Types for Proofs and Programs

Bernd Grobauer

Progress Report

BRICS

BRICS Ph.D. School
Department of Computer Science
University of Aarhus
Denmark

May 1999

Supervisor: Olivier Danvy

Acknowledgments

This progress report marks the half time on my way towards a Ph.D. in computer science. There are still two years to go, but I feel that already now thanks are due for all the help I received to reach this milestone. Let me start with my fellow students, most notably Wolfgang Naraschewski and Markus Wenzel in Munich, and Daniel Damian and Zhe Yang in Aarhus. In numerous discussions they not only "brought computer science to life", but also taught me a lot in the process.

I am indebted to Olaf Müller for his help and guidance in a research project carried out during my stay in Munich in the fall of 1998.

Thanks for advise both on a personal and scientific level are due to the three persons who have been supervising my studies, officially and unofficially: Tobias Nipkow in Munich, and Olivier Danvy and Mogens Nielsen in Aarhus. I recall various situations during the last two years, in which their guidance was indispensable. Olivier Danvy, my official supervisor, deserves special mention for all the time and effort he put into helping me to write this progress report. It gained immensely from his suggestions and observations. I am also grateful to Andrzej Filinski for timely comments.

I further would like to mention what a privilege it is to study and conduct research at BRICS. As a foreign Ph.D. student, I especially would like to thank Janne Christensen and Karen Kjær Møller for making BRICS such a hospitable place.

Last but not least I want to thank my parents and my sister Bärbel for their patience and support.

Contents

Chapter 1

Introduction and Overview

The present progress report is intended to give an introduction to the two areas of research I have been concentrating on in my second year of Ph.D. studies at the BRICS International Ph.D. School: interactive theorem proving and type-directed partial evaluation.

On the face of it, these areas seem to be quite disconnected. However the notion of types provides a conceptual link. The vast majority of the formal systems that are implemented by interactive theorem provers are so-called type theories. Chapter 2 focuses on using types for proofs: after an introduction to the history of formal systems and how they gave rise to types, the objectives and principles of interactive theorem proving are highlighted. Then an application of interactive theorem proving is described: Olaf Müller and myself developed a formalization of the theory of timed I/O automata, thus providing proof support for reasoning about timed distributed systems.

In the context of programming languages, types are ubiquitous. Chapter 3 first explores the basic concept underlying all applications of type systems in programming languages: restricting the number of possible programs that can be written in order to gain additional knowledge about the legal programs. Then, type-directed partial evaluation, a particular instance of using the information provided by types, is examined. Finally a small research project (joint work with Zhe Yang) in the context of type-directed partial evaluation is described.

Even though types provide a conceptual link between theorem proving and type-directed partial evaluation, it is not yet clear if a synthesis between both fields can be reached. Searching for possibilities of a synthesis is the logical next step of my research. Apart from some ideas for further research in both areas, chapter 4 outlines some initial ideas for a possible synthesis.

In the rest of the report, a working knowledge of logic and functional programming is assumed.

Chapter 2

Types for Proofs

At the turn of the century, interest into the foundations of mathematics gave rise to a number of proposals of formal systems in which general mathematics could be formalized. Bertrand Russell was the first to use a notion of type in a formal system. This led to a vast amount of research on so-called *type theories*. In the following, we first give an overview of the history of formal systems and describe how interactive theorem proving made the use of formal systems for reasoning feasible (see John Harrison's excellent article on *Formalized Mathematics* [34] for an in-depth treatment of both topics). We then turn to examine reasoning in Higher-Order Logic (HOL), a particular type-theory, and close with the description of a formalization of timed I/O automata in HOL.

2.1 Formal Systems

2.1.1 What is a Formal System?

While it is impossible to give a precise definition of what a formal system is, one can observe two features which seem to be common to most formal systems: one or more (formal) languages and judgment(s) over the language(s). Take propositional logic as an example. The language in question is an inductively defined language of propositions. By giving axioms and rules, one defines (again inductively) a judgment over these propositions, thus creating a class of derivable propositions.

Formal systems are used to reason about mathematical structures in a purely syntactical manner, namely by building derivations within the formal system. A prerequisite for this approach are theorems which relate the formal system to the mathematical structures one wants to reason about. Propositional logic, for example, is used to reason about truth functions. A so-called soundness result guarantees that the interpretation of each derivable proposition is a tautology. A completeness result tells one that indeed all tautologies are derivable in the system.

2.1.2 The Origins of Formal Systems

George Boole's formal system (formulated in 1854) for logical and set-theoretic reasoning can be regarded as the first successful attempt to reduce mathematical reasoning to symbolic reasoning within a formal system. Frege and Peano were the first to achieve this for general mathematics. Frege's formal system, the so-called *Begriffschrift* (1879), inspired Russell and Whitehead to their

work on the *Principia Mathematica* (1910 - 1913), in which they formalized a substantial part of mathematics within a formal system. Thus the Principia Mathematica can be seen as the first attempt to give a fully formal account of mathematics.

A number of different formal systems were proposed as a foundation for mathematics in the first half of the 20th century. The two most well-known, which have become somewhat of a standard, are probably Zermelo-Fraenkel set-theory and Gödel-Bernays set-theory.

It is interesting to note that Russell used a notion of types in the formal system introduced in the Principia Mathematica to get around the kind of inconsistency he himself had spotted in Frege's system. The so-called *Russell antinomy* involves the construction of a set $R := \{x \mid x \notin x\}$: from assuming $R \in R$ one can derive $R \notin R$ and vice versa. Russell introduced types which layered the universe of discourse and required that for $x \in y$ to make sense, y must be in layer $n+1$ if x is in layer n. Thus the antinomy could be avoided.

2.1.3 Formal Systems in Computer Science

The research of using formal systems as a foundation for mathematics, that started with Frege and Peano, had very little or no impact on work in most areas of mathematics. Reasoning is still mostly conducted in a naive set-theory. Most practitioners of mathematics seem to believe that the level of formality commonly employed is sufficient; after all, everyone with a mathematical education has some gut-feeling about what is trivially true in naive set-theory, i.e., what can be regarded as a basic step in a proof. Furthermore, conducting proofs in a formal system usually is quite intricate: firstly, the form of a problem and the mathematical concepts used in its proof have to be formalized in a far more detailed way than with usual mathematical reasoning. Secondly, writing down a large derivation in a formal system with pencil and paper is usually quite infeasible; errors made in the symbolic manipulations will produce a fair number of incorrect proofs, which is exactly what was to be avoided by using formal systems in the first place.

Advances in computer science created an additional necessity to reason both within formal systems and about formal systems. In mathematical logic, formal systems had been mainly used to give an axiomatic basis to various theories, but for many applications naive set-theory seemed to be good enough. In computer science on the other hand, meaningful reasoning about the central notion of computation is impossible without a precise formalization of what computation is. One of the most influential formalizations of computation, the λ-calculus, basically is a formal system — it had been the object of extensive research long before a mathematical model for it was found. Furthermore, many standard tools of computer-science such as structured operational semantics, typing systems, language grammars etc. can be seen as formal systems. Thus extending the ability to reason about formal systems automatically extends the ability to reason about large parts of computer science.

Curry [13] and Howard [39] were the first to notice that there is a strong link between some of the typing systems used in computer science and various logics — this observation become famous as the *Curry-Howard-Isomorphism*. It is central to a body of work which strives to create a foundational system for computer science, starting with the first formulation (1971) of Martin-Löf's type-theory. Due to an inconsistency discovered by Girard [27], it was substantially modified, giving rise to newer, predicative versions [47, 48] of Martin-Löf type-theory on one hand and the so-called Calculus of Constructions and its extensions [12, 45] on the other hand.

A number of other foundational systems for computer science has been proposed. Scott's PCF, introduced at the beginning of the 70's as a manuscript (unpublished until its appearance in the Böhm Festschrift [70]) is notable not only for the large impact it and the related work of finding

adequate mathematical models had on computer science, but also for the implementation of a *Logic of Computable Functions* inspired by it, giving rise to the LCF-system [62]. The basic ideas for designing an interactive theorem prover formulated then had a considerable influence on the way theorem provers were implemented afterwards: today there are LCF-like theorem provers for various logics — we talk of the so-called *LCF approach*.

2.2 Theorem Proving: Making Formal Systems Usable

2.2.1 Objectives of Interactive Theorem Proving

As mentioned above, using a formal system for pencil and paper proofs is usually infeasible; non-trivial proofs will almost always be of a size and complexity which is not manageable by hand. Apart from the danger of errors in symbolic manipulation, one will get lost in formal details at some point of time, thus losing the overview about the proof. Furthermore specifying all the mathematical objects needed from scratch in a given formal system is by itself a formidable task.

This is where interactive theorem proving comes in, making reasoning in formal systems feasible:

- Correctness of derivations is ensured, since the theorem prover will only carry out derivation steps valid with respect to the underlying formal system.
- Specifications are facilitated by providing high-level language support for defining mathematical objects, which otherwise would have to be coded from scratch.
- The user can focus on the essential parts of a proof, i.e., the steps which provide insight into how the proof works: starting an induction, using rule inversion for an inductively defined relation, etc. Proof steps which amount to pure rewriting or "bookkeeping tasks" necessary in the given formal system (weakening the context and what have you) should be automatized. Usually the user is confronted with a proof state and induces the next proof step by issuing some command. Ideally the theorem prover will provide for proof procedures which take care of at least the uninteresting parts of a proof.

2.2.2 How to Ensure Correctness

As any other program, a theorem prover might be faulty, and for example allow incorrect proof steps. Most theorem provers are simply too big to convince oneself that they do the right thing. Putting one's trust into such a complex program might seem too much to ask for. Total certainty about a complex proof in general may not even be achievable, but there should be a way to achieve roughly the same 'level' of certainty most people seem to feel comfortable with when it comes to believing proofs carried out by hand. Two approaches that strive to provide this kind of certainty advocate correctness relative to the correctness of a small and easily understandable piece of program:

- In type theories that are based on the Curry-Howard-Isomorphism, a proof is actually a concrete proof object, whose type expresses the proposition it proves. Here the idea is to make these proof objects explicit enough to allow a reasonably simple type checker to check whether a given proof object is indeed a proof for some theorem. This way one does not have to trust the possibly big theorem prover which was used to produce this proof object, but only the small type checker.

- Theorem provers which are implemented following the LCF-approach are written in a statically typed language like ML which offers the programmer abstract data types (ADT). An ADT for *theorems* is introduced, for which the basic axioms and rules given by the underlying formal system are implemented as the only operations on this ADT. Type-safety of the implementation language ensures that only these operations can be used to create theorems. Thus one only has to believe that the implementation of this ADT is correct. This implementation should ideally be a rather small part of the theorem-prover.

Randy Pollack's notes on *how to believe a machine-checked proof* [67] provide a deeper discussion on this subject.

2.2.3 How to Facilitate Definitions

In order to reason about mathematical objects, these objects first have to be defined. In informal proofs one will without much ado introduce objects like inductively defined relations, recursively defined data types, functions defined via primitive recursion over such data types, etc. Defining such objects from scratch in a formal system is at least tiresome. Furthermore, it also constitutes a source of errors. In the best case defining everything from scratch leads to objects that mirror not quite what one had in mind, in the worst case it introduces an inconsistency into the theory one is defining.

Theorem provers alleviate this problem via "syntactic sugar", to use a phrase coined in the programming languages community [43]. The user is provided with a high-level language to write definitions in, which will then be transformed into definitions within the formal system. Examples can be found in the literature [6, 54, 72, 73].

2.2.4 How to Facilitate Reasoning

As described above, reasoning is facilitated by offering the user a number of proof procedures with which at least the uninteresting parts of a proof can be conducted automatically. However, in addition to built-in proof procedures, the user should have the possibility to implement new proof procedures which are suited to a particular problem class at hand. Furthermore there is the question of how to combine different proof procedures.

