{))},d) &= \texttt{(4,(1,}n\texttt{))}\\
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(2,}e\texttt{))},d) &= \mathit{rep\_F}_{fst:exp\rightarrow exp}(\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(e,d))\\
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(3,}e\texttt{))},d) &= \mathit{rep\_F}_{snd:exp\rightarrow exp}(\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(e,d))
\end{aligned}
$$

$$
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(4,(5,(}x\texttt{,}y\texttt{))))},d)
$$

$$
= \texttt{(4,(4,(}\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(x,d)\texttt{,}\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(y,d)\texttt{)))}.
$$

$$
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(5,(5,(}x\texttt{,}y\texttt{))))},d)
$$

$$
= \mathit{rep\_F}_{=:exp\times exp\rightarrow exp}(\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(x,d), \mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(y,d))
$$

$$
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(6,(6,(}x\texttt{,(}y\texttt{,}z\texttt{)))))},d)
$$

$$
= \mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\mathit{rep\_F}_{if:exp\times exp\times exp\rightarrow exp}(\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(x,d),y,z),d)
$$

$$
\mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\texttt{(4,(7,(}i\texttt{,}e\texttt{)))},d)
$$

$$
= \mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(\mathit{rep\_F}_{replace:exp\times exp\rightarrow exp}(\mathit{body}, \mathit{rep\_F}_{apply:exp\times decs\rightarrow exp}(e,d)),d)
$$

where $\mathit{body} = \mathit{rep\_F}_{fetch:ident\times decs\rightarrow exp}(i,d)$

$$\begin{aligned}
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(0,0)),r) &= r\\
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(0,1)),r) &= (4,(0,1))\\
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(1,n)),r) &= (4,(1,n))\\
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(2,e)),r) &= \\
&\quad rep\_F_{fst:exp\rightarrow exp}(rep\_F_{replace:exp\times exp\rightarrow exp}(e,r))\\
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(3,e)),r) &= \\
&\quad rep\_F_{snd:exp\times exp\rightarrow exp}(rep\_F_{replace:exp\times exp\rightarrow exp}(e,r))
\end{aligned}$$

$$\begin{aligned}
&rep\_F_{replace:exp\times exp\rightarrow exp}((4,(5,(5,(x,y)))),r)\\
&\quad = rep\_F_{=:exp\times exp\rightarrow exp}(rep\_F_{replace:exp\times exp\rightarrow exp}(x,r), F_{replace:exp\times exp\rightarrow exp}(y,r))\\
&rep\_F_{replace:exp\times exp\rightarrow exp}((4,(4,(5,(x,y)))),r)\\
&\quad = (rep\_F_{replace:exp\times exp\rightarrow exp}(x,r), rep\_F_{replace:exp\times exp\rightarrow exp}(y,r))\\
&rep\_F_{replace:exp\times exp\rightarrow exp}((4,(6,(6,(x,(y,z))))),r)\\
&\quad = rep\_F_{if:exp\times exp\times exp\rightarrow exp}(x',y',z')
\end{aligned}$$

$$\text{where}\quad \begin{aligned}
x' &= rep\_F_{replace:exp\times exp\rightarrow exp}(x,r)\\
y' &= rep\_F_{replace:exp\times exp\rightarrow exp}(y,r)\\
z' &= rep\_F_{replace:exp\times exp\rightarrow exp}(z,r)
\end{aligned}$$

$$\begin{aligned}
rep\_F_{replace:exp\times exp\rightarrow exp}((4,(7,(7,(i,e)))),r) &= \\
&\quad (4,(7,(7,(i,rep\_F_{replace:exp\times exp\rightarrow exp}(e,r)))))
\end{aligned}$$

$$\begin{aligned}
&rep\_F_{fetch:ident\times decs\rightarrow exp}(x,(3,(0,0))) = (4,(0,1))\\
&rep\_F_{fetch:ident\times decs\rightarrow exp}(x,(3,(1,(8,(y,(e,d))))))\\
&\quad = rep\_F_{get:ident\times num\times exp\times decs\rightarrow exp}(x, rep\_F_{same:ident\times ident\rightarrow num}(x,y),e,d)
\end{aligned}$$

$$\begin{aligned}
rep\_F_{get:ident\times num\times exp\times decs\rightarrow exp}(x,(1,0),e,d) &= e\\
rep\_F_{get:ident\times num\times exp\times decs\rightarrow exp}(x,(1,n+1),e,d) &= rep\_F_{fetch:ident\times decs\rightarrow exp}(x,d)
\end{aligned}$$

$$\begin{aligned}
rep\_F_{same:ident\times ident\rightarrow num}((1,0),(1,0)) &= (1,0)\\
rep\_F_{same:ident\times ident\rightarrow num}((1,x+1),(1,0)) &= (1,1)\\
rep\_F_{same:ident\times ident\rightarrow num}((1,0),(1,x+1)) &= (1,1)\\
rep\_F_{same:ident\times ident\rightarrow num}((1,x+1),(1,y+1)) &= rep\_F_{same:ident\times ident\rightarrow num}(x,y)
\end{aligned}$$

