From Proof Normalization to Compiler Generation and Type-Directed Change-of-Representation

René Vestergaard[1]

DAIMI
Computer Science Department
Århus University
<jrvest@daimi.aau.dk>

Volen Center for Complex Systems
Computer Science Department
Brandeis University
<jrvest@cs.brandeis.edu>

May 15, 1997

[1]The present writing serves as the "Kandidat Speciale" for the author at DAIMI. The work was commenced in Århus but finished while visiting Prof. Harry Mairson at Brandeis University. The visit was kindly supported by ONR Grant N00014-93-1-1015, NSF Grant CCR-9216185, Knud Højgaards Fond, Fam.Hede Nielsens Fond, and BRICS.

Abstract

The main part of this thesis is a synthesis of considerations from Type Theory, Mathematical Logic/Proof Theory, and (Denotational) Semantics to perform various automatic program transformations ranging from normalization over currying and coercion-insertion to compiler derivation.

At the core of our technique we have what has been described as "An Inverse of the Evaluation Functional for Typed λ-calculus" [7]. It is essentially type-directed η-expansion followed by β-reduction on certain terms. Quite independently of [7], η-expansion has been studied for its use in Partial Evaluation, where among other things it has been used to obtain a one-pass CPS-transformer [20]. It is some of the consequences of this coincidence [19] that are described in the following.

Our approach will be purely syntactical and it is hoped that it marks a simplification on earlier treatments of the materiel. We have tried presenting the materiel based purely on the standard reduction properties for the simply typed λ-calculus albeit in a slightly generalized form.

After the initial treatment we will consider to which extend existing programming languages can be used to simulate the considered calculus and as it turns out the proposed method can be handled. Following this we will look at some practical implications and conclusively we will consider generalizations.

Contents

List of Figures

Chapter 1

Introduction

This work has its roots in the work by Olivier Danvy on "Type-Directed Partial Evaluation" [19], where it was shown how —in a two-level language[1]— type-directed, two-level η-expansion combined with full, one-level β-reduction[2] essentially performs partial evaluation on λ-terms. The usefulness of controlled $\beta\eta$-rewriting for partial evaluation has been described in [21, 22] but was already used along the same lines in [20] to obtain a one-pass CPS transformer. An interesting implication, which we will pursue in Section 3.3, is that when interpreting one level as syntax and the other as syntax construction we can, when using functional languages that adequately support the present notion of two levels (e.g. Scheme), in effect perform all the syntax computation in one pass and be left only with syntax construction. In this formalism it is possible to introduce syntax constructors which are $\alpha\beta\eta$-equivalent to the considered syntax. Thus we can up to $\alpha\beta\eta$-equivalence go from syntax to syntax construction solely by looking at types and by reducing on syntax. This was first recognized by Berger & Schwichtenberg as "An Inverse of the Evaluation Functional for Type λ-calculus" [7]. It was used to specify a proof(-term) normalizer and it is that description which is at the heart of this thesis.

The contributions of this thesis are considered to be – in order of presentation:

- The definition of a new, symmetric two-level λ-calculus, Subsection 1.1.4.

[1] The notion of two-level languages is taken from [55]. Due to technical reasons we will —in Section 1.1– introduce a new, symmetric two-level λ-calculus

[2] Which is possible in the given situation

- The definition of a calculus with explicit substitution isomorphic to a specific Sequent Calculus presentation of Minimal Logic [70, System G3m], Section 2.2. Based on this we consider an alternative, type-independent Hauptsatz without the use of an ordering on derivations.

- A reformulation of Berger & Schwichtenberg's proof normalizer extended to product types in the new two-level calculus. We also give a correctness proof. We then go on to show that the method can be simulated in an existing programming thus immediately giving correctness of the implementation, Chapter 3.

- A review of some of the practical issues related to the use of Type-Directed Partial Evaluation [19], Chapter 4. In particular for performing compiler generation [23], Chapter 5.

- Considerations on generalizations of Berger & Schwichtenberg's proof normalizer to perform Type-Directed Change-of-Representation, Chapter 6.

The materiel in Chapters 3 and 6 respectively Chapter 2 constitute my two main points of interest.

1.0.1 Acknowledgments

I have been lucky to meet people that have shared my interests in Computer Science and with whom I have spent some very pleasant hours. Olivier Danvy has been my advisor and as such has contributed significantly to more than the strictly academical subjects. I spent the academical year 1996-'97 at Brandeis University with Harry Mairson, Jakov Kucan and Luca Roversi. Our discussions on proof theory, logic, and semantics have been very gratifying to me. Beyond these, the usual suspects, I have had the pleasure of interacting with Ulrich Berger, Torben Braüner, Matthias Eberl, Bernhard Gramlich, Glenn Holloway, Andrew Kennedy, Assaf Kfoury (and the rest of the Church Group at Boston University), Albert Meyer, Bob Muller, Jens Palsberg, Kristoffer Rose, Helmut Schwichtenberg, Richard Statman, Anne Troelstra, Joe Wells, and Mitch Wand (and the rest of the Programming Language Semantics group at Northeastern University) on the areas covered in this thesis. I have also interacted with a lot of other people on other areas and I am grateful for these hours.

The diagrams have been generated with Rose's Xy-pic package which proved to be a helpful tool.

Jenny Frances kindly tried to improve on my English writing skills and patiently read through pages of oddly constructed sentences, grammatical errors, and theorems.

Thank you all.

1.1 Preliminaries

We will consider different λ-calculi. They will be referred to as $\lambda_{\mathrm{C}}^{\mathrm{TS}}$ where TS denotes the considered type system and C denotes the [type of] permissible syntax constructors. We will always consider terms up to α-equality. When using other notions of equality we will state it explicitly.

As the material is more or less standard we will present notational shorthands and explanations of particularities beforehand.

- Substitution (-[-/-]) is defined to be non-capturing as usual when considering terms up to α-equivalence.
- By $M\{N\}$ we will refer to the term M containing the *term occurrence* N. A term occurrence is a term occurring at a specific position within another term.
- When concerned with indexed constructs, such as pairs and projections, we will write only "i" to collectively mean all [correct] indices.
- For the explicitly typed calculus we will omit all but the relevant type annotations.
- We use nonstandard, but in our opinion very intuitive names for the equalities. This is of course subject to aesthetical judgement. However, using this notation will prove to be a notational simplification later on. We will use the terms β- and η-equality ($=_\beta, =_\eta$) to collectively mean all such equalities within each calculus. Equality without subscript ($=$) will be used to denote equality relative to the context of its use.
- Equivalence ($\equiv$) will be used for syntactical equality on terms.
- Rewriting terms according to the equality rules will, for obvious reasons, be called:

left-to-right: reduction of M to N, written $M \rightarrow_X N$ for equality X.

right-to-left: expansion of M to N, written $M \leftarrow_X N$ for equality X.

An equality will thus correspond to the symmetric, transitive and reflexive closure of a reduction or an expansion. A term matching the originating side of an oriented equality will be said to form a *redex* that can be *rewritten* to a *reduct*.

- We will write "types in the $\lambda_{\mathrm{C}}^{\mathrm{TS}}$-calculus" to mean "types for terms in the $\lambda_{\mathrm{C}}^{\mathrm{TS}}$-calculus" and similar blurrings for simplicity.
- By the *erasure*, $\mathrm{e}(\cdot)$, of an explicitly typed term we will be referring to the same term with all type annotations stripped off. Such a term is obviously implicitly typed.
- The two-level calculus have two copies of all the usual syntactical constructions separated by overlining and underlining. By the *collapse*, $\mathrm{c}(\cdot)$, of a two-level term we will mean the one-level counterpart obtained by removing overlining and underlining. The term is readily seen to be an ordinary, implicitly typed term.
- In the two-level calculus we will take variables to belong to the level of their binding abstraction and thus not overline or underline them.
- For any reduction $(\rightarrow)$ an asterix superscript $(\rightarrow^*)$ will denote the infixed relation obtained as the transitive and reflexive closure of the reduction.

We will throughout most of this thesis consider types as generated by:

$$\begin{array}{rcl} \tau & ::= & o \\ & | & \tau_1 \rightarrow \tau_2 \\ & | & \tau_1 \times \tau_2 \end{array}$$

o denotes the elements of a countable set of fixed base types (Int, Real, Cm, etc.) and type variables. $\rightarrow$ will be used right associatively. $\times$ will be taken to be left-associative and bind tighter than $\rightarrow$. We consider both explicitly and implicitly typed terms and we will in the latter parts of Section 2.1 very precisely characterize the relevant differences in the two approaches.

1.1.1 Basic notions

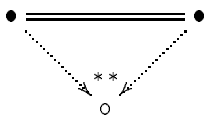
Definition 1 (Strong Normalization) *A reduction in a calculus is said to enjoy Strong Normalization (SN) if for all terms in the calculus, all reduction sequences are finite. That is, no matter what order redices are reduced in, you will reach a term without redices, a normal form.*

Definition 2 (Weak Normalization) *A reduction in a calculus is said to enjoy Weak Normalization (WN) if all terms have a normal form with respect to the reduction.*

Definition 3 (Church-Rosser) *A reduction, $\rightarrow$, in a calculus is said to enjoy Church-Rosser (CR) if terms equal by the transitive, symmetric and reflexive closure of the reduction have a common reduct by transitive, reflexive reduction ($\rightarrow^*$):*

$$\forall M_1, M_2.M_1 = M_2 \Rightarrow (\exists N.M_1 \rightarrow^* N \wedge M_2 \rightarrow^* N)$$

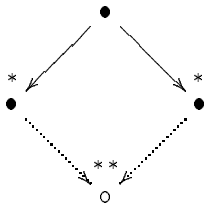
or in diagram form:



Proposition 1 *A reduction, $\rightarrow$, enjoys CR iff $\rightarrow^*$ has the diamond property*[3]*:*

$$\forall M, M_1, M_2.[(M \rightarrow^* M_1 \wedge M \rightarrow^* M_2) \Rightarrow \exists N.(M_1 \rightarrow^* N \wedge M_2 \rightarrow^* N)]$$

or in diagram form:



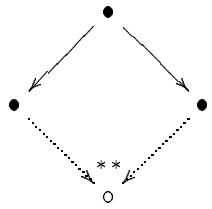
Definition 4 (Weak Church Rosser) *A reduction, $\rightarrow$, in a calculus is said to enjoy Weak Church-Rosser (WCR) if it has the weak diamond property*[4]*:*

$$\forall M, M_1, M_2.[(M \rightarrow M_1 \wedge M \rightarrow M_2) \Rightarrow \exists N.(M_1 \rightarrow^* N \wedge M_2 \rightarrow^* N)]$$

or in diagram form:

[3] When considering binary relations this is normally called confluence.

[4] ... and this is called weak confluence.



Lemma 1 (Newman's Lemma) *If a reduction in a calculus enjoys both Strong Normalization and Weak Church-Rosser, it enjoys Church-Rosser:*

$$SN \wedge WCR \Rightarrow CR$$

Proposition 2 (Unique Normal Forms) *If a reduction in a calculus enjoys (W)CR, the normal forms with respect to the reduction are unique (UNF).*

1.1.2 An implicitly typed calculus

Definition 5 (Implicitly simply typed λ-calculus) *Let the pre-terms of the implicitly typed λ-calculus with pairs ($\lambda^{Curry}_{\rightarrow\times}$) be given by:*

$$\begin{array}{rcl} e & ::= & x \\ & | & k \\ & | & \lambda x.e \\ & | & e_1\, e_2 \\ & | & \langle e_1,\, e_2 \rangle \\ & | & p_i(e) \end{array}$$

The terms –M, N– are given as the provable pre-terms of

$$x : \tau \quad (Asmp) \qquad\qquad k : o \quad (Const)$$

$$\frac{\begin{array}{c}[x:\tau]_k \\ \vdots \\ M : \sigma\end{array}}{\lambda x.M : \tau \rightarrow \sigma}\,(\rightarrow I)_k \qquad\qquad \frac{\overset{\pi_M}{M : \tau \rightarrow \sigma} \quad \overset{\pi_N}{N : \tau}}{M\,N : \sigma}\,(\rightarrow E)$$

$$\frac{\overset{\pi_{M_1}}{M_1 : \tau_1} \quad \overset{\pi_{M_2}}{M_2 : \tau_2}}{\langle M,\, N\rangle : \tau \times \sigma}\,(\times I) \qquad\qquad \frac{\overset{\pi_M}{M : \tau_1 \times \tau_2}}{p_i(M) : \tau_i}\,(\times E_i)$$

We will write $M \in \Lambda^\tau$ when M provably is of type τ. Observe that we only allow constants at base type. The different notions of term equality we will use are:

$$\begin{array}{rcll}
(\lambda x.M)\,N & =_{\beta_\to} & M[N/x] & \\
p_i(\langle M_1,\, M_2\rangle) & =_{\beta_\times} & M_i & \\
\lambda x.M\,x & =_{\eta_\to} & M & ;\textit{if } x \notin FV(M) \;\&\; M \in \Lambda^{\tau\to\sigma} \\
 & & & \;\&\; M \neq \lambda y.N \\
 & & & \;\&\; M \textit{ is not applied} \\
\langle p_1(M),\, p_2(M)\rangle & =_{\eta_\times} & M & ;\textit{if} \vdash M \in \Lambda^{\tau\times\sigma} \;\&\; M \neq \langle M_1,\, M_2\rangle \\
 & & & \;\&\; M \textit{ is not projected}
\end{array}$$

A few remarks should be added about the side conditions for the η-equalities [14]. First, FV is the set of free variables in the usual sense. Secondly regarding the terms not being constructors and not being destructed. The conditions serve the purpose of giving termination results on η-expansion as otherwise all function- and product-typed terms would be infinitely expandable, Figure 1.1. They appear to originate from Mints [51], were forgotten only to be revived recently [2, 16, 40]. The combined normal-forms of β-reduction and η-expansion are (cf. [40, p. 137]) Huet's long $\beta(\eta)$-normal forms [39, Definition 4.1.2]:

$$\lambda x_1 \ldots \lambda x_n.y\,M_1 \ldots M_k \in \Lambda^{\tau_1\to\ldots\to\tau_n\to o}$$

with the M_i's in long $\beta(\eta)$-normal form. The thing to notice is of course that y is fully applied, that is $y\,M_1 \ldots M_k \in \Lambda^o$. In the case of products the form is harder to characterize concisely, but for $M \in \Lambda^{o\times\tau} \;\wedge\; M \neq \langle M_1,\, M_2\rangle$ we have:

$$\langle \mathrm{p}_1(M),\, \widehat{\mathrm{p}_2(M)}\rangle \in \Lambda^{o\times\tau}$$

where $\widehat{\mathrm{p}_2(M)}$ is the long $\beta(\eta)$-normal form of $\mathrm{p}_2(M)$. That is, the components of a pair are either a pair or of base or function type. Another way to say it is that all terms of product type are pairs or are being projected. The intuitive property of long $\beta(\eta)$-normal forms is exactly that all typing structure is syntactically explicit.

With the side-conditions in Definition 5, η-equality is immediately seen not to be a congruence[5] hence η-expansion is not compatible[6] (cf. Subsection 1.1.4). However, we will not need either.

[5] We do not have $N =_\eta N' \Rightarrow M\{N\} =_\eta M\{N'\}$.

[6] Nor do we have $N \leftarrow_\eta N' \Rightarrow M\{N\} \leftarrow_\eta M\{N'\}$.

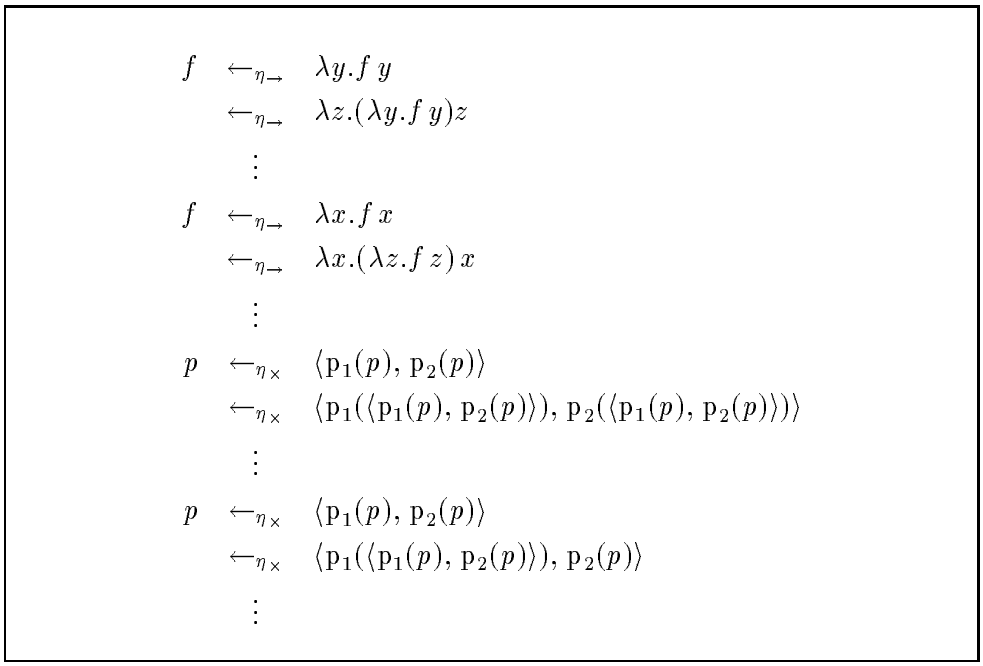


Figure 1.1: Infinite non-restricted η-expansion

Theorem 1 (Strong Normalization) *We will use the following known results:*

- *β-reduction, $\to_\beta$, enjoys SN in $\lambda_{\to\times}^{Curry}$ [30, Corollary – Section 6.3.3].*
- *η-expansion, $\leftarrow_\eta$, enjoys SN in $\lambda_{\to\times}^{Curry}$ [2, Theorem 8].*
- *$\beta\eta$-rewriting, $\to_\beta \cup \leftarrow_\eta$, enjoys SN in $\lambda_{\to\times}^{Curry}$ [2, Theorem 15].*

Theorem 2 (Church-Rosser) *We will use the following known results:*

- *β-reduction, $\to_\beta$, enjoys CR in $\lambda_{\to\times}^{Curry}$ – a corollary to Theorem 7.*
- *η-expansion, $\leftarrow_\eta$, enjoys CR in $\lambda_{\to\times}^{Curry}$ [2, Theorem 8].*
- *$\beta\eta$-rewriting, $\to_\beta \cup \leftarrow_\eta$, enjoys CR in $\lambda_{\to\times}^{Curry}$ [2, Theorem 15].*

1.1.3 An explicitly typed calculus

Definition 6 (Explicitly simply typed λ-calculus) *Let the terms of the explicitly typed λ-calculus with pairs ($\lambda_{\rightarrow\times}^{Church}$) be given by:*

$$\begin{array}{rcl} M^-, N^- & ::= & x^\tau \\ & | & k^o \\ & | & (\lambda x^\tau . M^\sigma)^{\tau\rightarrow\sigma} \\ & | & (M^{\tau\rightarrow\sigma}\, N^\tau)^\sigma \\ & | & \langle M_1^{\tau_1}, M_2^{\tau_2}\rangle^{\tau_1\times\tau_2} \\ & | & p_i(M^{\tau_1\times\tau_2})^{\tau_i} \end{array}$$

The different notions of term equality we will use are:

$$\begin{array}{rcll} (\lambda x^\tau . M^\sigma)\, N^\tau & =_{\beta_\rightarrow} & M^\sigma[N^\tau/x^\tau] & \\ p_i(\langle M_1^{\tau_1}, M_2^{\tau_2}\rangle) & =_{\beta_\times} & M_i^{\tau_i} & \\ \lambda x^\tau . M^{\tau\rightarrow\sigma}\, x^\tau & =_{\eta_\rightarrow} & M^{\tau\rightarrow\sigma} & ; \textit{if } x \notin FV(M) \textit{ \& } M \neq \lambda y.N \\ & & & \textit{\& } M \textit{ is not applied} \\ \langle p_1(M^{\tau_1\times\tau_2}), p_2(M^{\tau_1\times\tau_2})\rangle & =_{\eta_\times} & M^{\tau_1\times\tau_2} & ; \textit{if } M \neq \langle M_1, M_2\rangle \\ & & & \textit{\& } M \textit{ is not projected} \end{array}$$

Theorem 3 (Strong Normalization) *By Theorem 1 we have:*

- *β-reduction, $\rightarrow_\beta$, enjoys SN in $\lambda_{\rightarrow\times}^{Church}$.*
- *η-expansion, $\leftarrow_\eta$, enjoys SN in $\lambda_{\rightarrow\times}^{Church}$.*
- *$\beta\eta$-rewriting, $\rightarrow_\beta \cup \leftarrow_\eta$, enjoys SN in $\lambda_{\rightarrow\times}^{Church}$.*

Theorem 4 (Church-Rosser) *By Theorem 2 we have:*

- *β-reduction, $\rightarrow_\beta$, enjoys CR in $\lambda_{\rightarrow\times}^{Church}$.*
- *η-expansion, $\leftarrow_\eta$, enjoys CR in $\lambda_{\rightarrow\times}^{Church}$.*
- *$\beta\eta$-rewriting, $\rightarrow_\beta \cup \leftarrow_\eta$, enjoys CR in $\lambda_{\rightarrow\times}^{Church}$.*

1.1.4 A two-level calculus

While the former two calculi are well-known we will now introduce something new. It resembles the work in Nielson & Nielson: "Two-level Functional

Languages" [55]. We cannot immediately use any of their results[7] but to a certain extent we will use their notation and terminology. We will consider essentially $\lambda_{\rightarrow\times}^{\text{Curry}}$ but with two versions of every construct: An overlined (called static) and an underlined (called dynamic)[8].

Definition 7 (Implicitly simply typed two-level λ-calculus) *Let the pre-terms of $\lambda_{\rightarrow\times}^{Curry\text{-}II}$ be given by:*

$$\begin{array}{rcl} e & ::= & x \\ & | & k \\ & | & \overline{\lambda} x.e \\ & | & e_1 \,\overline{@}\, e_2 \\ & | & \overline{\langle} e_1,\, e_2 \overline{\rangle} \\ & | & \overline{p}_i(e) \\ & | & \underline{\lambda} x.e \\ & | & e_1 \,\underline{@}\, e_2 \\ & | & \underline{\langle} e_1,\, e_2 \underline{\rangle} \\ & | & \underline{p}_i(e) \end{array}$$

The terms $-M,\, N-$ are given as the provable pre-terms of an overlined and an underlined version of the type system (distinguished by $\overline{:}\,\&\,\underline{:}$) for $\lambda_{\rightarrow\times}^{Curry}$ plus the following two rules, for base types o:

$$\frac{\pi}{M \,\overline{:}\, o}{\scriptstyle\,(Dyn)} \qquad \frac{\pi}{N \,\underline{:}\, o}{\scriptstyle\,(Stat)}$$

That is, interfacing between the two levels is only allowed at base type (constants are thus both static and dynamic). We will write $M \in \overline{\Lambda}_{II}^{\tau}$ resp. $M \in \underline{\Lambda}_{II}^{\tau}$ when M statically resp. dynamically provably is of type τ (collectively $M \in \Lambda_{II}^{\tau}$). The different notions of term equality we will use are:

$$(\overline{\lambda} x.M)\,\overline{@}\,N \quad =_{\overline{\beta_{\rightarrow}}} \quad M[N/x]$$

[7] They allow terms which are dynamically of at least function type to be passed around but not applied in the static level [55, [Up]–Table 3.3, [()]–Table 3.4]. This is very different from our requirements.

[8] Again, please remember that the terminology is widely used with connotation unrelated to its present use.

$$
\begin{array}{rcll}
\overline{p}_i(\overline{\langle} M_1,\, M_2 \overline{\rangle}) & =_{\overline{\beta_\times}} & M_i & \\
\overline{\lambda} x.M\, \overline{@}\, x & =_{\overline{\eta_\rightarrow}} & M & ;\textit{if } x \notin FV(M) \;\&\; M \in \overline{\Lambda}_{II}^{\tau\rightarrow\sigma} \\
 & & & \;\&\; M \neq \overline{\lambda} y.N \\
 & & & \;\&\; M \textit{ is not applied} \\
\overline{\langle} \overline{p}_1(M),\, p_2(M) \overline{\rangle} & =_{\overline{\eta_\times}} & M & ;\textit{if } M \in \overline{\Lambda}_{II}^{\tau\times\sigma} \;\&\; M \neq \overline{\langle} M_1,\, M_2 \overline{\rangle} \\
 & & & \;\&\; M \textit{ is not projected} \\
(\underline{\lambda} x.M)\, \underline{@}\, N & =_{\underline{\beta_\rightarrow}} & M[N/x] & \\
\underline{p}_i(\underline{\langle} M_1,\, M_2 \underline{\rangle}) & =_{\underline{\beta_\times}} & M_i & \\
\underline{\lambda} x.M\, \underline{@}\, x & =_{\underline{\eta_\rightarrow}} & M & ;\textit{if } x \notin FV(M) \;\&\; M \in \underline{\Lambda}_{II}^{\tau\rightarrow\sigma} \\
 & & & \;\&\; M \neq \underline{\lambda} y.N \\
 & & & \;\&\; M \textit{ is not applied} \\
\underline{\langle} \underline{p}_1(M),\, \underline{p}_2(M) \underline{\rangle} & =_{\underline{\eta_\times}} & M & ;\textit{if } M \in \underline{\Lambda}_{II}^{\tau\times\sigma} \;\&\; M \neq \underline{\langle} M_1,\, M_2 \underline{\rangle} \\
 & & & \;\&\; M \textit{ is not projected}
\end{array}
$$

We will show that this calculus enjoys generalizations of some of the properties $\lambda_{\rightarrow\times}^{\text{Curry}}$ enjoy[9]. The calculus is intended to be essentially identical to $\lambda_{\rightarrow\times}^{\text{Curry}}$ while at the same time having a notion of two levels which will give us a very convenient framework in which to present our main results, Chapter 3. In this section we will prove the properties required for that work.

The single-most important feature of the calculus is believed to be what we call "the base-type level interface", namely the two typing rules stating that a term which can be statically proven to be of base type can also be dynamically proven to be of base type, and vice versa. This feature gives a totally symmetric calculus where each level is believed to inherit the features of $\lambda_{\rightarrow\times}^{\text{Curry}}$. As the level interface only is at base type, the levels are in a certain sense computationally independent, that is, no reduction in one level can introduce a redex in the other.

To allow the reader a look into the details we will start out by proving SN and CR for the β-reductions of $\lambda_{\rightarrow\times}^{\text{Curry-II}}$ and then go on to state the results we will be needing later on.

Before starting on the technicalities we would like to state that the results are believed to generalize to any number of levels.

[9] It is, however, believed that all properties carry through, cf. the last section of the present Chapter.

The classical results

A β-redex occurring within another term can be straightforwardly reduced, that is, β-reduction enjoys *compatibility*:

$$N \to_\beta N' \Rightarrow M\{N\} \to_\beta M\{N'\}$$

which imply:

$$N \to_\beta^* N' \Rightarrow M[N/x] \to_\beta^* M[N'/x]$$

Lemma 2 (Subject Reduction, β) *Λ_{II} is closed under $\to_\beta$ ($=\to_{\overline{\beta}} \cup \to_{\underline{\beta}}$).*

Proof: By compatibility we need to show that any term with an outermost redex reduces to a term of the same type.

$(\overline{\lambda}x.M)\,\overline{@}\,N$: The situation must be as follows for the type proof:

$$\dfrac{\dfrac{\begin{array}{c}[x\bar{:}\tau]_k \\ \pi_1 \\ M\bar{:}\sigma\end{array}}{\overline{\lambda}x.M\bar{:}\tau \to \sigma}(\to I)_k \qquad \begin{array}{c}\pi_2 \\ N\bar{:}\tau\end{array}}{M\ \overline{@}\ N\bar{:}\sigma}(\to E)$$

And substituting N in for x amounts to the following restructuring of the typing proof:

$$\dfrac{\begin{array}{c}\pi_2 \\ \cdots\ N\bar{:}\tau\ \cdots \\ \pi'_1 \\ M[N/x]\bar{:}\sigma\end{array}}{}$$

with π'_1 obtained from π_1 by insertion of π_2 for every member of the assumption class k[10]. It is immediately seen to still be a typing proof.

$(\underline{\lambda}x.M)\,\underline{@}\,N$: Symmetric.

$\overline{\mathbf{p}}_i(\overline{\langle} M_1, M_2 \overline{\rangle})$: Again the situation must be as such:

[10] For further explanation please refer to Chapter 2

$$\dfrac{\dfrac{\begin{matrix}\pi_1 & \pi_2\\ M_1\bar{:}\tau_1 & M_2\bar{:}\tau_2\end{matrix}}{\overline{\langle} M_1,\, M_2\overline{\rangle}\bar{:}\tau_1\times\tau_2}(\times I)}{\overline{\mathrm{p}}_i(\overline{\langle} M_1,\, M_2\overline{\rangle})\bar{:}\tau_i}(\times E_i)$$

And projecting is straightforward:

$$\begin{matrix}\pi_i\\ M_i\bar{:}\tau_i\end{matrix}$$

which necessarily is a type proof.

$\underline{\mathbf{p}}_i(\underline{\langle} M_1,\, M_2\underline{\rangle})$**:** Symmetric.

□

The result in effect means that (i) we can collapse $v \in \Lambda^{\tau}_{\mathrm{II}}$ into $c(v) \in \Lambda^{\tau}$ and perform β-reduction on $c(v)$ in $\lambda^{\text{Curry}}_{\to\times}$ while preserving existence of a corresponding reduction in $\lambda^{\text{Curry-II}}_{\to\times}$ [11] and (ii) any $\to_\beta$ step on v is valid on $c(v)$.