The concept of *tactics* and *tacticals*, which goes back to the LCF approach, is widely used to both provide for basic proof procedures and a generic way of writing new ones. A tactic can be seen as a function of type

$$goal \longrightarrow (goal\ list \times justification),$$

where the *justification* is a function mapping a list of theorems into a theorem. The intuition is that it suffices to prove the *subgoals* returned by the tactic in order to prove the original goal. So after all subgoals have been proven as theorems, the justification function is applied to the resulting list of theorems, hopefully returning the original goal as a theorem. Since theorems are implemented by an abstract data type, the justification function cannot "cheat" — all that can happen is that it either fails or proves a different theorem. Hence a flawed tactic will always be found out at some point of time during the proof — it cannot introduce invalid proofs.

Tacticals are higher-order functions which combine tactics, for example by sequencing two tactics, repeatedly applying a tactic, etc. If tactics are generalized such that they can return a (lazy) list of different possible outcomes, tacticals which perform backtracking can be written.

This opens the door to implementing all kinds of proof search strategies solely by combining tactics with tacticals.

Many LCF-style theorem provers offer a basic simplification tactic which performs higher-order rewriting. A popular way to implement these so-called *simplifiers* is based on the notion of *conversions* [60]: A conversion takes some term t and produces a theorem of form $t = u$, where u should be simpler than t. Just as tactics are combined by tacticals, basic conversions can be combined into more sophisticated conversions by suitable higher-order functions.

Built-in simplification procedures and tactics together with the ability to program customized versions in a relatively intuitive way help the user of an interactive LCF-style theorem prover to delegate a considerable amount of proof search to the machine. Proof procedures which cannot be expressed with the means described so far can be implemented in the language the theorem prover is written in. No invalid proofs can be constructed this way, since any such procedure can only produce theorems by using the functions exported by the abstract data type of theorems.

2.3 Reasoning in Higher-Order Logic (HOL)

2.3.1 Church's Simple Theory of Types

In 1940 Church proposed a foundational system which he called "Simple Theory of Types" [10]. It can be seen as a simplification of Russell's type-theory. Today a slightly modified version (adding polymorphism and a definitional mechanism) is known under the name of "Higher-Order Logic" (HOL), which was first implemented in the HOL system [29] — by now there exist a number of different implementations.

The predicate "simple" in the type-theory's original name does hold true: In the formulation of HOL given in [29], only eight rules, five axioms and ten constant definitions are used. Conceptually, HOL is a higher-order equational calculus with implication, λ-binding and binding via Hilbert's ϵ-combinator; this is extended by adding a number of constants by equational definitions and some rules defining additional properties of these constants.

As any other system, HOL has its strengths and weaknesses. It for example does not offer dependent types and can become quite inflexible for certain parts of mathematics such as abstract algebra. However, for many applications its typing system proves to be a blessing and reasoning within HOL comes close enough to informal mathematics. A large number of applications of HOL can be found in the literature [8, 35, 36, 52, 56, 57, 58, 59, 65, 74]. An extension [55, 75] of the type system with Haskell-style type classes [33] as implemented in Isabelle/HOL [63] adds further flexibility.

2.3.2 Deep vs. Shallow Embedding

Formalizing a logic or a language in a formal system is sometimes referred to as an *embedding*. Formalizing only the semantic operators, translating syntax to semantic structures by an extra-logical device, is referred to as *shallow embedding*. In a so-called *deep embedding* the abstract syntax of a language (or logic) and the semantic functions assigning meaning to syntax are also formalized. Both approaches have their advantages and disadvantages. As a rule of thumb one could say that a shallow embedding is probably more adequate if, for the example of formalizing a programming language, one wants to reason about specific programs rather than strive for general results about the language. Some general results are usually obtainable under a shallow embedding by proving properties of semantic operators, but results of the form "*For all programs*

... " generally are out of reach. Further discussion about embeddings can be found in the literature [8].

A particularly nice example of a shallow embedding in HOL is HOLCF [53], a conservative extension of HOL (as implemented in the Isabelle theorem prover) with LCF constructs. Firstly a type class of complete partial orders is defined, which gives rise to a notion of monotonicity and continuity. This is then extended with definitions of semantic combinators such as application, composition and abstraction for building LCF functions. Properties about these combinators are shown, e.g., that composition of two continuous functions results in a continuous function. Thus one receives a fair number of theorems describing characteristic properties of the semantic combinators. Using these theorems, the automatic proof tools of Isabelle are set up to discharge obligations to prove continuity wherever possible automatically.

Using theorems that relate the LCF function space with the HOL function space (total HOL functions are basically lifted into partial LCF functions), the user has the possibility to do parts of his reasoning in 'pure' HOL, using the more complicated LCF-part only where necessary. The formalization of timed I/O automata described in section 2.4 makes extensive use of this methodology.

2.3.3 Reasoning about Meta-Theory in HOL

It should not come as a surprise that HOL is expressive enough to formalize the reasoning behind many of the formal methods which are in use — after all Church proposed his simple theory of types as a *foundational* system. Hence when using a theorem prover to provide proof support for some formal method, one can chose to formalize the underlying theory of the formal method within the theorem prover.

Usually formal methods provide the user with certain proof principles, whose correctness has been shown by meta-theoretic reasoning, i.e., reasoning about the underlying theory of the formal method. One can decide to take these proof principles on faith. This means modeling only the parts of the theory which are needed to formalize and prove the proof obligations that arise from an application of such principles. Another way is to formalize the complete theory and *derive* the proof principles as theorems, i.e., to reason about meta-theory within the theorem prover. This approach obviously is far more elaborate, but also offers certain advantages. Not only does one avoid errors which stem from possible mismatches between the theory of the formal method and the parts which were formalized. One further gains additional certainty about the validity of the employed proof principles. Maybe most importantly, it is possible to derive new proof principles as the need arises.

The formalization of timed I/O automata in HOLCF described in section 2.4 includes substantial parts of the underlying meta-theory.

2.4 Reasoning about Timed I/O Automata in HOLCF

In this section we focus on an application of the concepts introduced in the previous sections, namely a formalization of the theory of timed I/O automata in HOLCF.

2.4.1 The Theory of Timed I/O Automata

A timed I/O automaton A is a labeled transition system, where the labels are called *actions*. These are divided into *external* and *internal* actions, denoted by $ext(A)$ and $int(A)$, respectively.

The external actions are divided into *input* and *output* actions (denoted by $inp(A)$ and $out(A)$) and a collection of special *time-passage* actions $\{\nu(t) \mid t \in \mathcal{T}\}$. Here $\mathcal{T}$ is a dense time domain which usually is taken to be $\mathbb{R}^{\geq 0}$. A timed I/O automaton A can be specified by defining

- a (necessarily infinite) set $states(A)$ of states
- a nonempty set $start(A) \subseteq states(A)$
- an action signature $sig(A) = (inp(A), out(A), int(A))$ where $inp(A)$, $out(A)$ and $int(A)$ are pairwise disjoint. None of the actions in the signature is allowed to be of the form $\nu(t)$ with $t \in \mathcal{T}$. We call $vis(A) := inp(A) \cup out(A)$ the set of *visible* actions; $ext(A)$ thus can be written as $vis(A) \cup \{\nu(t) \mid t \in \mathcal{T}\}$
- a transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$, where $acts(A)$ is defined as $ext(A) \cup int(A)$. Let $r \xrightarrow{a}_A s$ denote $(r, a, s) \in trans(A)$

A number of additional requirements a timed I/O automaton must satisfy give some insight about the intuition behind the definitions: for example a time-passage action $\nu(t)$ is supposed to stand for the passage of time t. Obviously if t time units pass and then another t' units, this should have the same effect as if $t+t'$ time units passed in one step. Therefore it is required that if $s \xrightarrow{\nu(t)}_A s'$ and $s' \xrightarrow{\nu(t')}_A s''$ then $s \xrightarrow{\nu(t+t')}_A s''$.

Timed I/O automata can be combined by *parallel composition*, which is denoted by $\|$. A step of a composed automaton $A \| B$ is caused either by an internal step of A or B alone, a combined step with an input action common to A and B, or a communication between A and B via an action that is input action to A and output action to B (or vice versa).

Executions and Traces The notion of behavior used for timed I/O automata is *executions* and, as behavior observable from the outside, *traces*.

As for executions, two models are commonly used. The first of them views an execution ex as the continuous passage of time with timeless actions occurring at discrete points in time:

$$ex = \omega_0 \, a_1 \omega_1 \, a_2 \omega_2 \ldots \quad .$$

Here the ω_i are so-called *trajectories*, that is mappings from an interval of the time domain into the state-space of the automaton — the exact definition of a trajectory makes sure that it defines only time-passage that is compatible with the specification of the automaton. These executions are called *timed* executions. We however are going to formalize a model which Lynch and Vaandrager [46] call *sampled* executions. Here an execution is a sequence in which states and actions alternate:

$$ex = s_0 \, a_1 s_1 \, a_2 s_2 \ldots$$

where an action a_i is either a discrete action or a time-passage action. Sampling can be seen as an abstraction of timed executions: instead of trajectories, which hold information about the state of an automaton for every point of time, explicit time-passage steps occur in executions.

The total amount of time which has passed up to a certain point in an execution can be deduced by adding up all the time-values which parameterized the time-passage actions that occurred so far. Notice that the execution model includes executions which, intuitively speaking, do not make sense — since time cannot be stopped, only executions with an infinite total amount of time-passage do. Executions with an infinite total amount of time-passage are called *admissible*.

An execution gives rise to a trace in a straightforward way by first filtering out the invisible actions, suppressing the states and associating each visible action with a time stamp of the point of time of its occurrence. Then the resulting sequences of (visible) actions paired with a time stamp is paired with the least upper bound of the maximal time value reached in the execution (a special symbol ∞ is used in case of an admissible execution). Traces that stem from admissible executions are themselves called admissible.

The observable behavior of a timed I/O automaton A is viewed to be the set of its traces $traces(A)$ — this may sometimes be restricted to certain kinds of traces, e.g., admissible traces. If the signatures of two automata A and C agree on the visible actions, then C is said to implement A iff $traces(C) \subseteq traces(A)$.

Proof Principles The most important proof methods for timed I/O automata are invariance proofs and simulation proofs. The former is used to show that a predicate over $states(A)$ holds for every reachable state in an automaton A. Simulation proofs are used to show that an automaton C implements an automaton A; they are based on a theorem about timed I/O automata which says that if there exists a so-called simulation relation $S \subseteq states(C) \times states(A)$, then C implements A.

2.4.2 A Formalization in HOLCF

In the following we describe how a significant fragment of the theory of timed I/O automata has been formalized in Isabelle. Often only minor changes to an already existing formalization of the theory of I/O automata had to be made. This can be seen as one more piece of evidence that laying the meta-theoretical foundations of a formal method within a theorem prover is worth the additional effort: there is a good chance of reusing the work already done. Further information about the original formalization of I/O automata can be found in Olaf Müller's Ph.D. thesis [52].

Some Preliminaries about Isabelle/HOLCF Several features of Isabelle/HOLCF help the user to write succinct and readable theories (compare with section 2.2.3). There is a mechanism for ML-style definitions of inductive data types [6], introduced by the keyword **datatype**. For every definition of a recursive data type, Isabelle will automatically construct a type together with the necessary proofs that this type has the expected properties. Another feature working along the same lines allows the inductive definition of relations (denoted by **inductive**). Built on top of the data type package is a package which provides for extensible record types [54]. Record definitions are introduced by the keyword **record**.

Isabelle's syntax mechanisms allow for intuitive notations; for example set comprehension is written as $\{e \mid P\}$, a record s which contains a field foo can be updated by writing $s(|\mathsf{foo} := bar|)$. A similar notation is used for function update: $f(e_1 := e_2)$ denotes a function which maps e_1 on e_2 and behaves like f on every other value of its domain.

The type constructor for functions in HOL is denoted by $\Rightarrow$, the projections for pairs by fst and snd. Simple non-recursive definitions are marked with the keyword **defs**, type definitions with **types**, and theorems proven in Isabelle with the keyword **thm**.

