Interpreting Specialization in Type Theory

Peter Thiemann*
Institut für Informatik
Universität Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Abstract

We define the static semantics of offline partial evaluation for the simply-typed lambda calculus using a translation into a Martin-Löf-style type theory with suitable extensions. Our approach clarifies that the distinction between specialization-time and run-time computation in partial evaluation can model the phase distinction between compile-time and run-time computation in a module language. Working backwards from that connection, we define partial evaluation for a core language with modules.

Key Words lambda calculus, dependent types

1 Introduction

Program specialization subsumes a whole range of automatic transformations that aim at making programs faster while preserving their semantics. One particular instance is offline partial evaluation, a specialization technique that transforms an annotated program and some part of the input into a specialized program. In an annotated program each expression is either marked executable at specialization time or it is marked executable at run time. The specializer executes the specialization-time expressions and generates code for the run-time expressions.

Lambdamix [21, 22] is a specializer for the lambda calculus. Its operation depends on the *well-formedness* of annotated expressions, a criterion which ensures that no type errors occur during specialization. Lambdamix employs a partial type system [20] to verify well-formedness. Specialization of a well-formed expression either loops or it returns a specialized program, but it cannot fail due to type mismatches.

There are many approaches to define the semantics of Lambdamix-style specializers, based on denotational semantics [22, 21], operational semantics [27], logical frameworks [26], modal and temporal logics [16, 15], category theory [35], and higher-order rewriting [14]. We add a further color to this palette by employing type theory as a semantic foundation. Our plan is as follows:

- define the static semantics by translating an annotated expression into a term in a suitable type theory;
- obtain the specialized expression by applying an extraction function to the type derivation of the translated expression;
- reverse the connection to obtain an extension of Lambdamix that encompasses a module calculus.

The translation establishes a semantics for specialization via the semantics of type theory. We use a Martin-Löf-style type theory [32, 8, 37] with some non-standard extensions. Employing type theory as a semantic metalanguage yields denotational as well as operational models for free. Furthermore, it installs a sound framework for formal reasoning about specialization and a way of comparing different methods of specialization if they can all be specified in the same type theory.

Our particular translation to type theory applies to Lambdamix restricted to simply-typed programs[1]. Type inference for the translated expression performs the specialization-time computation. Extracting the specialized program from the type derivation involves reducing all redexes involving types in the type theory.

The main technical results are:

1. If E is a well-formed annotated expression then its translation type checks and the extraction function returns the same expression as applying Lambdamix to E, up to α-conversion.
2. A well-formedness criterion for an extension of Lambdamix with a module language.

The rest of the paper is structured as follows. We start with an introduction to the basic principles of Lambdamix for the simply-typed lambda calculus in Section 2. Next, in Section 3, we investigate the translation from Lambdamix to type theory in a step-by-step manner, discovering the necessary features of the type theory along the way. In Section 4 we state the formal properties of our translation. The connection to the phase distinction and to module calculus is outlined in Sec. 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

Basic knowledge of partial evaluation [7, 29, 13] is helpful for understanding this paper. Appendix A defines some notation used throughout. For convenience, Appendix B summarizes all translations defined in this paper and Appendix C states the rules of the underlying type theory.

2 Principles of Lambdamix

This section recalls the principles of Lambdamix specialization restricted to the simply-typed lambda calculus [21, 29]. We define the syntax of annotated expressions and annotated types along with a type system formalizing the well-formedness criterion of types and expressions.

*This work has been done while at the School of Computer Science and Information Technology, University of Nottingham, UK. The author acknowledges support by EPSRC grant GR/M22840 “Semantics of Specialization”.