As we stated earlier η-expansion does not enjoy compatibility but we do have:

Lemma 3 (Subject Reduction, η**)** *Λ_{II} is closed under $\leftarrow_\eta$ ($=\leftarrow_{\overline{\eta}} \cup \leftarrow_{\underline{\eta}}$).*

Proof: We will consider $M\{N\}$ in which N is an $\overline{\eta}$-redex. It suffices to look at the different ways N can be η-expanded:

$N \in \overline{\Lambda}^{\tau\to\sigma}_{\mathbf{II}}$:

$$\dfrac{\dfrac{\begin{matrix}\pi\\ [x\bar{:}\tau]_k\end{matrix}\quad N\bar{:}\tau\to\sigma}{N\,\overline{@}\,x\bar{:}\sigma}(\to E)}{\overline{\lambda}x.N\,\overline{@}\,x\bar{:}\tau\to\sigma}(\to I)_k$$

which is seen to fit into the type proof of M in the place of:

$$\begin{matrix}\pi\\ N\bar{:}\tau\to\sigma\end{matrix}$$

[11] As the collapse trivially cannot create new redices by the base-type level-interface requirement.

$N \in \overline{\Lambda}_{\mathbf{II}}^{\tau_1 \times \tau_2}$:

$$\dfrac{\dfrac{\overset{\pi}{N \bar{:} \tau_1 \times \tau_2}}{\overline{\mathrm{p}}_1(N) \bar{:} \tau_1}(\times E_1) \quad \dfrac{\overset{\pi}{N \bar{:} \tau_1 \times \tau_2}}{\overline{\mathrm{p}}_2(N) \bar{:} \tau_2}(\times E_2)}{\overline{\langle} \overline{\mathrm{p}}_1(N), \overline{\mathrm{p}}_2(N) \overline{\rangle} \bar{:} \tau_1 \times \tau_2}(\times I)$$

which again is seen to fit for:

$$\overset{\pi}{N \bar{:} \tau_1 \times \tau_2}$$

□

The ease with which these results were proven is of course due to the calculus only allowing level interfacing at base type and we will see that the same holds for the following results.

Theorem 5 *$\rightarrow_\beta$ enjoys SN in $\lambda_{\rightarrow\times}^{Curry\text{-}II}$.*

Proof: By collapsing into $\lambda_{\rightarrow\times}^{\mathrm{Curry}}$.

□

Corollary 1 *Both $\rightarrow_{\overline{\beta}}$ and $\rightarrow_{\underline{\beta}}$ enjoys SN in $\lambda_{\rightarrow\times}^{Curry\text{-}II}$.*

Theorem 6 *$\rightarrow_\beta$ enjoys CR in $\lambda_{\rightarrow\times}^{Curry\text{-}II}$.*

Proof: By collapsing into $\lambda_{\rightarrow\times}^{\mathrm{Curry}}$.

□

We have already considered compatibility and before going on to proving CR for the individual β-reductions we must consider the dual situation of compatibility which requires a proof:

Lemma 4 (Substitution Lemma) *$\rightarrow_{\overline{\beta}}, \rightarrow_{\underline{\beta}}$ are substitutive in $\lambda_{\rightarrow\times}^{Curry\text{-}II}$, that is, for the static case:*

$$M \rightarrow_{\overline{\beta}} M' \Rightarrow M[N/x] \rightarrow_{\overline{\beta}} M'[N/x]$$

Proof: Again by compatibility it suffices to consider outermost redices and by usual symmetry will we only consider static such.

$M \equiv (\overline{\lambda} y.M_1)\,\overline{@}\,M_2$. As substitution is defined non-capturing we can W.L.O.G. assume $x \neq y \wedge y \notin FV(N)$.

$$\begin{array}{rcll} ((\overline{\lambda} y.M_1)\,\overline{@}\,M_2)[N/x] & \equiv & (\overline{\lambda} y.M_1[N/x])\,\overline{@}\,M_2[N/x] & \\ & \rightarrow_{\overline{\beta_{\rightarrow}}} & (M_1[N/x])[(M_2[N/x])/y] & (1.1) \end{array}$$

And by induction on M_1 we show this to be equal to the desired form:

$$(M_1[M_2/y])[N/x] \tag{1.2}$$

$M_1 \equiv z$: Equations (1.1), (1.2) are immediately seen to be equal to either N, $M_2[N/x]$, or to z different from x, y and we are done.

$M_1 \equiv \overline{\lambda} v.M_3$: We can again assume v to be unrelated to the incoming substitutions and by I.H. we have the desired property for M_3 thus we are done.

$M_1 \equiv M_3\,\overline{@}\,M_4$: Immediately by I.H. on M_3, M_4.

$M_1 \equiv \overline{\langle} M_3,\, M_4 \overline{\rangle}$: Equivalent to the above case.

$M_1 \equiv \overline{\mathbf{p}}_i(M_3)$: Again equivalent.

$M \equiv \overline{\mathbf{p}}_i(\overline{\langle} M_1,\, M_2 \overline{\rangle})$. We go straight ahead:

$$\begin{array}{rcl} \overline{\mathrm{p}}_i(\overline{\langle} M_1,\, M_2 \overline{\rangle})[N/x] & \equiv & \overline{\mathrm{p}}_i(\overline{\langle} M_1[N/x],\, M_2[N/x] \overline{\rangle}) \\ & \rightarrow_{\overline{\beta_{\times}}} & M_i[N/x] \end{array}$$

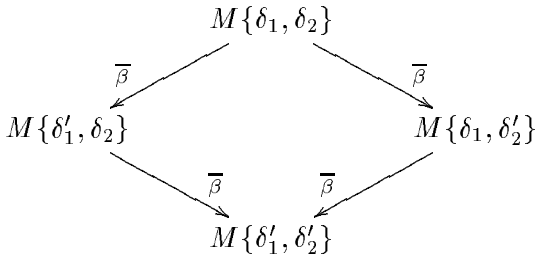
and we are done.

□

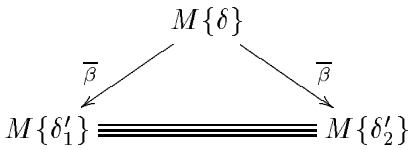
Lemma 5 *Both* $\rightarrow_{\overline{\beta}}$ *and* $\rightarrow_{\underline{\beta}}$ *enjoy WCR in* $\lambda_{\rightarrow\times}^{Curry\text{-}II}$*:*

Proof: (Only the static case; by case-splitting in the two redices relative positions)

δ_1, δ_2 **disjoint** The two redices do not interfere and we can reduce them independently:

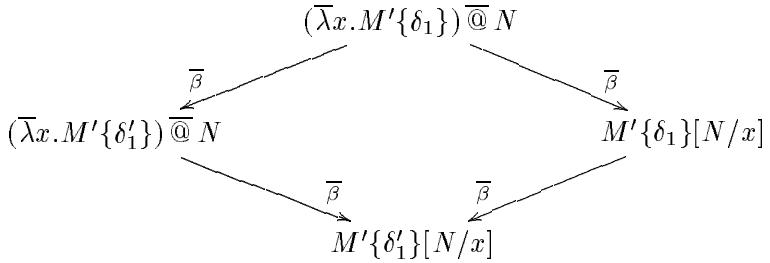


$\delta_1 = \delta_2 (= \delta)$ The two redices are the same redex:

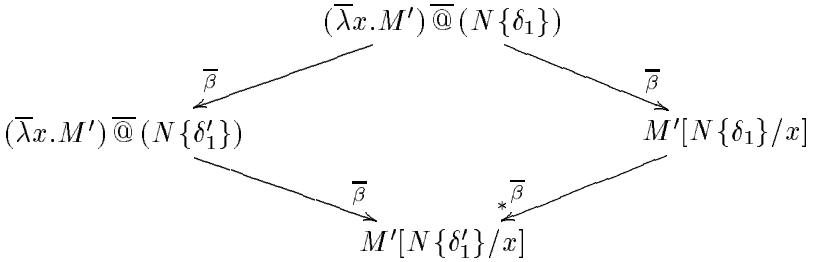


$\delta_1 \subset \delta_2$ By compatibility we need only consider terms where δ_2 is an outermost redex hence we have the following different situations:

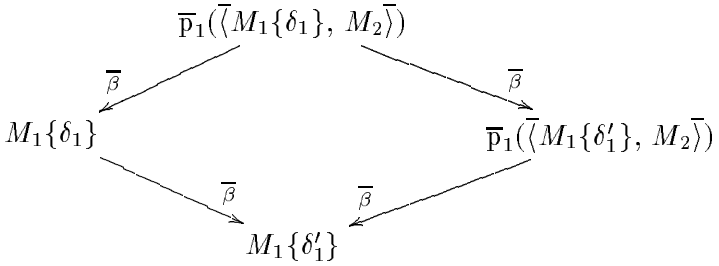
$\delta_2 = (\overline{\lambda}x.M')\,\overline{@}\,N \ \wedge\ \delta_1 \subseteq M'$. The lower right follows from Lemma 4 and compatibility while the lower left is ordinary reduction:



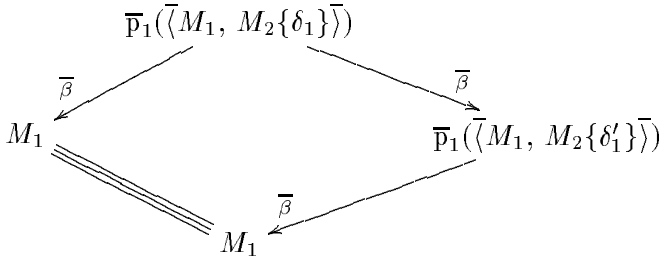
$\delta_2 = (\overline{\lambda}x.M')\,\overline{\overline{@}}\,N \ \wedge\ \delta_1 \subseteq N$. The lower right follows from compatibility while the lower left is ordinary reduction:



$\delta_2 = \overline{\mathbf{p}}_1(\overline{\langle} M_1,\ M_2\overline{\rangle}) \ \wedge\ \delta_1 \subseteq M_1$. Immediate:



$\delta_2 = \overline{\mathrm{p}}_1(\overline{\langle} M_1,\ M_2\overline{\rangle}) \ \wedge\ \delta_1 \subseteq M_2$. Immediate:



Symmetric for the other projection.

$\delta_1 \supset \delta_2$: Symmetric to the above.

Theorem 7 *Both* $\rightarrow_{\overline{\beta}}$ *and* $\rightarrow_{\underline{\beta}}$ *enjoy Church-Rosser in* $\lambda_{\rightarrow\times}^{Curry\text{-}II}$.

Proof: By Lemma 5 and Newman's Lemma (1).

□

Independency of levels

We will start with a few lemmas characterizing the different reductions and their normal forms.

Lemma 6 *In* $\lambda_{\rightarrow\times}^{Curry\text{-}II}$ *the* $\overline{\beta}$*-normal forms of base typed static terms without free static variables are purely dynamic, that is, do not contain overlined constructors. And vice versa.*

Proof: (By considering the different shapes of a static term)

x : Not closed.

k : Can be taken to be dynamic.

$\overline{\lambda}x.e$: Not of base type.

$\overline{\langle e_1, e_2 \rangle}$: Not of base type.

$\overline{e_1}\,\overline{@}\,e_2$, $\overline{\mathbf{p}}_i(\overline{e})$: $\overline{e_1}$, $\overline{e}$ cannot be variables (due to closedness), pairs (due to type/normal form), or abstractions (due to normal form/type). Therefore they must be either applications or projections, however, this [continued] requirement cannot produce a finite term.

□

And finally, the results we are after:

Theorem 8 *In $\lambda_{\rightarrow\times}^{Curry\text{-}II}$ the $\overline{\beta}$-normal form of a dynamic term without free static variables is purely dynamic. And vice versa.*

Proof: Given such a term we can consider its non-nested static constructs separately by compatibility. Each of these must be of base type and without free static variables and thus, by Lemma 6 their $\overline{\beta}$-normal forms are purely dynamic.

□

Proposition 3 *β-reduction in $\lambda_{\rightarrow\times}^{Curry\text{-}II}$ preserves type as well as level at higher type, while η-expansion preserves both type and level.*

Proof: Immediate from Lemmas 2, 3 (Subject Reduction).

□

Theorem 9 *$\overline{\beta}$-reduction cannot introduce a $\underline{\beta}$-redex nor an $\underline{\eta}$-redex. And symmetrically.*

Proof: Let δ be a $\overline{\beta}$-redex occurring in M: $M\{\delta\}$. We continue by structural induction on δ:

$\delta \equiv \overline{\mathbf{p}}_i(\overline{\langle M_1, M_2 \rangle})$: If M_i is of higher-type δ must occur in a static context and can thus trivially not introduce any dynamic redex. If it is of base type it cannot introduce a new redex at all.

$\delta \equiv (\overline{\lambda}x.M)\,\overline{@}\,N$: By the above reasoning we see that the reduct, $M[N/x]$, cannot introduce any dynamic redex in its context. The same reasoning also applies to N's role and type in $M[N/x]$ and we are done.

□

Corollary 2 *$\overline{\beta}$-reduction preserves $\underline{\beta\eta}$-normal forms. And vice versa.*

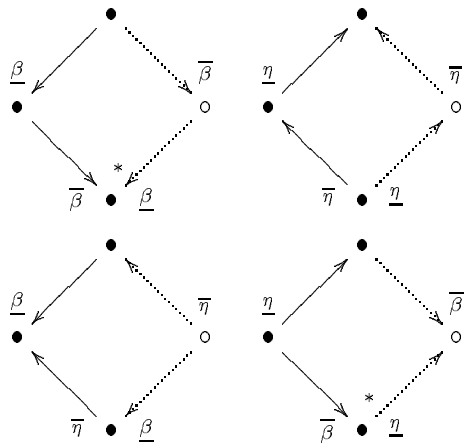
A more general result

The property underlying the present development is independence of reductions in the two levels. A direct proof of the following conjecture would be rather large and as we do not need it for our later results, we have not dwelled into the details. We believe, however, that the reader might benefit from being exposed to it.

Conjecture 1 (Commutivity of Reductions) *$\rightarrow_{\overline{\beta}}$ respectively $\leftarrow_{\overline{\eta}}$ commutes over $\rightarrow_{\underline{\beta}}$ and $\leftarrow_{\underline{\eta}}$, and vice versa.*

$$\forall M, M_1, M_2 . M \rightarrow_{\underline{\beta}} M_1 \rightarrow_{\overline{\beta}} M_2 \Rightarrow (\exists M' . M \rightarrow_{\overline{\beta}} M' \rightarrow^*_{\underline{\beta}} M_2)$$
$$\forall M, M_1, M_2 . M \leftarrow_{\underline{\eta}} M_1 \leftarrow_{\overline{\eta}} M_2 \Rightarrow (\exists M' . M \leftarrow_{\overline{\eta}} M' \leftarrow_{\underline{\eta}} M_2)$$
$$\forall M, M_1, M_2 . M \rightarrow_{\underline{\beta}} M_1 \leftarrow_{\overline{\eta}} M_2 \Rightarrow (\exists M' . M \leftarrow_{\overline{\eta}} M' \rightarrow_{\underline{\beta}} M_2)$$
$$\forall M, M_1, M_2 . M \leftarrow_{\underline{\eta}} M_1 \rightarrow_{\overline{\beta}} M_2 \Rightarrow (\exists M' . M \rightarrow_{\overline{\beta}} M' \leftarrow^*_{\underline{\eta}} M_2)$$

or in diagram form:



Chapter 2

The Curry-Howard Correspondence of Proof Normalization

In this chapter we will consider two different formalizations of Minimal Logic and present their Curry-Howard corresponding calculi. A standard result is equivalence of the systems [70] but we will not be concerned with that issue. Instead we will focus on the different computational meanings of obtaining normal forms in the two systems.

The two formalizations we will use are instances of the styles known as Natural Deduction and Sequent Calculus[1]. Both are due to Gentzen [29] but the Natural Deduction style was refined and developed by Prawitz [59]. Natural Deduction style is what traditionally gives the Curry-Howard Isomorphism, even though the standard reference [38] actually uses Sequent Calculus[2].

Our reason for giving this treatment is two-fold: (i) It is necessary to treat, at least, Natural Deduction to make the thesis self-contained, and (ii) so far the computational meaning of Sequent Calculus does not seem to have been promoted beyond folklore [70, Section 3.3.4] & [5, 37]. By this we mean that, even though the correspondence has been described, there does not seem to have been any cross-fertilizing effect (however, see [15]). We

[1] The third major style is what is known as Hilbert Systems. They consist of a number of axioms and *Modus Ponens*. The computational flavour of such systems is readily recognized as combinator-based languages.

[2] Remarks are made suggesting the inappropriateness of this [38, p. 484, "*Note added 1979 ...*"]

will point at an area where such an effect could arise and it is part of our ongoing work.

The work in this chapter has benefitted from interaction with –in alphabetical order– Torben Braüner, Olivier Danvy, Bernhard Gramlich, Jakov Kucan, Harry Mairson, Albert Meyer, Kristoffer Rose, Luca Roversi, Helmut Schwichtenberg, and Anne Troelstra.

2.1 Natural Deduction

Natural Deduction derivations are built –almost– as trees with assumed formulas in the leafs and with internal nodes corresponding to Introduction and Elimination rules for each type constructor. The derivations are not entirely trees as rules can be non-local. Hence they are presented with their subderivations. Normal forms are defined as derivations without introduction rules immediately followed by a corresponding elimination rule.

In the following we will consider a term annotated Natural Deduction presentation of Implicative, Conjunctive Minimal Logic, here –Figure 2.1– called Nm.

$$x : \tau \qquad (\text{Asmp})$$

$$\frac{\begin{array}{c}[x:\tau]_k \\ \vdots \\ M : \sigma\end{array}}{\lambda x.M : \tau \to \sigma}\,(\to I)_k \qquad \frac{\begin{array}{cc}\pi_M & \pi_N \\ M : \tau \to \sigma & N : \tau\end{array}}{M\ N : \sigma}\,(\to E)$$

$$\frac{\begin{array}{cc}\pi_{M_1} & \pi_{M_2} \\ M_1 : \tau_1 & M_2 : \tau_2\end{array}}{\langle M, N\rangle : \tau_1 \times \tau_2}\,(\times I) \qquad \frac{\begin{array}{c}\pi_M \\ M : \tau_1 \times \tau_2\end{array}}{\mathrm{p}_i(M) : \tau_i}\,(\times E_i)$$

Figure 2.1: Nm; Term-annotated Implicative and Conjunctive Minimal Logic in Natural Deduction style.

Assumptions are said to occur free in a derivation, unless they are discharged (or bound) by the $[-]_k$ operation in $(\to I)_k$. Discharging can be performed on any number of free occurrences of a given formula. We will refer to such a set as an assumption class. The $(\to I)_k$ rule is linked to

its assumption class through the index k which is assumed unique. When the proofs are term-annotated the variable names determine the assumption classes up to α-equivalence.

Considering Definition 5, we can see that to any $\lambda_{\to\times}^{\text{Church}}$-term[3] there is a corresponding derivation in the above logic. Furthermore, as each rule introduces unique syntax, and as this syntax in a certain sense is type annotated, the correspondence is unique. We conclude from Figure 2.1 that any derivation is annotated with a unique $\lambda_{\to\times}^{\text{Church}}$-term. That is, we have a 1-1 correspondence between $\lambda_{\to\times}^{\text{Church}}$-terms and Natural Deduction derivations for Minimal Logic.

Remembering the notion of normal forms, we obtain the following method for rearranging derivations [attempting] to normalize proofs:

$$\dfrac{\dfrac{\begin{array}{c}[x:\tau]_k\\ \pi_1\\ M:\sigma\end{array}}{\lambda x.M:\tau\to\sigma}(\to I)_k \qquad \begin{array}{c}\pi_2\\ N:\tau\end{array}}{M\,N:\sigma}(\to E) \quad \longmapsto_{normalize} \quad \begin{array}{c}\pi_2\\ \cdots\ N:\tau\ \cdots\\ \pi'_1\\ M[N/x]:\sigma\end{array}$$

where π'_1 is obtained from π_1 by substituting π_2 in for all assumptions in the class k.

$$\dfrac{\dfrac{\begin{array}{cc}\pi_1 & \pi_2\\ M_1:\tau_1 & M_2:\tau_2\end{array}}{\langle M_1,\,M_2\rangle:\tau_1\times\tau_2}(\times I)}{\mathrm{p}_i(\langle M_1,\,M_2\rangle):\tau_i}(\times E_i) \quad \longmapsto_{normalize} \quad \begin{array}{c}\pi_i\\ M_i:\tau_i\end{array}$$

These rewritings are seen to correspond exactly to $\to_\beta$. That is, the presentation of minimal logic given in Figure 2.1 is isomorphic to $\lambda_{\to\times}^{\text{Church}}$ with β-reduction. Furthermore, if proofs are considered equal up to substitution of type variables, we have a logic isomorphic to $\lambda_{\to\times}^{\text{Curry}}$ with $\to_\beta$.

We have from results about $\lambda_{\to\times}^{\text{Church}}$ that derivations normalize to unique normal forms.

[3] Notice the discrepancy between the referred Definition (implicit types) and the terms in consideration (explicit types).

A few comments about the relationship between logics and explicitly resp. implicitly typed calculi seem to be in order. We will, especially in Chapter 3, treat implicitly typed terms according to their overall type. As seen from:

$$\frac{\begin{array}{cc}\pi_1 & \pi_2 \\ M : \tau \to \sigma & N : \tau\end{array}}{M\,N : \sigma}\,(\to E)$$

this does not go in the general case, as N's type is unspecified in the type of $M\,N$. However, the normal formed fragment of Nm enjoys the so-called Subformula Property [30].

Definition 8 (Subformula Property) *A logic is said to enjoy the Subformula Property if for all derivations, π, a formula in π is a subformula of π's conclusion or is a free assumption in π.*

We therefore see, that in the case of the β-normal formed fragment of $\lambda_{\to\times}^{\text{Curry}}$, the overall type of a term uniquely associates implicitly typed terms with [normal formed] derivations of Nm.

The Subformula Property implies the Separation Property [70]:

Definition 9 (Separation Property) *A logic is said to enjoy the Separation Property if to prove a formula you only require logical rules corresponding to the connectives which occur in free assumptions and in the considered formula.*

The logical rules can of course be both introduction and elimination rules, but observe that the property relates to occurrences of formulas within formulas and free assumptions not within proofs. The separation property is what, on the logical level, allows us to use a type-directed approach to construct the normal forms of [closed] terms in the following Chapters.

2.2 Sequent Calculus

Sequent calculi[4] logics do not have specific elimination rules for each constructor. Instead they have a general rule, Cut, which handles elimination of all constructors. To achieve this, sequents over formulas are two-sided[5]

[4] Read: calculi over sequents

[5] More precisely is there a notion of formula duality in Sequent Calculi logics. A formula is thus dual to itself occurring on the other side of the sequent, so to say.

and correspondingly there are introduction rules on both sides for each constructor. The left hand side, the antecedent, is to be considered as a set of sufficient assumptions to prove the right hand side, the succedent. A consequence of this is that rules are fully local and derivations become proper trees. The formulas in the antecedent behave as the points of interaction between the succedent and the rest of the proof. The interaction happens through the Cut rule. We will return to this point in Figure 2.5, after having considered Cut.

2.2.1 Specifics

As in Natural Deduction logics, Sequent Calculi logics have a notion of normal formed derivations. Unlike the structural criterion for Natural Deduction logics, a normal formed derivation in a Sequent Calculus logic is described simply as a cut-free derivation. For this reason, our core logic –Figure 2.2– will not include a Cut rule. We will consider Cut later on –Figure 2.3– and show how to normalize derivations containing Cut-rule occurrences by performing so-called Cut Elimination, Theorem 10.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\,(Ax)$$

$$\frac{x : \tau, y : \sigma, \Gamma \vdash M : \upsilon}{p : \tau \times \sigma, \Gamma \vdash \text{let } \langle x,\, y\rangle{=}p \text{ in } M}\,(\times L) \qquad \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash \langle M,\, N\rangle : \tau \times \sigma}\,(\times R)$$

$$\frac{f : \tau \to \sigma, \Gamma \vdash N : \tau \quad x : \sigma, \Gamma \vdash M : \upsilon}{f : \tau \to \sigma, \Gamma \vdash \text{let } x{=}f\ N \text{ in } M}\,(\to L) \qquad \frac{x : \tau, \Gamma \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \tau \to \sigma}\,(\to R)$$

Figure 2.2: Sm; Term-annotated Implicative and Conjunctive Minimal Logic in Sequent Calculus style. The logic is a term-annotated subset of G3m as given by Troelstra & Schwichtenberg. G3m is originally due to Kleene and Dragalin.

The Γ, Δ's in the rules of Figures 2.2 & 2.3 are called the context of the rules, as they do not have any influence on how the sequent below the line (the conclusion) is formed. The formulas of the sequents above the line (the premises), which influence the formation, are called the principal

formulas, as are the resulting formulas in the conclusion. We observe that, modulo excessive context (cf. Appendix A.3) and a suitable notion of type annotations on the proof terms (cf. implicit and explicit types relative to Nm), there is a one-to-one correspondence between derivations in Sm+Cut and the proof terms. As in the case of Nm, Sm enjoys the Subformula Property [70] and parallel comments thus apply.

Rules that do not allow a logic to prove additional formulas when added are usually called "admissible". The Cut rule is the archetypical admissible rule. The relative merit of Sm, compared to other Sequent Calculi logics, is that, what is traditionally known as, the structural rules –weakening and contraction in Figure 2.3– now also become admissible. Structural rules are ways of changing sequents without it having any traditional logical consequences[6]. We will not be overly concerned with their computational meaning. In the case of the contraction rule we will require the contracted variables to be identical. Granted, this is a rather unusual restriction and we will offer an explanation in Appendix A.2.

$$\frac{\Gamma \vdash M : \tau \quad x : \tau, \Delta \vdash N : \sigma}{\Gamma, \Delta \vdash N\langle\!\langle M/x \rangle\!\rangle : \sigma}\,(Cut)$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma, x : \tau \vdash M : \sigma}\,(Weak)$$

$$\frac{\Gamma, x : \tau, x : \tau \vdash M : \sigma}{\Gamma, x : \tau \vdash M : \sigma}\,(Cont)$$

Figure 2.3: Cut and structural rules for Sm. $-\langle\!\langle -/- \rangle\!\rangle$ is to be considered as an operator.

Antecedents are multisets of formula annotated variables. As we are only interested in terms up to α-equivalence and as all rules of Sm are context sharing, variables can be chosen to be distinct. Cut, however, cannot in the current logic be presented with shared contexts, as we would not be able to prove cut elimination — and cut elimination is our main concern. We will therefore require the two contexts in Cut to have identical variables

[6] Linear logic, however, takes the stand that structural rules have logical meaning.

annotated to formulas that become contracted and require all others to be distinct.

If we where to present these naming criteria in traditional calculus style we would, for all terms, define two sets, (i) the free variables, FV(-), which correspond to the variables in the antecedent and (ii) the bound variables, BV(-), corresponding to how the antecedents change when going from a premise to the conclusion. Please refer to Appendix A.1 for an overview of these in the case of Sm+Cut (without the extra conditions generated by contraction –Appendix A.2– and weakening –Appendix A.3).

2.2.2 The traditional results

Before going on to prove cut elimination, we will state the following result, where we write $\Gamma \vdash_n M : \tau$ for a given derivation when the longest path in the derivation has length n.

Lemma 7 (Lemma 3.3.4, Proposition 3.4.5 in [70]) *Sm is closed under weakening and contraction (Weak), (Cont) in Figure 2.3):*

- $\Gamma \vdash_n M : \tau \Rightarrow \Gamma, \Delta \vdash_n M : \tau$
- $\Gamma, x : \tau, x : \tau \vdash_n M : \sigma \Rightarrow \Gamma, x : \tau \vdash_n M : \sigma$

We are now ready to start the proof of cut elimination for Sm. To that end we need a few definitions relating to cuts.

Definition 10 (Level, Depth [70]) *The level, $l(\cdot)$, of a cut is the sum of the depths of the deductions in the premises. The depth of a deduction is the maximum length of a path in the deduction.*

Definition 11 (Rank, Size, Cutrank [70]) *The rank, $r(\cdot)$, of a cut is one plus the size of the formula being cut. The size of a formula is the number of connectives plus the number of basic formulas (fixed base types and type variables) in it. The cutrank, $cr(\cdot)$, of a derivation is the maximum of the ranks of the cuts occurring in it.*

Definition 12 (Eliminatable Cut [70]) *A cut, χ, ending a derivation, π, with sub-derivations π_1, π_2:*

$$\frac{\begin{array}{cc}\pi_1 & \pi_2 \\ \Gamma \vdash M : \tau & x : \tau, \Delta \vdash N : \sigma\end{array}}{\Gamma, \Delta \vdash N\langle\!\langle M/x \rangle\!\rangle : \sigma}\,(Cut)$$

is called eliminatable if:

$$r(\chi) = cr(\pi) \ \wedge \ cr(\pi_1), cr(\pi_2) < r(\chi)$$

That is, the rank of eliminatable cuts is uniquely maximal in the derivation ending in that cut. As is usual we will also require that not both π_1 and π_2 end in a cut.

It is immediately observed that topmost cuts are eliminatable. Hence, any derivation containing cuts contains an eliminatable cut.

Theorem 10 (Hauptsatz/Cut Elimination) *Sm enjoys cut elimination. That is, (Cut) is admissible for Sm and can be eliminated.*

Before starting we should say that the structure of our proof is similar to that of the proof in [70], however, it is has some minor differences. Furthermore it is our belief that the present approach marks a slight simplification to their proof. Afterwards we will discuss the process of obtaining cut-free derivations and consider an alternative to the present proof inspired by the notion of reduction in a calculus.

Proof: (by induction on the number of [pre-existing] cuts with a well-founded sub-induction on derivations ordered lexicographically by cutrank, maximal level of eliminatable cuts.)