Timed I/O Automata All the proofs that have been carried out in order to formalize the necessary fragment of the theory of timed I/O automata only require the time domain $\mathcal{T}$ to be an ordered abelian group. Therefore we keep the whole Isabelle theory of timed I/O automata parametric with respect to a type representing an ordered group. This can be achieved in an elegant way using the mechanism of *axiomatic type classes* [75] of Isabelle: a type class timeD is

defined which represents all types τ that satisfy the axioms of an ordered group with respect to a signature $\langle \tau, \oplus, \ominus, \mathbf{0}, \leq \rangle$. The notation τ_{timeD} signifies that τ is such a type.

Actions of timed I/O automata can either be discrete actions or time-passage actions; we introduce a special data type:

$$\textbf{datatype}\ (\sigma, \tau_{\mathsf{timeD}})\,\mathsf{action} = \mathsf{Discrete}\ \sigma \mid \mathsf{Time}\ \tau$$

In the following a discrete action Discrete a will be written as $\lhd a \rhd$, a time-passage action Time t as $\nu(t)$.

We further chose to make timing information explicit in the states, as for example done by Lynch and Vaandrager [46]. There only one special time-passage action ν is used, which suffices as time-passage can be read from the states. We, however, keep timing information both in the states and in the actions — we argue that specifications of timed I/O automata will gain clarity. Therefore states will always be of type $(\sigma, \tau_{\mathsf{timeD}})\,\mathsf{state}$, where

$$\begin{array}{l}\textbf{record}\ (\sigma, \tau_{\mathsf{timeD}})\,\mathsf{state} = \\ \quad \mathsf{content} :: \sigma \\ \quad \mathsf{now}\ \ :: \tau\end{array}$$

With these modifications in mind, the definitions made in Section 2.4.1 give rise to the following type definitions in a straightforward way:

$$\begin{array}{lrcl}\textbf{types} & \alpha\,\mathsf{signature} & = & (\alpha\,\mathsf{set} \times \alpha\,\mathsf{set} \times \alpha\,\mathsf{set}) \\ & (\alpha, \sigma, \tau_{\mathsf{timeD}})\,\mathsf{transition} & = & (\sigma, \tau)\,\mathsf{state} \times (\alpha, \tau)\,\mathsf{action} \times (\sigma, \tau)\,\mathsf{state} \\ & (\alpha, \sigma, \tau_{\mathsf{timeD}})\,\mathsf{tioa} & = & \alpha\,\mathsf{signature} \times (\sigma, \tau)\,\mathsf{state\ set} \times (\alpha, \sigma, \tau)\,\mathsf{transition\ set}\end{array}$$

Thus, $(\alpha, \sigma, \tau)\,\mathsf{tioa}$ stands for a timed I/O automaton where discrete actions are of type α, the state contents (i.e., the state without its time stamp) of type σ and the used time domain of type τ, which is required to be in the axiomatic type class timeD. For tioa and signature also selectors have been defined, namely

- inputs, outputs and internals, giving access to the different kinds of (discrete) actions that form a signature, together with derived functions like visibles s = inputs $s \cup$ outputs s, which are the visible discrete actions,
- sig-of, starts-of and trans-of giving access to the signature, the start states and the transition relation of a timed I/O automaton.

The additional requirements on timed I/O automata as introduced in section 2.4.1 are formalized as predicates over the transition relation of an automaton. Since we have chosen to carry time stamps within the states, an additional well-formedness condition has to be formulated:

$$\begin{array}{ll}\textbf{defs} & \mathsf{well\text{-}formed\text{-}trans}\ A = \\ & \forall (s, a, r) \in (\mathsf{trans\text{-}of}\ A).\ \ldots \wedge (\mathsf{case}\ a\ \mathsf{of} \\ & \qquad \lhd a' \rhd \rightarrow \mathsf{now}\ s = \mathsf{now}\ r \\ & \qquad \mid \nu(t) \rightarrow \mathsf{now}\ r = (\mathsf{now}\ s) \oplus t)\end{array}$$

This and the other requirements on timed I/O automata are combined in a predicate TIOA over type tioa.

Executions and Traces As mentioned above we are going to formalize sampled executions. Lynch and Vaandrager [46] have shown that reachability is the same for both execution models. Further, for each sampling execution there is a timed execution that gives rise to the same trace and vice versa. Since the sampling model preserves traces and reachability, it is sufficient for our purposes. After all we are not so much interested in meta-theory dealing with the completeness of refinement notions (cf. [46]), but want to provide a foundation for actual verification work. Executions thus boil down to sequences in which states and actions alternate. Only technical modifications to the representation of executions used in the already existing formalization of I/O automata had to be made. Here the LCF-part of HOLCF comes in — lazy lists are used to model possibly infinite sequences (a comparison of several methods to formalize infinite sequences can be found in the literature [23]). Consider the following type declarations:

$$\begin{array}{rrcl} \textbf{types} & (\alpha, \sigma, \tau_{\mathsf{timeD}})\,\mathsf{pairs} & = & ((\alpha, \tau)\,\mathsf{action} \times (\sigma, \tau)\,\mathsf{state})\,\mathsf{Seq} \\ & (\alpha, \sigma, \tau)\,\mathsf{execution} & = & (\sigma, \tau)\,\mathsf{state} \times (\alpha, \sigma, \tau)\,\mathsf{pairs} \\ & (\alpha, \tau)\,\mathsf{trace} & = & ((\alpha, \tau)\,\mathsf{action} \times \tau)\,\mathsf{Seq} \end{array}$$

Here Seq is a type of lazy lists specialized for a smooth interaction between HOL and HOLCF (for details see Olaf Müller's Ph.D. thesis [52]). An execution consists of a start state paired with a sequence of action/state pairs, whereas a trace is a sequence of actions paired with a time stamp (we chose not to include the end time reached in the execution which gave rise to the trace, since no information useful for actual verification work is added by it). All further definitions in the theory of timed I/O automata are formalized with functions and predicates over the types defined above. For example is-exec-frag A ex checks whether ex is an execution fragment of an automaton A. Its definition is based upon a continuous LCF-function $\mathsf{is\text{-}exec\text{-}frag}_{\mathsf{C}}$. To explain our formalization of an execution fragment, it suffices to display some properties of is-exec-frag which have been derived automatically from its definition:

$$\begin{array}{lrcl} \textbf{thm} & \mathsf{is\text{-}exec\text{-}frag}\ A\ (s, []) & & \\ \textbf{thm} & \mathsf{is\text{-}exec\text{-}frag}\ A\ (s, (a, r)\hat{}\, ex) & = & ((s, a, r) \in \mathsf{trans\text{-}of}\ A \wedge \mathsf{is\text{-}exec\text{-}frag}\ A\ (r, ex)) \end{array}$$

Here [] stands for the empty sequence and ^ for the cons-operation.

Further definitions include a function mk-trace which maps an execution to the trace resulting from it, executions and traces which give the set of executions and set of traces of an automaton, respectively.

Reachability can be defined via an inductive definition:

$$\textbf{inductive} \qquad \frac{s \in \mathsf{starts\text{-}of}\ C}{s \in \mathsf{reachable}\ C} \qquad \frac{s \in \mathsf{reachable}\ C \quad (s, a, t) \in \mathsf{trans\text{-}of}\ C}{t \in \mathsf{reachable} C}$$

Invariance Proofs and Simulation Proofs for Trace Inclusion Neither showing the correctness of the proof principle for invariants nor showing the correctness of simulation proofs required significant changes of the formalization for untimed I/O automata. An invariant is defined as

$$\textbf{defs} \qquad \mathsf{invariant}\ A\ P = \forall s.\ \mathsf{reachable}\ A\ s \implies P(s)$$

The principle of invariance proofs, namely

$$\textbf{thm} \qquad \frac{\begin{array}{l} \forall s \in \mathsf{starts\text{-}of}\ A.\ P(s) \\ \forall s\ a\ t.\ \mathsf{reachable}\ A\ s \wedge P(s) \wedge s \xrightarrow{a}_A t \implies P(t) \end{array}}{\mathsf{invariant}\ A\ P}$$

is easily shown by rule induction over reachable. The correctness of simulation proofs has been shown for three frequently used kinds of simulation relations, namely *refinement mappings*, *forward simulations* and *weak forward simulations*. Since it turned out that *weak forward simulations* are preferable in practice, we will restrict our presentation to this proof principle. All the simulation relations mentioned are based on the notion of a *move*: Let $s_{ex} \stackrel{\langle a,t \rangle}{\Longrightarrow}_A r$ denote an a-move from state s to state r in an automaton A by way of an execution fragment ex. The definition of a move requires, that the execution fragment ex has s as first and r as last state; further the trace arising from ex is either the sequence holding only $\langle a, t \rangle$ if a is a visible action, or the empty sequence if a is not visible. The intuition is that A performs an action a and possibly some invisible actions before and after that in going from s to r.

A forward simulation R between an automaton C and an automaton A basically says: any step $s \stackrel{a}{\longrightarrow}_C r$ from a reachable state s of C can be matched by a corresponding move of A:

$$
\begin{array}{lll}
\textbf{defs} & \text{is-w-simulation } R\ C\ A = & \\
 & \quad (\forall s\ u.\ u \in R[s] \Longrightarrow \text{now } u = \text{now } s) \wedge & \\
 & \quad (\forall s \in \text{starts-of } C.\ R[s] \cap \text{starts-of} A \neq \emptyset) \wedge & \\
 & \quad (\forall s\ s'\ r\ a.\ \text{reachable } C\ s & \\
 & \qquad \wedge\ s \stackrel{a}{\longrightarrow}_C r & \\
 & \qquad \wedge\ (s, s') \in R & \\
 & \qquad \wedge\ \text{reachable } A\ s' & \\
 & \qquad \Longrightarrow \exists r'\ ex.\ (r, r') \in R\ \wedge\ {s'}_{ex} \stackrel{\langle a, \text{now}\ r' \rangle}{\Longrightarrow}_A r') &
\end{array}
$$

The first line of the given definition requires a weak forward simulation R to be *synchronous*, i.e., to relate only states with the same time stamp ($R[s]$ denotes the image of s under R). The second line requires the set of all states related to some start state s of C to contain at least one start state of A. The requirement that s' be reachable in A and s reachable in C characterizes a *weak* forward simulation.

The corresponding proof principle, which has been derived in Isabelle, is

$$
\textbf{thm} \qquad \frac{\text{visibles } C = \text{visibles } A \quad \text{is-w-simulation } R\ C\ A}{\text{traces } C \subseteq \text{traces } A}
$$

At first it is surprising that showing the correctness of simulation relations required hardly any changes to the corresponding proof for untimed I/O automata. In particular no information about the time domain is needed. The point is that *synchronous* simulation relations restrain time-passage so much that time-passage actions do not differ significantly from other invisible actions in this context.

2.4.3 Simulation proofs for Execution Inclusion

In a case study carried out with timed I/O automata by Heitmeyer and Lynch [38], inclusion of executions under projection to a common component automaton is derived from the existence of a simulation relation between two automata. In doing so, the authors refer to general results about composition of timed automata. When trying to formalize this step we realized that one of the automata involved cannot be regarded as constructed by parallel composition. Therefore results about the composition of timed automata cannot be directly applied; further the very notion of projecting executions to a component automaton is not well-defined.

In order to formalize the meta-theory necessary for carrying out this particular case study, we tried to find a suitable reformulation of the employed concept, which is as well interesting in its own right. The basic idea is to find a notion of projecting executions to parts of an automaton which is compatible with the concept of simulation proofs: inclusion of executions under projection to a common part of two automata should be provable by exhibiting a simulation relation.

In which setting will execution inclusion with respect to some projection be employed? Say timed I/O automata are to be used for specifying a controller which influences an environment. Both the environment and the implementation of the controller will be modelled using automata — their parallel composition C describes the actual behavior of the controlled environment. Trying to show correctness via the concept of simulation proofs, the *desired* behavior of the controlled environment will be specified as another automaton A. For constructing A, the automaton specifying the environment is likely to be reused, i.e., A arises from modifying this automaton such that its behavior is restricted to the desired behavior. Hence the state space of A will contain the environment's state space, which can be extracted by a suitable projection.