A.6 The *self-interpreter* program

The *rep_interpreter* function is implemented in *Toy* by the program shown below. Note that we shall use numerical characters to represent numbers rather than the *Toy* representation using succ and 0. For the sake of readability we also use meaningful identifiers rather than strings of x as the *Toy* syntax specifies.

```
*where(arg) where *where  = (10,(9,(*apply(arg),(3,(0,0)))));


*apply = if fst(fst(arg)) = 4 then
         if fst(snd(fst(arg))) = 0 then fst(arg)
         else if fst(snd(fst(arg))) = 1 then fst(arg)
         else if fst(snd(fst(arg))) = 2 then
                 *fst(*apply((snd(fst(arg)),snd(arg)))
         else if fst(snd(fst(arg))) = 3 then
                 *snd(*apply((snd(fst(arg)),snd(arg)))
         else if fst(snd(fst(arg))) = 4 then
                 (4,(4,(*apply((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                        *apply((snd(snd(snd(snd(fst(arg))))),snd(arg))))))
         else if fst(snd(fst(arg))) = 5 then
                 *=(*apply((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                       *apply((snd(snd(snd(snd(fst(arg))))),snd(arg))))
         else if fst(snd(fst(arg))) = 6 then
             *apply((*if((*apply((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                        (fst(snd(snd(snd(snd(fst(arg)))))),
                        snd(snd(snd(snd(snd(fst(arg))))))))),
                 snd(arg)))
         else if fst(snd(fst(arg))) = 7 then
             *apply((*replace((*fetch(fst(snd(snd(fst(arg))))),snd(arg))),
                      *apply((snd(snd(snd(fst(arg)))),snd(arg)))))


         else error
     else error;
```

```
*fst = if fst(arg) = 4 then
           if fst(snd(arg)) = 0 then
                if sdn(snd(arg)) = 1 then (4,(0,1))
                else error
           else if fst(snd(arg)) = 1 then arg
           else if fst(snd(arg)) = 4 then fst(snd(snd(snd(arg))))
           else (4,(2,arg))
       else (4,(2,arg));

*snd = if fst(arg) = 4 then
           if fst(snd(arg)) = 0 then
                if sdn(snd(arg)) = 1 then (4,(0,1))
                else error
           else if fst(snd(arg)) = 1 then arg
           else if fst(snd(arg)) = 4 then snd(snd(snd(snd(arg))))
           else (4,(3,arg))
       else (4,(3,arg));

*= = if *and((fst(fst(arg))=4,fst(snd(arg))=4)) then
         if *and((fst(snd(fst(arg)))=1,fst(snd(snd(arg)))=1)) then
              (4,(1,*rep_equal((snd(snd(fst(arg))),snd(snd(snd(arg)))))))
         else if *or((*and((fst(snd(fst(arg)))=4,fst(snd(snd(arg)))=4)),
                     *and((*and((fst(snd(fst(arg)))=0,
                                 snd(snd(fst(arg)))=1)),
                          *and((fst(snd(snd(arg)))=0,
                                 snd(snd(snd(arg)))=1)))))) then
              (4,(0,1))
         else (4,(5,arg))
     else error;

*rep_equal = if *and((fst(fst(arg))=2,fst(snd(arg))=2)) then
                (2,snd(fst(arg))=snd(snd(arg)))
```

```
             else error;

*if = if fst(fst(arg)) = 4 then
         if *and((fst(snd(fst(arg)))=0,snd(snd(fst(arg)))=1)) then
            (4,(0,1))
         else if *and((fst(snd(fst(arg)))=1,snd(snd(fst(arg)))=0)) then
            fst(snd(arg))
         else if *or((fst(snd(fst(arg)))=1,fst(snd(fst(arg)))=4)) then
            snd(snd(arg))
         else (4,(6,(6,(fst(arg),(fst(snd(arg)),snd(snd(arg)))))))
      else (4,(6,(6,(fst(arg),(fst(snd(arg)),snd(snd(arg)))))));

*replace = if fst(fst(arg)) = 4 then
              if fst(snd(fst(arg))) = 0 then
                 if snd(snd(fst(arg))) = 0 then snd(arg)
                 else (4,(0,1))
              else if fst(snd(fst(arg))) = 1 then fst(arg)
              else if fst(snd(fst(arg))) = 2 then
                      *fst(*replace((snd(snd(fst(arg))),snd(arg))))
              else if fst(snd(fst(arg))) = 3 then
                      *snd(*replace((snd(snd(fst(arg))),snd(arg))))
              else if fst(snd(fst(arg))) = 5 then
                      *=((*replace((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                          *replace((snd(snd(snd(snd(fst(arg))))),snd(arg)))))
              else if fst(snd(fst(arg))) = 4 then
                      (*replace((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                       *replace((snd(snd(snd(snd(fst(arg))))),snd(arg))))
              else if fst(snd(fst(arg))) = 6 then
                      *if((*replace((fst(snd(snd(snd(fst(arg))))),snd(arg))),
                          (*replace((fst(snd(snd(snd(snd(fst(arg)))))),
                                     snd(arg))),
                           *replace((snd(snd(snd(snd(snd(fst(arg)))))),
                                     snd(arg))))))
              else if fst(snd(fst(arg))) = 7 then
                      (4,(7,(7,(fst(snd(snd(snd(fst(arg))))),
                          *replace((snd(snd(snd(snd(fst(arg))))),
                                    snd(arg)))))));

*fetch = if fst(snd(arg)) = 3 then
            if fst(snd(snd(arg))) = 0 then (4,(0,1))
            else if fst(snd(snd(arg))) = 1 then
                 *get((fst(arg),
                      (*same((fst(arg),fst(snd(snd(snd(snd(arg))))))),
                      (fst(snd(snd(snd(snd(snd(arg)))))),
                       snd(snd(snd(snd(snd(snd(arg)))))))))))
            else error;
         else error;

*get = if fst(fst(snd(arg))) = 1 then
          if snd(fst(snd(srg))) = 0 then fst(snd(snd(arg)))
          else *fetch((fst(arg),snd(snd(snd(arg)))))
       else error;

*same = if *and((fst(fst(arg))=1,fst(snd(arg))=1)) then
           (1,snd(fst(arg))=snd(snd(arg)))
        else error;

*and = if fst(arg) then snd(arg) else fst(arg);

*or  = if fst(arg) then fst(arg) else snd(arg);
```