We show that any derivation ending in an eliminatable cut can be transformed into a cut-free derivation of the same sequent. It is done by stepwise rewriting the derivation in consideration. Derivations in this chain will be ordered to the preceding derivation by the above mentioned ordering. The ordering essentially says that either the cutrank of the derivation strictly decreases or it remains the same and the levels of any new cuts with the same rank[7] as the eliminated one are strictly lower. This ordering is readily seen to be well-founded[8]. We will annotate the new cuts with either l, if they are of the same rank but have strictly lower level, or r, if they are of strictly lower rank.

[7] Cuts with the same rank as the original cut are necessarily new as eliminatable cuts are required to have uniquely maximal rank. Observe that there actually is no strict requirement for the rank of eliminatable cuts to be uniquely maximal.

[8] That is, without infinitely descending chains.

We proceed by considering three cases – we use implicit weakening and contraction throughout and we indicate this by *. That is, a derivation π^* is obtained from π by weakening or contraction:

At least one axiom is being cut .

- Principal cut formula on the right:

$$\dfrac{\overset{\pi}{\Gamma \vdash u : \tau} \quad \dfrac{}{\Gamma', x : \tau \vdash x : \tau}\,(Ax)}{\Gamma, \Gamma' \vdash x\langle\!\langle u/x \rangle\!\rangle : \tau}\,(Cut) \tag{2.1}$$

$$\longmapsto_{CE} \qquad \overset{\pi^*}{\Gamma, \Gamma' \vdash u : \tau}$$

- Non-principal cut formula on the right:

$$\dfrac{\overset{\pi}{\Gamma \vdash u : \tau} \quad \dfrac{}{x : \tau, y : \sigma, \Gamma' \vdash y : \sigma}\,(Ax)}{\Gamma, \Gamma', y : \sigma \vdash y\langle\!\langle u/x \rangle\!\rangle : \sigma}\,(Cut) \tag{2.2}$$

$$\longmapsto_{CE} \qquad \dfrac{}{\Gamma, \Gamma', y : \sigma \vdash y : \sigma}\,(Ax)$$

- Principal cut formula on the left:

$$\dfrac{\dfrac{}{\Gamma, x : \tau \vdash x : \tau}\,(Ax) \quad \overset{\pi}{y : \tau, \Gamma' \vdash u : \sigma}}{x : \tau, \Gamma, \Gamma' \vdash u\langle\!\langle x/y \rangle\!\rangle : \sigma}\,(Cut) \tag{2.3}$$

$$\longmapsto_{CE} \qquad \overset{\pi^*}{x : \tau, \Gamma, \Gamma' \vdash u[x/y] : \sigma}$$

It should be remarked that π^* besides being weakened also has x syntactically substituted for y. This is immediately seen to be allowed.

- Non-principal cut formula on the left: Not possible

No axiom, at least one non-principal cut formula .

- $(\times L)$ on the right with non-principal cut formula:

$$\dfrac{\begin{array}{c}\pi_1\\ \Gamma\vdash u:\upsilon\end{array}\quad\dfrac{\begin{array}{c}\pi_2\\ x:\upsilon,y:\tau,z:\sigma,\Gamma'\vdash v:\rho\end{array}}{x:\upsilon,s:\tau\times\sigma,\Gamma'\vdash \text{let } \langle x,\, y\rangle{=}s \text{ in } v:\rho}\,(\times L)}{\Gamma,s:\tau\times\sigma,\Gamma'\vdash(\text{let } \langle y,\, z\rangle{=}s \text{ in } v)\langle\!\langle u/x\rangle\!\rangle:\rho}\,(Cut) \tag{2.4}$$

$$\longmapsto_{CE}\quad\dfrac{\begin{array}{c}\pi_1\\ \Gamma\vdash u:\upsilon\end{array}\quad\dfrac{\begin{array}{c}\pi_2\\ x:\upsilon,y:\tau,z:\sigma,\Gamma'\vdash v:\rho\end{array}}{\Gamma,y:\tau,z:\sigma,\Gamma'\vdash v\langle\!\langle u/x\rangle\!\rangle:\rho}\,(Cut)_l}{\Gamma,s:\tau\times\sigma,\Gamma'\vdash\text{let } \langle y,\, z\rangle{=}s \text{ in } v\langle\!\langle u/x\rangle\!\rangle:\rho}\,(\times L)$$

- $(\times R)$ on the right with non-principal cut formula:

$$\dfrac{\begin{array}{c}\pi_1\\ \Gamma\vdash u:\upsilon\end{array}\quad\dfrac{\begin{array}{cc}\pi_2 & \pi_3\\ x:\upsilon,\Gamma'\vdash v:\tau & x:\upsilon,\Gamma'\vdash w:\sigma\end{array}}{x:\upsilon,\Gamma'\vdash\langle v,\, w\rangle:\tau\times\sigma}\,(\times R)}{\Gamma,\Gamma'\vdash\langle v,\, w\rangle\langle\!\langle u/x\rangle\!\rangle:\tau\times\sigma}\,(Cut) \tag{2.5}$$

$$\longmapsto_{CE}\quad\dfrac{\dfrac{\begin{array}{c}\pi_1\\ \Gamma\vdash u:\upsilon\end{array}\quad\begin{array}{c}\pi_2\\ x:\upsilon,\Gamma'\vdash v:\tau\end{array}}{\Gamma,\Gamma'\vdash v\langle\!\langle u/x\rangle\!\rangle:\tau}\,(Cut)_l\quad\dfrac{\begin{array}{c}\pi_1\\ \Gamma\vdash u:\upsilon\end{array}\quad\begin{array}{c}\pi_3\\ x:\upsilon,\Gamma'\vdash w:\sigma\end{array}}{\Gamma,\Gamma'\vdash w\langle\!\langle u/x\rangle\!\rangle:\sigma}\,(Cut)_l}{\Gamma,\Gamma'\vdash\langle v\langle\!\langle u/x\rangle\!\rangle,\, w\langle\!\langle u/x\rangle\!\rangle\rangle:\tau\times\sigma}\,(\times R)$$

- $(\to L)$ on the right with non-principal cut formula:

$$\dfrac{\begin{array}{c}\pi_1\\ \Gamma\vdash w:\upsilon\end{array}\quad\dfrac{\begin{array}{cc}\pi_2 & \pi_3\\ f:\tau\to\sigma,y:\upsilon,\Gamma'\vdash u:\tau & x:\sigma,y:\upsilon,\Gamma'\vdash v:\gamma\end{array}}{f:\tau\to\sigma,y:\upsilon,\Gamma'\vdash\text{let } x{=}(f\,u) \text{ in } v:\gamma}\,(\to L)}{f:\tau\to\sigma,\Gamma,\Gamma'\vdash(\text{let } x{=}(f\,u) \text{ in } v)\langle\!\langle w/y\rangle\!\rangle:\gamma}\,(Cut) \tag{2.6}$$

$$\longmapsto_{CE} \dfrac{\dfrac{\overset{\pi_1}{\Gamma \vdash w : v} \quad \overset{\pi_2}{f : \tau \to \sigma, y : v, \Gamma' \vdash u : \tau}}{\Gamma, f : \tau \to \sigma, \Gamma' \vdash u\langle\!\langle w/y \rangle\!\rangle : \tau}(Cut)_l \quad \dfrac{\overset{\pi_1}{\Gamma \vdash w : v} \quad \overset{\pi_3}{x : \sigma, y : v, \Gamma' \vdash v : \gamma}}{\Gamma, x : \sigma, \Gamma' \vdash v\langle\!\langle w/y \rangle\!\rangle : \gamma}(Cut)_l}{\Gamma, f : \tau \to \sigma, \Gamma' \vdash \text{let } x{=}(f\ u\langle\!\langle w/y \rangle\!\rangle) \text{ in } v\langle\!\langle w/y \rangle\!\rangle : \gamma}(\to L)$$

- $(\to R)$ on the right with non-principal cut formula:

$$\dfrac{\overset{\pi_1}{\Gamma \vdash w : v} \quad \dfrac{\overset{\pi_2}{x : v, \Gamma', y : \tau \vdash u : \sigma}}{x : v, \Gamma' \vdash \lambda y.u : \tau \to \sigma}(\to R)}{\Gamma, \Gamma' \vdash (\lambda y.u)\langle\!\langle w/x \rangle\!\rangle : \tau \to \sigma}(Cut) \tag{2.7}$$

$$\longmapsto_{CE} \quad \dfrac{\dfrac{\overset{\pi_1}{\Gamma \vdash w : v} \quad \overset{\pi_2}{x : v, \Gamma', y : \tau \vdash u : \sigma}}{\Gamma, y : \tau, \Gamma' \vdash u\langle\!\langle w/x \rangle\!\rangle : \sigma}(Cut)_l}{\Gamma, \Gamma' \vdash \lambda y.u\langle\!\langle w/x \rangle\!\rangle : \tau \to \sigma}(\to R)$$

- $(\times L)$ on the left with non-principal cut formula:

$$\dfrac{\dfrac{\overset{\pi_1}{x : \tau, y : \sigma, \Gamma \vdash u : v}}{z : \tau \times \sigma, \Gamma \vdash \text{let } \langle x,\, y\rangle{=}z \text{ in } u : v}(\times L) \quad \overset{\pi_2}{t : v, \Gamma' \vdash v : \gamma}}{z : \tau \times \sigma, \Gamma, \Gamma' \vdash v\langle\!\langle \text{let } \langle x,\, y\rangle{=}z \text{ in } u/t \rangle\!\rangle : \gamma}(Cut) \tag{2.8}$$

$$\longmapsto_{CE} \quad \dfrac{\dfrac{\overset{\pi_1}{x : \tau, y : \sigma, \Gamma \vdash u : v} \quad \overset{\pi_2}{t : v, \Gamma' \vdash v : \gamma}}{x : \tau, y : \sigma, \Gamma, \Gamma' \vdash v\langle\!\langle u/t \rangle\!\rangle : \gamma}(Cut)_l}{z : \tau \times \sigma, \Gamma, \Gamma' \vdash \text{let } \langle x,\, y\rangle{=}z \text{ in } v\langle\!\langle u/t \rangle\!\rangle : \gamma}(\times L)$$

- $(\times R)$ on the left with non-principal cut formula: Not possible
- $(\to L)$ on the left with non-principal cut formula:

$$\dfrac{\dfrac{\overset{\pi_1}{f : \tau \to \sigma, \Gamma \vdash u : \tau} \quad \overset{\pi_2}{x : \sigma, \Gamma \vdash v : v}}{f : \tau \to \sigma, \Gamma \vdash \text{let } x{=}f\ u \text{ in } v : v}(\to L) \quad \overset{\pi_3}{s : v, \Gamma' \vdash w : \gamma}}{f : \tau \to \sigma, \Gamma, \Gamma' \vdash w\langle\!\langle (\text{let } x{=}f\ u \text{ in } v)/s \rangle\!\rangle : \gamma}(Cut) \tag{2.9}$$

$$\longmapsto_{CE} \dfrac{\dfrac{\pi_1^*}{f:\tau\to\sigma,\Gamma,\Gamma'\vdash u:\tau} \quad \dfrac{\begin{matrix}\pi_2 & \pi_3\\ x:\sigma,\Gamma\vdash v:\upsilon & s:\upsilon,\Gamma'\vdash w:\gamma\end{matrix}}{x:\sigma,\Gamma,\Gamma'\vdash w\langle\!\langle v/s\rangle\!\rangle:\gamma}\,(Cut)_l}{f:\tau\to\sigma,\Gamma,\Gamma'\vdash \text{let } x{=}f\ u \text{ in } w\langle\!\langle v/s\rangle\!\rangle:\gamma}\,(\to L)$$

- $(\to R)$ on the left with non-principal cut formula: Not possible

Both principal cut formulas .

- On $\times$

$$\dfrac{\dfrac{\begin{matrix}\pi_1 & \pi_2\\ \Gamma\vdash u:\tau & \Gamma\vdash v:\sigma\end{matrix}}{\Gamma\vdash\langle u,\,v\rangle:\tau\times\sigma}\,(\times R) \quad \dfrac{\pi_3}{z:\tau\times\sigma,\Gamma'\vdash \text{let } \langle x,\,y\rangle{=}z \text{ in } w:\upsilon}\,(\times L)}{\Gamma,\Gamma'\vdash(\text{let } \langle x,\,y\rangle{=}z \text{ in } w)\langle\!\langle\langle u,\,v\rangle/z\rangle\!\rangle:\upsilon}\,(Cut) \tag{2.10}$$

$$\longmapsto_{CE} \quad \dfrac{\begin{matrix}\pi_2\\ \Gamma'\vdash v:\sigma\end{matrix} \quad \dfrac{\begin{matrix}\pi_1 & \pi_3\\ \Gamma\vdash u:\tau & x:\tau,y:\sigma,\Gamma'\vdash w:\upsilon\end{matrix}}{\Gamma,y:\sigma,\Gamma'\vdash w\langle\!\langle u/x\rangle\!\rangle:\upsilon}\,(Cut)_r}{\Gamma',\Gamma,\Gamma'\vdash w\langle\!\langle u/x\rangle\!\rangle\langle\!\langle v/y\rangle\!\rangle:\upsilon}\,(Cut)_r$$

and the result follows after full cut elimination by closure under weakening.

- On $\to$

$$\dfrac{\dfrac{\pi_1}{\Gamma,x:\tau\vdash u:\sigma}\,(\to R) \quad \dfrac{\begin{matrix}\pi_2 & \pi_3\\ f:\tau\to\sigma,\Gamma'\vdash v:\tau & y:\sigma,\Gamma'\vdash w:\upsilon\end{matrix}}{f:\tau\to\sigma,\Gamma'\vdash \text{let } y{=}f\ v \text{ in } w:\upsilon}\,(\to L)}{\Gamma,\Gamma'\vdash(\text{let } y{=}f\ v \text{ in } w)\langle\!\langle\lambda x.u/f\rangle\!\rangle:\upsilon}\,(Cut) \tag{2.11}$$

$$\longmapsto_{CE} \dfrac{\dfrac{\dfrac{\pi_1}{\Gamma,x:\tau\vdash u:\sigma}\,(\to R)}{\Gamma\vdash\lambda x.u:\tau\to\sigma} \quad \dfrac{\pi_2}{f:\tau\to\sigma,\Gamma'\vdash v:\tau}\,(Cut)_l \quad \dfrac{\begin{matrix}\pi_1 & \pi_3\\ \Gamma,x:\tau\vdash u:\sigma & y:\sigma,\Gamma'\vdash w:\upsilon\end{matrix}}{\Gamma,x:\tau,\Gamma'\vdash w\langle\!\langle u/y\rangle\!\rangle:\upsilon}\,(Cut)_r}{\Gamma,\Gamma',\Gamma,\Gamma'\vdash w\langle\!\langle u/y\rangle\!\rangle\langle\!\langle v\langle\!\langle\lambda x.u/f\rangle\!\rangle/x\rangle\!\rangle:\upsilon}\,(Cut)_r$$

and the result follows after full cut elimination by closure under weakening.

□

2.2.3 The new results

The Hauptsatz is essentially a WN result for the process of obtaining cut free derivations. The proof structure for the Hauptsatz is thus a terminating reduction strategy on redices given as eliminatable cuts. By proof structure we mean the inter-dependencies between the different inductions that the proof is built from. A sub-induction on some cut in a derivation, for example, will prescribe that the derivation ending in the considered cut is made cut free before we proceed to consider the full derivation. By the structure of the above proof it is appearant that the reduction strategy employed involves full normalization and not just one-step reduction on derivations. We will proceed to consider one-step rewriting, which we will call Cut Propagation. It will be introduced together with a very simple SN result.

Definition 13 (Innermost Strong Normalization) *A reduction on a calculus is said to enjoy Innermost Strong Normalization, ISN, if it enjoys SN when restricted to innermost redices.*

Corollary 3 (ISN for cut propagation) *The process of continuously picking a topmost cut and rewriting it one step according to the rewriting steps in the Hauptsatz terminates.*

The elimination steps performed in the Hauptsatz are purely local, but for step 2.3. These substitutions are, however, only on the derivations ending in the considered cut. Two topmost cuts can thus safely be eliminated interleaved without changing any properties of the process.

Proof: (by induction on the number of [pre-existing] topmost cuts with a sub-induction as for the Hauptsatz).

Consider a proof-tree with n topmost cuts:

$$\dfrac{\dfrac{\begin{matrix}\pi_1 & \delta_1\end{matrix}}{\Gamma_1 \vdash M_1 : \tau_1}(Cut) \quad \cdots \quad \dfrac{\begin{matrix}\pi_n & \delta_n\end{matrix}}{\Gamma_n \vdash M_n : \tau_n}(Cut)}{\Gamma \vdash N : \sigma}(Rule)$$

As the cuts are topmost we can eliminate them interleaved. When one elimination terminates, at most one other cut can become topmost:

$$\dfrac{\begin{array}{ccccc} \theta_1 & & \phi_i & & \theta_n \\ \Gamma'_1 \vdash M'_1 : \tau_1 & \cdots & \Gamma_i \vdash M'_1 : \tau_1 & \cdots & \Gamma'_n \vdash M'_n : \tau_n \\ ? & & \vdots & & ? \\ & & \dfrac{}{\Gamma' \vdash M' : \tau'}\,(Cut) & & \\ & & \cdots & & \end{array}}{\Gamma \vdash N : \sigma}\,(Rule)$$

This new cut is, again, seen to be immediately eliminatable even though the overall proof structure is not in correct form (cf. the ?'s above).

Therefore we can continuously apply the rules from the Hauptsatz to either of the derivations stemming from pre-existing top-most cuts. Each derivation has a top-most cut which vacuously is eliminatable.

□

$$\begin{array}{rcl} e & ::= & x \\ & | & e_1\langle\!\langle e_2/x \rangle\!\rangle \\ & | & \text{let } \langle x_1, x_2\rangle = x_3 \text{ in } e \\ & | & \langle e_1, e_2\rangle \\ & | & \text{let } x_1 = x_2\, e_1 \text{ in } e_2 \\ & | & \lambda x.e \end{array}$$

Figure 2.4: Syntax of proof terms for Sm

Given the one-to-one correspondence between derivations in Sm+Cut and the proof terms we have shown:

Proposition 4 *The Hauptsatz calculus –as induced by the Hauptsatz and Corollary 3– enjoys ISN.*

The pre-terms of the calculus are shown in Figure 2.4, the terms are the pre-terms provable in Nm, and the reductions are given in Figure 2.5.

As for Figure 2.5, it is worth noticing that the rules on $\cdot\langle\!\langle\cdot/\cdot\rangle\!\rangle$ come in three flavours (i) instantiation, (ii) promotion, and (iii) interaction. They are seen to very precisely describe a calculus with explicit substitutions even though the form is different from that of others [49, 64].

Instantiation

$$
\begin{array}{rcl}
x\langle\!\langle u/x\rangle\!\rangle & \rightarrow_{2.1} & u \\
y\langle\!\langle u/x\rangle\!\rangle & \rightarrow_{2.2} & y \\
u\langle\!\langle x/y\rangle\!\rangle & \rightarrow_{2.3} & u[x/y]
\end{array}
$$

Promotion

$$
\begin{array}{rcl}
(\text{let } \langle y, z\rangle{=}s \text{ in } v)\langle\!\langle u/x\rangle\!\rangle & \rightarrow_{2.4} & \text{let } \langle y, z\rangle{=}s \text{ in } v\langle\!\langle u/x\rangle\!\rangle \\
\langle v, w\rangle\langle\!\langle u/x\rangle\!\rangle & \rightarrow_{2.5} & \langle v\langle\!\langle u/x\rangle\!\rangle, w\langle\!\langle u/x\rangle\!\rangle\rangle \\
(\text{let } x{=}f\ u \text{ in } v)\langle\!\langle w/y\rangle\!\rangle & \rightarrow_{2.6} & \text{let } x{=}f\ u\langle\!\langle w/y\rangle\!\rangle \text{ in } v\langle\!\langle w/y\rangle\!\rangle \\
(\lambda y.u)\langle\!\langle w/x\rangle\!\rangle & \rightarrow_{2.7} & \lambda y.u\langle\!\langle w/x\rangle\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
v\langle\!\langle \text{let } \langle x, y\rangle{=}z \text{ in } u/t\rangle\!\rangle & \rightarrow_{2.8} & \text{let } \langle x, y\rangle{=}z \text{ in } v\langle\!\langle u/t\rangle\!\rangle \\
w\langle\!\langle \text{let } x{=}f\ u \text{ in } v/s\rangle\!\rangle & \rightarrow_{2.9} & \text{let } x{=}f\ u \text{ in } w\langle\!\langle v/s\rangle\!\rangle
\end{array}
$$

Interaction

$$
\begin{array}{rcl}
(\text{let } \langle x, y\rangle{=}z \text{ in } w)\langle\!\langle\langle u, v\rangle/z\rangle\!\rangle & \rightarrow_{2.10} & w\langle\!\langle u/x\rangle\!\rangle\langle\!\langle v/y\rangle\!\rangle \\
(\text{let } y{=}f\ v \text{ in } w)\langle\!\langle\lambda x.u/f\rangle\!\rangle & \rightarrow_{2.11} & w\langle\!\langle u/y\rangle\!\rangle\langle\!\langle v\langle\!\langle\lambda x.u/f\rangle\!\rangle/x\rangle\!\rangle
\end{array}
$$

In the above the meta-variables u, v, w range over arbitrary terms, while x, y, z, s, f range over variable names. For the latter case, different meta-variables are required to denote different variable names.

Figure 2.5: Reduction as induced by the Hauptsatz

There are two flavours of promotion rules depending on which argument of the operator we are considering. The outermost argument is the term originating as the right premise of the cut rule, while the innermost one is originating from the left premise. The third argument is the variable annotating the cut formula in the right premise. The difference in numbers between these two flavours are due to the fact that the left premise cannot

have a non-principal cut formula because antecedents are restricted to contain exactly one formula[9]. The interaction and instantiation rules prescribes how the $\cdot\langle\!\langle\cdot/\cdot\rangle\!\rangle$ operator interacts with the rest of the syntax. Especially concerning the interaction rules, it is worth noticing how they fit our description of left rules as introducing points of interaction.

Comparing Hauptsatz proofs

As already remarked the present Hauptsatz proof greatly resembles the proof in [70][10]. We will compare the two proofs:

Proof terms: We use a term-annotated logic, as we are interested in the computational correspondence of cut elimination, Figure 2.6.

Proof structure: Instead of a well-founded sub-induction they proceed by induction on cut-rank with a sub-induction on levels of eliminatable cuts. Their structure would not have allowed us to prove Corollary 3 without analyzing how this proceeds. As it turns out a topmost cut being rewritten can only introduce cuts with the same rank that are topmost and the result would actually carry through.

Step 2.11: We have changed step 2.11 slightly for increased readability of the resulting proof term reduction. Compare $w\langle\!\langle(u\langle\!\langle(v\langle\!\langle\lambda x.u/f\rangle\!\rangle)/x\rangle\!\rangle)/y\rangle\!\rangle$ to Figure 2.5, rule 2.11.

2.2.4 The future results

We will in this section sketch how the close connection between the Hauptsatz and its induced calculus can be exploited to come up with even stronger SN results for the process of making derivations cut-free. Such results appear only to have been considered very recently[25, 66][11] and for that reason we have dared to suggest this scenario as a possible future path. It is based on the following result where 'non-overlapping TRS' will be defined later.

[9] Intuitionistic logic is obtained from Minimal logic by adding a designated falsity formula, $\bot$, and rules on it. In turn Classical logic is obtained from Intuitionistic logic by allowing several formulas in the antecedent.

[10] Actually most Hauptsatz proofs are of the same form but with different proof structure.

[11] Brought to the attention of the present author by Anne S. Troelstra. The latter kindly made available by the author.

Logic		Calculus
Cut	$\leftrightarrow$	Explicit substitution
Eliminatable cut	$\leftrightarrow$	Permissible redex
Proof structure	$\leftrightarrow$	Reduction strategy
Cut Elimination	$\leftrightarrow$	Normalization of subterm
Cut Propagation	$\leftrightarrow$	One-step reduction
Hauptsatz	$\leftrightarrow$	WN for reduction
Corollary 3	$\leftrightarrow$	ISN for reduction

Figure 2.6: The Curry-Howard correspondence of Sequent Calculus Logic and Explicit Substitution Calculus

Theorem 11 ([32, Theorem 5 (a)]) *For any non-overlapping TRS we have:*

$$ISN \Rightarrow SN$$

We can remark that in [34], Gramlich presents several sufficient conditions under which SN follows from ISN. The above is the most immediate.

Definition 14 (Term Rewriting Systems) *A TRS consists of a countable signature, $\mathcal{F}$, of capital-letter function symbols, each with an associated arity and of a disjoint, countable set of variables, $\mathcal{V}$, written with lower-case letters. The terms over a signature and a set of variables, $\mathcal{T}(\mathcal{F},\mathcal{V})$, are the set of "fully applied" functions built from other such functions and variables. Furthermore there is a rewriting relation $\mathcal{R} \subseteq \mathcal{T}(\mathcal{F},\mathcal{V}) \times \mathcal{T}(\mathcal{F},\mathcal{V})$, s.t., for all $(l,r) \in \mathcal{R}$, written $l \to r$, the variables in r all occur in l where furthermore $l \notin \mathcal{V}$. Term rewriting is performed according to rewriting rules up to variable substitution and is defined to be compatible (cf. Chapter 1).*

We will be slightly more general for ease of notation and to be able to come up with a tentative formalization of future work.

Definition 15 (Conditional Term Rewriting System [33]) *A CTRS is a TRS which furthermore are allowed to have positive side-conditions on rewriting rules:*

$$l \to r \qquad \Leftarrow \bigwedge_i (l_i = r_i)$$

The l_i, r_i can be subterms of l, r respectively. The intended semantics is that the rewrite rule is permitted for variable instantiations that satisfy the conjunction.

Proposition 5 *(*[**33, Theorem 3.13**]*) Theorem 11 also holds when considering a CTRS.*

We can observe that by countable expansion of each rule, side-conditions like the following are allowed where the T_i's are terms:

$$x \in \{T_1, \ldots\}$$

After this little intermezzo, we will return to the basic definitions, and then go on to sketch our proposal.

Definition 16 (Critical Pair, Non-Overlapping) *Two rules $l_1 \to r_1, l_2 \to r_2$ are said to constitute a critical pair if, say, l_1, up to unification, is a non-variable subterm of l_2. A TRS is non-overlapping if there are no critical pairs.*

Rule 2.3 of Figure 2.5 is immediately seen not to be expressible as a rewriting rule without taking $\cdot[\cdot/\cdot]$ to be an explicit operator, in which case we would have accomplished very little. Fortunately we have:

Proposition 6 *Rule 2.3 is admissible for the calculus.*

Proof: By case-splitting on u's form one of the rules 2.1, 2.2, 2.4, ..., 2.7 is seen to be applicable.

□

While making sure we are able to distinguish between terms, variables and explicit substitutions, we arrive at an abstract syntax for the pre-terms of the calculus written as a TRS, Figure 2.7. In the figure we have for simplicity precluded all cuts on cuts. Eliminatable cuts only precluded cuts on two cuts. The relevant cases can be added.

A well-known property of Sequent Calculus logic is the lack of WCR:

$$\begin{aligned} &\to_{2.4} \quad \text{let } \langle x_1, y_1\rangle{=}z_1 \text{ in } v\langle\!\langle(\text{let } \langle x_2, y_2\rangle{=}z_2 \text{ in } w)/s\rangle\!\rangle \\ &(\text{let } \langle x_1, y_1\rangle{=}z_1 \text{ in } v)\langle\!\langle(\text{let } \langle x_2, y_2\rangle{=}z_2 \text{ in } w)/s\rangle\!\rangle \\ &\to_{2.8} \quad \text{let } \langle x_2, y_2\rangle{=}z_2 \text{ in } (\text{let } \langle x_1, y_1\rangle{=}z_1 \text{ in } v)\langle\!\langle w/s\rangle\!\rangle \end{aligned}$$

$$\mathcal{V} = \{x, y, x_i, e, e_i\}$$
$$\mathcal{F} = \{\mathrm{V}^1, \mathrm{S}^3, \mathrm{T}^1, \mathrm{LP}^4, \mathrm{P}^2, \mathrm{LF}^4, \mathrm{A}^2, X_1^0, X_2^0, \ldots\}$$

V $\sim$ variables; S $\sim$ expl.subst.; T $\sim$ term;
LP $\sim$ let on pairs; P $\sim$ pairs;LF $\sim$ let on fcts.;
A $\sim$ abstraction; $X_i \sim$ variable names

$$\begin{array}{rcl} e & ::= & \mathrm{V}(X_i) \\ & | & \mathrm{S}(e_1, e_2, \mathrm{V}(X_i)) \\ & | & \mathrm{T}(\mathrm{LP}(\mathrm{V}(X_{i_1}), \mathrm{V}(X_{i_2}), \mathrm{V}(X_{i_3}), e)) \\ & | & \mathrm{T}(\mathrm{P}(e_1, e_2)) \\ & | & \mathrm{T}(\mathrm{LF}(\mathrm{V}(X_{i_1}), \mathrm{V}(X_{i_2}), e_1, e_2)) \\ & | & \mathrm{T}(\mathrm{A}(\mathrm{V}(X_i), e)) \end{array}$$

Where we furthermore restrict the e_i's in S
to be either T or V (four possibilities).

Figure 2.7: The Hauptsatz pre-terms as a TRS

This example exactly addresses that Rules 2.4 & 2.8 constitute a critical pair and we need to remove all of these before we have a non-overlapping system. A way to do so is to fix an order in which we apply the different rules. We will do it by not allowing explicit substitutions on outermost let-constructs if there is in innermost let-construct, Figure 2.8. Observe that we use negative side-conditions to resolve the critical pairs arising, e.g., from the first two rules. We thus go beyond Proposition 5. We also do not consider Rule 2.3 nor explicit substitutions on explicit substitutions. Notice also that, as we only consider terms and not all pre-terms, there are a number of implicit conditions on the rules in the figure which need not be made explicit.