The notion of observable behavior one wants to use in this case might very well be executions rather than traces: executions hold information also about states, and it often is more natural to define desired behavior by looking at the states rather than visible actions only. What one wants to do is to define projections for both automata, which extract the environment. Using these projections, executions of both automata can be made comparable. As the following theorem shows, a simulation indeed implies inclusion of executions under the given projections, if these fulfill certain requirements:

$$\textbf{thm}\quad \frac{\begin{array}{l}\mathsf{is\text{-}w\text{-}simulation}\ R\ C\ A\\ \forall u\ u'.\ (u,u') \in R \Longrightarrow p_c\, u = p_a\, u'\\ \forall a\ s\ s'.\ a \notin actS \wedge s \xrightarrow{\triangleleft a \triangleright}_A s' \Longrightarrow p_a\, s = p_a\, s'\\ actS \subseteq \mathsf{visibles(sig\text{-}of}\ A) \cap \mathsf{visibles(sig\text{-}of}\, C)\end{array}}{\begin{array}{ll} ex \in & \mathsf{executions}\ C \Longrightarrow\\ & \exists ex'.\ \mathsf{exec\text{-}proj}\ A\ p_a\ actS\ ex' = \mathsf{exec\text{-}proj}\ C\ p_c\ actS\ ex\end{array}}$$

Here p_c and p_a are projections from the state space of C and A on a common subcomponent. The set $actS$ is a subset of the shared visible discrete actions of C and A (see the fourth premise). The third premise of the theorem expresses that $actS$ has to include all the actions which in A (describing the desired behavior) affect the part of the state space that is projected out. The second premise gives a wellformedness condition of the simulation relation R with respect to p_c and p_a — only states which are equal under projection may be related.

It remains to clarify the rôle of exec-proj (actually defined by an elaborate continuous function $\mathsf{exec\text{-}proj}_c$; all the following properties of exec-proj have been proven correct by reasoning over $\mathsf{exec\text{-}proj}_c$ within the LCF-part of HOLCF). The function exec-proj in effect defines a new notion of observable behavior based on executions. Certainly one would expect exec-proj $A\ p\ actS$ to use projection p on the states of a given execution and to remove actions not in $actS$:

$$\begin{array}{ll}\textbf{thm} & a \in actS \Longrightarrow \mathsf{exec\text{-}proj}\ A\ p\ actS\ (s, (\triangleleft a \triangleright, s')\hat{}\, xs)\\ & \qquad = (p\ s, (\triangleleft a \triangleright, p\ s')\hat{}(\mathsf{snd}\ (\mathsf{exec\text{-}proj}\ A\ p\ actS\ (s', xs))))\\ \textbf{thm} & a \notin actS \Longrightarrow \mathsf{exec\text{-}proj}\ A\ p\ actS\ (s, (\triangleleft a \triangleright, s')\hat{}\, xs)\\ & \qquad = (p\ s, \mathsf{snd}\ (\mathsf{exec\text{-}proj}\ A\ p\ actS\ (s', xs)))\end{array}$$

However it is also necessary to abstract over subsequent time-passage steps:

$$\begin{array}{ll}\textbf{thm} & \mathsf{exec\text{-}proj}\ A\ p\ actS\ (s, (\nu(t'), s')\hat{}(\nu(t''), s'')\hat{}\, xs)\\ & \qquad = \mathsf{exec\text{-}proj}\ A\ p\ actS\ (s, (\nu(t' \oplus t''), s'')\hat{}\, xs)\end{array}$$

By removing actions that are not in *actS* it can happen that time-passage steps which before were separated by discrete actions appear in juxtaposition. Hence we further require that

$$\textbf{thm} \quad a \in actS \Longrightarrow \mathsf{exec\text{-}proj}\ A\ p\ actS\ (s, (\nu(t'), s')\hat{}(\triangleleft a \triangleright, s'')\hat{}xs)$$
$$= (p\ s, (\nu(t'), p\ s')\hat{}(\triangleleft a \triangleright, p\ s'')\hat{}(\mathsf{snd}\ (\mathsf{exec\text{-}proj}\ A\ p\ actS\ (s, xs))))$$
$$\textbf{thm} \quad a \notin actS \Longrightarrow \mathsf{exec\text{-}proj}\ A\ p\ actS\ (s, (\nu(t'), s')\hat{}(\triangleleft a \triangleright, s'')\hat{}xs)$$
$$= \mathsf{exec\text{-}proj}\ A\ p\ actS\ (s, (\nu(t'), s')\hat{}xs)$$

Notice that the notion of observable behavior implied by exec-proj may not be suitable for all applications. It suffices for cases in which time-passage changes nothing but the time-stamp in an automaton's state. Only because of this we can abstract away from subsequent time-passage steps: all the states dropped by this abstraction differ only in their time stamp, i.e., no information is lost.

The proof of the correctness theorem displayed above is rather involved. However for meta-theoretic proofs, which only have to be carried out once and for all, complicated proofs are acceptable. The interaction of HOL and LCF in the formalization of (timed) I/O automata is such that users can carry out actual specifications and their verification with (timed) I/O automata strictly in the simpler logic HOL. For the proof we also required assumptions about timeD — as mentioned above it was sufficient to know that every type τ in class timeD is an ordered group with respect to $\langle \tau, \oplus, \ominus, \mathbf{0}, \leq \rangle$.

2.4.4 A Case Study in Reasoning with Timed I/O Automata

Building upon the meta-theory introduced above, substantial parts of a solution to the so-called *Generalized Railroad Crossing* (GRC) as presented by Heitmeyer and Lynch [38] was formalized within Isabelle/HOLCF. A formalization undertaken in PVS [2] did not treat meta-theoretical questions and formalized only invariance proofs — a simulation proof which is central to the solution has not been covered.

Carrying out the correctness proof of the GRC, we found ourselves constantly performing invariance proofs (see section 2.4.2). Indeed, showing invariants of automata seems to be the proof principle used most frequently when doing verification work within the theory of (timed) I/O automata. This is because firstly safety properties often are invariants, and secondly because invariants are used to factorize both invariance and simulation proofs: lemmas usually are formulated as invariants. Factorizing proofs is not only important for adding clarity, but in the context of interactive theorem proving also for keeping the intermediate proof states of a manageable size.

The practicability of invariants even suggests to prefer weak simulation relations in favour of forward simulations. Lynch and Vaandrager [46] show that the notions of forward simulation and weak forward simulation are theoretically equivalent, i.e., whenever a forward simulation can be established, there exists also a weak forward simulation and vice versa. However, since for establishing the latter only *reachable* states of both automata have to be considered, invariants can be used to factor out information which otherwise would have to be made explicit in the simulation relation.

Having established that invariance proofs may be regarded as the most important proof principle, it is clear that a high degree of automation for invariance proofs will be of considerable help. The guiding principle is that at least those parts of invariance proofs which in a paper proof would be considered as "obvious" should be handled automatically.

The proof rule for invariance proofs has been given in section 2.4.2. One has to show that an invariant holds for each start state, which usually is trivial, and that any transition from a reachable state for which the invariant holds leads again into a state which satisfies the invariant. This is done by making a case analysis over all the actions of an automaton. In a paper proof certain cases will be regarded as "trivial", e.g., when a certain action will always invalidate the premise of the invariant to be shown.

It turned out that Isabelle's generic simplifier and its classical reasoners [64, 65] would usually handle all the trivial cases and even more fully automatically when provided with information about the effect of an action in question. For a primitive automaton, i.e., an automaton not built up by parallel composition, a theorem providing such information is easy to formulate and usually proven automatically. For composite automata, care has to be taken to keep the proof state of moderate size. This can be achieved by using specially derived rules which from the definition of parallel composition between two automata derive the effect of parallel composition between more automata.

An invariance proof can now be carried out as follows. One starts with a special tactic which sets up an invariance proof by showing that the invariance holds for the start states and then performs a case analysis over the different actions. For each of these cases one first supplies information about the effect of the action in question with the respective lemma and then uses Isabelle's simplifier and classical reasoner. This will often solve the case completely. Otherwise the resulting proof obligation will be simplified as far as possible for the user to continue by interactive proof.

A serious drawback of the current version of Isabelle is its lack of support for arithmetical reasoning — carrying out arithmetical reasoning by hand is not only cumbersome, but also hinders the use of built-in solvers. Because of the latter point, adding provers for arithmetic should enhance the automatization of proofs about timed I/O automata significantly.

More details about our treatment of the GRC using timed I/O automata in Isabelle/HOLCF can be found in a joint paper with Olaf Müller [30].

Chapter 3

Types for Programs

The notion of types probably was used for the first time when Russell, as mentioned before, introduced a layering into a universe of mathematical objects. By forbidding the use of mathematical objects in a way that violated this layering, an antinomy like the one in Frege's system could be avoided. In effect, a typing system was used to fine-tune the ways in which the system could be used, thus avoiding troublesome cases.

The same holds true for the initial motivation to use types in programs: Milner's statement "*Typed programs don't go wrong*" [49] has become somewhat of a mantra; Cardelli and Wegner draw an analogy between types and *a protecting armor* [9] — the basic argument is in both cases that types prevent certain misuses. Milner's statement is best suited for static type systems: if a program type checks at compile time, one has certain guarantees about the dynamic behavior of the program. From the programmer's point of view, static type systems enable "static debugging" (and, if there is a mechanism for automatic type inference, the information gained via type inference usually proves to be very helpful while programming). Dynamic type systems provide a somewhat weaker guarantee about a program's dynamic behavior: certain 'bad' behavior at execution time will be prevented, e.g., by making the program abort with a run-time error.

The underlying theme of using types in programming languages can be summed up as follows: introducing a type system reduces the number of possible programs that can be written. However, along with this reduction comes some additional knowledge about the legal programs. The development of new type systems often means finding a balance between being not too restrictive, and yet providing enough information to be usefully exploited. This balance is addressed for example in Schwartzbach's lecture notes on polymorphic type inference [69].

In a statically typed functional language, the type of an expression provides information about the intended use of the value it denotes. For example, a function type characterizes the way a function can be used, namely for mapping a value from the function's domain to the codomain. Numerous examples of exploiting the information provided by an expression's type, e.g., for program optimization, can be found in the literature [50, 66, 71]. We will focus on the following application of this principle: for a restricted class of values, one can write a *type-indexed* function *reify*. For any value v in this restricted class, an instance of *reify* for v's type will, when applied to v, return the representation of an expression in normal form with denotation v. This process is an instance of a concept called *normalization by evaluation* (NbE).

NbE can be used to perform a technique called *partial evaluation* (PE) differently from the traditional approaches; we speak of *type-directed partial evaluation* (TDPE). In the following we

first introduce the basic concepts of PE and NbE. Having done this, we show how TDPE uses NbE to perform partial evaluation. Finally we will report on work which examines how the so-called second Futamura projection, a PE concept that will be explained below, can be instantiated for TDPE.

3.1 Partial Evaluation

There is much to be said about partial evaluation — here we will only sketch some basics, explaining the *what* and giving only the vaguest idea about the *how*. A complete account of both the concepts of partial evaluation and many of its techniques can be found in Jones, Gomard and Sestoft's textbook [40]. A concise survey can also be found in Consel and Danvy's tutorial notes [11].

3.1.1 The Basic Concept

Partial evaluation is a technique for program optimization. It works by specializing programs with respect to parts of their input, the so-called *static* input. This process effectively stages computation: Computations which depend only on the static input are carried out during partial evaluation. At the same time, residual code is produced for computations dependent on *dynamic* input, i.e., input which is unknown at partial evaluation time.

Partial evaluation is carried out automatically by so-called *partial evaluators*. The extent to which PE pays off depends largely on how much can be precomputed during partial evaluation and how often the specialized code is used. If there is a lot of computation to be done that depends on the static input only, or the specialized program is used very often, then specialization and subsequent use of the specialized version of a program can improve performance significantly.