[1] Full Lambdamix admits partially-typed programs where the untyped parts must be run-time expressions.

$$
\begin{array}{llcl}
\text{binding times} & b & ::= & S \mid D \quad \text{ordered by} \quad S < D \\
\text{expressions} & \text{2Expr} \ni e & ::= & k \mid \text{lift } e \mid e +^b e \mid x \mid \lambda^b x.e \mid e @^b e \mid \text{if0}^b\, e\, e\, e \mid \text{fix}^b\, x.e \\
\text{types} & \text{2Type} \ni \tau & ::= & \text{int}^b \mid \tau \xrightarrow{b} \tau \\
\text{top annotation} & & & \text{Top}(\text{int}^b) = b \\
 & & & \text{Top}(\tau_2 \xrightarrow{b} \tau_1) = b
\end{array}
$$

Figure 1: Syntax of Lambdamix

Figure 1 shows the syntax of annotated expressions and types. The metavariable b ranges over binding times which denote phases in the processing of an expression, S for specialization-time or D for run-time. Constants k are specialization-time values by definition. They are restricted to numbers. The expression lift e converts a specialization-time integer into the corresponding run-time value. Addition $e + e$ serves as a representative primitive operation. Lambda abstraction $\lambda x.e$ and application $e@e$ have the usual call-by-name meaning. The conditional $\text{if0}\, e_1\, e_2\, e_3$ tests the value of e_1 for the number zero and returns the value of e_2 or e_3. The fixpoint operator $\text{fix}\, x.e$ computes the fixpoint of $\lambda x.e$.

There are only integer and function types. Both have specialization-time and run-time variants. The Top function extracts the top-level binding-time annotation from a type.

Lambdamix insists that annotated types are *well-formed.* A type τ is well-formed if τ wft is a consequence of the rules in Fig. 2. Well-formedness restricts run-time types with a top-level annotation of D to have only run-time components. For a type assumption A, A wft means that τ wft for all $x : \tau$ in A.

Annotated expressions have annotated types as described in Fig 3. Apart from the annotations, it defines a type system with simple types. Regarding the annotations, all introduction rules (b-cst), (b-abs), and (b-fix) enforce the well-formedness of the type of the introduced object and their expressions carry the same annotation as the top-level annotation of the type. The annotation of the expression in the elimination rules (b-add), (b-app), and (b-if) is equal to the top-level annotation of the type of the eliminated object. The (b-lift) rule is special because its argument is always a specialization-time value while its result is a run-time value. The (b-if) rule enforces the usual restriction: if the condition has a run-time type (i.e., the specializer cannot evaluate it) then both branches need to have a run-time type, too.

There is a tiny addition with respect to the usual system: the type assumption A contains annotated pairs $x :^b \tau$ where b indicates the binding time of the binding expression for x. This does not change the set of typable terms or the assigned types in any way but it is necessary for proving one of the properties in Sec. 4.

2.1 Denotational specification

The Lambdamix specializer is the interpreter of annotated expressions shown in Fig. 4. The intended semantics extends a call-by-name lambda calculus. The interpretation of the run-time constructs is code generation in all cases, indicated by the underlined syntax constructors which are considered operators on an abstract type RExpr of run-time expressions. The cases for λ^D and $\mathbf{fix}^D$ involve the generation of a fresh variable name for n. Although name generation leads to complications in a semantic soundness-proof with respect to the rules in Fig. 3 (as demonstrated and repaired by Moggi [35]), the issue is not central to the topic of this paper.

The defining equations of the specializer rely on a number of auxiliary operators. $\mathbf{int}(k)$ returns the integer value of the syntactic representation of the constant k. **if0** tests its first argument for zero and returns its second or third argument. **fix** is a fixpoint operator of type (Val $\to$ Val) $\to$ Val. $\mathbf{quote}(v)$ maps the integer v to its syntactic representation of type RExpr.

2.2 Operational specification

The denotational framework just outlined follows the standard definition of Lambdamix. However, to prove the properties that we are interested in, an operational framework is more appropriate. Therefore, we define an alternative semantics of 2Expr in terms of reductions. For Lambdamix, all reductions are *static reductions*, that is, every redex only involves constructs which are annotated as specialization-time.

$$
\begin{array}{lcl}
i +^S j & \to & i + j \\
(\lambda^S x.e_1) @^S e_2 & \to & e_1\{x := e_2\} \\
\text{if0}^S\, 0\, e_1\, e_2 & \to & e_1 \\
\text{if0}^S\, n\, e_1\, e_2 & \to & e_2 \text{ if } n \neq 0 \\
\text{fix}^S\, x.e & \to & e\{x := \text{fix}^S\, x.e\}
\end{array}
$$

The usual static reduction relation $\Rightarrow$ is the compatible closure of $\to$ and $\stackrel{*}{\Rightarrow}$ is the reflexive and transitive closure of $\Rightarrow$. It is easy to prove the following lemma by induction on the definition of $\Rightarrow$.