To apply Theorem 11 to this system we would either need to look for generalizations to negative side-conditions [74] or to remove them by, e.g., considering a calculus with De Bruijn indices [49] which in itself could create critical pairs.

Let us for a while neglect these concerns and recapitulate the known results that could lead to a stronger SN result and analyze the expected nature of it.

Proposition 7 *The [negative] CTRS of Figure 2.8 enjoys ISN.*

Proof: Immediate.

□

Proposition 8 *The [negative] CTRS of Figure 2.8 has no critical pairs.*

Proof: By straightforward case-splitting.

□

Now if we for a while assumed that Theorem 11 applied to CTRS's with inequalities between zero-arity function symbols we would obtain:

Non-Theorem 1 *The [negative] CTRS of Figure 2.8 enjoys SN.*

Non-Corollary 1 *The Hauptsatz calculus with similar restrictions enjoys SN.*

Non-Corollary 2 *Cut Propagation with similar restrictions enjoys SN*

We would thus have shown that rewriting on a derivation with cuts would terminate when we rewrite cuts[12] that have no immediately preceding cuts. The rewriting is one-step, deterministic, and precludes the use of Rule 2.3.

This would be a rather unusual cut elimination result, as it is rather strong and furthermore type-independent. That is, it needs no notion of eliminatable cut defined by rank and cutrank. The price is that the process becomes deterministic and thus vacuously enjoys (W)CR which might not be desirable. However, the result extends to: Any restriction on the rewriting relation that makes it deterministic[13] produce a strongly normalizing rewriting system.

[12] Observe that we consider all cuts and not just eliminatable cuts.
[13] By using [32, Theorem 9] this can be strengthened to WCR.

$$
\begin{array}{lll}
\mathrm{S}(\mathrm{V}(x),e,\mathrm{V}(x)) & \to_{2.1} \quad e & \Leftarrow e \in \{\mathrm{V},\mathrm{T}\} \\
\mathrm{S}(\mathrm{V}(x),e,\mathrm{V}(y)) & \to_{2.2} \quad \mathrm{V}(x) & \Leftarrow x \neq y \wedge e \in \{\mathrm{V},\mathrm{T}\}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{S}(\mathrm{T}(\mathrm{LP}(\mathrm{V}(x_1),\mathrm{V}(x_2),\mathrm{V}(x_3),e_1)),e_2,\mathrm{V}(x_4)) \\
\qquad \to_{2.4} \quad \mathrm{T}(\mathrm{LP}(\mathrm{V}(x_1),\mathrm{V}(x_2),\mathrm{V}(x_3),\mathrm{S}(e_1,e_2,\mathrm{V}(x_4)))) \\
\qquad \Leftarrow \quad x_3 \neq x_4 \wedge e_2 \in \{\mathrm{V},\mathrm{T}(\mathrm{A}),\mathrm{T}(\mathrm{P})\} \\
\mathrm{S}(\mathrm{T}(\mathrm{P}(e_1,e_2)),e_3,\mathrm{V}(x)) \\
\qquad \to_{2.5} \quad \mathrm{T}(\mathrm{P}(\mathrm{S}(e_1,e_3,\mathrm{V}(x)),\mathrm{S}(e_2,e_3,\mathrm{V}(x)))) \\
\qquad \Leftarrow \quad e_3 \in \{\mathrm{V},\mathrm{T}(\mathrm{A}),\mathrm{T}(\mathrm{P})\} \\
\mathrm{S}(\mathrm{T}(\mathrm{LF}(\mathrm{V}(x_1),\mathrm{V}(x_2),e_1,e_2)),e_3,\mathrm{V}(x_3)) \\
\qquad \to_{2.6} \quad \mathrm{T}(\mathrm{LF}(\mathrm{V}(x_1),\mathrm{V}(x_2),\mathrm{S}(e_1,e_3,\mathrm{V}(x_3)),\mathrm{S}(e_2,e_3,\mathrm{V}(x_3)))) \\
\qquad \Leftarrow \quad x_2 \neq x_3 \wedge e_3 \in \{\mathrm{V},\mathrm{T}(\mathrm{A}),\mathrm{T}(\mathrm{P})\} \\
\mathrm{S}(\mathrm{T}(\mathrm{A}(\mathrm{V}(x_1),e_1)),e_2,\mathrm{V}(x_2)) \\
\qquad \to_{2.7} \quad \mathrm{T}(\mathrm{A}(\mathrm{V}(x_1),\mathrm{S}(e_1,e_2,\mathrm{V}(x_2)))) \\
\qquad \Leftarrow \quad e_2 \in \{\mathrm{V},\mathrm{T}(\mathrm{A}),\mathrm{T}(\mathrm{P})\} \\
\mathrm{S}(e_1,\mathrm{T}(\mathrm{LP}(\mathrm{V}(x_1),\mathrm{V}(x_2),\mathrm{V}(x_3),e_2)),x_4) \\
\qquad \to_{2.8} \quad \mathrm{T}(\mathrm{LP}(\mathrm{V}(x_1),\mathrm{V}(x_2),\mathrm{V}(x_3),\mathrm{S}(e_1,e_2,x_4))) \\
\qquad \Leftarrow \quad x_3 \neq x_4 \wedge e_1 \in \{\mathrm{V},\mathrm{T}\} \\
\mathrm{S}(e_1,\mathrm{T}(\mathrm{LF}(\mathrm{V}(x_1),\mathrm{V}(x_2),e_2,e_3)),x_3) \\
\qquad \to_{2.9} \quad \mathrm{T}(\mathrm{LF}(\mathrm{V}(x_1),\mathrm{V}(x_2),e_2,\mathrm{S}(e_1,e_3,x_3))) \\
\qquad \Leftarrow \quad x_2 \neq x_3 \wedge e_1,e_2 \in \{\mathrm{V},\mathrm{T}\}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{S}(\mathrm{T}(\mathrm{LP}(\mathrm{V}(x_1),\mathrm{V}(x_2),\mathrm{V}(x_3),e_1)),\mathrm{T}(\mathrm{P}(e_2,e_3)),\mathrm{V}(x_3)) \\
\qquad \to_{2.10} \quad \mathrm{S}(\mathrm{S}(e_1,e_2,\mathrm{V}(x_1)),e_3,\mathrm{V}(x_2)) \\
\mathrm{S}(\mathrm{T}(\mathrm{LF}(\mathrm{V}(x_1),\mathrm{V}(x_2),e_1,e_2)),\mathrm{T}(\mathrm{A}(\mathrm{V}(x_3),e_3)),\mathrm{V}(x_2)) \\
\qquad \to_{2.11} \quad \mathrm{S}(\mathrm{S}(e_2,e_3,\mathrm{V}(x_1)),\mathrm{S}(e_1,\mathrm{T}(\mathrm{A}(\mathrm{V}(x_3),e_3)),\mathrm{V}(x_2)),\mathrm{V}(x_3))
\end{array}
$$

Figure 2.8: The rewriting relation on the Hauptsatz negative CTRS

Chapter 3

Proof Normalization and Type-Directed Partial Evaluation in $\lambda_{\to\times}^{\text{Curry-II}}$

3.1 Introduction

In this chapter we will use $\lambda_{\to\times}^{\text{Curry-II}}$, as defined in Chapter 1, to give a syntactical presentation with correctness proof of Berger & Schwichtenberg's proof normalizer [7] and with that also of Danvy's type-directed partial evaluator [19]. We would like to point out that two other correctness proofs are known to the author. One is a semantical proof [7] and the other is a strictly formal proof followed by a realization interpretation [6]. The present approach is thus hoped to complete the picture.

We will start by introducing the method as performing coercions between the levels of $\lambda_{\to\times}^{\text{Curry-II}}$, the formalization of which coincides exactly with Danvy's presentation of Berger & Schwichtenberg's proof normalizer [19, Figure 1]. The method will then be proven correct, which after a few considerations immediately gives correctness of Danvy's type-directed partial evaluator that implements the method. We then go on to present a less general version of the correctness proof given in [7]. We finish the chapter by briefly mentioning the proof in [6]. We have not been able to present a simplified version. In Chapter 4 we will investigate the practical implications of type-directed partial evaluation and consider one particular and very promising application.

3.1.1 Disclaimer

We will remind the reader of some of the comments made in Chapter 1 pertaining to terminology. Even though we use the words 'static' and 'dynamic' in ways reminiscent of their use in other areas, specifically in Partial Evaluation, there are significant differences. Most notably does our calculus allow base typed terms to be considered both as dynamic and as static.

Furthermore, the calculus is not directly designed to be simulated in any existing programming language, but we will show that Scheme [10] implements the method at hand[1]. Rather the calculus is intended to give as straightforward a proof of correctness as possible while at the same time providing us with a suitable frame for the work to be presented in Chapter 6. It is the authors hope that the reader finds this performed to a satisfactory degree.

3.2 Two-level coercions

3.2.1 Prologue

We will present a number of examples each highlighting different relevant issues for the technique. They will slowly progress from immediate to advanced.

- Consider $\overline{\lambda}v.v \in \overline{\Lambda}_{\mathrm{II}}^{o \to o}$. If we wish to change its level we can do the following: First as we, from the type, know the term is a function let us construct a function abstracting a fresh variable, say x, of type o, then let us construct the body of this function according to the original term. Do this by applying the original term, at base type, to x:

$$\underline{\lambda}x.(\overline{\lambda}v.v)\,\overline{@}\,x \quad =_{\overline{\beta_{\rightarrow}}} \quad \underline{\lambda}x.x$$

- Next look at $\overline{\lambda}v.\overline{\lambda}w.v\,\overline{@}\,w \in \overline{\Lambda}_{\mathrm{II}}^{(o \to o) \to o \to o}$. We cannot apply this term to a dynamically abstracted variable, s, of correct type as they belong to different levels and are not of base type but rather of type $o \to o$. However, we can change the level of s by: $\overline{\lambda}z.s\,\underline{@}\,z$, in which all interfaces between levels again are at base type. The result of applying the original term to this term is of type $o \to o$ wish we know how to

[1] It is worth noticing already at this point that the functions performing the coercions are not definable in $\lambda_{\rightarrow\times}^{\text{Curry-II}}$ but instead gives rise to $\lambda_{\rightarrow\times}^{\text{Curry-II}}$-terms.

handle by the above example hence the resulting term is:

$$\begin{aligned}
&\underline{\lambda}s.\underline{\lambda}x.((\overline{\lambda}v.\overline{\lambda}w.v\,\overline{@}\,w)\,\overline{@}\,(\overline{\lambda}z.s\,\underline{@}\,z))\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}s.\underline{\lambda}x.(\overline{\lambda}w.(\overline{\lambda}z.s\,\underline{@}\,z)\,\overline{@}\,w)\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}s.\underline{\lambda}x.(\overline{\lambda}w.s\,\underline{@}\,w)\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}s.\underline{\lambda}x.s\,\underline{@}\,x
\end{aligned}$$

- Following this example let us look at $\overline{\lambda}v.\overline{\lambda}w.v\,\overline{@}\,(v\,\overline{@}\,w) \in \overline{\Lambda}_{\mathrm{II}}^{(o\to o)\to o\to o}$. The method described above will treat this example exactly as the former and we have:

$$\begin{aligned}
&\underline{\lambda}s.\underline{\lambda}x.((\overline{\lambda}v.\overline{\lambda}w.v\,\overline{@}\,(v\,\overline{@}\,w))\,\overline{@}\,(\overline{\lambda}z.s\,\underline{@}\,z))\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}s.\underline{\lambda}x.\overline{\lambda}w.(\overline{\lambda}z.s\,\underline{@}\,z)\,\overline{@}\,((\overline{\lambda}z.s\,\underline{@}\,z)\,\overline{@}\,w)\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}s.\underline{\lambda}x.\overline{\lambda}w.(\overline{\lambda}z.s\,\underline{@}\,z)\,\overline{@}\,(s\,\underline{@}\,w)\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}s.\underline{\lambda}x.(\overline{\lambda}w.s\,\underline{@}\,(s\,\underline{@}\,w))\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}s.\underline{\lambda}x.s\,\underline{@}\,(s\,\underline{@}\,x)
\end{aligned}$$

- Another example is $(\overline{\lambda}v.v)\,\overline{@}\,(\overline{\lambda}w.w) \in \overline{\Lambda}_{\mathrm{II}}^{o\to o}$.

$$\begin{aligned}
&\underline{\lambda}x.(\overline{\lambda}v.v\,\overline{@}\,\overline{\lambda}w.w)\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}x.(\overline{\lambda}w.w)\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}x.x
\end{aligned}$$

  Where we see a β-normalizing effect.

- It is also interesting to look at $\overline{\lambda}v.v \in \overline{\Lambda}_{\mathrm{II}}^{(o\to o)\to o\to o}$

$$\begin{aligned}
&\underline{\lambda}s.\underline{\lambda}x.(\overline{\lambda}v.v\,\overline{@}\,(\overline{\lambda}w.s\,\underline{@}\,w))\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}s.\underline{\lambda}x.(\overline{\lambda}w.s\,\underline{@}\,w)\,\overline{@}\,x\\
&\qquad =_{\overline{\beta_{\to}}} \quad \underline{\lambda}s.\underline{\lambda}x.s\,\underline{@}\,x
\end{aligned}$$

  This example on the other hand shows an η-normalizing effect.

- Finally we will consider a single example with pairs, $\overline{\lambda}a.a \in \overline{\Lambda}_{\mathrm{II}}^{o\times o\to o\times o}$. Whenever we meet a pair we create a pair in the new level consisting of properly treated projections of the considered term in the old level:

$$\underline{\lambda}x.\underline{\langle}\overline{\mathrm{p}}_1(((\overline{\lambda}a.a)\,\overline{@}\,\overline{\langle}\underline{\mathrm{p}}_1(x),\,\underline{\mathrm{p}}_2(x)\overline{\rangle})),\,\overline{\mathrm{p}}_2(((\overline{\lambda}a.a)\,\overline{@}\,\overline{\langle}\underline{\mathrm{p}}_1(x),\,\underline{\mathrm{p}}_2(x)\overline{\rangle}))\underline{\rangle}$$

$$
\begin{aligned}
=_{\overline{\beta_{\to}}} \quad & \underline{\lambda} x.\underline{\langle}\overline{\mathrm{p}}_1(\overline{\langle}\underline{\mathrm{p}}_1(x),\, \underline{\mathrm{p}}_2(x)\overline{\rangle}),\, \overline{\mathrm{p}}_2(((\overline{\lambda} a.a)\,\overline{@}\,\overline{\langle}\underline{\mathrm{p}}_1(x),\, \underline{\mathrm{p}}_2(x)\overline{\rangle}))\underline{\rangle} \\
=_{\overline{\beta_{\to}}} \quad & \underline{\lambda} x.\underline{\langle}\overline{\mathrm{p}}_1(\overline{\langle}\underline{\mathrm{p}}_1(x),\, \underline{\mathrm{p}}_2(x)\overline{\rangle}),\, \overline{\mathrm{p}}_2(\overline{\langle}\underline{\mathrm{p}}_1(x),\, \underline{\mathrm{p}}_2(x)\overline{\rangle})\underline{\rangle} \\
=_{\overline{\beta_{\times}}} \quad & \underline{\lambda} x.\underline{\langle}\underline{\mathrm{p}}_1(x),\, \overline{\mathrm{p}}_2(\overline{\langle}\underline{\mathrm{p}}_1(x),\, \underline{\mathrm{p}}_2(x)\overline{\rangle})\underline{\rangle} \\
=_{\overline{\beta_{\times}}} \quad & \underline{\lambda} x.\underline{\langle}\underline{\mathrm{p}}_1(x),\, \underline{\mathrm{p}}_2(x)\underline{\rangle}
\end{aligned}
$$

What we did throughout these examples was to use [a hybrid of] η-expansion heavily. Notice, however, that dynamic expansions are performed on a static terms and vice versa which is not allowed by the side conditions of Definition 7. That is, our method cannot be defined in $\lambda_{\to\times}^{\text{Curry-II}}$ but we will in Theorem 12 prove that the method is well-defined and always produces $\lambda_{\to\times}^{\text{Curry-II}}$-terms. What we will gain from the presentation in $\lambda_{\to\times}^{\text{Curry-II}}$ is the characterization of normal forms we obtained in Theorem 8 and Corollary 2, namely that the $\overline{\beta}$-normal form[2] of a dynamic term is completely dynamic and that $\overline{\beta}$-reduction preserves $\underline{\beta\eta}$-normal forms. The former result tells us that we for free have obtained a separation between a computational relevant level (the static) and a presentation relevant level (the dynamic). The latter result tells us that if we only introduce normal-formed dynamic terms the final result will be in normal form. Furthermore this presentation allows us –in Chapter 6– to immediately generalize the method to change the representation of the final form and show a completeness result for the expressiveness of the method at hand.

3.2.2 η-expansion

The used η-expansion can be formalized as the type-indexed family of functions given in Figure 3.1. The reader is encouraged to read these six lines with the intuition that they are coercing a term from one level to the other while only allowing level interface at base type.

Theorem 12 *For proofs*

$$
\begin{array}{cc}
\pi_1 & \pi_2 \\
v\,\overline{:}\,\tau & e\,\underline{:}\,\tau
\end{array}
$$

we have respectively:

$$
\begin{array}{cc}
\pi_1 & \pi_2 \\
v\,\overline{:}\,\tau & e\,\underline{:}\,\tau \\
\vdots & \vdots \\
\downarrow^{\tau} v\,\underline{:}\,\tau & \uparrow_{\tau} e\,\overline{:}\,\tau
\end{array}
$$

[2] Overlined is static and underlined is dynamic

$$
\begin{array}{rcll}
\downarrow^{o} v & = & v & \\
\downarrow^{\tau_1 \times \tau_2} v & = & \underline{\langle} \downarrow^{\tau_1} \overline{\mathrm{p}}_1(v), \downarrow^{\tau_2} \overline{\mathrm{p}}_2(v) \underline{\rangle} & \\
\downarrow^{\tau_1 \rightarrow \tau_2} v & = & \underline{\lambda} x.\downarrow^{\tau_2} (v \,\overline{@}\, (\uparrow_{\tau_1} x)) & \text{; for x fresh} \\
\uparrow_{o} e & = & e & \\
\uparrow_{\tau_1 \times \tau_2} e & = & \overline{\langle} \uparrow_{\tau_1} \underline{\mathrm{p}}_1(e), \uparrow_{\tau_2} \underline{\mathrm{p}}_2(e) \overline{\rangle} & \\
\uparrow_{\tau_1 \rightarrow \tau_2} e & = & \overline{\lambda} v.\uparrow_{\tau_2} (e \,\underline{@}\, (\downarrow^{\tau_1} v)) & \text{; for v fresh}
\end{array}
$$

Figure 3.1: Level coercers for $\lambda_{\rightarrow \times}^{\text{Curry-II}}$.

Proof: (Double induction on τ but only showing static to dynamic)

$\tau \equiv o$:

$$\frac{\begin{array}{c}\pi \\ v \,\overline{:}\, o\end{array}}{v \,\underline{:}\, o}\,(Dyn)$$

$\tau \equiv \tau_1 \times \tau_2$ For π given by assumption and the $\vdots$'s given by I.H. we have:

$$\frac{\begin{array}{cc}\begin{array}{c}\pi \\ \dfrac{v \,\overline{:}\, \tau_1 \times \tau_2}{\overline{\mathrm{p}}_1(v) \,\overline{:}\, \tau_1}\,(\times E_1) \\ \vdots \\ \downarrow^{\tau_1} \overline{\mathrm{p}}_1(v) \,\underline{:}\, \tau_1\end{array} & \begin{array}{c}\pi \\ \dfrac{v \,\overline{:}\, \tau_1 \times \tau_2}{\overline{\mathrm{p}}_2(v) \,\overline{:}\, \tau_2}\,(\times E_2) \\ \vdots \\ \downarrow^{\tau_2} \overline{\mathrm{p}}_2(v) \,\underline{:}\, \tau_2\end{array}\end{array}}{\underline{\langle} \downarrow^{\tau_1} \overline{\mathrm{p}}_1(v), \downarrow^{\tau_2} \overline{\mathrm{p}}_2(v) \underline{\rangle} \,\underline{:}\, \tau_1 \times \tau_2}\,(\times I)$$

$\tau \equiv \tau_1 \to \tau_2$ For π given by assumption and the $\vdots$'s given by I.H. we have:

$$\dfrac{\dfrac{\pi \qquad\qquad [x\underline{:}\tau_1]_k \\ \qquad\qquad\qquad\vdots \\ v\bar{:}\tau_1 \to \tau_2 \quad \uparrow_{\tau_1} x\bar{:}\tau_1}{v\,\overline{@}\uparrow_{\tau_1} x\bar{:}\tau_2}(\to E) \\ \qquad\qquad\vdots \\ \downarrow^{\tau_2}(v\,\overline{@}\uparrow_{\tau_1} x)\underline{:}\tau_2}{\underline{\lambda}x.\downarrow^{\tau_2}(v\,\overline{@}\uparrow_{\tau_1} x)\underline{:}\tau_1 \to \tau_2}(\to I)_k$$

□

By the above we immediately obtain:

Corollary 4 *For any type, τ, we have that:*

$$\downarrow^{\tau} : \overline{\Lambda}^{\tau}_{II} \to \underline{\Lambda}^{\tau}_{II}$$
$$\uparrow_{\tau} : \underline{\Lambda}^{\tau}_{II} \to \overline{\Lambda}^{\tau}_{II}$$

are well-defined functions.

This method will be applied to terms in Λ^{τ} by considering them as completely static terms of $\overline{\Lambda}^{\tau}_{\mathrm{II}}$, that is, we embed using overlining $\Lambda^{\tau} \hookrightarrow \overline{\Lambda}^{\tau}_{\mathrm{II}}$.

3.2.3 $\overline{\beta}$-reduction

Given η-expanded terms we will now look at their $\overline{\beta}$-normal forms which are well-defined by:

Theorem 13 (Termination) *For closed $v \in \Lambda^{\tau}$ we have that the $\overline{\beta}$-normal form of $\downarrow^{\tau} v$ uniquely exists and is reached by all $\overline{\beta}$-reduction strategies.*

Proof: Follows from Corollary 1 (Strong Normalization) and Proposition 2 (Unique Normal Forms) using Theorem 7 (Church-Rosser).

□

Therefore we will write $\mathrm{nf}_{\overline{\beta}}(\downarrow^{\tau} v)$ for the $\overline{\beta}$-normal forms. At this point we could be let to conclude that we have shown $v =_{\beta\eta} \mathrm{c}(\mathrm{nf}_{\overline{\beta}}(\downarrow^{\tau} v))$ in $\lambda^{\mathrm{Curry}}_{\to\times}$. But observe that even though the [hybrid] η-expansion was shown

to terminate[3] it actually does not collapse to the η-expansion of $\lambda^{\mathrm{Curry}}_{\to\times}$, as witnessed by:

$$\begin{array}{lll} \downarrow^{o\to o} \overline{\lambda} v.v & = & \underline{\lambda} x.(\overline{\lambda} v.v)\,\overline{@}\,x \\ & =_{\overline{\beta\to}} & \underline{\lambda} x.x \end{array}$$

whose collapse would be:

$$\begin{array}{lll} \lambda v.v & =_{\eta}{}^{\div} & \lambda x.(\lambda v.v)x \\ & =_{\beta\to} & \lambda x.x \end{array}$$

which is not allowed by our restricted notion of η-expansion. However, in [14, page 5], Di Cosmo says about the present form of restricted η-expansion that "no equality is lost" and in fact we are immediately seen to have[4]:

$$\begin{array}{lll} \lambda x.(\lambda y.M)\,x & =_{\beta\to} & \lambda x.M[x/y] \\ & =_{(\alpha)} & \lambda y.M \end{array}$$

$$\langle \mathrm{p}_1(\langle M_1,\, M_2\rangle),\, \mathrm{p}_2(\langle M_1,\, M_2\rangle)\rangle \quad =_{\beta\times} \quad \langle M_1,\, M_2\rangle$$

Hence $(\alpha)\beta\eta$-equality is preserved and in fact we do have:

Proposition 9 *For all $v \in \Lambda^{\tau}$:*

$$v =_{\beta\eta} c(\mathrm{nf}_{\overline{\beta}}(\downarrow^{\tau} v))$$

3.2.4 Normalization

We will now consider the format of $\mathrm{nf}_{\overline{\beta}}(\downarrow^{\tau} v)$. It turns out that they are also in normal form with respect to $\underline{\beta}$-reduction and $\underline{\eta}$-expansion. As the terms $\mathrm{nf}_{\overline{\beta}}(\downarrow^{\tau} v)$ by Theorem 8 are completely dynamic we can subsequently remove the underlinings and thus have a $\beta\eta$-normal $\lambda^{\mathrm{Curry}}_{\to\times}$-term which is $\beta\eta$-equal to the original $\lambda^{\mathrm{Curry}}_{\to\times}$-term. This means the we actually have obtained the unique long $\beta(\eta)$-normal of the original term.

Lemma 8 *$\mathrm{nf}_{\overline{\beta}}(\downarrow^{\tau} v)$ are in $\underline{\eta}$-normal form.*

Proof: By Corollary 2 it suffices to show that $\downarrow^{\tau} v$ contains no $\underline{\eta}$-redices. As the original term is completely static we need only show that $\downarrow$ cannot introduce an $\underline{\eta}$-redex.

[3] We actually still need to see that the result is in η-normal form.

[4] Remembering that $\cdot[\cdot/\cdot]$ is defined non-capturing.

$\downarrow^o$: Vacuously not an $\underline{\eta}$-redex.

$\downarrow^{\tau_1 \times \tau_2}$: We introduce a dynamic pair which by definition cannot be $\underline{\eta}$-expanded. Each component follows by I.H.

$\downarrow^{\tau_1 \to \tau_2}$: We introduce a dynamic abstraction which by definition cannot be $\underline{\eta}$-expanded. The body follows by I.H., however, we also need to prove that the variable cannot give rise to an $\underline{\eta}$-redex in either of the contexts it can end up in.

By a parallel induction we therefore prove that no argument to $\uparrow$ can be used in a context where it gives rise an $\underline{\eta}$-redex:

$\uparrow_o$: Vacuously not an $\underline{\eta}$-redex.

$\uparrow_{\tau_1 \times \tau_2}$: The dynamic term is being projected and can thus not be $\underline{\eta}$-expanded.

$\uparrow_{\tau_1 \to \tau_2}$: The dynamic term is being applied and can thus not be $\underline{\eta}$-expanded. The argument of the dynamic application is by I.H. not an $\underline{\eta}$-redex.

□

Lemma 9 *$\underline{nf}_{\beta}(\downarrow^{\tau} v)$ are in $\underline{\beta}$-normal form.*

Proof: Same setup as in Lemma 8. We need to show that no introduced dynamic destructor (projection and application) can give rise to a $\underline{\beta}$-redex. We have the following two cases:

$\uparrow_{\tau_1 \times \tau_2}$: We must show that e cannot be a dynamic pair. We see that $\uparrow_{\tau}$ is only ever applied to a dynamic variable, dynamic projections, and a dynamic application.

$\uparrow_{\tau_1 \to \tau_2}$: We must show that e cannot be a dynamic abstraction which we already have.

□

Theorem 14 (Normalization) *For closed $v \in \Lambda^{\tau}$ we have that the $\underline{nf}_{\beta}(\downarrow^{\tau} v)$ is completely dynamic and is in long $\underline{\beta(\eta)}$-normal form.*

Proof: By Lemma 8 and Lemma 9.

□

And Theorems 13 and 14 together gives correctness of the method which can be thus described: Embed $v \in \Lambda^{\tau}$ into $\overline{\Lambda}^{\tau}_{\Pi}$, coerce that term into $\underline{\Lambda}^{\tau}_{\Pi}$, and $\overline{\beta}$-reduce it to a complete dynamic term, which can then be projected back to Λ^{τ}. The result is the long $\beta(\eta)$-normal form of the original term.

3.3 Type-Directed Partial Evaluation

For a straightforward implementation of this method we have a few things to consider:

1. The implementing language must support two levels, s.t., $\overline{\beta}$-reduction is well-defined:
   (a) $\overline{\beta}$-reduction can be performed independently of dynamic context.
   (b) Base-typed objects can be passed both statically and dynamically.
2. The [hybrid of] η-expansion performing the two-level coercions must be well-defined, that is, the language must allow the definition of the type-indexed functions, $\uparrow$ $\downarrow$. Thus the programming language should allow a certain degree of type-dependent definitions[5]

We will separate the two levels by quasiquote and unquote in Scheme [10], such that, static terms are Scheme values and dynamic terms are symbols. We immediately see that the first part of the first item is fulfilled. With respect to the second part we observe that all dynamic base-typed terms are symbols and thus vacuously can be passed as static values. As for the static base-typed terms they are necessarily constants as Scheme handles binding by the environment model and not by the substitution model we have considered so far. For numerical, boolean etc. constants we have that they *evaluate "to themselves"; they need not be quoted* [10, p. 7 last par.], that is, static base typed constants and their representation are freely interchangeable. And all in all are both the conditions in the first item fulfilled.

The condition in the second item is by Theorem 12 seen not to cause any problems either and we only need to make sure that the language adheres to the technical results presented in the chapter so far.

Theorem 13 : As we restrict ourselves to the simply typed case we have termination.

Theorem 14 : We must show that we end up only with symbols and base-typed constants. The lemmas that lead to this result hinge on Theorem 8 which in turn follows directly from Lemma 6. That is, if we are able to prove that static base-typed terms are dynamic we are done. And this was done above.

[5] However, the method has supposedly also been implemented in ML by Filinski. The present author is unaware of the details.

A direct implementation wil therefore produce long $\beta(\eta)$-normal forms.