3.1.2 A Simple Example of Partial Evaluation

The standard example of partial evaluation is specializing the power function, where power (x, y) calculates x^y. Here is an implementation of power in an ML-like language:

$$\begin{array}{ll} \textbf{fun} & \mathsf{power}\ (x, 0) = 1 \\ \mid & \mathsf{power}\ (x, n) = x * (\mathsf{power}\ (x, n-1)) \end{array}$$

The function power takes two arguments. Specializing it with respect to the second parameter, e.g., to the value 3, could lead to a residual program of form

$$\textbf{fun}\quad \mathsf{power\text{-}d\text{-}3}\ x = x * x * x$$

In this case, partial evaluation seems to pay off: a considerable amount of calculation could be precomputed, producing a residual program without any recursive calls. However, the result of specializing with respect to the first argument, again to the value 3, is rather un-exciting:

$$\begin{array}{ll} \textbf{fun} & \mathsf{power\text{-}3\text{-}d}\ 0 = 1 \\ \mid & \mathsf{power\text{-}3\text{-}d}\ n = 3 * (\mathsf{power\text{-}3\text{-}d}\ (n-1)) \end{array}$$

Without knowledge about the second argument to power, there is hardly any computation that can be carried out — inlining the value of the first argument is about all that can be done.

3.1.3 Correctness of Partial Evaluation

Intuitively it is quite clear what correctness of partial evaluation means: specializing a program to a part of the input and then executing the residual program over some remaining input should yield the same result as executing the original program over the complete input, provided that both cases terminate.

In order to give an equational account of this correctness requirement, we write $[\![\mathsf{prog}]\!]$ for the denotation of a program(text) prog. For partial evaluation we consider programs whose input can be divided into two parts, namely a static part s and a dynamic part d. Hence running prog on the total input is written as $[\![\mathsf{prog}]\!]\ \langle s, d\rangle$. Let prog_s denote the specialization of prog to the static input s. Then the correctness requirement for specialization can be expressed as

$$[\![\mathsf{prog}_s]\!]\ d\ =\ [\![\mathsf{prog}]\!]\ \langle s, d\rangle \qquad (*)$$

Say that the program pe performs partial evaluation, i.e., $[\![\mathsf{pe}]\!]\ \langle \mathsf{prog}, s\rangle = \mathsf{prog}_s$. Combining this with $(*)$ yields

$$[\![[\![\mathsf{pe}]\!]\ \langle \mathsf{prog}, s\rangle]\!]\ d = [\![\mathsf{prog}]\!]\ \langle s, d\rangle \qquad (**)$$

as requirement for the correctness of pe.

Notice that for an ML program prog, the expression

$$\textbf{fn}\ x \Rightarrow \mathsf{prog}\ (s, x)$$

trivially fulfills the correctness requirement for a specialization prog_s — the real challenge of partial evaluation is to find non-trivial specializations, for which the parts of the computation that depend only on the static input have been carried out.

3.1.4 Applications of Partial Evaluation

The example of the power function given in section 3.1.2 suggests that a program "specializes well" if its control flow depends largely on the static input. This situation can be understood as the program "interpreting" the static input. Hence partial evaluation is particularly well suited to removing an interpretive overhead.

In many applications of partial evaluation, the interpretive overhead to be removed is easily identified. Take for example functions for formatting and reading like `printf` and `scanf` in C. They are passed a *control string* which determines their behavior. If such functions are to be used for writing or reading a large amount of data line by line, specializing them will certainly pay off. A similar situation arises with a number of Unix utilities such as `grep` or `awk`.

Even if the control flow of a program depends not solely on the static input, partial evaluation can pay off: as we saw in section 3.1.2, partial evaluation if nothing else performs in-lining. Going even further and specializing a program to *no* input may still result in a simplified program. Of course usually no large simplifications are to be expected for programs that have been written in one piece with at least some thought. However, programs that have been generated or are the result of combining a number of modules may offer real possibilities for simplification. The same holds true for programs in which a large number of macros have been used, since macro expansion often produces code which is amenable to simplification.

In general, programs that are well-suited for reuse are often also well-suited for partial evaluation. This is because reuse is often achieved by making programs general and programming them

in a modular way. As there is a clear trend to practice more code reuse, the relevance of partial evaluation is likely to grow.

Other applications of partial evaluation often require a thorough analysis of the problem at hand to assess circumstances in which partial evaluation may pay off. Partial evaluation has for example been successfully applied to numerical computing [28] and ray-tracing [1] — in both cases it was initially not clear at all how partial evaluation could help.

In the next section we turn to a classical application of partial evaluation: compiling and compiler generation.

3.1.5 The Futamura Projections

The possibility to use partial evaluation for compilation and compiler generation has caused a lot of interest into the topic. Since partial evaluation can remove an *interpretive* overhead, it seems logical to examine the partial evaluation of an interpreter with respect to a source program. Observe that an interpreter for a programming language $\mathcal{L}$ can be understood as a function taking two arguments: a source program to be interpreted and some input for this program. Let $[\![\cdot]\!]_{\mathcal{L}}$ denote a semantic function assigning meanings to programs written in $\mathcal{L}$. A defining equation for an interpreter interpreter is then

$$[\![\mathsf{interpreter}]\!]\ \langle \mathsf{source}, \mathit{input}\rangle = [\![\mathsf{source}]\!]_{\mathcal{L}}\ \mathit{input}$$

Using a partial evaluator to specialize the interpreter to some source input, we can thus conclude (using $(**)$) that

$$[\![[\![\mathsf{pe}]\!]\ \langle \mathsf{interpreter}, \mathsf{source}\rangle]\!]\ \mathit{input} = [\![\mathsf{source}]\!]_{\mathcal{L}}\ \mathit{input}$$

We see that the result of specializing an interpreter with a source program has the same denotation as the source program itself under the semantics for $\mathcal{L}$-programs. Consider now what a compiler does: When compiling some source to a target, the following equation should hold:

$$[\![\mathsf{target}]\!]\ \mathit{input} = [\![\mathsf{source}]\!]_{\mathcal{L}}\ \mathit{input}$$

By equational reasoning and extensionality follows the *first Futamura projection*:

$$[\![\mathsf{pe}]\!]\ \langle \mathsf{interpreter}, \mathsf{source}\rangle = \mathsf{target}$$

Recall from section 3.1.3 that the equational treatment of partial evaluation is only half the story, since it only captures the extensional behavior of the resulting residual program. However since the control flow of an interpreter usually depends on the program to be interpreted, one can expect that all computations dealing with destructing the source program are carried out during partial evaluation. Thus we indeed can regard $[\![\mathsf{pe}]\!]\ \langle \mathsf{interpreter}, \mathsf{source}\rangle$ as a compiled program.

Using a similar line of reasoning, one arrives at the *second Futamura projection*

$$[\![\mathsf{pe}]\!]\ \langle \mathsf{pe}, \mathsf{interpreter}\rangle = \mathsf{compiler},$$

i.e., specializing a partial evaluator with respect to an interpreter for some language $\mathcal{L}$ yields a compiler for $\mathcal{L}$. The *third Futamura projection* highlights a link between partial evaluation and compiler generation:

$$[\![\mathsf{pe}]\!]\ \langle \mathsf{pe}, \mathsf{pe}\rangle = \mathsf{compiler\text{-}generator}$$

Notice that both the second and the third Futamura projection require an application of a partial evaluator to itself. See for example Consel and Danvy's tutorial notes [11] for a more general account on self application in partial evaluation, with the Futamura projections as a corollary.

3.1.6 Online and Offline Partial Evaluation

The first partial evaluators to be written were so-called *online* partial evaluators. Online partial evaluation works by interpreting a program text over the static and dynamic input in a non-standard way: the partial evaluator always keeps track of which values are static and which are dynamic. A computation that only depends on static values is carried out while for all other computations code is generated (residualizing the involved static values into syntax).

Online partial evaluation offers the most finely grained distinction between static and dynamic values that is possible. This however comes at a cost. Since for any function taking n arguments, 2^n cases have to be considered (any argument may be static or dynamic), online partial evaluators are rather large, consisting of numerous case distinctions. While this is acceptable for normal applications, self application of online partial evaluators turns out to be infeasible.

By developing a technique called *offline* partial evaluation, Jones, Sestoft and Søndergaard [41] were able to write the first self-applicable partial evaluator in a language of recursive equations, which later gave rise to the Mix system [42]. The basic idea behind offline partial evaluation is to stage the process of partial evaluation itself into (1) a so-called binding-time analysis and (2) the specialization of a program annotated with binding-time information.

Binding-time analysis pre-computes information about which values are static and which are dynamic — online partial evaluation determines this on the fly. Hence a partial evaluator operating on a program with binding-time annotations is much more lightweight than a online partial evaluator. This makes it more feasible to attempt self-application for offline partial evaluators. Notice that self application is performed by applying the partial evaluator to the binding-time-annotated source program of the partial evaluator.

3.2 Type-Directed Partial Evaluation

TDPE can be seen as the combination of two concepts. The first concept could be called *partial evaluation by normalization*: the basic observation is that under certain circumstances partial evaluation can be carried out by normalization.

The second concept is called normalization by evaluation. It provides one with a very speedy normalization function, and is thus of interest to any area of computer science in which normalization plays a role. NbE has for example been examined in the context of proof theory [4, 5]. See the proceedings of the first APPSEM workshop on NbE [19] for an overview of various other treatments.

TDPE is the instantiation of the first concept with NbE as normalization function. It is due to Danvy [15, 14, 16] — an introductory account can be found in Danvy's lecture notes on TDPE [18].

In the following we will introduce the concept of NbE for the pure simply typed λ-calculus and discuss the implementation of NbE in ML. We then examine the connection between normalization and partial evaluation in the restricted setting of the pure simply typed λ-calculus and give an example of using NbE for partial evaluation. Finally we sketch how the resulting concept of TDPE can be extended to richer functional languages.

3.2.1 A Two-Level λ-Calculus

Since the λ-calculus forms the basis of any functional programming language, we explain the concept of normalization by evaluation in terms of a two-level pure simply typed λ-calculus. *Two-level* means that the syntactic constructs of the λ-calculus have been duplicated — we distinguish them with over-lines and under-lines, where overlined and underlined constructs are called *static* and *dynamic*, respectively.

$$e ::= x \mid \overline{\lambda} x.e \mid e_0 \overline{@} e_1 \mid \underline{\lambda} x.e \mid e_0 \underline{@} e_1$$

The dynamic level models the syntax of a programming language. Dynamic β-reduction, which is given by the rule

$$(\underline{\lambda} x.e_1)\underline{@} e_0 \longrightarrow e_1[x \leftarrow e_0],$$

thus corresponds to a symbolic manipulation on expressions. The static level on the other hand models values — hence static β-reduction, as given by

$$(\overline{\lambda} x.e_1)\overline{@} e_0 \longrightarrow e_1[x \leftarrow e_0],$$

corresponds to evaluation.