In Figure 3.2 we have transliterated Figure 3.1 to Scheme using quasiquote and unquote. We use *gensym!* to create fresh names. As we observed in the proof of Lemma 8 the only dynamic terms are the ones we introduce so, in fact, this method gives fresh variables. Notice that we cannot do the same for fresh static variables. However, as binding in Scheme is performed in the environment (cf. [10, Sect. 3.5 Storage model: *Variables ... denote locations*]) we can just use the same name every time and the different occurrences are separated in the environment.

```
(define (reify t v)
  (case-record t
    [(Base b)
     v]
    [(Prd t1 t2)
     `(cons ,(reify t1 (car v)) ,(reify t2 (cdr v)))]
    [(Fct dom cod)
     (let ([var (gensym! "x")])
       `(lambda (,var) ,(reify cod (v (reflect dom var)))))]))

(define (reflect t e)
  (case-record t
    [(Base b)
     e]
    [(Prd t1 t2)
     (cons (reflect t1 `(car ,e)) (reflect t2 `(cdr ,e)))]
    [(Fct dom cod)
     (lambda (v) (reflect cod `(,e ,(reify dom v))))]))
```

Figure 3.2: Type-Directed Partial Evaluation in Scheme

3.4 Earlier proofs

In this section we will review Berger & Schwichtenberg's work on this normalizer as published in [6, 7]. The following is very close to their presentation, however, a few typographical errors and inconsistencies have been

adjusted.

3.4.1 A semantical proof

In [7], Berger & Schwichtenberg gives a semantical correctness proof of the method in the case of $\lambda_{\rightarrow}^{\text{Church}}$ (not $\lambda_{\rightarrow\times}^{\text{Curry}}$) in a purely functional setting, that is, while constructing fresh variables. More precisely do they show that in a certain class of admissible λ-models it is possible to invert the evaluational functional. The models are required to contain syntactical material and indices. Their main Theorem is in three parts:

Inversion: For any object, "a" –given as the meaning of a closed term–, in an admissible model the meaning (valuation) of "a" inverted is "a".

Normalization and substitution: For any term r it is possible to come up with an environment binding all free variables to a self-evaluating form, such that, the inverse of the valuation of r is r's long $\beta(\eta)$-normal form.

Completeness: If two terms are equal in an admissible model they are provably equal in $\lambda_{\rightarrow}^{\text{Church}}$ with $\beta\eta$-equality.

The notion of admissible models is complex and we will simplify the requirements somewhat and carry out the given proofs for inversion respectively normalization and substitution in this simpler setting. The simplifications are the ones used in the article to obtain an efficient normalization algorithm [7, Section 5].

Consider λ-models of $\lambda_{\rightarrow}^{\text{Church}}$ consisting of a pre-structure, M, that for all types, ρ, σ, consists of sets M^{ρ}, mappings $A_{\rho,\sigma} : M^{\rho} \to \sigma \times M^{\rho} \to M^{\sigma}$, and congruent and extensional equivalence relations, $=_{\rho}$ for the $A_{\rho,\sigma}$, that is equivalence relations such that

$$\begin{array}{ll} (Cong) & a =_{\rho\to\sigma} a', b =_{\rho} b' \Rightarrow A_{\rho,\sigma}ab =_{\sigma} A_{\rho,\sigma}a'b' \\ (Ext) & \forall b \in M^{\rho}.A_{\rho,\sigma}ab =_{\sigma} A_{\rho,\sigma}a'b \Rightarrow a =_{\rho\to\sigma} a' \end{array}$$

At higher types the pre-structure is required to be defined by exponentiation: $M^{\rho\to\sigma} = (M^{\sigma})^{M^{\rho}}$, why application, $Ap_{\rho,\sigma}$, will be taken to be ordinary function application.

Together with the prestructure we have meaning-mappings, where ENV is type-preserving environments in the usual sense:

$$|\cdot|^{\rho}\cdot : \Lambda^{\rho} \times ENV \to M^{\rho}$$

These functions are straightforwardly defined.

We will also require that it at base type contains representations of λ-syntax and (0,1)-sequences (for indices) plus the following operations between syntax and representation of syntax in the model: $\mathrm{EMB}_\rho \in (M^o)^{\Lambda^\rho}$ and $\mathrm{EXTRACT}_\rho \in (\Lambda^\rho)^{M^o}$ for embedding and extracting terms to and from their representation. We will represent terms by their equivalence class up to α. This is accomplished by requiring the pre-structure to –at base type– contain representations of terms like:

$$\lambda k^{\{0,1\}^*}.\lambda x^\tau_k.x_k$$

That is, the indices will form the base of an á la De Bruijn naming of a terms bound variables. The extraction is performed such that no free variables become bound.

To be precise the pre-structure will at base type contain ordinary λ-syntax and λ-syntax parameterized by an index: $\Lambda, \Lambda^{\{0,1\}^*} \subseteq M^o$, as well as indices: $\{0,1\}^* = M^\iota \subseteq M^o$. Therefore we will have $\mathrm{EMB}_\rho \in M^{o \to o}$ (from syntax to α-class) and $\mathrm{EXTRACT}_\rho \in M^{(\iota \to o) \to o}$ (from index-parameterized syntax to syntax)[6].

We will use $\#$ as infix notation for application of [representations of] index-parameterized syntax, s.t.:

$$\mathrm{EMB}(r) \,\#\, \mathrm{EMB}(s) = \mathrm{EMB}(r\,s)$$

NB! Such a model exists [7].

$$\begin{array}{rcl}
\Phi_\tau & \in & M^{\tau \to \iota \to o} \\
\Phi_o f k & = & f \\
\Phi_{\rho \to \sigma} a k & = & \lambda x^\rho_k.\Phi_\sigma(a\,(\Psi_\rho \mathrm{EMB}(x_k)))(k * 0) \\
 & & \\
\Psi_\tau & \in & M^{o \to \tau} \\
\Psi_o f & = & f \\
\Psi_{\rho \to \sigma} f b & = & \Psi_\sigma f \,\#\, (\Phi_\rho b)
\end{array}$$

Figure 3.3: An inverse of the evaluation functional.

[6]The difference between the codomain of EMB and the domain of EXTRACT is unessential as we have $\Lambda, \Lambda^{\{0,1\}^*} \subseteq M^o$.

Define for type respecting substitutions, θ, the associated environment, η_θ, s.t.

$$\eta_\theta(y^\rho) \quad = \quad \Psi_\rho \mathrm{EMB}(\theta y^\rho)$$

Lemma 10 (Simplified Main Lemma.) *For every $\lambda_{\to\times}^{Church}$-term, r^ρ, in long $\beta(\eta)$-normal form and every type respecting substitution θ, we have:*

$$\Phi_\rho |\, r \,|\eta_\theta \quad = \quad EMB(\theta r)$$

Proof: (by induction on r).

$\rho \equiv o; r \equiv x^{\rho_1 \to \ldots \to \rho_n \to o}\, s_1^{\rho_1} \ldots s_n^{\rho_n}$:

$$\begin{array}{lll}
|\, x\, s_1 \ldots s_n \,|\eta_\theta & & \\
\quad = |\, x \,|\eta_\theta \,|\, s_1 \,|\eta_\theta \ldots |\, s_n \,|\eta_\theta & ; \text{ def of } |\cdot| \\
\quad = \eta_\theta(x) \,|\, s_1 \,|\eta_\theta \ldots |\, s_n \,|\eta_\theta & ; \text{ def of } |\, x \,| \\
\quad = (\Psi_{\rho_1 \to \ldots \to \rho_n \to o}(\mathrm{EMB}(\theta x))) \,|\, s_1 \,|\eta_\theta \ldots |\, s_n \,|\eta_\theta & ; \text{ def of } \eta_\theta \\
\quad = (\mathrm{EMB}(\theta x) \,\#\, \Phi_{\rho_1} |\, s_1 |\eta_\theta) \,\#\, \ldots \,\#\, \Phi_{\rho_n} |\, s_n \,|\eta_\theta & ; \text{ def of } \Psi \\
\quad = (\mathrm{EMB}(\theta x) \,\#\, \mathrm{EMB}(\theta s_1)) \,\#\, \ldots \,\#\, \mathrm{EMB}(\theta s_n) & ; \text{ IH} \\
\quad = \mathrm{EMB}((\theta x)\,(\theta s_1) \ldots (\theta s_n)) & ; \text{ def of } \# \\
\quad = \mathrm{EMB}(\theta r) & ; \text{ def of subst.}
\end{array}$$

$\rho \equiv \sigma \to \sigma'; r \equiv \lambda x^\sigma . s^{\sigma'}$: As we only consider terms up to α-equivalence we can assume $\theta x = x$ and x not free in θy for any $y \neq x$ free in s. We need to show that

$$\Phi_{\sigma \to \sigma'}(|\, \lambda x.s \,|\eta_\theta) = \mathrm{EMB}(\theta \lambda x.s)$$

By extensionality it suffices to show, that for all k, we have

$$\Phi_{\sigma \to \sigma'}(|\, \lambda x.s \,|\eta_\theta)k = \mathrm{EMB}(\theta \lambda x.s)k$$

So:

$$\begin{array}{lll}
\Phi_{\sigma \to \sigma'}(|\, \lambda x.s \,|\eta_\theta)k & & \\
\quad = \lambda x_k^\rho . \Phi_{\sigma'}((|\, \lambda x.s \,|\eta_\theta)\,(\Psi_\sigma \mathrm{EMB}(x_k)))(k * 0) & ; \text{ def of } \Phi_{\sigma \to \sigma'} \\
\quad = \lambda x_k^\rho . \Phi_{\sigma'}(|\, s \,|\eta_\theta[x \mapsto \Psi_\sigma \mathrm{EMB}(x_k)])(k * 0) & ; \text{ def of } |\, \lambda x.s \,| \\
\quad = \lambda x_k^\rho . \Phi_{\sigma'}(|\, s \,|\eta_{\theta[x \mapsto x_k]})(k * 0) & ; \text{ def of } \eta_\theta \\
\quad = \lambda x_k^\rho . (\mathrm{EMB}(\theta[x \mapsto x_k]s))(k * 0) & ; \text{ IH} \\
\quad = \mathrm{EMB}(\lambda x.\theta s)k & ; \text{ def of EMB} \\
\quad = \mathrm{EMB}(\theta \lambda x.s)k & ; \text{ ass on } \theta
\end{array}$$

□

Theorem 15 (Inversion) *For closed $r \in \Lambda^\tau$, we have for suitable k and any η:*

$$|(\Phi_\tau(|r|\eta)k)| = |r|$$

That is, by abuse of notation:

$$|\cdot|^\tau \circ \Phi_\tau = Id_{M^\tau \restriction Image_{\|}(\Lambda^0)}$$

Proof:

$|(\Phi_\tau(|r|\eta)k)|$
$= |(\mathrm{EMB}(r)k)|$; Lemma 10 as r closed
$= |r|$; as $r =_\alpha (\mathrm{EMB}(r)k)$

□

Theorem 16 (Normalization and substitution) *For every term r^ρ, with long $\beta(\eta)$-normal form r^* and every substitution θ of at least the free variables of r, we have, for suitable k:*

$$\Phi_\rho(|r|\eta_\theta)k =_\alpha \theta r^*$$

Proof:

$\Phi_\rho(|r|\eta_\theta)k$
$= \Phi_\rho(|r^*|\eta_\theta)k$; by extensionality
$= \mathrm{EMB}(\theta r^*)k$; by Lemma 10
$=_\alpha \theta r^*$; def of EMB

□

3.4.2 A strictly formal proof

In [6], Berger derives the inverse of the evaluation functional for the typed λ-calculus without product types[7]. He applies Kreisel's modified realizability interpretation for intuitionistic logic to a constructive proof of strong normalization:

$$\exists s.N_\tau(r,s)$$

Where $r, s \in \Lambda^\tau$ and N states that s is the long $\beta(\eta)$-normal form of r. The proof uses Tait's computability predicates, Strong Computability, and

[7]The proof is, however, expected to carry through straightforwardly.

derives Φ from a proof that Strong Computability implies Strong Normalization. In turn Ψ is derived from a proof stating that Strong Argument Application[8] implies Strong Computability. This is seen to go well in thread with our approach in which we take a static term and via $\downarrow, \Phi$ comes up with the dynamic skeleton of its normal form. The body of the skeleton is constructed by using $\uparrow, \Psi$. The proof uses induction over types on the meta-level and the result is thus a type-indexed family of functions.

Modified realizability as opposed to realizability addresses the issue that parts of a proof can be deemed computationally irrelevant. They can henceforth be annotated to not cause the introduction of witnessing syntax.

The proof is therefore in two layers, namely a computationally relevant part and a computationally irrelevant part. The distinction between the two parts greatly resembles our treatment. We saw that the two-level coercers where not definable in $\lambda_{\rightarrow\times}^{\text{Curry}}$ and hence we had to give an independent correctness proof of these, Theorem 12. This corresponds to half of the computationally irrelevant part of Berger's proof. More precisely are the considered predicates universally quantified with terms (that is, *for all terms* ...) and this gives rise to a λ-abstraction in the proof term which is annotated as irrelevant. The other half of the computationally irrelevant part is the generation of fresh variables. We simply require the variables to be fresh while he takes the approach layed out in the preceding chapter and leaves the parts where they are actually used to construct fresh variables as computationally irrelevant. The actual abstraction of the De Bruijn indices is taken to be computationally relevant which corresponds to the result of using `gensym!` in our case. How the produced names become different is irrelevant in our treatment.

[8] An application of a variable to a number of terms has a counter part where all argument terms are in normal form. This of course addresses the shape of Huet's long $\beta(\eta)$-normal forms.

Chapter 4

Practical issues for TDPE; Compiler Generation

We will investigate the practical implications of the work in Chapter 3. We saw that the proof normalizer can be implemented in Scheme and there are a number of issues raised from that. Most notably, we have that Scheme is latently typed[1], which allows us to define much more than the proof terms of Nm. We will investigate to which extent we can take advantage of that. We will also make a few comments preparing for Chapter 5, in which we have included [23] in its entirety.

We will happily refer to simply typed values in the context of Scheme which, as said, is latently typed. One aspect of simple types is that terms of fixed based types can only be used as strictly being of that type. This is seen not to be compatible with the notion of latent types. When referring to the full set of Scheme values –which we have not done so far– we will not consider fixed base types. Rather, we simply consider all base types to be type variables. We can still put orderings on base types and treat terms accordingly, but this only makes sense on the representation level, and not in the actual implementation.

We will thorughout this chapter consider the untyped λ-calculus, $\lambda_{\rightarrow\times}$, given as the pre-terms of $\lambda_{\rightarrow\times}^{\text{Curry}}$ with β-reduction straightforwardly defined:

$$\lambda x.M \quad \rightarrow_\beta \quad M[N/x]$$

[1] E.g., terms that are applied must be functions, but there are no coherence requirements on their domain type and the type of the arguments.

$$\mathrm{p}_i(\langle M_1,\, M_2\rangle) \quad \rightarrow_\beta \quad M_i$$

4.1 Eventually well-typed terms

In Chapter 3, we saw that to prove the normalizing effect of our approach (Theorem 14) we needed two results about $\lambda_{\rightarrow\times}^{\text{Curry-II}}$:

- The result of $\overline{\beta}$-reduction is purely dynamic, Theorem 8. This theorem hinges on Lemma 6, which proves that the base typed β-normal forms of closed terms in $\lambda_{\rightarrow\times}^{\text{Curry}}$ are constants.
- No dynamic constructs can be introduced by $\overline{\beta}$-reduction, Corollary 2.

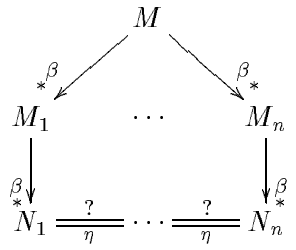
The actual normalization result, Theorem 14, then followed by some easy considerations on the two-level coercers, Figure 3.1. As long as we adhere to these two conditions (of which the former also prescribes termination), we see that the method will carry through. We observe that the central definition is the following:

Definition 17 (Eventually Well-typed) *We will say that a $\lambda_{\rightarrow\times}$-term, e, is eventually well-typed at type τ, $e \in EW_\tau$, if*

$$\exists M \in \Lambda^\tau . e \rightarrow_\beta^* M$$

Observe that even though we call a term eventually well-typed, it can have reduction sequences which never lead to a simply-typeable term.

In the case of $\lambda_{\rightarrow\times}^{\text{Curry}}$, we saw that a result was obtained when considering a term's principal type and all instatiations of it. All of the results were seen to be η-equal in the unrestricted sense. In the present situation we cannot *a priori* expect that to hold. More specifically, the notion of a principal type for which we can obtain a result, is not necessarily well-defined. So far we have not been able to find conclusive evidence for either case, and we will thus take the most general approach. To give an impression of the difficulties we will look at the following situation with M an $\lambda_{\rightarrow\times}$-term:



Where $M_i \in \Lambda^{\tau_i}$ are simply typeable terms, all of which have a β-normal form, $N_i \in \Lambda^{\tau_i}$. However, at present it is not known to the author whether or not $\rightarrow_\beta$-reduction on $\lambda_{\rightarrow\times}$ enjoys UNF, which would allow us to conclude $\forall i,j.N_i =_\eta N_j$, again with unstricted η-equality.

Evaluation in Scheme corresponds to left-most, inner-most β-reduction to weak head normal form. This reduction strategy is deterministic, albeit unspecified at places and we see that there can be at most one M_i. When it exists it is always reached. In the process of performing the two-level coercions, Figure 3.1, we bring all static terms to base type by a controlled [hybrid of] η-expansion. The process was seen to depend only on the type according to which it was performed, Theorem 12. At base type, weak head normal forms are seen to coincide with β-normal forms. At higher types the difference is only in the function case. Hence we can apply the technique succesfully as long as considered Scheme values, v, of type, $\tau = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow o$, gives rise to termination of the evaluation of the following base typed Scheme value:

$$v\, \eta_{\tau_1}(x_1) \ldots \eta_{\tau_n}(x_n)$$

with $\eta(\cdot)$ defined as follows for any value, w:

$$\eta_\tau(w) = \begin{cases} w & \tau = o \\ \langle \eta_{\tau_1}(\mathrm{p}_1(w)),\, \eta_{\tau_2}(\mathrm{p}_2(w)) \rangle & \tau = \tau_1 \times \tau_2 \\ \lambda v.\eta_{\tau_2}(w\, \eta_{\tau_1}(v)) & \tau = \tau_1 \rightarrow \tau_2 \end{cases}$$

We saw that this way of deconstructing function values does not lead to non-terminating behaviour which would not have occured had we considered full left-most, inner-most reduction, $\rightarrow_{li}$, Theorems 8, 12. Termination of the above evaluation is thus seen to coincide with our previous notion of eventually well-typed terms for li-reduction:

Proposition 10 *By using the implementation given in Figure 3.2 we can normalize all Scheme values which are eventually well-typed by* li*-reduction, EW^{li}.*

It still remains to prove more constructive results than the above. On the other hand, it is a rather strong result, as it says that we can produce a representation of the normal form of any Scheme value which, when abstracting from weak head normal forms, always terminates in a simply typed value[2].

4.2 Compiler Generation

We have included [23] as the next chapter. In it we use a more general type-directed partial evaluator than the one presented here [19]. It allows the user to annotate functions to be strict. The reason is that if we in a call-by-value regime use a normalized term, as opposed to the original term, computation can end up being dublicated [19]. Thus the normal form of:

$$\lambda f.\lambda g.\lambda x.(\lambda y.(f\ y\ y))\,(g\ x)$$

is

$$\lambda f.\lambda g.\lambda x.(f\,(g\ x)\,(g\ x))$$

Where $g\ x$ is going to get evaluated twice. That is not a problem in our pure treatment. However, as we are considering actual applications this is a real concern and must be addressed. We will refer to [19] for further details.

If we were to perform provably correct compiler generation we would have to modify this approach slightly, as the more general type-directed partial evaluator has not yet been proven correct. We could, however, also preclude the above situation by requiring that strict functions are evaluated exactly once. The standard way of obtaining this is by using a continuation passing semantics [62]. In [35] Harrison and Kamin take exactly that approach to reproduce Reynold's results for a functor-category semantics for Algol [63].

[2] Remember that simple types in this case mean that all base types are type variables that can be unified with each other.

Chapter 5

Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation

Olivier Danvy René Vestergaard
BRICS[1]
Computer Science Department
Aarhus University [2]

5.1 Abstract

We illustrate a simple and effective solution to semantics-based compiling. Our solution is based on "type-directed partial evaluation", and

- our compiler generator is expressed in a few lines, and is efficient;
- its input is a well-typed, purely functional definitional interpreter in the style of denotational semantics;
- the output of the generated compiler is effectively three-address code, in the fashion and efficiency of the Dragon Book;

[1] Basic Research in Computer Science,
Centre of the Danish National Research Foundation.
[2] Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: {danvy, jrvest}@brics.dk

- the generated compiler processes several hundred lines of source code per second.

The source language considered in this case study is imperative, block-structured, higher-order, call-by-value, allows subtyping, and obeys stack discipline. It is bigger than what is usually reported in the literature on semantics-based compiling and partial evaluation.

Our compiling technique uses the first Futamura projection, *i.e.*, we compile programs by specializing a definitional interpreter with respect to the program. Specialization is carried out using type-directed partial evaluation, which is a mild version of partial evaluation akin to λ-calculus normalization.

Our definitional interpreter follows the format of denotational semantics, with a clear separation between semantic algebras, and valuation functions. It is thus a completely straightforward stack-based interpreter in direct style, which requires no clever staging technique (currying, continuations, binding-time improvements, *etc.*), and does not rely on any other framework (attribute grammars, annotations, *etc.*) than the typed λ-calculus. In particular, it uses no other program analysis than traditional type inference.

The overall simplicity and effectiveness of the approach has encouraged us to write this paper, to illustrate this genuine solution to denotational semantics-directed compilation, in the spirit of Scott and Strachey. Our conclusion is that λ-calculus normalization suffices for compiling by specializing an interpreter.

5.2 Introduction

5.2.1 Denotational semantics and semantics-implementation systems

Twenty years ago, when denotational semantics was developed [50, 69], there were high hopes for it to be used to specify most, if not all programming languages. When Mosses developed his Semantics Implementation System [53], it was with the explicit goal of generating compilers from denotational specifications.

Time passed, and these hopes did not materialize as concretely as was wished. Other semantic frameworks are used today, and other associated semantics-implementation systems as well. Two explanations could be that (1) domains proved to be an interesting area of research *per se*, and they are studied today quite independently of programming-language design and specification; and (2) the λ-notation of denotational semantics was deemed

untamable — indeed writing a denotational specification can be compared to writing a program in a module-less, lazy functional language without automatic type-checker.

As for semantics-implementation systems, there have been many [1, 28, 41, 48, 53, 57, 58, 61, 65, 67, 71, 72, 73], and they were all quite complicated. (Note: this list of references is by no means exhaustive. It is merely meant to be indicative.)

5.2.2 Partial evaluation

For a while, partial evaluation has held some promise for compiling and compiler generation, through the Futamura projections [42, 43]. The first Futamura projection states that specializing a definitional interpreter with respect to a source program "compiles" the source program from the defined language to the defining language [62]. This idea has been applied to a variety of interpreters and programming-language paradigms, as reported in the literature [12, 42].

One of the biggest and most successful applications is Jørgensen's BAWL, which produces code that is competitive with commercially available systems, given a good Scheme implementation [44]. The problem with partial evaluation, however, is the same as for most semantics-implementation system: it requires an expert to use it successfully.

5.2.3 Type-directed partial evaluation

Recently, the first author has developed an alternative approach to partial evaluation which is better adapted to specializing interpreters and strikingly simpler [19]. The approach is type-directed and amounts to normalizing a closed, well-typed program, given its type (see Figure 5.1). The approach has been illustrated on a toy programming language, by transliterating a denotational specification into a definitional interpreter, making this term closed by abstracting all the (run-time) semantics operators, applying it to the source program, and normalizing the result.

Type-directed partial evaluation thus merely requires the user to write a purely functional, well-typed definitional interpreter, to close it by abstracting its semantic operators, and to provide its type. The type is obtained for free, using ML or Haskell. No annotations or binding-time improvements are needed.

The point of the experiment with a toy programming language [19] is that it can be carried out at all. The point of this paper is to show that the

approach scales up to a non-trivial language.

$$
\begin{array}{rcl}
t \in \text{Type} & ::= & b \mid t_1 \times t_2 \mid t_1 \to t_2 \\
\\
\text{reify} & = & \lambda t.\lambda v.{\downarrow^t}\, v \\
{\downarrow^b}\, v & = & v \\
{\downarrow^{t_1 \times t_2}}\, v & = & \underline{\text{pair}}({\downarrow^{t_1}}\, \overline{\text{fst}}\, v, {\downarrow^{t_2}}\, \overline{\text{snd}}\, v) \\
{\downarrow^{t_1 \to t_2}}\, v & = & \underline{\lambda} x_1.{\downarrow^{t_2}}\, (v\, \overline{@}\, ({\uparrow_{t_1}}\, x_1)) \\
 & & \text{where } x_1 \text{ is fresh.} \\
\\
\text{reflect} & = & \lambda t.\lambda e.{\uparrow_t}\, e \\
{\uparrow_b}\, e & = & e \\
{\uparrow_{t_1 \times t_2}}\, e & = & \overline{\text{pair}}({\uparrow_{t_1}}\, \underline{\text{fst}}\, e, {\uparrow_{t_2}}\, \underline{\text{snd}}\, e) \\
{\uparrow_{t_1 \to t_2}}\, e & = & \overline{\lambda} v_1.{\uparrow_{t_2}}\, (e\, \underline{@}\, ({\downarrow^{t_1}}\, v_1)) \\
\\
\text{residualize} & = & \text{statically-reduce} \circ \text{reify}
\end{array}
$$

The down arrow is read *reify*: it maps a static value and its type into a two-level λ-term [55] that statically reduces to the dynamic counterpart of this static value. Conversely, the up arrow is read *reflect*: it maps a dynamic expression into a two-level λ-term representing the static counterpart of this dynamic expression.

In residualize, reify (resp. reflect) is applied to types occurring positively (resp. negatively) in the source type.

N.B. In practice, residual let expressions need to be inserted to maintain the order of execution, *à la* Similix [8]. This feature makes it possible to specialize direct-style programs with the same advantages one gets when specializing continuation-passing programs, and is described elsewhere [18].

Figure 5.1: Type-directed partial evaluation (a.k.a. residualization)

5.2.4 Disclaimer

We are not suggesting that well-typed, compositional and purely definitional interpreters written in the fashion of denotational semantics are the way to go always. This method of definition faces the same problems as denotational semantics. For example, it cannot scale up easily to truly large languages, which has motivated *e.g.*, the development of Action Semantics

[54].

Our point is that given such a definitional interpreter, type-directed partial evaluation provides a remarkably simple and effective solution to semantics-based compiling.

5.2.5 This paper

We consider an imperative language with block structure, higher-order and mutually recursive procedures, call-by-value, and that allows subtyping. We write a direct-style definitional interpreter that is stack-based, reflecting the traditional call/return strategy of Algol 60 [60], but is otherwise completely straightforward.[3] We specialize it and obtain three-address code that is comparable to what can be expected from a compiler implemented *e.g.*, according to the Dragon Book [1].

This experiment is significant for the following reasons:

Semantics-implementation systems: Our source language is realistic. Our compiler generator is extremely simple (it is displayed in Figure 5.1). Our language specification is naturally in direct style and requires no staging transformations [45]. Our compiler is efficient (several hundred lines per second). Our target code is reasonable.

We are not aware of other semantics-implementation systems that yield similar target code with a similar efficiency. In any case, our primary goal is not efficiency. Our point here is that the problem of semantics-based compiling can be stated in such a way that a solution exists that is extremely simple and yields reasonable results.

Partial evaluation: Similar results can be obtained using traditional partial evaluation, but not as simply and not as efficiently.

An online partial evaluator incurs a significant interpretive overhead. An offline partial evaluator necessitates a binding-time analysis, and then either incurs interpretive overhead by direct specialization, or requires its user to generate a generating extension. Furthermore, an offline partial evaluator usually requires binding-time improvements, which are an art in themselves [42, Chapter 12].

[3] Being stack-based is of course not a requirement. An unstructured store would also need to be threaded throughout, but its implementation would require a garbage collector [50].

It is our experience that in a way, partial evaluators today are too powerful: our present experiment shows that λ-calculus normalization is enough for compiling by interpreter specialization.

5.2.6 Overview

Section 5.3 presents the essence of semantics-based compiling by type-directed partial evaluation. Section 5.4 briefly outlines our source programming language. Based on this description, it is simple to transcribe its denotational specification into a definitional interpreter [62]. We describe one such interpreter in Section 5.5, and display the outline of it in Figure 5.6. It is well-typed. Given its type and a source program such as the one in Figure 5.3, we can residualize the application of the interpreter to the source program and obtain a residual program such as the one in Figure 5.4. Each residual program is a specialized version of the definitional interpreter and appears as three-address code. We use the syntax of Scheme to represent this three-address code, but it is trivial to map it into assembly code.

5.3 The essence of semantics-based compiling by type-directed partial evaluation

Our point here is that to compile programs by interpreter specialization, λ-calculus normalization is enough.

Interpreter specialization: the first Futamura projection states that specializing an interpreter with respect to a program (*i.e.*, performing all the interpreter operations that only depend on the program, but not on its input) amounts to compiling this program into the defining language of the interpreter [42]. Along with the second and third Futamura projections, this property has been instrumental to the renaissance of partial evaluation in the 80's [43].

Normalization is enough: the technique of type-directed partial evaluation is otherwise used to normalize simply typed programs extracted from proofs. There, it is referred to as "normalization by evaluation" [4, 7]. Type-directed partial evaluation slightly differs in that to make it fit call-by-value, we insert let expressions naming residual expressions to avoid code and computation duplication.