3.2.2 Reification and Reflection

The process of turning a static construct into a dynamic one is called *reification*. *Reflection* goes the other way, turning a dynamic construct into a static one. For the two-level λ-calculus introduced above, we can specify both transformations in a type-directed way for any type scheme (where '$\bullet$' stands for any base type):

$$\begin{array}{rcll} \downarrow^{\bullet} e & = & e & \\ \downarrow^{\tau_1 \to \tau_2} e & = & \underline{\lambda} x.\downarrow^{\tau_2} (e \overline{@} (\uparrow_{\tau_1} x)) & \text{where } x \text{ is fresh} \end{array}$$

$$\begin{array}{rcll} \uparrow_{\bullet} e & = & e & \\ \uparrow_{\tau_1 \to \tau_2} e & = & \overline{\lambda} x.\uparrow_{\tau_2} (e \underline{@} (\downarrow^{\tau_1} x)) & \text{where } x \text{ is fresh} \end{array}$$

For our purposes it suffices to consider what happens when reifying a completely static term (that is to turn a value into an expression). Let's look at a couple of examples:

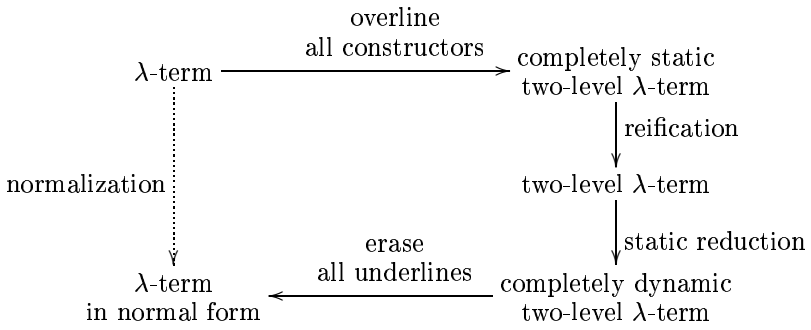
$$\downarrow^{\bullet \to \bullet} e = \underline{\lambda} x_1.e \overline{@} x_1 \tag{1}$$

$$\downarrow^{\bullet \to \bullet \to \bullet} e = \underline{\lambda} x_1.\underline{\lambda} x_2.(e \overline{@} x_1) \overline{@} x_2 \tag{2}$$

Instantiating the static expression e in (1) with the static identity function $\overline{\lambda} x.x$ and performing static β-reduction yields $\underline{\lambda} x_1.x_1$ as result. Instantiating the e in (2) with the static K-combinator $\overline{\lambda} x.\overline{\lambda} y.x$, the result after static β-reduction is $\underline{\lambda} x_1.\underline{\lambda} x_2.x_1$. The very same result is obtained for $\overline{\lambda} x.\overline{\lambda} y.(\overline{\lambda} z.z)x$, which is β-equivalent to $\overline{\lambda} x.\overline{\lambda} y.x$. This seems to suggest that reification and subsequent static β-normalization not only turns a static term into a dynamic term — the resulting dynamic term is furthermore in normal form with respect to *dynamic* β-reduction.

3.2.3 The Concept of Normalization by Evaluation

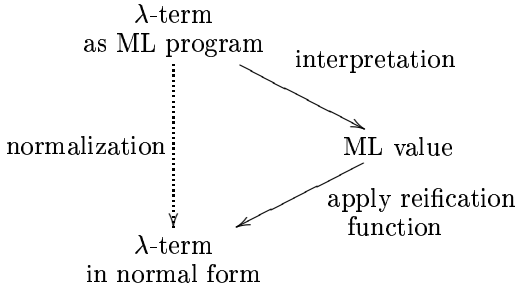
It can be shown that reifying a completely static term t_s followed by static β-normalization results in a completely dynamic term t_d. Furthermore, forgetting about the over- and under-lines, t_d is the long β-η-normal form of t_s. The following diagram taken from Danvy's lecture notes [18] illustrates how this property of reification can be used to perform normalization by evaluation:



Some intuition about what is happening can be gained by observing that reification with respect to a certain type performs a (two-level) η-expansion "around" the expression e to be reified. The subsequent static β-reduction will "execute" e. It is because of this execution of e that β-η-equivalent terms give the same result — only the *extension* of e matters.

3.2.4 Naive Normalization by Evaluation in ML

In section 3.2.1 the two-level λ-calculus was motivated by drawing an analogy between dynamic terms and expressions on one hand, and static terms and values on the other hand. If reflection and reification as introduced in section 3.2.2 can be coded in ML, then reification gives one the possibility to *decompile* a simply typed value into syntax. Furthermore the resulting expression is in normal form. The diagram from section 3.2.3 can be recast as



In effect, normalization is carried out using the ML evaluation engine on the program to be normalized instead of doing syntactic manipulations on a representation of the program's syntax. Obviously a clear gain in performance could be expected.

Reification and reflection are type-indexed functions. This poses a problem for their implementation in ML, since ML does not offer dependent types. One possible solution is using an implementation technique due to Filinski and Yang [17, 76]; see also Rhiger's derivation [68]. For every type constructor $\mathcal{T}$, a combinator is written, which takes a *pair* of reification and reflection

```
structure Exp =
    struct
        datatype exp =
            VAR of string | LAM of string * exp | APP of exp * exp
          | PAIR of exp * exp | PFST of exp | PSND of exp
    end

structure Gensym
= struct
    local val n = ref 0
    in fun init () = n := 0
       fun new base = (n := !n + 1; base ^ Int.toString (!n))
    end
  end;

structure naive_Nbe =
    struct
        local open Exp
        in datatype 'a rr = RR of ('a -> exp) * (exp -> 'a)

            val rra = RR (fn e => e, fn e => e)

            fun rrf (RR (reify1, reflect1), RR (reify2, reflect2))
                = RR (fn f => let val x = Gensym.new "x"
                              in LAM (x,reify2 (f (reflect1 (VAR x))))
                              end,
                      fn e => fn v => reflect2 (APP (e,reify1 v)))

            fun rrp (RR (reify1, reflect1), RR (reify2, reflect2))
                = RR (fn (v1,v2) => PAIR (reify1 v1, reify2 v2),
                      fn e => (reflect1 (PFST e), reflect2 (PSND e)))

            fun reify (RR (reify',reflect')) v =
                  (Gensym.init (); reify' v)

            fun reflect (RR (reify',reflect')) v =
                reflect' v
        end
    end
```

Figure 3.1: Naive NbE

functions for every parameter to $\mathcal{T}$, defining the reification-reflection pair for $\mathcal{T}$ in terms of this input. Thus a reification function for a certain type τ can be created by combining the combinators according to the structure of τ and projecting out the reification function from the resulting function pair.

Figure 3.1 gives the implementation of naive NbE in ML as presented in Danvy's lecture

note [18]. In addition to what has been described so far, the given implementation of NbE can also deal with pairing — the extension is straightforward. Structure `Exp` defines a data type for representing expressions, structure `Gensym` defines the basic functions needed for generating fresh symbols. Structure `naive_Nbe` finally defines naive NbE: `rra`, `rrf` and `rrp` are the combinators which correspond to base type, function type and pair type, respectively. The functions `reify` and `reflect` project the reification resp. reflection function out of a function pair built up using these combinators.

Defining `-->` as infix operator for `rrf` and `a'` as shortcut for `rra`, we can for example write $\downarrow^{\bullet\to\bullet} v$ as `naive_Nbe.reify (a' --> a') v` in ML. From now on we take $\downarrow^{\tau} v$ to mean the reification of an ML value v into an expression and $\uparrow_{\tau} e$ the reflection of an expression e, where e and v fit the type scheme τ. Obviously the output of a reification in ML is of type `Exp.exp` — we however write output pretty-printed as ML syntax. ML values are written in λ-notation, ML syntax in typewriter font.

3.2.5 Partial Evaluation by Normalization in the Pure Simply Typed λ-Calculus

As pointed out in section 3.1.1, when partially evaluating a program, one tries to pre-compute as much as possible. For the pure simply typed λ-calculus, a reduction to β-normal form seems like a natural choice for a notion of precomputation. A binding-time analysis is unnecessary — top-level abstractions will bind dynamic input to variables, which normalization passes around until they are to be used. Since the only way of 'use' in the pure simply typed λ-calculus is to use something as a function, further normalization will stop automatically, because applying a variable to something does not constitute a β-redex. Hence partial evaluation for the pure simply typed λ-calculus corresponds to applying the static input to the expression which is to be partially evaluated. One may have to use a "wrapping" expression which reorders the arguments such that all the static input is taken before the dynamic input.

Let us for example specialize $\mathsf{p} := \lambda x.\lambda f.f@x$ with respect to $\mathsf{id} := \lambda x.x$ in its second argument. This can be done by normalizing $(\mathsf{C}\ \mathsf{p})\mathsf{id}$, where $\mathsf{C} := \lambda f.\lambda x.\lambda y.f@y@x$ is applied to p to change the order in which the arguments are taken. The result of normalization is $\lambda x.x$.

3.2.6 Using NbE for Partial Evaluation in the Pure Simply Typed λ-Calculus

Having established that partial evaluation in the pure simply typed λ-calculus corresponds to normalization, we can use NbE as normalization function. For the ML implementation, one makes the top-level definitions

```
val p     =  fn x => fn f => f x
val id    =  fn x => x
val C     =  fn f => fn x => fn y => f y x
```

and then uses reification, which carries out the specialization via normalization:

$$\downarrow^{\bullet\to\bullet} \texttt{(C p id)} = \texttt{fn x0 => x0}$$

To put it in a nutshell: partial evaluation for the pure simply typed λ-calculus can be performed by applying the function to be specialized to the static input and reifying the result.

3.2.7 Partial Evaluation via NbE for General ML Programs

We have seen that NbE can be used to perform partial evaluation in the pure simply typed λ-calculus. It works so nicely because in this setting partial evaluation boils down to normalization.

For general ML programs, several problems arise, both in extending the idea of using NbE for partial evaluation and in proving the correctness of the resulting scheme (see [25] for an in-depth discussion):

- In the pure λ-calculus, the only notion of computation is β-reduction. General ML programs however use a number of primitives which perform computation. In order to receive good results, i.e., performing as much static computation as possible during partial evaluation, NbE has to be set up in a way such that static computation involving primitives is carried out, while residual code is produced for computation which depends on dynamic input.
- Whereas in the pure λ-calculus β-reduction always leads to a normal form eventually, ML or Haskell programs can diverge. Hence the concept of carrying out partial evaluation by normalization has to be adjusted.
- NbE is formally defined in a normal-order setting. ML however is a call-by-value language with computational effects. It is known that β-reduction is not a sound transformation rule for such languages. Hence NbE has to be modified such that only valid reductions are performed.

In the following we sketch how the problems mentioned above have been solved. The resulting scheme has been proven correct by Filinski [26] for call-by-name and call-by-value PCF using domain theoretical reasoning. From now on $\downarrow^\tau$ and $\uparrow_\tau$ will refer to the ML implementation of reification and reflection for call-by-value.

Dealing with primitives As pointed out above, primitives whose arguments depend on static input only should perform computation during partial evaluation. This can be achieved by making a binding-time analysis of the program to be partially evaluated. All the primitives which only depend on static input will be left unchanged — since partial evaluation is performed via NbE, the computation will be carried out in the evaluation process. Primitives which depend on dynamic input on the other hand will be substituted by code-generating functions, using reflection. By using ML functors, the instantiation of primitives with code-generating functions can be carried out in a structured way.

Coping with the loss of strong normalization The additional expressive power over the pure λ-calculus offered by recursive definitions or, equivalently, the presence of fixed point operators in ML, leads to the possibility of non-termination.

Consider the example from section 3.1.2 of specializing the power function. When specializing the first argument, the recursion is completely static and is guaranteed to terminate. Hence it can be left implicit or rewritten in terms of a static fixed point operator — during partial evaluation it will be unrolled automatically. However, when specializing to the second argument, the depth of the recursion depends on dynamic input. In this case, the recursion has to be made explicit by using a fixed point operator. It is introduced like any other primitive and, since it depends on dynamic input, substituted by a code-generating function for partial evaluation.

In the case of the power function, we were lucky: the recursion we classified as static indeed terminates when unrolled during partial evaluation. Furthermore, we correctly classified the recursion as dynamic when specializing with respect to the second argument — unfolding it would

lead to non-termination. This suggests that one way of dealing with the possibility of divergence is to equip the binding-time analysis with a (necessarily conservative) termination analysis.

For the time being, the practiced approach is to shift the responsibility to the programmer, who either has to declare recursions as static or dynamic, or puts guards in the definitions of his fixed point operators. The latter technique is illustrated in Danvy's lecture notes [18].

NbE for cbv-languages with computational effects The rule of β-reduction is not valid in cbv-languages with computational effects. This is because computation which is carried out exactly once under a call-by-value regime may either not be carried out at all or be carried out several times. Dynamic let-insertion [7, 37, 44] solves this problem — it can be implemented using control operators such as *shift* and *reset* [20, 24].

3.2.8 The Status Quo of TDPE

TDPE offers two main advantages. Firstly, it is very efficient — in an application to compiling actions [21] by partially evaluating an interpreter for Action Semantics [51], TDPE performed substantially better than a syntax-directed technique. Secondly, the implementation is provably correct.

However in comparison with traditional methods for partial evaluation, TDPE is still rather restricted — like lambda-Mix, it performs for example only monovariant program-point specialization [40, chapters 4,8]. Current research focuses both on overcoming such restrictions and on understanding the relationship between TDPE and traditional techniques of partial evaluation. Another line of research tries to exploit the fact that the output of TDPE is in normal form. This can for example be advantageous for compiling to machine language as shown in an application of TDPE to runtime code generation [3].

3.3 Instantiating the 2nd Futamura Projection for TDPE

As was mentioned in section 3.1.5, the Futamura projections can be generalized. Consider for example the 2nd Futamura projection for any program p instead of an interpreter:

$$[\![\mathsf{pe}]\!]\ \langle \mathsf{pe}, \mathsf{p}\rangle = \mathsf{pe}_{\mathsf{p}} \qquad (***)$$

Here pe_{p} is the so-called *generating extension* [40, chapter 5] of p. When passed some static input s, the program pe_{p} generates the code of p_s. Equation (***) says that a generating extension for some program p can be obtained by specializing a partial evaluator to p.

In the following we describe joint work with Zhe Yang (Danvy and Rhiger developed a scheme to apply the second Futamura projection to TDPE independently from us [22]). We first examine how self-application can be achieved for reification and reflection. Then we show that this enables one to instantiate the second Futamura projection for TDPE and sketch some implementational issues. We close with an assessment of what has been achieved, drawing also on experiences with deriving a compiler from an interpreter for a toy imperative language.

3.3.1 Towards Self-Application for TDPE

Self-application is difficult because a partial evaluator has to be able to handle all the language constructs which were used to write it. A more expressive language on one hand makes it easier to

write a partial evaluator, but on the other hand this partial evaluator has to be more powerful. This consideration should make it clear that for discussing a scheme of self-application one has to fix both the implementation language and the intended way of implementing the partial evaluator. So when talking about self-application for TDPE, we really mean self-application for an ML implementation of TDPE, which uses combinators to encode type-indexed functions as described in section 3.2.7.

The basic technique The fact that TDPE is implemented as a set of combinators (one for every type constructor TDPE can handle) is a first "stumbling block" when considering self-application. Traditional partial evaluators are one program which take a (possibly annotated) program text as input — self-application is then "just" a matter of making sure that the partial evaluator can handle its own code as input.

Consider first how TDPE specializes a given program `p`: As pointed out in section 3.2.7, `p` has to be "prepared" for partial evaluation by replacing any primitive that depends on dynamic input with a code generating function — we speak of *instrumenting* `p`. The idea behind self-application of TDPE is to instrument every combinator on its own. When combining these combinators according to a type scheme τ, this gives rise to so-called *residualizing* reification and reflection functions, which we write as $\Downarrow^\tau$ and $\Uparrow_\tau$, respectively.

Instrumenting the implementation of naive NbE We demonstrate the technique of instrumenting the implementation of NbE by carrying out the necessary steps for the implementation of naive NbE as given in section 3.2.7. The first step is to parameterize over the primitives that are used, namely the constructors of data type `Exp` and the functions defined in `Gensym`. This is shown in Figure 3.2. The signatures `EXP` and `GENSYM` are given in Figure 3.3. For typing reasons we had to abstract away also from the types. Furthermore the declaration of a residualzing function `qstring` for strings has been added to `GENSYM`.

The rationale behind parameterizing naive NbE instead of simply hardwiring code-generating primitives is to use the implementation both for producing the normal and the residualizing version of NbE. Only trivial modifications to the structures `Exp` and `Gensym` as introduced in section 3.2.7 are necessary to make them fit the signatures `EXP` and `GENSYM`. Instantiating `mk_naiveNbe` with `Exp` and `Gensym` will result in an implementation of naive NbE equivalent to the one given in section 3.2.7.

For the residualzing version of NbE, the functor `mk_naiveNbe` is instantiated with code-generating versions of `EXP` and `GENSYM`. In Figure 3.4 we give a code-generating structure of signature `EXP` as example.

Instantiating `mk_naiveNbe` with `DynamicExp` and a code-generating structure of signature `GENSYM` results in a residualizing implementation of naive NbE.

What is missing? Above we showed how to instrument the ML implementation of naive NbE. The implementation of NbE for call-by-value as presented in Danvy's lecture notes [18] is somewhat more complicated. This is mainly due to the use of control operators to implement let insertion. Instrumenting works the same way as above; however certain complications have to be overcome — we will expand on this in section 3.3.3.

```
functor mk_naiveNbe (structure Exp: EXP
                     structure Gensym: GENSYM
                     sharing type Exp.string_ = Gensym.string_)
    = struct
          local open Exp
          in
              type exp_ = Exp.exp_
              datatype 'a rr = RR of ('a -> exp_) * (exp_ -> 'a)
              val rra = RR (fn e => e, fn e => e)

              fun rrf (RR (reify1, reflect1), RR (reify2, reflect2))
                  = RR (fn f => let val x = Gensym.new (Gensym.qstring "x")
                                in LAM (x,reify2 (f (reflect1 (VAR x))))
                                end,
                        fn e => fn v => reflect2 (APP (e,reify1 v)))

              fun rrp (RR (reify1, reflect1), RR (reify2, reflect2))
                = RR (fn (v1,v2) => PAIR (reify1 v1, reify2 v2),
                      fn e => (reflect1 (PFST e), reflect2 (PSND e)))

              fun reify (RR (reify',reflect')) v =
                  (Gensym.init (); reify' v)

              fun reflect (RR (reify',reflect')) v =
                  reflect' v
          end
      end
```

Figure 3.2: Naive NbE, parameterized

```
signature EXP =
    sig
        type exp_
        type string_

        val VAR : string_ -> exp_
        val LAM : string_ * exp_ -> exp_
        val APP : exp_ * exp_ -> exp_
        val PAIR : exp_ * exp_ -> exp_
        val PFST : exp_ -> exp_
        val PSND : exp_ -> exp_

    end;
```

```
signature GENSYM =
    sig
        type string_
        val qstring: string -> string_
        val new: string_ -> string_
        val init: unit -> unit
    end;
```

Figure 3.3: Signatures `EXP` and `GENSYM`

```
structure DynamicExp
= struct
      type string_ = Exp.exp
      type exp_    = Exp.exp

      fun unary oper op1 = Exp.APP (Exp.VAR oper, op1)
      fun binary oper op1 op2 = Exp.APP (Exp.VAR oper, Exp.PAIR (op1, op2))


      fun VAR s = unary "VAR" s
      fun LAM (s, e) = binary "LAM" s e
      fun APP (e1, e2) = binary "APP" e1 e2

      fun PAIR (e1,e2) = binary "PAIR" e1 e2
      fun PFST e = unary "PFST" e
      fun PSND e = unary "PSND" e

    end
```

Figure 3.4: A code-generating version of `Exp`

3.3.2 Applications of Self-Application

One application of self-application for TDPE is to *visualize* what TDPE does. This is achieved by decompilation via reification. Consider for example $\downarrow^{\bullet\to\bullet}$. We decompile it by reifying its residualizing counterpart $\Downarrow^{\bullet\to\bullet}$. Since $\Downarrow^{\bullet\to\bullet}$ expects an argument which fits the type scheme $\bullet \to \bullet$ and returns an expression, we reify at type scheme $(\bullet \to \bullet) \to \bullet$:

$\downarrow^{(\bullet\to\bullet)\to\bullet} (\Downarrow^{\bullet\to\bullet})$ =

```
fn x1 =>
  let val r2=(Gensym.init ()) in
     let val r3=(Gensym.new "x") in
        let val r4=(x1 (VAR r3)) in
           (LAM (r3, r4))
        end
     end
  end
```

This indeed visualizes the process of reification at the type scheme $\bullet \to \bullet$: The first let-binding results in a call to `Gensym.init`. Then `r3` is bound to a fresh variable. Finally an abstraction is created, where the body consists of the application of `x1` to the fresh variable.

Recall the very first example of reifying the identity function — the partial evaluation equation reads

$$\downarrow^{\bullet\to\bullet} (\lambda x.x) = \texttt{fn x0 => x0}$$

It is easy to see that the code that resultes from visualizing $\downarrow^{\bullet\to\bullet}$ produces the same result when applied to the identity function.

Self-application gives us the possibility to look reification and reflection "over the shoulder". To see how the second Futamura projection can be instantiated for TDPE, remember that partially evaluating f to some value v is done by first applying the function to be specialized to v and then using reification. So if f has a type scheme $\tau \to \sigma$, type-directed partial evaluation with respect to the argument of type τ can be written as

$$\lambda x^{\tau}.\downarrow^{\sigma} (f\ x) : \tau \to exp$$

Self-application of reification results in the instantiation of the second Futamura projection for TDPE:

$$\downarrow^{\tau \to \bullet} (\lambda x^{\tau}.\Downarrow^{\sigma} (f\ x))$$

It turns out that all the primitives in f have to be instrumented. To see this, recall "normal" TDPE for a function f: the primitives dependent on dynamic input had to be replaced by code-generating functions. When specializing a partial evaluator to f instead of applying it to f, computations that were static before now have to generate code. By the same reasoning, computations that were dynamic before, and thus were set up to generate code, now have to be set up to generate code that generates code.

3.3.3 Implementational Issues

Implementing self-application for TDPE in ML posed a number of challenges.

Polymorphic occurrences of primitives In the original implementation of reification and reflection one primitive (namely the control operator `shift` (cf. section 3.2.7) is used polymorphically. Here is a piece of the code performing let-insertion:

```
Ctrl.shift (fn k =>
            LET (r,
                 APP (e, reify1 v),
                 Ctrl.reset (fn () => k (reflect2 (VAR r)))))
```

The types `shift` and `reset` are $((\alpha \to \texttt{exp\_}) \to \texttt{exp\_}) \to \alpha$ and $(\texttt{unit} \to \texttt{exp\_}) \to \texttt{exp\_}$, respectively. It is not possible to specify α more precisely, since the combinator in whose definition the cited code occurs must be polymorphic.

However the exact type for which a primitive is going to be used has to be known when instrumenting the primitive. Hence the scheme for creating instrumented versions of reification and reflection as described above does not work.

The problem is solved by a local program transformation which is due to Zhe Yang (April '99):

```
shift (fn k => f (k e))   ⇒   shift (fn k => f (k ())); e
```

where `f` can be an evaluation context in general. This way `k` is always applied to an expression of type `unit`, so that `shift` is only used monomorphically with type $((\texttt{unit} \to \texttt{exp\_}) \to \texttt{exp\_}) \to \texttt{unit}$. The validity of this program transformation is shown by expanding the definitions of `shift` and `reset`.

Organizing the self-application The general pattern of self-application is

$$\downarrow^{\tau_1} (\cdots (\Downarrow^{\tau_2} \text{ or } \Uparrow_{\tau_2}) \cdots v_{\tau_1,\tau_2} \cdots),$$

i.e., two different interpretations of the combinators implementing reification and reflection are needed. Code duplication for the normal and residualizing reification and reflection functions can be avoided by coding TDPE as a functor which is parameterized over the primitives.

For using the second Futamura projection on some expression f of type scheme $\tau \to \sigma$, one wants to specify both f and the encoding of $\tau \to \sigma$ only once. As outlined in section 3.3.2, the encoding of the codomain σ has to be interpreted by the instrumented version of reification, while $\tau \to \bullet$ is interpreted by "normal" reification. Using higher-order functors it is still possible to specify the encoding of $\tau \to \sigma$ only a single time: one encodes τ and σ separately in a functor parameterized over the type-encoding combinators. This functor, together with the specification of f, gives rise to an input structure for a higher-order functor which performs the second Futamura projection on f.