Throughout, we consider Schmidt-style denotational specifications [65], *i.e.*, specifications with a clear separation between semantic algebras, and val-

```
(define-record (Stop))
(define-record (Sequence i c))
(define-record (Zero))
(define-record (One))

(define meaning
  (lambda (c)
    (lambda (f g)
      (letrec ([meaning-command
                (lambda (c)
                  (case-record c
                    [(Stop) (lambda (s) s)]
                    [(Sequence i c)
                     (lambda (s)
                       ((meaning-command c)
                        ((meaning-instruction i) s)))]))]
               [meaning-instruction
                (lambda (i)
                  (case-record i
                    [(Zero) f]
                    [(One) g]))])
        (meaning-command c)))))

(define-base-type sto "s")
(define-compound-type one (sto -!> sto) "f" alias)
(define-compound-type two (sto -!> sto) "g" alias)
(define-compound-type denotation-type ((one two) => sto -!> sto))
```

Figure 5.2: Microscopic definitional interpreter

uation functions. We normalize the result of applying the main valuation function to a program, given the type of its denotation. The result is a combination of semantic-algebra operations, sequentialized with let expressions, and thus corresponding to three-address code [1].

Let us illustrate this point with one extremely simple example. We consider a microscopic imperative language and its definitional interpreter (Figure 5.2). Applying this interpreter to a source program yields a functional value. Normalizing it yields a textual representation of the dynamic semantics of the source program, *i.e.*, its compiled version: a sequence of semantic-algebra operations.

5.3.1 Syntax

The following BNF specifies our microscopic imperative language. A program consists of a sequence of zeros and ones terminated by a stop.

$$\begin{array}{rcl} p \in \text{Program} & ::= & c \\ c \in \text{Command} & ::= & \text{stop} \mid i\ ;\ c \\ i \in \text{Instruction} & ::= & \text{zero} \mid \text{one} \end{array}$$

5.3.2 Semantic algebra

The valuation functions uses a semantic algebra consisting of a store and two store-transforming operations f and g.

5.3.3 Valuation functions

We respectively interpret zero and one with f and g.

$$\begin{array}{rcl} \mathcal{C} : \text{Command} & \to & \text{Store} \to \text{Store} \\ \mathcal{C}[\![\text{stop}]\!] & = & \lambda\sigma.\sigma \\ \mathcal{C}[\![i\ ;\ c]\!] & = & \lambda\sigma.\mathcal{C}[\![c]\!](\mathcal{I}[\![i]\!]\sigma) \end{array} \qquad \begin{array}{rcl} \mathcal{I} : \text{Instruction} & \to & \text{Store} \to \text{Store} \\ \mathcal{I}[\![\text{zero}]\!] & = & f \\ \mathcal{I}[\![\text{one}]\!] & = & g \end{array}$$

5.3.4 Compiling by normalization

Figure 5.2 displays a direct-style definitional interpreter, expressed in Scheme. This interpreter is curried: when applied to a source program, it returns a higher-order procedure expecting f and g and returning a store transformer. We have parameterized it with f and g merely for convenience (precisely: to make it a closed term, *i.e.*, a term without free variables that can be characterized with its type).

The figure also shows the definition of the abstract syntax as records, and the definition of the types of the store, of f, and g. The function type of f and g is annotated with "!" to indicate that they operate on a single-threaded value, and thus that their application should be named with a let expression [18].

Let `p` denote the source program "zero; one; zero; one; stop". In the following Scheme session, we residualize the application of the definitional interpreter with respect to the type of the denotation. The result is the text of the corresponding normal form, *i.e.*, of the dynamic semantics of `p`.

```
> (load "tdpe.scm")
> (load "mic-def-int.scm")
```

```
> (residualize (meaning p) 'denotation-type)
(lambda (f g)
  (lambda (s0)
    (let* ([s1 (f s0)]
           [s2 (g s1)]
           [s3 (f s2)])
      (g s3))))
```

This residual Scheme program is the specialized version of the interpreter of Figure 5.2 with respect to the source program `p`. It performs the run-time actions of `p`, sequentially. The interpretive overhead is gone.

5.3.5 Assessment

This microscopic example is significant for at least three reasons.

This is semantics-based compilation: we have compiled a program based on the semantics of a programming language.

This is partial evaluation: we have specialized a definitional interpreter with respect to a source program.

This is normalization: all we have used is a λ-calculus normalizer.

Furthermore, the target language of the normalizer (a flat sequence of let expressions) has the same structure as three-address code as described in the Dragon book [1]. Translating residual programs into assembly language therefore does not amount to writing a compiler for the λ-calculus, but more simply to writing the back-end of a standard compiler. The method is thus not a step sideways, but an actual step forward in the process of compiling a source program. In essence, it makes it possible to derive the front-end of a compiler from a definitional interpreter.

Methodologically, our type-directed partial evaluator is simpler than a traditional semantics-implementation system or a standard partial evaluator for three more reasons:

1. Normalization is simpler than partial evaluation.
2. Type-directed partial evaluation does not require one to re-implement the λ-calculus: it relies on an existing implementation (presently Scheme, but ML or Haskell would do just as well).

3. There is no need to massage (a.k.a. stage [45]) the definitional interpreter to derive [the front-end of] the compiler. It is enough to follow the format of denotational semantics: semantic algebras, and valuation functions.

We come back to these issues in Section 5.7.

The example language of this section is deliberately microscopic, but it captures the essence of our case study. Examples of semantics-based compiling by type-directed partial evaluation have been published for Paulson's Tiny language [19, 18], which is an imperative while-language traditional in the semantics community [41]. In this paper, we explore semantics-based compiling with a more substantial programming language, in an effort to explore the applicability of type-directed partial evaluation.

5.4 A block-structured procedural language with subtyping

The following programming language is deliberately reminiscent of Reynolds's idealized Algol [63], although it uses call-by-value. We briefly present it here. A full description is available in a companion technical report [24].

5.4.1 Abstract Syntax

The language is imperative: its basic syntactic units are commands. It is block-structured: any command can have local declarations. It is procedural: commands can be abstracted and parameterized. It is higher-order: procedures can be passed as arguments (though not returned, to enable stack discipline for activation records) to other procedures. Finally it is typed and supports subtyping in that an integer value can be assigned to a real variable, a procedure expecting a real can be passed instead of a procedure expecting an integer, *etc.*

$p, \langle pgm \rangle$	$\in$	Pgm	—domain of programs
$c, \langle cmd \rangle$	$\in$	Cmd	—domain of commands
$e, \langle exp \rangle$	$\in$	Exp	—domain of expressions
$i, \langle ide \rangle$	$\in$	Ide	—domain of identifiers
$d, \langle decl \rangle$	$\in$	$Decl$	—domain of declarations
$t, \langle type \rangle$	$\in$	$Type$	—domain of types
$o, \langle btype \rangle$	$\in$	$BType$	—domain of base types

$$
\begin{array}{rcl}
\langle pgm \rangle & ::= & \langle cmd \rangle \\
\langle decl \rangle & ::= & \textbf{Var}\ \langle ide \rangle : \langle btype \rangle\ =\ \langle exp \rangle \\
& | & \textbf{Proc}\ \langle ide \rangle\ (\langle ide \rangle{:}\langle type \rangle, \ldots, \langle ide \rangle{:}\langle type \rangle)\ =\ \langle cmd \rangle \\
\langle cmd \rangle & ::= & \textbf{skip} \mid \textbf{write}\ \langle exp \rangle \mid \textbf{read}\ \langle ide \rangle \mid \langle cmd \rangle\ ;\ \langle cmd \rangle \\
& | & \langle ide \rangle := \langle exp \rangle \mid \textbf{if}\ \langle exp \rangle\ \textbf{then}\ \langle cmd \rangle\ \textbf{else}\ \langle cmd \rangle \\
& | & \textbf{while}\ \langle exp \rangle\ \textbf{do}\ \langle cmd \rangle \mid \textbf{call}\ \langle ide \rangle\ (\langle exp \rangle, \ldots, \langle exp \rangle) \\
& | & \textbf{block}\ (\langle decl \rangle, \ldots, \langle decl \rangle)\ \textbf{in}\ \langle cmd \rangle \\
\langle exp \rangle & ::= & \langle lit \rangle \mid \langle ide \rangle \mid \langle exp \rangle\ \langle op \rangle\ \langle exp \rangle \\
\langle lit \rangle & ::= & \langle bool \rangle \mid \langle int \rangle \mid \langle real \rangle \\
\langle op \rangle & ::= & + \mid \times \mid - \mid < \mid = \mid \textbf{and} \mid \textbf{or} \\
\langle btype \rangle & ::= & \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} \\
\langle type \rangle & ::= & \langle btype \rangle \mid \textbf{Proc}\ (\langle type \rangle, \ldots, \langle type \rangle)
\end{array}
$$

5.4.2 Semantics

The language is block structured, procedural, and higher-order. Since it is also typed, we define the corresponding domain of values inductively, following its typing structure. Our implementation is stack-based, and so we want to pass parameters on top of the stack. We thus define the denotation of procedures as a store transformer, whose elements are functions accepting a stack whose top frame contains parameters of an appropriate type. We furthermore index the denotation of a procedure with its free variables.

The store can hold untagged booleans, integers, reals, and procedures. It is accessed with typed operators. We define expressible values (the result of evaluating an expression) to be type-annotated storable values. A denotable value (held in the environment) is a reference to the stack, paired with the type of the corresponding stored value.

The stack holds a sequence of activation records which are statically (for the environment) and dynamically (for the continuation) linked via base pointers, as is traditional in Algol-like languages [1, 60]. An activation record is pushed at each procedure call, and popped at each procedure return. A block (of bindings) is pushed whenever we enter the scope of a declaration, and popped upon exit of this scope. Each block extends the current activation record. Procedures are call-by-value: they are passed the (storable) values of their arguments on the stack [1, 60]. The global store pairs a push-down stack and i/o channels. It is addressed through type-indexed operators.

The language is statically typed: any type mismatch yields an error. In addition, subtyping is allowed: any context expecting a value of type t (*i.e.*, assignments and parameter passing) accepts an expressible value whose

type is a subtype of t. Our coercions at higher type simulate intermediate procedures that coerce their arguments as required by their types [63]. Along the same lines, the type-directed partial evaluator of Figure 5.1 is simply a coercion from static to dynamic [19].

We want the language to obey a stack discipline. Stack discipline can only be broken when values may outlive the scope of their free variables. This can only occur when an identifier i of higher type is updated with the value of an identifier j which was declared in the scope of i. We thus specify that at higher type, i must be local to j.

5.5 A definitional interpreter and its specialization

We have transcribed the denotational specification of Section 5.4 into a definitional interpreter which is compositional and purely functional [62, 68]. We make the interpreter a closed term by abstracting all its free variables at the outset (*i.e.*, the semantic operators fix, test, *etc.*). Figure 5.6 displays the skeleton of the interpreter.

As in Section 5.3, we can then residualize the application of the definitional interpreter to any source program, with respect to the codomain of the definitional interpreter, *i.e.*, with respect to the type of the meaning of a source program. The result is a textual representation of the dynamic semantics of the source program, *i.e.*, of its step-by-step execution without interpretive overhead.

Figure 5.3 displays such a source program. Figures 5.4 and 5.5 display the resulting target program, unedited (except for the comments).

```
block Var x : Int = 100
      Proc print (y : Real) = write y
      Proc p (q : Proc(Int), y : Int) = call q (x + y)
in call p (print, 4)
```

Figure 5.3: Sample source program

The source program of Figure 5.3 illustrates block structure, non-local variables, higher-order procedures, and subtyping. The body of Procedure p refers to the two parameters of p and also to the global integer-typed variable x. Procedure p expects an integer-expecting procedure and an integer. It

is passed the real-expecting procedure *print*, which is legal in the present subtyping discipline. Procedure *print* needs to be coerced into an integer-expecting procedure, which is then passed to Procedure p.

```
(lambda (fix test int-to-real conj disj eq-bool read-int read-real
         read-bool write-int write-real write-bool add-int mul-int
         sub-int less-int eq-int add-real mul-real sub-real less-real
         eq-real push-int push-real push-bool push-proc push-func
         push-al push-base-pointer pop-block update-base-pointer
         pop-frame lookup-int lookup-real lookup-bool lookup-proc
         lookup-func update-int update-real update-bool update-proc
         update-func current-al lookup-al update-al)
  (lambda (s)
    (let* ([s   (push-int 100 s)]  ;;; decl. of x
           [a0  (current-al s)]
           [s   (push-proc
                  (lambda (s)      ;;; decl. of print (code pointer)
                    (let ([r1 (lookup-real 0 0 s)])
                      (write-real s r1)))
                  s)]
           [s   (push-al a0 s)]    ;;; decl. of print (access link)
           [a2  (current-al s)]
           ...
```

Figure 5.4: Sample target program (specialized version of Fig. 5.6 with respect to Fig. 5.3)

The residual program of Figure 5.4 is a specialized version of the definitional interpreter of Figure 5.6 with respect to the source program of Figure 5.3. It is a flat Scheme program threading the store throughout and reflecting the step-by-step execution of the source program. The static semantics of the source program, *i.e.*, all the interpretive steps that only depend on the text of the source program, has been processed at partial-evaluation time: all the location offsets are solved and all primitive operations are properly typed. The coercion, in particular, has been residualized as a call to an intermediate procedure coercing its integer argument to a real and calling Procedure *print*.

The target language of this partial evaluator is reminiscent of continuation-passing style without continuations, otherwise known as *nqCPS*, *A-normal forms* [26], or *monadic normal forms* [36], *i.e.*, for all practical purposes, three-address code, as in the Dragon book [1]. It can be indifferently con-

```
...
        [s   (push-proc
               (lambda (s)   ;;; decl. of p (code pointer)
                 (let* ([p3 (lookup-proc 0 0 s)]
                        [a4 (lookup-al 0 1 s)]
                        [i5 (lookup-int 1 0 s)]
                        [i6 (lookup-int 0 2 s)]
                        [i7 (add-int i5 i6)]
                        [s  (push-al a4 s)]
                        [s  (push-base-pointer s)]
                        [s  (push-int i7 s)]
                        [s  (update-base-pointer s 1)]
                        [s  (p3 s)])
                   (pop-frame s)))
               s)]
        [s   (push-al a2 s)] ;;; decl. of p (access link)
        [p8  (lookup-proc 0 3 s)]
        [a9  (lookup-al 0 4 s)]
        [p10 (lookup-proc 0 1 s)]
        [a11 (lookup-al 0 2 s)]
        [s   (push-al a9 s)]
        [s   (push-base-pointer s)]
        [s   (push-proc
               (lambda (s)   ;;; coercion of print (code pointer)
                 (let* ([i12 (lookup-int 0 0 s)]
                        [a13 (current-al s)]
                        [s   (push-al a13 s)]
                        [s   (push-base-pointer s)]
                        [s   (push-real (int-to-real i12) s)]
                        [s   (update-base-pointer s 1)]
                        [s   (p10 s)])
                   (pop-frame s)))
               s)
        [s   (push-al a11 s)] ;;; coercion of print (access link)
        [s   (push-int 4 s)]
        [s   (update-base-pointer s 3)]
        [s   (p8 s)]          ;;; actual call to p
        [s   (pop-frame s)])
    (pop-block s 5))))
```

Figure 5.5: Fig. 5.4 (continued and ended)

sidered as a Scheme program or be translated into assembly language.

5.6 Assessment

5.6.1 The definitional interpreter

Our definitional interpreter has roughly the same size as the denotational specification of Section 5.4: 530 lines of Scheme code and less than 16 Kb. This however also includes the treatment of functions, which we have elided here.

The semantic operations occupy 120 lines of Scheme code and about 3.5 Kb.

5.6.2 Compiler generation

Generating a compiler out of an interpreter, using type-directed partial evaluation, amounts to specializing the type-directed partial evaluator with respect to the type of the interpreter (applied to a source program) [19, Section 2.4]. The improvement obtained by eliminating the type interpretive overhead is negligible in practice.

5.6.3 Compiling efficiency

We have constructed a number of source programs of varying size (up to 18,000 lines), and have observed that on the average, compiling takes place at about 400 lines per second on a SPARC station 20 with two 75 Mhz processors running Solaris 2.4, using R. Kent Dybvig's Chez Scheme Version 4.1u. On a smaller machine, a SPARC Station ELC with one 33 Mhz processor running SunOS 4.1.1, about 100 lines are compiled per second, again using Chez Scheme.

5.6.4 Efficiency of the compiled code

The compiled code is of standard, Dragon-book quality [1]. It accounts for the dynamic interpretation steps of the source program. As this paper is going to press, we do not have actual figures yet about the efficiency of assembly code (only of intermediate code).

5.6.5 Interpreted vs. compiled code

Compiled intermediate code runs four times faster than interpreted code, on the average, in Scheme. This is consistent with traditional results in partial evaluation [12, 42].

5.7 Related work

5.7.1 Semantics-implementation systems

Our use of type-directed partial evaluation to specialize a definitional interpreter very precisely matches the goal of Mosses's Semantics Implementation System [53], as witnessed by the following recent quote [54]:

> "SIS [...] took denotational descriptions as input. It transformed a denotational description into a λ-expression which, when applied to the abstract syntax of a program, reduced to a λ-expression that represented the semantics of the program in the form of an input-output function. This expression could be regarded as the 'object code' of the program for the λ-reduction machine that SIS provided. By applying this code to some input, and reducing again, one could get the output of the program according to the semantics."

When SIS was developed, functional languages and their programming environment did not exist. Today's definitional interpreters can be (1) type-checked automatically and (2) interactively tested. Correspondingly, today's λ-reduction machines are simply Scheme, ML, or Haskell systems. Alternatively, though, we can translate our target three-address code directly into assembly language.

SIS was the first semantics-implementation system, but as mentioned in Section 5.2, it was followed by a considerable number of other systems. All of these systems are non-trivial. Some of them are definitely on the sophisticated side. None of them are so simple that they can, in a type-directed fashion and in a few lines, as in Figure 5.1,

1. take a well-typed, runnable, unannotated definitional interpreter in direct style, as in Figure 5.6, together with a source program, as in Figure 5.3; and

2. produce a textual representation of its dynamic semantics, as in Figure 5.4.

5.7.2 Compiler derivation

Deriving compilers from interpreters is a well-documented exercise by now [27, 52, 73]. These derivations are significantly more involved than the present work, and require significant more handcraft and ingenuity. For example, the source interpreter usually has to be expressed in continuation-passing style.

5.7.3 Partial evaluation

The renaissance of partial evaluation we see in the 90s originates in Jones's Mix project [43], which aimed at compiling by interpreter specialization and at generating compilers by self-application. This seminal work has paved the way for further work on partial evaluation, namely with offline strategies and binding-time improvements [12, 42]. Let us simply mention two such works.

Definitional interpreter for an imperative language: In the proceedings of POPL'91 [11], Consel and Danvy report the successful compilation and compiler generation for an imperative language that is much simpler than the present one (procedureless and stackless). The quality of the residual code is comparable, but a full-fledged partial evaluator is used, including source annotations, and so are several non-trivial binding-time improvements, most notably continuations.

Definitional interpreter for a lazy language: In the proceedings of POPL'92 [44], Jørgensen reports the successful compilation and compiler generation for a lazy language of a commercial scale. The target language is the high-level programming language Scheme. The quality of the result is competitive with contemporary implementations, given an efficient Scheme implementation. Again, a full-fledged partial evaluator is used, including source annotations, and so is a veritable arsenal of binding-time improvements, including continuations.

This work: In our work, we use λ-calculus normalization, no annotations, no binding-time analysis, no binding-time improvements, and no continuations. Our results are of Dragon-book quality.

5.8 Conclusion

We hope to have shown that type-directed partial evaluation of a definitional interpreter does scale up to a realistic example with non-trivial programming features. We would also like to point out that our work offers evidence that Scott and Strachey were right, each in their own way, when they developed Denotational Semantics: Strachey in that the λ-calculus provides a proper medium for encoding at least traditional programming languages, as illustrated by Landin [47]; and Scott for organizing this encoding with types and domain theory. The resulting format of denotational semantics proves ideal to apply a six-lines λ-calculus normalizer (using a higher-order functional language) and obtain the front-end of a semantics-based compiler towards three-address code — a strikingly concise and elegant solution to the old problem of semantics-based compiling.

5.9 Limitations

Modern programming languages (such as ML) have more type structure than Algol-like languages, in that whereas the type of the denotation of an Algol program is always the same, the type of the denotation of an ML program depends on the type of this program. Such languages are beyond the reach of the current state-of-the-art of type-directed partial evaluation, which is simply typed. Higher type systems are necessary — a topic for future work.

Acknowledgements

Grateful thanks to John Hatcliff, Nevin Heintze, Julia L. Lawall, Peter Lee, Karoline Malmkjær, Peter D. Mosses, and Peter O'Hearn for discussions and comments; and to the referees, for their lucid reviews.

```
(define meaning
  (lambda (p)
    (lambda (fix test ...)
      (lambda (s)
        (letrec ([meaning-program (lambda (p s) ...)]
                 [meaning-command
                  (lambda (c r s)
                    (case-record c
                      ...
                      [(Assign i1 e)
                       (case-record e
                         [(Ide i2)
                          (let ([(Pair x1 t1) ((cdr r) i1)]
                                [(Pair x2 t2) ((cdr r) i2)])
                            (if (or (is-local? x1 x2)
                                    (and (is-base-type? t1)
                                         (is-base-type? t2)))
                                (update t1 x1
                                        (coerce t2 t1 (lookup t2 x2 s))
                                        s)
                                (wrong "not stackable")))]
                         [else
                          (let ([(Pair x1 t1) ((cdr r) i1)]
                                [(ExpVal t2 v2)
                                 (meaning-expression e r s)])
                            (update t1 x1 (coerce t2 t1 v2) s))])]
                      ...))]
                 [meaning-expression (lambda (e r s) ...)]
                 [meaning-operator (lambda (op v1 v2) ...)]
                 [meaning-declarations (lambda (ds r s) ...)]
                 [meaning-declaration (lambda (d r s) ...)])
          (meaning-program p s))))))
```

Figure 5.6: Skeleton of the medium definitional interpreter

Chapter 6

Type-Directed Change-of-Representation

This Chapter considers ways of generalizing the work from Chapter 3. It is based on observations made while studying the dynamics of type-directed partial evaluation. As the method is type-directed, it is reasonable to consider different orderings on types. Figure 6.1 shows the orderings we have experimented with.

	Fixed base types	Higher order types
Partial Orders	Subtyping, e.g., integer $\leq$ real	Inheritance, e.g., $\tau_1 \times \ldots \times \tau_{n+k} \preceq \tau_1 \times \ldots \times \tau_n$
Equivalence Relations	Commensurate units, e.g., cm $\simeq$ inches	Isomorphic types, e.g., $\tau_1 \times \tau_2 \cong \tau_2 \times \tau_1$

Figure 6.1: Type orderings

Inheritance as a partial order on higher-order types was suggested by Palsberg [56]. The equivalence relation on fixed base types is taken from Kennedy "Relational Parametricity and Units of Measure" [46] and refers to units of the same property (dimension) but in different measures. The problems are similar to those for subtypes, where coercions need to be inserted to have subtyped values occur in supertyped contexts. Kennedy says that values expressed in a certain unit can be *converted* to any commensurate[1]

[1]Webster's entry for commensurate: *Having a common measure; commensurable; reducible to a common measure.*

unit.

Before we go on we would like to give the reader an idea of what we mean by Type-Directed Change-of-Representation. The technique we presented in Chapter 3 changed the representation of terms to their normal forms and the present work is considering ways of generalizing that. More specifically will we address the properties of terms that can be changed based on type information. An example could be insertion of subtype coercions, such that terms with implicit coercions have them made explicit. A typing system could thus have a rule for implicit coercions:

$$\frac{x : \mathrm{Int}}{x : \mathrm{Real}}$$

as opposed to a rule for explicit coercions:

$$\frac{x : \mathrm{Int}}{\mathrm{Int2Real}(x) : \mathrm{Real}}$$

Both have their qualities but in some cases, such as language implementations, it is desirable to be able to automatically obtain the latter form from the former[2]. In a sense we see terms as proof terms where the change-of-representation leaves the proof unchanged but changes the representation of its designated term. We will be more general, especially when considering higher-order types, but the above captures very precisely the approach we are taking[3].

To be more specific we can notice that the method in Chapter 3 produces normal forms by introducing syntax which is $\beta\eta$-equal to the original term. It could be hoped that some desirable effects could be obtained by instead introducing modified, that is not $\beta\eta$-equal, syntax. For currying it could look like:

$$e \rightarrow_{\mathrm{Curry}} \lambda x^{\tau_1}.\lambda y^{\tau_2}.e\,\langle x, y\rangle$$

and we would define the following double-type indexed family of functions and their symmetric versions:

$$\Downarrow^{\tau_1\times\tau_2\rightarrow\tau_3}_{\tau_1\rightarrow\tau_2\rightarrow\tau_3}\ v \quad = \quad \underline{\lambda} x.\underline{\lambda} y.{\downarrow}^{\tau_3}\ v\ \overline{@}\ \underline{\langle}\ {\uparrow}_{\tau_1}\ x,\ {\uparrow}_{\tau_2}\ y\underline{\rangle}$$

[2] Observe that the calculus of proof terms in the former case is not isomorphic to the logic, while this in fact is so in the latter case.

[3] It would thus be natural to relate the present treatment to notions like interdeducability of proofs and we leave it as future work.

Insertion of coercions and conversions would be obtained by :

$$\Downarrow^{\mathrm{Int}}_{\mathrm{Real}} v = \mathrm{Int2Real}(v)$$

We will show that such an approach is actually not tractable as the technology is readily available. Howver, the approach will set up a framework for us, in which we can investigate more complex situations.

In "Isomorphisms of Types: from λ-calculus to information retrieval and language design" [13] Di Cosmo treats isomorphic types to great extent and we will start by reviewing the relevant material rather thoroughly and thus use his treatment to derive the form suggested above.

6.1 A theory of isomorphic types

We will start by considering a semantical and a syntactical notion of isomorphism and show that they for $\lambda^{\mathrm{Church}}_{\to\times}$ coincide.

Definition 18 *Two types, τ and σ, in $\lambda^{Church}_{\to\times}$ are said to be isomorphic ($\tau \cong_{\lambda^{Church}_{\to\times}} \sigma$) iff their meanings are (model-)isomorphic in every model of the calculus.*

Definition 19 *Two types, τ and σ, in $\lambda^{Church}_{\to\times}$ are provably isomorphic*[4] *($\tau \cong^{p}_{\lambda^{Church}_{\to\times}} \sigma$) iff there exists invertible terms $M^{\tau\to\sigma}$ and $N^{\sigma\to\tau}$, called witnesses ($M : \tau \cong^{p}_{\lambda^{Church}_{\to\times}} \sigma : N$), such that:*

$$\begin{aligned} M \circ N &=_{\beta\eta} \lambda x^{\sigma}.x \\ N \circ M &=_{\beta\eta} \lambda x^{\tau}.x \end{aligned}$$

where $\circ$ is infix for the following [type-indexed family of] $\lambda^{Church}_{\to\times}$-term[s]:

$$\circ_{\tau_1,\tau_2,\tau_3} = (\lambda g^{\tau_2\to\tau_3}.\lambda f^{\tau_1\to\tau_2}.\lambda t^{\tau_1}.g\,(f\,t))^{(\tau_2\to\tau_3)\to(\tau_1\to\tau_2)\to\tau_1\to\tau_3}$$

And we immediately obtain:

Theorem 17 (Di Cosmo: Theorem 1.8.3) *For $\lambda^{Church}_{\to\times}$ type isomorphisms are exactly the provable isomorphisms:*

$$\forall\tau,\sigma.\tau \cong_{\lambda^{Church}_{\to\times}} \sigma \Leftrightarrow \tau \cong^{p}_{\lambda^{Church}_{\to\times}} \sigma$$

[4] There are several different names for this notion of isomorphism. Di Cosmo [13] calls them definable isomorphisms, while Bruce & Longo [9] calls them provable isomorphisms. We have chosen the latter as we, by the Curry-Howard correspondence, see terms as proofs.

Proof:

"$\Rightarrow$": M and N, proving $\tau \cong^{p}_{\lambda^{\text{Church}}_{\rightarrow\times}} \sigma$, compose to the identity both ways. In every model their meanings will thus compose to the relevant identity in the model and we are done.

"$\Leftarrow$": $\lambda^{\text{Church}}_{\rightarrow\times}$ has an open term model thus giving the $\lambda^{\text{Church}}_{\rightarrow\times}$-witnesses immediately.

□

By considering the syntactical notion of isomorphic type we see that any term whose representation we change can be retrieved again up to $\beta\eta$-equivalence. All we need to know is the inverse of the witness that lead to the change. Addressing isomorphic types we find that its theory can be finitely axiomatized hence there is some hope of obtaining an inverse should we be interested.

Axiomatization

Definition 20 (Di Cosmo: Table 1.5) *Let* $\mathbf{Th}_{\rightarrow\times}$ *be the following equational theory:*

$$\begin{aligned} A \times B &= B \times A \\ A \times (B \times C) &= (A \times B) \times C \\ A \times B \rightarrow C &= A \rightarrow B \rightarrow C \\ A \rightarrow B \times C &= (A \rightarrow B) \times (A \rightarrow C) \end{aligned}$$

And write $\mathbf{Th}_{\rightarrow\times} \vdash A{=}B$ *when* A *and* B *are provably equal in the* $\mathbf{Th}_{\rightarrow\times}$[5]

Relationship

Theorem 18 (Di Cosmo: Theorem 1.9.9, Theorem 4.3.4, Remark 4.2.4) $\mathbf{Th}_{\rightarrow\times}$ *is a sound and complete theory of provable isomorphisms for* $\lambda^{Church}_{\rightarrow\times}$*:*

$$\forall \tau, \sigma. \mathbf{Th}_{\rightarrow\times} \vdash \tau = \sigma \Leftrightarrow \tau \cong^{p}_{\lambda^{Church}_{\rightarrow\times}} \sigma$$

Proof: Soundness is proved in Proposition 11, while the completeness proof is rather elaborate and is the essence of Di Cosmo's thesis.

[5] Notice that equality thus defined is a reflexive, transitive and symmetric congruence.

□

It can be mentioned that the completeness proof takes a form that immediately renders itself as a decision procedure for whether two types are isomorphic hence we obtain the following corollary.

Corollary 5 (Di Cosmo: Corollary 4.3.6) $\mathbf{Th}_{\rightarrow\times}$ *is decidable.*

Theorem 18 essentially says that for any pair of provably isomorphic types there is a [finite] proof of their equality in $\mathbf{Th}_{\rightarrow\times}$and more to our use: To all pairs of $\mathbf{Th}_{\rightarrow\times}$-equal types there are terms witnessing them as provably isomorphic. Moreover these witnesses can be defined according to the $\mathbf{Th}_{\rightarrow\times}$-proof. That is, by composing the properly substituted[6] witnesses of each rule application.

Proposition 11 *For types* τ_1, τ_2, τ_3 *the ground equalities of* $\mathbf{Th}_{\rightarrow\times}$ *are witnessed by* $(DC^{\tau}_{\sigma} : \tau \cong^{p}_{\lambda^{Church}_{\rightarrow\times}} \sigma : DC^{\sigma}_{\tau})$*:*

$$
\begin{aligned}
DC^{\tau_1\times\tau_2}_{\tau_2\times\tau_1} &= (\lambda p^{\tau_1\times\tau_2}.\langle p_2(p),\, p_1(p)\rangle)^{\tau_1\times\tau_2\rightarrow\tau_2\times\tau_1}\\
DC^{\tau_2\times\tau_1}_{\tau_1\times\tau_2} &= (\lambda p^{\tau_2\times\tau_1}.\langle p_2(p),\, p_1(p)\rangle)^{\tau_2\times\tau_1\rightarrow\tau_1\times\tau_2}\\
DC^{\tau_1\times(\tau_2\times\tau_3)}_{(\tau_1\times\tau_2)\times\tau_3} &= (\lambda t^{\tau_1\times(\tau_2\times\tau_3)}.\langle\langle p_1(t),\, p_1(p_2(t))\rangle,\, p_2(p_2(t))\rangle)^{\tau_1\times(\tau_2\times\tau_3)\rightarrow(\tau_1\times\tau_2)\times\tau_3}\\
DC^{(\tau_1\times\tau_2)\times\tau_3}_{\tau_1\times(\tau_2\times\tau_3)} &= (\lambda t^{(\tau_1\times\tau_2)\times\tau_3}.\langle p_1(p_1(t)),\, \langle p_2(p_1(t)),\, p_2(t)\rangle\rangle)^{(\tau_1\times\tau_2)\times\tau_3\rightarrow\tau_1\times(\tau_2\times\tau_3)}\\
DC^{\tau_1\times\tau_2\rightarrow\tau_3}_{\tau_1\rightarrow\tau_2\rightarrow\tau_3} &= (\lambda f^{\tau_1\times\tau_2\rightarrow\tau_3}.\lambda a^{\tau_1}.\lambda b^{\tau_2}.f\,\langle a,\, b\rangle)^{(\tau_1\times\tau_2\rightarrow\tau_3)\rightarrow\tau_1\rightarrow\tau_2\rightarrow\tau_3}\\
DC^{\tau_1\rightarrow\tau_2\rightarrow\tau_3}_{\tau_1\times\tau_2\rightarrow\tau_3} &= (\lambda f^{\tau_1\rightarrow\tau_2\rightarrow\tau_3}.\lambda p^{\tau_1\times\tau_2}.(f\,p_1(p))\,p_2(p))^{(\tau_1\rightarrow\tau_2\rightarrow\tau_3)\rightarrow\tau_1\times\tau_2\rightarrow\tau_3}\\
DC^{\tau_1\rightarrow\tau_2\times\tau_3}_{(\tau_1\rightarrow\tau_2)\times(\tau_1\rightarrow\tau_3)} &= (\lambda f^{\tau_1\rightarrow\tau_2\times\tau_3}.\langle\lambda a^{\tau_1}.p_1(f\,a),\, \lambda a^{\tau_1}.p_2(f\,a)\rangle)^{(\tau_1\rightarrow\tau_2\times\tau_3)\rightarrow(\tau_1\rightarrow\tau_2)\times(\tau_1\rightarrow\tau_3)}\\
DC^{(\tau_1\rightarrow\tau_2)\times(\tau_1\rightarrow\tau_3)}_{\tau_1\rightarrow\tau_2\times\tau_3} &= (\lambda p^{(\tau_1\rightarrow\tau_2)\times(\tau_1\rightarrow\tau_3)}.\lambda a^{\tau_1}.\langle p_1(p)\,a,\, p_2(p)\,a\rangle)^{(\tau_1\rightarrow\tau_2)\times(\tau_1\rightarrow\tau_3)\rightarrow\tau_1\rightarrow\tau_2\times\tau_3}
\end{aligned}
$$

Furthermore if $M : \tau_1 \cong^{p}_{\lambda^{Church}_{\rightarrow\times}} \tau_2 : M^{-1}$ *and* $N : \tau_3 \cong^{p}_{\lambda^{Church}_{\rightarrow\times}} \tau_4 : N^{-1}$ *we have the substitution witnesses:*

$$
\begin{aligned}
DC[M,N]^{\tau_1\times\tau_3}_{\tau_2\times\tau_4} &= (\lambda p^{\tau_1\times\tau_3}.\langle M\,p_1(p),\, N\,p_2(p)\rangle)^{\tau_1\times\tau_3\rightarrow\tau_2\times\tau_4}\\
DC[M^{-1},N^{-1}]^{\tau_2\times\tau_4}_{\tau_1\times\tau_3} &= (\lambda p^{\tau_2\times\tau_4}.\langle M^{-1}\,p_1(p),\, N^{-1}\,p_2(p)\rangle)^{\tau_2\times\tau_4\rightarrow\tau_1\times\tau_3}\\
DC[M^{-1},N]^{\tau_1\rightarrow\tau_3}_{\tau_2\rightarrow\tau_4} &= (\lambda f^{\tau_1\rightarrow\tau_3}.\lambda x^{\tau_2}.N\,(f\,(M^{-1}\,x)))^{(\tau_1\rightarrow\tau_3)\rightarrow\tau_2\rightarrow\tau_4}\\
DC[M,N^{-1}]^{\tau_2\rightarrow\tau_4}_{\tau_1\rightarrow\tau_3} &= (\lambda f^{\tau_2\rightarrow\tau_4}.\lambda x^{\tau_1}.N^{-1}\,(f\,(M\,x)))^{(\tau_2\rightarrow\tau_4)\rightarrow\tau_1\rightarrow\tau_3}
\end{aligned}
$$

[6] This point will become clear in a moment.

Pairwise they witness $\tau_1 \times \tau_3 \cong^p_{\lambda^{Church}_{\rightarrow\times}} \tau_2 \times \tau_4$ *resp.* $\tau_1 \rightarrow \tau_3 \cong^p_{\lambda^{Church}_{\rightarrow\times}} \tau_2 \rightarrow \tau_4$ *hence* $\cong^p_{\lambda^{Church}_{\rightarrow\times}}$ *is a congruence – observe the contravariant behaviour of the substitution witnesses for the domain type of a function. Finally, provable isomorphisms also enjoy reflexivity, symmetry, and reflexivity as, for all types, the identity is self inverse, witnesses are mutually inverses, and composition is provable and invertible.*

Proof: The types of the terms are correct, and all that is left is to prove these terms pairwise invertible, which is immediate, and the details will be omitted, however, it should be noted that $\circ$ has the following property

$$(M^{\tau_2\rightarrow\tau_3} \circ_{\tau_1,\tau_2,\tau_3} N^{\tau_1\rightarrow\tau_2})^{-1} = (N^{-1})^{\tau_2\rightarrow\tau_1} \circ_{\tau_3,\tau_2,\tau_1} (M^{-1})^{\tau_3\rightarrow\tau_2}$$

□

Remark [Proposition 11]: Observe that, for M, N, τ, and σ properly defined, we have the following notational equalities:

$$\begin{aligned} (\mathrm{DC}^{\tau}_{\sigma})^{-1} &= \mathrm{DC}^{\sigma}_{\tau} \\ (\mathrm{DC}[\mathrm{M},\mathrm{N}]^{\tau}_{\sigma})^{-1} &= \mathrm{DC}[\mathrm{M}^{-1},\mathrm{N}^{-1}]^{\sigma}_{\tau} \end{aligned}$$

And we will only give one of the witnesses for an isomorphism.

We will consider a few examples of this:

- Consider $\sigma_1 = \tau_1 \times \tau_2 \rightarrow \tau_3$ and $\sigma_2 = \tau_2 \times \tau_1 \rightarrow \tau_3$ and let us prove $\sigma_1 \cong^p_{\lambda^{\mathrm{Church}}_{\rightarrow\times}} \sigma_2$, thus illustrating how to compose the ground witnesses through the substitution witnesses. We immediately have that $\mathrm{DC}^{\tau_1\times\tau_2}_{\tau_2\times\tau_1}$ witnesses that the domain types are isomorphic, while Id_{τ_3} proves the codomain types isomorphic, so σ_1 and σ_2 are provably isomorphic by $\mathrm{DC}[(\mathrm{DC}^{\tau_1\times\tau_2}_{\tau_2\times\tau_1})^{-1},\mathrm{Id}_{\tau_3}]^{\sigma_1}_{\sigma_2}$ with normal form:

  $$(\lambda f^{\sigma_1}.\lambda x^{\tau_2\times\tau_1}.f\,\langle \mathrm{p}_2(x),\, \mathrm{p}_1(x)\rangle)^{\sigma_1\rightarrow\sigma_2}$$

  As can be induced from the following automatic normalization of the erasure of the witness[7]

```
> (define-base-type t1)
> (define-base-type t2)
> (define-base-type t3)
```

[7] By use of Danvy's Type-Directed Partial Evaluator [17, 18]

```
> (define-compound-type sigma1 ((t1 * t2) -> t3) "f" alias)
> (define-compound-type p (t2 * t1) "x" alias)
> (define-compound-type sigma2 (p -> t3))
> (define DC1 (lambda (p) (cons (cdr p) (car p))))
> (define DC2 (lambda (dc1 dc2)
                      (lambda (f)
                              (lambda (x)
                                      (dc2 (f (dc1 x)))))))
> (define DC3 (DC2 DC1 (lambda (a) a)))
> (residualize DC3 '(sigma1 -> sigma2))
(lambda (f) (lambda (x) (f (cons (cdr x) (car x)))))
>
```

- Next consider $\sigma_1 = (\tau_1 \times \tau_2) \times \tau_3 \to \tau_4 \cong^{p}_{\lambda^{\text{Church}}_{\to\times}} \tau_3 \times \tau_1 \to \tau_2 \to \tau_4 = \sigma_2$. It is immediate to see that symmetry, followed by associativity and ending with currying proves them equal in $\mathbf{Th}_{\to\times}$:

$$\begin{array}{rcl} \mathbf{Th}_{\to\times} & \vdash & (\tau_1 \times \tau_2) \times \tau_3 \to \tau_4 \\ & = & \tau_3 \times (\tau_1 \times \tau_2) \to \tau_4 \\ & = & (\tau_3 \times \tau_1) \times \tau_2 \to \tau_4 \\ & = & \tau_3 \times \tau_1 \to \tau_2 \to \tau_4 \end{array}$$

  and thus by the transitivity witnesses (that is, composition) we get that the isomorphism is witnessed by:

$$\begin{array}{rcl} \mathrm{DC}^{(\tau_3\times\tau_1)\times\tau_2\to\tau_4}_{\tau_3\times\tau_1\to\tau_2\to\tau_4} & \circ & \mathrm{DC}[\mathrm{DC}^{(\tau_3\times\tau_1)\times\tau_2}_{\tau_3\times(\tau_1\times\tau_2)}, Id_{\tau_4}]^{\tau_3\times(\tau_1\times\tau_2)\to\tau_4}_{(\tau_3\times\tau_1)\times\tau_2\to\tau_4} \\ & \circ & \mathrm{DC}[\mathrm{DC}^{\tau_3\times(\tau_1\times\tau_2)}_{(\tau_1\times\tau_2)\times\tau_3}, Id_{\tau_4}]^{(\tau_1\times\tau_2)\times\tau_3\to\tau_4}_{\tau_3\times(\tau_1\times\tau_2)\to\tau_4} \end{array}$$

  This term has the following β-normal form:

$$(\lambda f^{\sigma_1}.\lambda p^{\tau_3\times\tau_1}.\lambda a^{\tau_2}.f\,\langle\langle \mathrm{p}_2(p),\, a\rangle,\, \mathrm{p}_1(p)\rangle)^{\sigma_1\to\sigma_2}$$

  As can be seen from

```
> (define-base-type t1)
> (define-base-type t2)
> (define-base-type t3)
> (define-base-type t4)
> (define-compound-type sigma1 (((t1 * t2) * t3) -> t4) "f" alias)
> (define-compound-type p (t3 * t1) "p" alias)
> (define-compound-type a t2 "a" alias)
```

```
> (define-compound-type sigma2 (p -> a -> t4))
> (define DC1 (lambda (t)
                      (cons (car (car t))
                            (cons (cdr (car t)) (cdr t)))))
> (define DC2 (lambda (x) x))
> (define DC3 (lambda (dc1 dc2)
                      (lambda (f)
                              (lambda (x)
                                      (dc2 (f (dc1 x)))))))
> (define DC4 (DC3 DC1 DC2))
> (define DC5 (lambda (p) (cons (cdr p) (car p))))
> (define DC6 (DC3 DC5 DC2))
> (define DC7 (lambda (f)
                      (lambda (a)
                              (lambda (b)
                                      (f (cons a b))))))
> (define DC8 (lambda (x) (DC7 (DC4 (DC6 x)))))
> (residualize DC8 '(sigma1 -> sigma2))
(lambda (f)
  (lambda (p)
    (lambda (a) (f (cons (cons (cdr p) a) (car p))))))
>
```

6.2 Type-Directed Change-of-Representation on isomorphic types

We will now consider the effect of combining our two-level coercers, Figure 3.1, with the finite family of witnesses for the type isomorphisms, Proposition 11. Having

$$\Downarrow^{\tau_1\times\tau_2\to\tau_3}_{\tau_1\to\tau_2\to\tau_3} v \;=\; \underline{\lambda}x.\underline{\lambda}y.{\downarrow^{\tau_3}}\, v\, \overline{@}\overline{\langle}\uparrow_{\tau_1} x, \uparrow_{\tau_2} y\overline{\rangle}$$

suggests to us to compose the two-level coercers with dynamic versions of the witnesses. However, as both $\lambda^{\text{Curry-II}}_{\to\times}$ and the two-level coercers are symmetric, we see that the exact same result would be obtained by using the witnesses statically and in fact we shall do so, for reasons to become clear. Assuming for a while to have $\downarrow$, $\uparrow$ defined in $\lambda^{\text{Curry-II}}_{\to\times}$ we wish to define:

$$\begin{aligned}\Downarrow^{\sigma}_{\tau} &\approx \downarrow^{\tau} \circ \overline{\mathrm{DC}}^{\sigma}_{\tau}\\ \Uparrow^{\sigma}_{\tau} &\approx \overline{\mathrm{DC}}^{\tau}_{\sigma} \circ \uparrow_{\tau}\end{aligned}$$

Which, however, does not go as we encounter the same problem that left $\downarrow, \uparrow$ undefinable in $\lambda_{\to\times}^{\text{Curry-II}}$:

$$\overline{\lambda} v.{\downarrow}^{\sigma}\ (\overline{\mathrm{DC}}_{\sigma}^{\tau}\ \overline{@}\ v) \notin \overline{\Lambda}_{\mathrm{II}}^{\tau\to\sigma}, \underline{\Lambda}_{\mathrm{II}}^{\tau\to\sigma}$$

That is, they are not $\lambda_{\to\times}^{\text{Curry-II}}$-terms because at any type they have a static domain and a dynamic codomain. We do, however, expect such a composition to always produce $\lambda_{\to\times}^{\text{Curry-II}}$-terms so we proceed by considering:

$$\begin{array}{lcl} \Downarrow_{\tau}^{\sigma}\ v & = & {\downarrow}^{\tau}\ (\overline{\mathrm{DC}}_{\tau}^{\sigma}\ \overline{@}\ v) \in \overline{\Lambda}_{\mathrm{II}}^{\tau} \\ \Uparrow_{\tau}^{\sigma}\ v & = & \overline{\mathrm{DC}}_{\sigma}^{\tau}\ \overline{@}\ ({\uparrow}_{\tau}\ v) \in \underline{\Lambda}_{\mathrm{II}}^{\sigma} \end{array}$$

Our hope is that the resulting terms will $\overline{\beta}$-reduce to long $\underline{\beta(\eta)}$-normal forms and we will in the following perform the initial $\overline{\beta}$-reductions.

$$\begin{array}{l} \Downarrow_{\tau_2\times\tau_1}^{\tau_1\times\tau_2}\ v \\ \quad =_{\overline{\beta}}\ {\downarrow}^{\tau_2\times\tau_1}\ \overline{\langle \overline{\mathrm{p}}_2(v), \overline{\mathrm{p}}_1(v)\rangle} \\ \quad =\ \underline{\langle}\ {\downarrow}^{\tau_2}\ \overline{\mathrm{p}}_1(\overline{\langle \overline{\mathrm{p}}_2(v), \overline{\mathrm{p}}_1(v)\rangle}), {\downarrow}^{\tau_1}\ \overline{\mathrm{p}}_2(\overline{\langle \overline{\mathrm{p}}_2(v), \overline{\mathrm{p}}_1(v)\rangle})\underline{\rangle} \\ \quad =_{\overline{\beta}}\ \underline{\langle}\ {\downarrow}^{\tau_2}\ \overline{\mathrm{p}}_2(v), {\downarrow}^{\tau_1}\ \overline{\mathrm{p}}_1(v)\underline{\rangle} \end{array}$$

$$\begin{array}{l} \Downarrow_{(\tau_1\times\tau_2)\times\tau_3}^{\tau_1\times(\tau_2\times\tau_3)}\ v \\ \quad =_{\overline{\beta}}\ {\downarrow}^{(\tau_1\times\tau_2)\times\tau_3}\ (\overline{\langle\langle \overline{\mathrm{p}}_1(v), \overline{\mathrm{p}}_{12}(v)\rangle}, \overline{\mathrm{p}}_{22}(v)\rangle) \\ \quad =_{\overline{\beta}}\ \underline{\langle\langle}\ {\downarrow}^{\tau_1}\ \overline{\mathrm{p}}_1(v), {\downarrow}^{\tau_2}\ \overline{\mathrm{p}}_{12}(v)\underline{\rangle}, {\downarrow}^{\tau_3}\ \overline{\mathrm{p}}_{22}(v)\underline{\rangle} \end{array}$$

$$\begin{array}{l} \Downarrow_{\tau_1\times(\tau_2\times\tau_3)}^{(\tau_1\times\tau_2)\times\tau_3}\ v \\ \quad =_{\overline{\beta}}\ {\downarrow}^{\tau_1\times(\tau_2\times\tau_3)}\ \overline{\langle \overline{\mathrm{p}}_{11}(v), \overline{\langle \overline{\mathrm{p}}_{21}(v), \overline{\mathrm{p}}_2(v)\rangle}\rangle} \\ \quad =_{\overline{\beta}}\ \underline{\langle}\ {\downarrow}^{\tau_1}\ \overline{\mathrm{p}}_{11}(v), \underline{\langle}\ {\downarrow}^{\tau_2}\ \overline{\mathrm{p}}_{21}(v), {\downarrow}^{\tau_3}\ \overline{\mathrm{p}}_2(v)\underline{\rangle}\underline{\rangle} \end{array}$$

$$\begin{array}{l} \Downarrow_{\tau_1\to\tau_2\to\tau_3}^{\tau_1\times\tau_2\to\tau_3}\ v \\ \quad =_{\overline{\beta}}\ {\downarrow}^{\tau_1\to\tau_2\to\tau_3}\ \overline{\lambda} a.\overline{\lambda} b.v\ \overline{@}\ \overline{\langle} a, b\overline{\rangle} \\ \quad =_{\overline{\beta}}\ \underline{\lambda} x_1.\underline{\lambda} x_2.{\downarrow}^{\tau_3}\ (v\ \overline{@}\ \overline{\langle}\ {\uparrow}_{\tau_1}\ x_1, {\uparrow}_{\tau_2}\ x_2\overline{\rangle}) \end{array}$$

$$\begin{array}{l} \Downarrow_{\tau_1\times\tau_2\to\tau_3}^{\tau_1\to\tau_2\to\tau_3}\ v \\ \quad =_{\overline{\beta}}\ {\downarrow}^{\tau_1\times\tau_2\to\tau_3}\ \overline{\lambda} p.(v\ \overline{@}\ \overline{\mathrm{p}}_1(p))\ \overline{@}\ \overline{\mathrm{p}}_2(p) \end{array}$$

$$=_{\overline{\beta}} \quad \underline{\lambda} x.{\downarrow}^{\tau_3}\ (v\,\overline{@}\,({\uparrow}_{\tau_1}\ \underline{\mathrm{p}}_1(x)))\,\overline{@}\,({\uparrow}_{\tau_2}\ \underline{\mathrm{p}}_2(x))$$

$$\begin{aligned}
&{\Downarrow}^{\tau_1\to\tau_2\times\tau_3}_{(\tau_1\to\tau_2)\times(\tau_1\to\tau_3)}\ v\\
&\quad =_{\overline{\beta}} \quad {\downarrow}^{(\tau_1\to\tau_2)\times(\tau_1\to\tau_3)}\ \underline{\langle}\overline{\lambda} w.\overline{\mathrm{p}}_1(v\,\overline{@}\,w),\ \overline{\lambda} w.\overline{\mathrm{p}}_2(v\,\overline{@}\,w)\underline{\rangle})\\
&\quad =_{\overline{\beta}} \quad \underline{\langle\underline{\lambda} x.{\downarrow}^{\tau_2}\ \overline{\mathrm{p}}_1(v\,\overline{@}\,({\uparrow}_{\tau_1}\ x)),\ \underline{\lambda} y.{\downarrow}^{\tau_3}\ \overline{\mathrm{p}}_2(v\,\overline{@}\,({\uparrow}_{\tau_1}\ y))\rangle}
\end{aligned}$$

$$\begin{aligned}
&{\Downarrow}^{(\tau_1\to\tau_2)\times(\tau_1\to\tau_3)}_{\tau_1\to\tau_2\times\tau_3}\ v\\
&\quad =_{\overline{\beta}} \quad {\downarrow}^{\tau_1\to\tau_2\times\tau_3}\ \overline{\lambda} a.\overline{\langle}(\overline{\mathrm{p}}_1(v)\,\overline{@}\,a),\ (\overline{\mathrm{p}}_2(v)\,\overline{@}\,a)\overline{\rangle}\\
&\quad =_{\overline{\beta}} \quad \underline{\lambda} x.\underline{\langle}\ {\downarrow}^{\tau_2}\ (\overline{\mathrm{p}}_1(v)\,\overline{@}\,({\uparrow}_{\tau_1}\ x)),\ {\downarrow}^{\tau_3}\ (\overline{\mathrm{p}}_2(v)\,\overline{@}\,({\uparrow}_{\tau_1}\ x))\underline{\rangle}
\end{aligned}$$

Now consider the composition with the substitution witnesses. The result will be parameterized versions of $\Downarrow$.

$$\begin{aligned}
&{\Downarrow}^{\tau_1\times\tau_3}_{\tau_2\times\tau_4}\ [\overline{\mathrm{DC}}[\overline{\mathrm{DC}}^{\tau_1}_{\tau_2},\overline{\mathrm{DC}}^{\tau_3}_{\tau_4}]^{\tau_1\times\tau_3}_{\tau_2\times\tau_4}]\,v\\
&\quad =_{\overline{\beta}} \quad \underline{\langle}\ {\Downarrow}^{\tau_1}_{\tau_2}\ [\overline{\mathrm{DC}}^{\tau_1}_{\tau_2}]\,\overline{\mathrm{p}}_1(v),\ {\Downarrow}^{\tau_3}_{\tau_4}\ [\overline{\mathrm{DC}}^{\tau_3}_{\tau_4}]\,\overline{\mathrm{p}}_2(v)\underline{\rangle}
\end{aligned}$$

$$\begin{aligned}
&{\Downarrow}^{\tau_1\to\tau_3}_{\tau_2\to\tau_4}\ [\overline{\mathrm{DC}}[\overline{\mathrm{DC}}^{\tau_2}_{\tau_1},\overline{\mathrm{DC}}^{\tau_3}_{\tau_4}]^{\tau_1\to\tau_3}_{\tau_2\to\tau_4}]\,v\\
&\quad =_{\overline{\beta}} \quad \underline{\lambda} x.{\Downarrow}^{\tau_3}_{\tau_4}\ [\overline{\mathrm{DC}}^{\tau_3}_{\tau_4}]\,(v\,\overline{@}\,{\Uparrow}^{\tau_1}_{\tau_2}\ [\overline{\mathrm{DC}}^{\tau_2}_{\tau_1}]\,x)
\end{aligned}$$

In the inductive cases we take parameterizing with the identity to be $\uparrow$, $\downarrow$ while parameterizing with one of the ground witnesses gives the corresponding case. Next for $\Uparrow$:

$$\begin{aligned}
&{\Uparrow}^{\tau_2\times\tau_1}_{\tau_1\times\tau_2}\ e\\
&\quad = \quad (\overline{\lambda} p.\overline{\langle}\overline{\mathrm{p}}_2(p),\ \overline{\mathrm{p}}_1(p)\overline{\rangle})\,\overline{@}\,\overline{\langle}\ {\uparrow}_{\tau_1}\ \underline{\mathrm{p}}_1(e),\ {\uparrow}_{\tau_2}\ \underline{\mathrm{p}}_2(e)\overline{\rangle}\\
&\quad =_{\overline{\beta}} \quad \overline{\langle}\ {\uparrow}_{\tau_2}\ \underline{\mathrm{p}}_2(e),\ {\uparrow}_{\tau_1}\ \underline{\mathrm{p}}_1(e)\overline{\rangle}
\end{aligned}$$

$$\begin{aligned}
&{\Uparrow}^{(\tau_1\times\tau_2)\times\tau_3}_{\tau_1\times(\tau_2\times\tau_3)}\ e\\
&\quad = \quad (\overline{\lambda} t.\overline{\langle\langle}\overline{\mathrm{p}}_1(t),\ \overline{\mathrm{p}}_{12}(t)\overline{\rangle},\ \overline{\mathrm{p}}_{22}(t)\overline{\rangle})\,\overline{@}\,\overline{\langle}\ {\uparrow}_{\tau_1}\ \underline{\mathrm{p}}_1(e),\ \overline{\langle}\ {\uparrow}_{\tau_2}\ \underline{\mathrm{p}}_{12}(e),\ {\uparrow}_{\tau_3}\ \underline{\mathrm{p}}_{22}(e)\overline{\rangle\rangle}\\
&\quad =_{\overline{\beta}} \quad \overline{\langle\langle}\ {\uparrow}_{\tau_1}\ \underline{\mathrm{p}}_1(e),\ {\uparrow}_{\tau_2}\ \underline{\mathrm{p}}_{12}(e)\overline{\rangle},\ {\uparrow}_{\tau_3}\ \underline{\mathrm{p}}_{22}(e)\overline{\rangle}
\end{aligned}$$

$$\begin{aligned}
&{\Uparrow}^{\tau_1\times(\tau_2\times\tau_3)}_{(\tau_1\times\tau_2)\times\tau_3}\ e\\
&\quad = \quad (\overline{\lambda} t.\overline{\langle}\overline{\mathrm{p}}_{11}(t),\ \overline{\langle}\overline{\mathrm{p}}_{21}(t),\ \overline{\mathrm{p}}_2(t)\overline{\rangle}\overline{\rangle})\,\overline{@}\,\overline{\langle\langle}\ {\uparrow}_{\tau_1}\ \underline{\mathrm{p}}_{11}(e),\ {\uparrow}_{\tau_2}\ \underline{\mathrm{p}}_{21}(e)\overline{\rangle},\ {\uparrow}_{\tau_3}\ \underline{\mathrm{p}}_2(e)\overline{\rangle}\\
&\quad =_{\overline{\beta}} \quad \overline{\langle}\ {\uparrow}_{\tau_1}\ \underline{\mathrm{p}}_{11}(e),\ \overline{\langle}\ {\uparrow}_{\tau_2}\ \underline{\mathrm{p}}_{21}(e),\ {\uparrow}_{\tau_3}\ \underline{\mathrm{p}}_2(e)\overline{\rangle\rangle}
\end{aligned}$$

$$\Uparrow^{\tau_1\to\tau_2\to\tau_3}_{\tau_1\times\tau_2\to\tau_3}\ e$$
$$\begin{array}{ll} = & (\overline{\lambda}f.\overline{\lambda}a.\overline{\lambda}b.f\,\overline{@}\,\overline{\langle}a,\,b\overline{\rangle})\,\overline{@}\,\overline{\lambda}p.{\uparrow_{\tau_3}}\,(e\,\underline{@}\,\underline{\langle}\,{\downarrow^{\tau_1}}\,\overline{\mathrm{p}}_1(p),\,{\downarrow^{\tau_2}}\,\overline{\mathrm{p}}_2(p)\underline{\rangle}) \\ =_{\overline{\beta}} & \overline{\lambda}a.\overline{\lambda}b.{\uparrow_{\tau_3}}\,(e\,\underline{@}\,\underline{\langle}\,{\downarrow^{\tau_1}}\,a,\,{\downarrow^{\tau_2}}\,b\underline{\rangle}) \end{array}$$

$$\Uparrow^{\tau_1\times\tau_2\to\tau_3}_{\tau_1\to\tau_2\to\tau_3}\ e$$
$$\begin{array}{ll} = & (\overline{\lambda}f.\overline{\lambda}p.(f\,\overline{@}\,\overline{\mathrm{p}}_1(p))\,\overline{@}\,\overline{\mathrm{p}}_2(p))\,\overline{@}\,\overline{\lambda}a.\overline{\lambda}b.{\uparrow_{\tau_3}}\,((e\,\underline{@}\,{\downarrow^{\tau_1}}\,a)\,\underline{@}\,{\downarrow^{\tau_2}}\,b) \\ =_{\overline{\beta}} & \overline{\lambda}p.{\uparrow_{\tau_3}}\,(e\,\underline{@}\,({\downarrow^{\tau_1}}\,\overline{\mathrm{p}}_1(p)))\,\underline{@}\,({\downarrow^{\tau_2}}\,\overline{\mathrm{p}}_2(p)) \end{array}$$

$$\Uparrow^{(\tau_1\to\tau_2)\times(\tau_1\to\tau_3)}_{\tau_1\to\tau_2\times\tau_3}\ e$$
$$\begin{array}{ll} = & (\overline{\lambda}f.\overline{\langle}\overline{\lambda}a.\overline{\mathrm{p}}_1(f\,\overline{@}\,a),\,\overline{\lambda}a.\overline{\mathrm{p}}_2(f\,\overline{@}\,a)\overline{\rangle}) \\ & \overline{@}\,\overline{\lambda}a.\overline{\langle}\,{\uparrow_{\tau_2}}\,\underline{\mathrm{p}}_1(e\,\underline{@}\,({\downarrow^{\tau_1}}\,a)),\,{\uparrow_{\tau_3}}\,\underline{\mathrm{p}}_2(e\,\underline{@}\,({\downarrow^{\tau_1}}\,a))\overline{\rangle} \\ =_{\overline{\beta}} & \overline{\langle}\overline{\lambda}a.{\uparrow_{\tau_2}}\,\underline{\mathrm{p}}_1(e\,\underline{@}\,({\downarrow^{\tau_1}}\,a)),\,\overline{\lambda}a.{\uparrow_{\tau_3}}\,\underline{\mathrm{p}}_2(e\,\underline{@}\,({\downarrow^{\tau_1}}\,a))\overline{\rangle} \end{array}$$

$$\Uparrow^{\tau_1\to\tau_2\times\tau_3}_{(\tau_1\to\tau_2)\times(\tau_1\to\tau_3)}\ e$$
$$\begin{array}{ll} = & (\overline{\lambda}p.\overline{\lambda}a.\overline{\langle}(\overline{\mathrm{p}}_1(p)\,\overline{@}\,a),\,(\overline{\mathrm{p}}_2(p)\,\overline{@}\,a)\overline{\rangle}) \\ & \overline{@}\,\overline{\langle}\overline{\lambda}a.{\uparrow_{\tau_2}}\,(\underline{\mathrm{p}}_1(e)\,\underline{@}\,({\downarrow^{\tau_1}}\,a)),\,\overline{\lambda}a.{\uparrow_{\tau_3}}\,(\underline{\mathrm{p}}_2(e)\,\underline{@}\,({\downarrow^{\tau_1}}\,a))\overline{\rangle} \\ =_{\overline{\beta}} & \overline{\lambda}a.\overline{\langle}\,{\uparrow_{\tau_2}}\,(\underline{\mathrm{p}}_1(e)\,\underline{@}\,({\downarrow^{\tau_1}}\,a)),\,{\uparrow_{\tau_3}}\,(\underline{\mathrm{p}}_2(e)\,\underline{@}\,({\downarrow^{\tau_1}}\,a))\overline{\rangle} \end{array}$$