Pragmatics of self-application When applying the second Futamura projection, instrumenting the program to be specialized can become quite intricate. As pointed out above, all primitives have to be instrumented, however on different levels, i.e., either generating code or generating code that generates code. Some pragmatics for keeping the process of instrumentation as simple as possible had to be worked out.

As was mentioned above, instrumenting a program can be done in a structured way using functors — the program to be specialized is parameterized over all primitives that depend on dynamic input. This gives one the possibility to either run the program by instantiating the resulting functor with the standard interpretation of the primitives, or to instrument the program by instantiation with code generating functions.

For the second Futamura projection it is convenient to parameterize over three structures which we call `statics`, `dynamics` and `static_&_dynamic`. The first of them holds all primitives which only depend on static input, the second all the primitives which also depend on dynamic input. The structure `static_&_dynamic` holds primitives which depend on static input but map into the type which represents expressions (having this third structure minimizes explicit type-sharing annotations).

Given such a parameterization, a program can either be run, partially evaluated or used in the second Futamura projection by instantiating the underlying structures accordingly. The following table gives an overview which instantiation has to be used for which case:

	`statics`	`s_&_d`	`dynamics`
run	SI	SI	SI
partial evaluation	SI	SI	$\uparrow_\tau$
2nd Futamura proj.	$\uparrow_\tau$	$\uparrow_\tau$	$\Uparrow_\tau$

Here 'SI' stands for the *standard interpretation*. *Normal* reflection $\uparrow_\tau$ is used to implement code-generating primitives and the instrumented version of reflection $\Uparrow_\tau$ to implement primitives that produce code-generating code.

The example treated in the following section may be helpful in understanding the instrumentation process better.

3.3.4 A Generating Extension of the Power Function

In section 3.1.2 we observed that when specializing the power function with respect to its first argument, the recursion can be unrolled statically. This is because the recursion is controlled by repeatedly decreasing the first argument and comparing it against zero. Hence when instrumenting the power function, we introduce the primitives `eqi` for testing equality and `pred` for calculating the predecessor of an integer. Both will reside in the structure `statics` as introduced in section 3.3.3. Multiplication on the other hand is clearly dynamic — we introduce a primitive `mul` and place it in structure `dynamic`. The fixed point operator which implements the recursion itself is static, but its result type is dynamic. Therefore the primitive `fix` will be placed in structure `static_&_dynamic`. Furthermore several functions like `qint` and `ubool` are needed to transfer values between the different levels. The completely instrumented power function is displayed in Figure 3.5. Instantiating it with the proper structures and performing the second Futamura projection results in a generating extension for the power function as depicted in Figure 3.6. When applied to a non-negative integer n, it indeed generates the same code as partial evaluation of `power` with respect to n does.

```
functor Pow  (structure S : STATICS
              structure D : DYNAMICS
              structure SandD : STATIC_AND_DYNAMIC
              sharing
              type S.int_ = SandD.int_            and
              type D.int__ = SandD.int__          and
              type D.int_ = S.int_ )
    =
    struct
        fun power n x
            = SandD.fix (fn loop => fn n =>
                         if S.ubool (S.eqi (n, S.qint 0)) then
                          D.qint_ (S.qint 1)
                         else
                          D.mul (x, loop (S.pred n))) n
    end;
```

Figure 3.5: The power function, parameterized

3.3.5 What Has Been Achieved?

As described above, self-application gives one the possiblity to visualize the process of TDPE. We examined various instances of reification and reflection, thus gaining further understanding about the implementation of TDPE. Using the second Futamura projection implemented as above, we derived a compiler from an interpreter for Paulson's *Tiny* language [61], a toy imperative language. However, the need to fully instrument the interpreter makes it harder to write. Consider for example the way one writes an interpreter: Usually a number of mutually recursive functions are defined which operate on the abstract syntax tree of the program. The underlying primitive, a

```
(fn x1 =>
   let val r2=(Gensym.init ()) in
      let val r3=(Gensym.new "x") in
         let val r4= (fix (fn x6 =>
                             (fn x7 =>
                               (if (eqi (x7, 0))
                                 then (qint_ 1)
                                 else (mul ((VAR r3), (x6 (pred x7)))))))) x1
   in
            LAM (r3, r4)
         end
      end
   end
)
```

Figure 3.6: A generating extension for the power function

fixed point operator, can be left implicit, since the functions depend only on static input. For the second Futamura projection however, the fixed point operator has to be made explicit. As this holds true for any primitive, the interpreter has to be modified considerably.

The additional instrumentation of primitives also slows down performance: using a compiler derived via the second Futamura projection from the modified interpreter could be slower than compiling by partially evaluating the original interpreter with respect to the source program.

To put it in a nutshell: It is not clear yet whether the second Futamura projection is of practical use in the context of type-directed partial evaluation. However instantiating the second Futamura projection for TDPE gave us new insight into type-directed partial evaluation. Further investigations of the concept of self-application for TDPE are on the way.

Bibliography

[1] Peter Holst Andersen. Partial evaluation applied to ray tracing. In W. Mackens and S. M. Rump, editors, *Software Engineering in Scientific Computing*, pages 78–85. Vieweg, 1996.

[2] Myla M. Archer and Constance L. Heitmeyer. Mechanical verification of timed automata: A case study. Technical Report NRL/MR/5546-98-8180, Naval Research Laboratory, 1998.

[3] Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation (preliminary version). In *Second International Workshop on Types in Compilation, TIC '98 Preliminary Proceedings*, pages 240 – 252, 1997.

[4] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Albert R. Meyer, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 203–213, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[5] Ulrich Berger and Helmut Schwichtenberg. Normalization by evaluation. In B. Möller and J.V. Tucker, editors, *Prospects for hardware foundations, NADA*, 1998.

[6] Stefan Berghofer. Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL. Master's thesis, TU München, 1998.

[7] 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.

[8] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156, 1993.

[9] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):480–521, December 1985.

[10] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(56-68), 1940.

[11] 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.

[12] Thierry Coquand and Gérard Huet. *Constructions: A Higher Order Proof System for Mechanizing Mathematics*, volume 203 of *Lecture Notes in Computer Science*. Springer Verlag, 1985.

[13] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North Holland Publishing Company, 1958.

[14] Olivier Danvy. Pragmatic aspects of type-directed partial evaluation. In *Partial Evaluation, Proceedings*, volume 1110 of *Lecture Notes in Computer Science*, pages 73–94, 1996.

[15] Olivier Danvy. Type-Directed Partial Evaluation. In *Proceedings of 21st Annual Symposium on Principles of Programming Languages*, 1996.

[16] Olivier Danvy. Online type-directed partial evaluation. In *Third Fuji International Symposium on Functional and Logic Programming, FLOPS '98 Proceedings*, pages 271–295, 1998.

[17] Olivier Danvy. A simple solution to type specialization. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of ICALP '98*, volume 1443 of *Lecture Notes in Computer Science*. Springer, 1998.

[18] Olivier Danvy. Type-directed partial evaluation, 1998. Lecture Notes for the DIKU International Summer School *Partial Evaluation — Practice and Theory.*

[19] Olivier Danvy and Peter Dybjer, editors. *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Gothenburg, Sweden, May 8–9, 1998), number NS-98-1 in Notes Series, Department of Computer Science, University of Aarhus, May 1998. BRICS.

[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 and Morten Rhiger. Compiling actions by partial evaluation, revisited. Research Series RS-98-13, BRICS, Department of Computer Science, University of Aarhus, June 1998. 25 pp.

[22] Olivier Danvy and Morten Rhiger. Private communication, spring 1999.

[23] Marco Devillers, David Griffioen, and Olaf Müller. Possibly infinite sequences in theorem provers: A comparative study. In Gunter and Felty [32], pages 89–104.

[24] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.

[25] Andrzej Filinski. From normalization-by-evaluation to type-directed partial evaluation. In Danvy and Dybjer [19].

[26] Andrzej Filinski. A semantic account of type-directed partial evaluation. In *PPDP'99, International Conference on Principles and Practice of Declarative Programming*, 1999. To appear.

[27] Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Thèse d'État, Université de Paris VII, Paris, France, 1972.

[28] Robert Glück, Ryo Nakashige, and Robert Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.

[29] Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[30] Bernd Grobauer and Olaf Müller. From I/O automata to timed I/O automata — a solution to the 'generalized railroad crossing' in Isabelle/HOLCF. In Y. Bertot, G. Dowek, Ch. Paulin-Mohring, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, Lecture Notes in Computer Science, Nice, France, 1999. Springer-Verlag. To appear.

[31] Jim Grundy and Malcolm Newey, editors. *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1497 of *Lecture Notes in Computer Science*, Canberra, Australia, 1998. Springer-Verlag.

[32] Elsa L. Gunter and Amy Felty, editors. *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, Murray Hill, NJ, 1997. Springer-Verlag.

[33] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in Haskell. In Donald Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 241–256, Edinburgh, U.K., 11–13 April 1994. Springer.

[34] John Harrison. Formalized mathematics. Technical Report 36, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland, 1996. Available on the Web as `http://www.cl.cam.ac.uk/users/jrh/papers/form-math3.html`.

[35] John Harrison. Verifying the accuracy of polynomial approximations in HOL. In Gunter and Felty [32], pages 137–152.

[36] John Harrison. Formalizing Dijkstra. In Grundy and Newey [31], pages 171–188.

[37] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, pages 507–541, 1997. Extended version available as the technical report BRICS RS-96-34.

[38] Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, Dec. 1994.

[39] W.A. Howard. The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *To H.B. Curry: Essays on Combinatorial Logic*. Academic Press, 1980.

[40] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[41] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer Verlag, 1985.

[42] 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, February 1989. DIKU Report 91/12.

[43] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[44] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[45] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[46] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations – Part II: timing based systems. Technical Report CS-R9314, CWI, 1993.

[47] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Proceedings Logic Colloquium '73, Bristol, UK, July 1973*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, Amsterdam, 1975.

[48] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[49] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[50] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1995. Tech Report CMU-CS-95-226.

[51] Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Computer Science. Cambridge University Press, Cambridge, 1992.

[52] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, TU München, 1998.

[53] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*.

[54] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In Grundy and Newey [31], pages 349 – 365.

[55] Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

[56] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). In M. McRobbie and J.K. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 733–747. Springer Verlag, 1996.

[57] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[58] Tobias Nipkow and Leonor Prensa Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *FASE'99, Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science. Springer Verlag, 1999. To appear.

[59] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer, 1998. To appear.

[60] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

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

[62] Lawrence C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[63] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[64] Lawrence C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and its Applications*, chapter 3. MIT Press, 1997.

[65] Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. In *CADE-15 Workshop on Integration of Deductive Systems*, 1998.

[66] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens. In R. J. M. Hughes, editor, *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, New York, NY, 1991. Springer-Verlag.

[67] Robert Pollack. What we learn from formal checking; part I: How to believe a machine-checked proof. BRICS autumn school on verification. Notes Series NS-96-8, BRICS, Department of Computer Science, University of Aarhus, October 1996. iv+19 pp.

[68] Morten Rhiger. Deriving a statically typed type-directed partial evaluator. In O. Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99), Proceedings*, 1999.

[69] Michael I. Schwartzbach. Polymorphic type inference. Lecture Series LS-95-3, BRICS, Department of Computer Science, University of Aarhus, June 1995. viii+24 pp.

[70] Dana S. Scott. A type-theoretical alternative to CUCH, ISWIM and OWHY. *Theoretical Computer Science*, 121(411-440), 1993.

[71] Zhong Shao. Flexible representations analysis. In *1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97), Proceedings*, pages 85–98. ACM Press, June 1997.

[72] Konrad Slind. Function definition in higher order logic. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–398, Turku, Finland, 1996. Springer Verlag.

[73] Konrad Slind. Derivation and use of induction schemes in higher-order logic. In Gunter and Felty [32].

[74] Haykal Tej and Burkhardt Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Proceedings of the FME '97 — Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337. Springer Verlag, 1997.

[75] Markus Wenzel. Type classes and overloading in higher-order logic. In Gunter and Felty [32], pages 307 — 322.

[76] Zhe Yang. Encoding types in ML-like languages. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 1998.