And the composites:

$$\Uparrow^{\tau_2\times\tau_4}_{\tau_1\times\tau_3}\,[\overline{\mathrm{DC}}[\overline{\mathrm{DC}}^{\tau_1}_{\tau_2},\overline{\mathrm{DC}}^{\tau_3}_{\tau_4}]^{\tau_1\times\tau_3}_{\tau_2\times\tau_4}]\,e$$
$$=_{\overline{\beta}}\ \ \overline{\langle}\,\Uparrow^{\tau_2}_{\tau_1}\,[\overline{\mathrm{DC}}^{\tau_1}_{\tau_2}]\,\overline{\mathrm{p}}_1(e),\,\Uparrow^{\tau_4}_{\tau_3}\,[\overline{\mathrm{DC}}^{\tau_3}_{\tau_4}]\,\overline{\mathrm{p}}_2(e)\overline{\rangle}$$

$$\Uparrow^{\tau_2\to\tau_4}_{\tau_1\to\tau_3}\,[\overline{\mathrm{DC}}[\overline{\mathrm{DC}}^{\tau_2}_{\tau_1},\overline{\mathrm{DC}}^{\tau_3}_{\tau_4}]^{\tau_1\to\tau_3}_{\tau_2\to\tau_4}]\,e$$
$$=_{\overline{\beta}}\ \ \overline{\lambda}x.\Uparrow^{\tau_4}_{\tau_3}\,[\overline{\mathrm{DC}}^{\tau_3}_{\tau_4}]\,(e\,\underline{@}\,\Downarrow^{\tau_2}_{\tau_1}\,[\overline{\mathrm{DC}}^{\tau_2}_{\tau_1}]\,x)$$

Analyzing the result

We see that for any proof of two types being isomorphic there is a corresponding term that, when statically composed with the level coercers, allows us to perform the associated change-of-representation. Furthermore the resulting terms actually have the expected form which as an aside was what

lead us to the present treatment. However, we also saw that it is the witnesses in themselves that perform the change in types not the composition with the two-level coerers. By CR for $\overline{\beta}$-reduction it is actually the case that the long $\beta(\eta)$-normal form is obtained under any reduction strategy. The implementation in Chapter 3 can thus be used to perform change between isomorphic types as it is. All which is needed is that the term at hand is having its type changed beforehand by application of the witness for the change.

6.3 Type-Directed Change-of-Representation

For isomorphic types we saw that the changes-of-representation we can perform are all reversible up to $\beta\eta$-equivalence. Therefore they in a certain sense are safe. We will now go beyond isomorphic types and by the above analysis we obtain the following completeness result for change-of-representation:

Proposition 12 $\downarrow, \uparrow$ *as defined in Figure 3.1, can perform any change-of-representation that can be described as a closed* $\lambda_{\rightarrow\times}^{Church}$*-term and furthermore return a long* $\beta(\eta)$*-normal form.*

Corollary 6 *We can change terms from one type to any other type which is related to it by the relations in Figure 6.1.*

Proof: The following families of $\lambda_{\rightarrow\times}^{\mathrm{Church}}$-terms consists of closed terms. Observe that by the substitution witnesses of Proposition 11 we need only consider the simplest instances.

Subtyping For base types $o_1 \leq o_2$ and coercions $o_1 2 o_2$, we have

$$(\lambda x^{o_1}.o_1 2 o_2(x))^{o_1 \rightarrow o_2}$$

Commensurate units Parallel.

Inheritance For all sets of types, $\{\tau_1 \ldots, \tau_{n+k}\}$, we have

$$(\lambda x^{\tau_1 \times \ldots \times \tau_{n+k}}.\langle \langle \mathrm{p}_1(x), \ldots \rangle \ldots, \mathrm{p}_{\underbrace{2 \ldots 2}_{n}}(x) \rangle)^{\tau_1 \times \ldots \times \tau_{n+k} \rightarrow \tau_1 \times \ldots \times \tau_n}$$

And the full result follows by closing the family under associativity.

Isomorphic types See Proposition 11.

□

6.4 Going even further

In the beginning of this chapter, we posed the problem of whether it was possible to perform type-directed change-of-representation in such a way that, e.g., currying could be performed on only a subset of the terms that could be curried according to their types. The question was answered in the affirmative by the use of the substitution witnesses of Proposition 11. They essentially allowed us to pick the particular subtype of the overall type we would like to have curried. However, once picked we just let things take their course. Reflecting on this we immediately see that we have not achieved the other of our original goals namely to make implicit coercions explicit in the terms. For example, is it easy to see that no choice based on the type of x can be made that would make the implicit coercion explicit in the following example if we are restricted to the approach used so far.

$$(\lambda x.\lambda f.f\,x\,x)^{\mathrm{Int}\to(\mathrm{Int}\to\mathrm{Real}\to\mathrm{Real})\to\mathrm{Real}}$$

The reason is that we can only treat the term associated with a specific subtype in a uniform manner as we are guided purely by the types not by the interdependencies between the types.

We remember how we in Chapter 2 analyzed the role of the separation property for the normal formed fragment of Nm. We used it to justify a type-directed approach when treating β-normal forms in $\lambda_{\to\times}^{\mathrm{Curry}}$. However, we see that the approach apparently falls short in the present situation. The reason being that we can extend Nm with the rule allowing coercions but it seems as if we do not have any type-directed way of determining when it is used. This can of course not be so, as extending a logic that enjoys the subformula property with a subtyping rule as proposed, does not make the subformula property brake. And the separation property is a consequence of the subformula property.

6.4.1 The separation property on fixed base types

Let us study the separation property on fixed base types to see what the issues are. To do this we will consider a very simple logic over the atomic formulas: Int & Real, the connective: $\to$, and let ϕ, ψ be meta-variables over formulas defined according to the following rules:

$$\text{Int} \qquad \text{Real} \qquad \frac{\text{Int}}{\text{Real}}\,(\text{Int2Real}) \qquad \frac{\begin{array}{c}[\phi]_k \\ \vdots \\ \psi\end{array}}{\phi \to \psi}\,(\to I)_k$$

We see that the first three formulas adhere to the separation property as they are. However, look at:

$$\frac{\dfrac{[\text{Int}]_1}{\text{Real}}\,(\text{Int2Real})}{\text{Int} \to \text{Real}}\,(\to I)_1$$

At a first glance, we could be tempted to conclude that the logic does not enjoy the separation property, as the rule (Int2Real) appears to be unaccounted for amongst the constructors, $\to$, in the conclusion [and free assumptions] of the above derivation. However, as witnessed by the first three rules both Real & Int should be counted as 0-arity constructors[8] as they introduce a formula which has meaning in itself. Actually we see that there are two constructor rules for Real which is what leads to the conflict.

We will now go back to our type-directed method of coercing between levels and see that the above considerations actually manifests themselves. To that end, we will not use the two-level coercers as the identity on base types, but leave them as they are, and see what happens when considering the above example:

$$\begin{array}{ll}
& \downarrow^{\text{Int}\to(\text{Int}\to\text{Real}\to\text{Real})\to\text{Real}} \overline{\lambda}x.\overline{\lambda}f.f\,\overline{@}\,x\,\overline{@}\,x \\
= & \underline{\lambda}y.\underline{\lambda}g.{\downarrow^{\text{Real}}}\,((\overline{\lambda}x.\overline{\lambda}f.f\,\overline{@}\,x\,\overline{@}\,x)\,\overline{@}\,({\uparrow_{\text{Int}}}\,y)\,\overline{@}\,\overline{\lambda}v.\overline{\lambda}w.{\uparrow_{\text{Real}}}\,(g\,\underline{@}\,({\downarrow^{\text{Int}}}\,v)\,\underline{@}\,({\downarrow^{\text{Real}}}\,w))) \\
=_{\overline{\beta}} & \underline{\lambda}y.\underline{\lambda}g.g\,\underline{@}\,({\downarrow^{\text{Int}}}\,({\uparrow_{\text{Int}}}\,y))\,\underline{@}\,({\downarrow^{\text{Real}}}\,({\uparrow_{\text{Int}}}\,y)) \\
=^{?} & \underline{\lambda}y.\underline{\lambda}g.g\,\underline{@}\,y\,\underline{@}\,\text{Int2Real}(y)
\end{array}$$

And we see that the information is actually available. The only question is how to take advantage of it in order to justify the last equality.

6.4.2 Fixed base types and Type-Directed Change-of-Representation

We see that we already have answered the question, as we concluded that we needed a way of considering inter-dependencies between types. More specifically, we have to be able to determine by which criterion we have

[8] In fact, this is also standard when considering designated formulas [70, Section 3.2]

concluded Real: By use of (Int2Real) or because it was actually supplied as a Real from either some elimination rule or by a [discharged] assumption[9].

We will turn to type theory to justify the approach we take. Observe that from the mutual recursive structure of $\downarrow, \uparrow$ we can see that $\downarrow$ is annotated with types occurring positively while $\uparrow$ is annotated with types occurring negatively in the overall type, when using them to normalize terms as we did in Chapter 3. Another way to say this is that we use $\downarrow$ according to types of terms, while $\uparrow$ is used according to types of contexts. When considering terms as derivations this becomes: $\downarrow$ is used when trying to decide which rule to use coming from above in the derivation, while $\uparrow$ tries the same coming from below. The reason we have failed to accomplish the coercion-insertion so far is that at base type we need both kinds of information to determine whether to put a (Int2Real) rule there or not. In Figure 6.2 we do this by having $\uparrow_o$ introduce type-annotated terms.

$$\begin{array}{rcl} \downarrow^o v^{o'} & = & \underline{o'2o}(v) \\ \uparrow_o e & = & e^o \end{array}$$

Figure 6.2: Making coercions and conversions explicit

And the last two lines from our running example becomes

$$\begin{array}{rcl} \downarrow^{\mathrm{Int}\to(\mathrm{Int}\to\mathrm{Real}\to\mathrm{Real})\to\mathrm{Real}} \overline{\lambda}x.\overline{\lambda}f.f\,\overline{@}\,x\,\overline{@}\,x & & \\ & =_{\overline{\beta}} & \underline{\lambda}y.\underline{\lambda}g.g\,\underline{@}\,(\downarrow^{\mathrm{Int}} y^{\mathrm{Int}})\,\underline{@}\,(\downarrow^{\mathrm{Real}} y^{\mathrm{Int}}) \\ & = & \underline{\lambda}y.\underline{\lambda}g.g\,\underline{@}\,y\,\underline{@}\,\mathrm{Int2Real}(y) \end{array}$$

6.5 Conclusion and future work

In a certain sense, we have in the last section been considering first-order type-direction as opposed to zeroth-order in the preceding sections. We went beyond having types just being types and considered one way of relating types which was unrelated to their structure. We saw that on the logical level this corresponded to having several introduction rules for the same constructor.

[9] Observe that, as we only consider closed terms, it cannot be by free assumption.

In [19], Danvy considers the use of shift/reset to insert let constructs to preserve strictness of functions under normalization and to be able to treat sum typed terms. This appears to capture a notion of higher-order type-direction. To which extent this is *ad hoc* is not known at present. It is also not known whether an ordering on type-direction makes sense at all. We leave the further study of methods for using type-directed approaches to future work.

Chapter 7

Conclusion

7.1 Overview

This thesis has been concerned with proof normalization and various computational aspects of it. We have seen the basic notion of proof normalization for two styles of presenting conjunctive, implicative minimal logic: Sequent Calculus and Natural Deduction, Chapter 2. As for the Natural Deduction, we presented a way of obtaining a representation of normal forms given the proof term corresponding to the original proof, Chapter 3. The method was presented and proved correct in a purely syntactical setting. It was done by using a new, symmetric, two-level λ-calculus. We believe that this is the first time the material has been given a thorough syntactical treatment. Furthermore it is the first correctness proof that actually treats products as well as functions.

We observed that the method at hand actually was implementable in Scheme. We were concerned with to which extent the implementation had practical implications, Chapters 4 & 5. There are a number of technical details in our presentation, which we see as crucial to the results we were able to obtain.

Our two-level λ-calculus was defined to only allow interface between the levels at base type, Chapter 1. This results in a totally symmetric calculus for which we, in a more general setting, could define two functions that coerced any term to the other level. We saw that this base-type level-interface requirement goes very well in thread with the intuitive property of Huet's long $\beta(\eta)$-normal forms. It also proved to be crucial to the implementation as it allowed us to circumvent that Scheme "only" evaluates to weak head

normal forms.

We would like to point out that the symmetry of the calculus allowed us to treat generalizations of the method in a uniform way, Chapter 6. We saw how trying to generalize the two-level coercers gave the same result whether we approached it statically or dynamically. This gave us a completeness result with respect to the expressiveness of the two-level coercers.

Our initial attempts to generalize were immediately seen to be a sort of zeroth-order approach and we successfully considered a first-order generalization. We saw that first-order type-direction was strictly more expressive than the zeroth-order case. As for the higher-order case, no results were given.

With respect to proof normalization for the Sequent Calculus presentation of the logic, Chapter 2, we saw that we were able to slightly alter the traditional approach, Cut Elimination, to obtain a new method, Cut Propagation. Cut Propagation was seen to be more compelling when we treated the proof terms of the logic as a calculus of explicit substitutions.

The proof normalizer, for the Natural Deduction presentation of the logic, was seen to be sufficiently powerful to allow us to perform Semantics-Based Compiling via the first Futamura projection for a non-trivial stack-based language, Chapter 5. The approach we took was seen to be justified by Schmidt-style denotational semantics.

7.2 Future work

We proposed a number of areas as future work. Most formerly these include a further investigation of

- the use of results from Term Rewriting Systems to obtain type-independent constructive proofs for Cut being admissible, and
- different ways of using type-directed approaches to make properties of derivations explicit in the associated proof terms.

7.3 Final word

We would like to stress one particular issue regarding the treatment we have given the material. We have sought to use as simple means as possible.

We believe that *a priori* there is no reason why simplicity would preclude obtaining interesting results. Actually we believe mathematical simplicity is the one thing to strive for when explaining and substantiating ones work. We hope to have shown that altering ones viewpoint on a subject relative to available means can simplify matters.

Appendix A

Variables and Sm+Cut+Cont+Weak

A.1 Variables and Sm+Cut

$$
\begin{array}{rcl}
x & : & x \in \mathrm{FV}(x) \\
\text{let } \langle x_1,\, x_2\rangle{=}x_3 \text{ in } e & : & \mathrm{BV}(\text{let } \langle x_1,\, x_2\rangle{=}x_3 \text{ in } e) = \mathrm{BV}(e) \cup \{x_1, x_2\} \\
 & \& & \mathrm{FV}(\text{let } \langle x_1,\, x_2\rangle{=}x_3 \text{ in } e) = (\mathrm{FV}(e) \cup \{x_3\}) \setminus \{x_1, x_2\} \\
 & \& & x_3 \notin \mathrm{FV}(e) \\
\langle e_1,\, e_2\rangle & : & \mathrm{FV}(\langle e_1,\, e_2\rangle) = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
 & \& & \mathrm{BV}(\langle e_1,\, e_2\rangle) = \mathrm{BV}(e_1) \cup \mathrm{BV}(e_2) \\
\text{let } x_1{=}x_2\, e_1 \text{ in } e_2 & : & \mathrm{FV}(\text{let } x_1{=}x_2\, e_1 \text{ in } e_2) = (\mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \cup \{x_2\}) \setminus \{x_1\} \\
 & \& & \mathrm{BV}(\text{let } x_1{=}x_2\, e_1 \text{ in } e_2) = \mathrm{BV}(e_1) \cup \mathrm{BV}(e_2) \cup \{x_1\} \\
 & \& & x_2 \notin \mathrm{FV}(e_2) \\
 & \& & x_1 \notin \mathrm{FV}(e_1) \\
\lambda x.e & : & \mathrm{BV}(\lambda x.e) = \mathrm{BV}(e) \cup \{x\} \\
 & \& & \mathrm{FV}(\lambda x.e) = \mathrm{FV}(e) \setminus \{x\} \\
e_1\langle\!\langle e_2/x\rangle\!\rangle & : & \mathrm{BV}(e_1\langle\!\langle e_2/x\rangle\!\rangle) = \mathrm{BV}(e_1) \cup \mathrm{BV}(e_2) \\
 & \& & \mathrm{FV}(e_1\langle\!\langle e_2/x\rangle\!\rangle) = (\mathrm{FV}(e_1) \cup \mathrm{FV}(e_2)) \setminus \{x\} \\
 & \& & (\{x\} \cup \mathrm{FV}(e_1)) \cap \mathrm{FV}(e_2) = \emptyset \\
 & \& & \mathrm{BV}(e_1) \cap \mathrm{FV}(e_2) = \emptyset \\
 & \& & \mathrm{FV}(e_1) \cap \mathrm{BV}(e_2) = \emptyset
\end{array}
$$

The two last restrictions are deduced from the fact that only variables

that become contracted can be equal. This is only possible if both are either free or bound –in the above sense– at any point in the derivation.

A.2 Variables and Contraction

As seen in Appendix A.1 there are non-local requirements on our rules when we add proof terms. Consider the following example that uses a more general contraction rule than the one we considered:

$$\dfrac{\dfrac{x:\tau \vdash u:\sigma \quad y:\sigma, z:\tau \vdash v:\gamma}{x:\tau, z:\tau \vdash v\langle\!\langle u/y \rangle\!\rangle : \gamma}\,(Cut)}{s:\tau \vdash (v\langle\!\langle u/y \rangle\!\rangle)[s/x][s/z]:\gamma}\,(Cont)$$

The issue is that our logic is closed under contraction. Thus the concluding proof-term above has to be equal to the proof-term that concludes the corresponding contraction-free derivation, in order to induce a calculus of proof-terms which is isomorphic to the logic. As we are only interested in using the calculus for reasoning purposes, we will enforce the restriction that any variables, that are contracted in the process of making derivations cut-free, must be identical:

$$\dfrac{\dfrac{x:\tau \vdash u:\sigma \quad y:\sigma, x:\tau \vdash v:\gamma}{x:\tau, x:\tau \vdash v\langle\!\langle u/y \rangle\!\rangle : \gamma}\,(Cut)}{x:\tau \vdash v\langle\!\langle u/y \rangle\!\rangle : \gamma}\,(Cont)$$

This is seen to not have any effect on the expressiveness of the logic, however odd the restriction might seem.

A.3 Variables and Weakening

Our logic is to be considered closed under weakening. Weakening adds a formula to the antecedent and thus according to Appendix A.1 extends the set of variables defined to be free. As the actual free variables of a proofterm in the succedent of a conclusion is included in FV() (cf. Appendix A.1), this is seen to have no immediate computational effect, and we will refer to such variables as excessive context. Observe once again that excessive contexts are required to make rules context-sharing.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Yohji Akama. On Mints' reduction for ccc-calculus. In *Typed Lambda Calculi and Applications*, volume 664, pages 1–12, 1993.

[3] France E. Allen, editor. *Proceedings of the 1982 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No 6, Boston, Massachusetts, June 1982. ACM Press.

[4] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt and David E. Rydeheard, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, 1995.

[5] Andrea Asperti. Special light linear logic. Available via anonymous ftp as ftp.cs.unibo.it/pub/asperti/SLLL.ps.

[6] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993.

[7] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[8] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[9] Kim B. Bruce and Giuseppe Longo. Provable isomorphisms and domain equations in models of typed languages (preliminary version). In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 263–272, Providence, Rhode Island, 6–8 May 1985.

[10] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[11] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[12] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[13] Roberto Di Cosmo. *Isomorphisms of types: from lambda-calculus to information retrieval and language design.* Progress in theoretical computer science. Birkhauser, 1995.

[14] Roberto Di Cosmo. A brief history of rewriting with extensionality. `ftp://ftp.ens.fr/pub/dmi/users/dicosmo/Slides/GLA96.ps.gz`, September 1996. Lecture given at the International Summer School on Type Theory and Rewriting, Glasgow.

[15] Roberto Di Cosmo and Delia Kesner. Strong normalization of explicit substitutions via cut elimination in proof nets. In *12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, 1997.

[16] D. Cubric. Embedding of a free CCC into the category of sets. `ftp://triples.math.mcgill.ca/pub/cubric/frccc.ps.gz`, March 1992. Manuscript announced on the Types mailing-list.

[17] Olivier Danvy. A documented implementation of type-directed partial evaluation. Available upon request from the author: `<danvy@brics.dk>`.

[18] Olivier Danvy. Pragmatics of type-directed partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 73–94, Dagstuhl, Germany, February 1996.

[19] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[20] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[21] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation*, pages 209–227, 1993. An earlier version appeared in the proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.

[22] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

[23] Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Extended version available as the technical report BRICS-RS-96-13.

[24] Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation (extended version). Technical report BRICS-RS-96-13, Computer Science Department, Aarhus University, Aarhus, Denmark, May 1996.

[25] Roy Dyckhoff and Luis Pinto. Permutability of proofs in intuitionistic sequent calculi. Draft, November 1996. Sofar unavailable to the present author.

[26] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[27] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill, 1991.

[28] Harald Ganzinger, Robert Giegerich, Ulrich Mönke, and Reinhard Wilhem. A truly generative semantics-directed compiler generator. In Allen [3], pages 172–184.

[29] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[30] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[31] Susan L. Graham, editor. *Proceedings of the 1984 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No 6, Montréal, Canada, June 1984. ACM Press.

[32] Bernhard Gramlich. Relating innermost, weak, uniform and modular termination of term rewriting systems. In *International Conference on Logic Programming and Automated Reasoning, 1992*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 285–296. Springer-Verlag, 1992.

[33] Bernhard Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, September 1995. Preliminary version appeared as Relating Innermost, Weak, Uniform and Modular Termination of Term Rewriting Systems, LPAR'92.

[34] Bernhard Gramlich. On proving termination by innermost termination. In *Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA-96)*, volume 1103 of *LNCS*, pages 93–107. Springer-Verlag, July 27–30 1996.

[35] William Harrison and Sam Kamin. Compilation as partial evaluation of functor category semantics. Available from `http://www-sal.cs.uiuc.edu/~kamin/`.

[36] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994. ACM Press.

[37] H. Herbelin. A λ-calculus structure isomorphic to gentzen-style sequent calculus structure. In *Annual Conference of the European Association for Computer Science Logic, CSL'94, Kazimierz (Poland) , Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 1995.

[38] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 470–490. Academic Press, 1980.

[39] Gérard Huet. Résolution d'équations dans les langages d'ordre 1, 2, ..., ω. Thèse d'État, Université de Paris VII, Paris, France, 1976.

[40] C. Barry Jay and Neil Ghani. The virtues of eta-expansion. *Journal of Functional Programming*, 5(2):135–154, 1995.

[41] Neil D. Jones, editor. *Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, Aarhus, Denmark, 1980.

[42] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.

[43] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[44] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 258–268, Albuquerque, New Mexico, January 1992. ACM Press.

[45] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In Mark Scott Johnson and Ravi Sethi, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg, Florida, January 1986.

[46] Andrew Kennedy. Relational parametricity and units of measure. In Neil D. Jones, editor, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997. ACM Press. To appear.

[47] Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965.

[48] Peter Lee. *Realistic Compiler Generation.* MIT Press, 1989.

[49] Pierre Lescanne. From $\lambda\sigma$ to λv: A journey through calculi of explicit substitutions. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 60–69, Portland, Oregon, January 17–21, 1994. ACM Press.

[50] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, and John Wiley, New York, 1976.

[51] G.E. Mints. Theory of catefories and theory of proofs. *Urgent Questions of Logic and the Methodology of Science*, 1979. In Russian.

[52] Lockwood Morris. The next 700 formal language descriptions. In Carolyn L. Talcott, editor, *Special issue on continuations (Part I)*, LISP and Symbolic Computation, Vol. 6, Nos. 3/4, pages 249–258. Kluwer Academic Publishers, December 1993.

[53] Peter D. Mosses. SIS — semantics implementation system, reference manual and user guide. Technical Report MD-30, DAIMI, Computer Science Department, Aarhus University, Aarhus, Denmark, 1979.

[54] Peter D. Mosses. Theory and practice of Action Semantics. In *Proceedings of the 1996 Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, 1996. To appear.

[55] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

[56] Jens Palsberg. Personal communication, December 1996. A presentation of a preliminary version of Type-Directed Change-of-Representation spawned some questions about handling of inheritance.

[57] Larry Paulson. Compiler generation from denotational semantics. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.

[58] Uwe Pleban. Compiler prototyping using formal semantics. In Graham [31], pages 94–105.

[59] Dag Prawitz. *Natural Deduction.* Almquist and Wiksell, Uppsala, 1965.

[60] B. Randell and L. J. Russell. *Algol 60 Implementation.* Academic Press, New York, 1964.

[61] Martin R. Raskovsky. Denotational semantics as a specification of code generators. In Allen [3], pages 230–244.

[62] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972.

[63] John C. Reynolds. The essence of Algol. In van Vliet, editor, *International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, The Netherlands, 1982. North-Holland.

[64] Kristoffer S. Rose. Explicit substitutions – tutorial & survey. Technical Report LS-96-13, BRICS, Aarhus University, Denmark, September 1996.

[65] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[66] Helmut Schwichtenberg. Termination of permutative conversions in intuitionistic gentzen calculi. Available on the authors homepage: `http://www.mathematik.uni-muenchen.de/~schwicht`, April 1997. Brought to the attention of the present author by Anne S. Troelstra. Kindly made available by the author.

[67] Ravi Sethi. Control flow aspects of semantics-directed compiling. In Allen [3], pages 245–260.

[68] Joseph Stoy. Some mathematical aspects of functional programming. In John Darlington, Peter Henderson, and David A. Turner, editors, *Functional Programming and its Applications.* Cambridge University Press, 1982.

[69] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[70] Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996.

[71] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

[72] Mitchell Wand. A semantic prototyping system. In Graham [31], pages 213–221.

[73] Mitchell Wand. From interpreter to compiler: a representational derivation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 306–324, Copenhagen, Denmark, October 1985.

[74] C.-P. Wirth and B. Gramlich. A constructor-based approach to positive/negative-conditional equational specifications. *Journal of Symbolic Computation*, 17(1):51–90, January 1994.