Lemma 1 *If $e_1 \Rightarrow e_2$ then, for all ρ, $\mathcal{S}[\![e_1]\!]\rho = \mathcal{S}[\![e_2]\!]\rho$.*

A further simple induction on the length of a reduction sequence yields the next lemma.

Lemma 2 *If $e_1 \stackrel{*}{\Rightarrow} e_2$ then, for all ρ, $\mathcal{S}[\![e_1]\!]\rho = \mathcal{S}[\![e_2]\!]\rho$.*

3 From Lambdamix to type theory

We discuss a translation from annotated expressions to type theory by going through the different syntactic constructs. The basic idea is to translate a compile-time expression to an expression on the type level and a run-time expression to a value expression. The translation also requires to explicitly state the type of a translated compile-time expression, which is a kind.

For these reasons, there are two functions that work on 2Expr (more precisely, on a Lambdamix type derivation):

- $\mathcal{E}$ maps a typed 2Expr to a term at the value level and
- $\mathcal{T}$ maps a typed 2Expr to a term at the type level.

In addition, there is a function to classify terms at the type level:

- $\mathcal{K}$ which maps a 2Type to a kind.

$$\mathrm{int}^b \ \mathrm{wft}$$

$$\frac{\tau_1 \ \mathrm{wft} \quad \tau_2 \ \mathrm{wft} \quad b \le \mathrm{Top}(\tau_1) \quad b \le \mathrm{Top}(\tau_2)}{\tau_1 \xrightarrow{b} \tau_2 \ \mathrm{wft}}$$

Figure 2: Well-formedness of annotated types

$$\text{(b-cst)}\ \frac{A \ \mathrm{wft}}{A \vdash_{\mathrm{lm}} k : \mathrm{int}^S}$$

$$\text{(b-lift)}\ \frac{A \vdash_{\mathrm{lm}} e : \mathrm{int}^S}{A \vdash_{\mathrm{lm}} \mathrm{lift}\ e : \mathrm{int}^D}$$

$$\text{(b-add)}\ \frac{A \vdash_{\mathrm{lm}} e_1 : \mathrm{int}^b \quad A \vdash_{\mathrm{lm}} e_2 : \mathrm{int}^b}{A \vdash_{\mathrm{lm}} e_1 +^b e_2 : \mathrm{int}^b}$$

$$\text{(b-var)}\ \frac{A \ \mathrm{wft} \quad x :^b \tau \ \mathrm{in}\ A}{A \vdash_{\mathrm{lm}} x : \tau}$$

$$\text{(b-abs)}\ \frac{A\{x :^b \tau_2\} \vdash_{\mathrm{lm}} e : \tau_1 \quad \tau_2 \xrightarrow{b} \tau_1 \ \mathrm{wft}}{A \vdash_{\mathrm{lm}} \lambda^b x.e : \tau_2 \xrightarrow{b} \tau_1}$$

$$\text{(b-app)}\ \frac{A \vdash_{\mathrm{lm}} e_1 : \tau_2 \xrightarrow{b} \tau_1 \quad A \vdash_{\mathrm{lm}} e_2 : \tau_2}{A \vdash_{\mathrm{lm}} e_1 @^b e_2 : \tau_1}$$

$$\text{(b-if)}\ \frac{A \vdash_{\mathrm{lm}} e_0 : \mathrm{int}^b \quad A \vdash_{\mathrm{lm}} e_1 : \tau \quad A \vdash_{\mathrm{lm}} e_2 : \tau \quad b \le \mathrm{Top}(\tau)}{A \vdash_{\mathrm{lm}} \mathrm{if0}^b\ e_0\ e_1\ e_2 : \tau}$$

$$\text{(b-fix)}\ \frac{A\{x :^b \tau\} \vdash_{\mathrm{lm}} e : \tau \quad \tau \ \mathrm{wft} \quad b \le \mathrm{Top}(\tau)}{A \vdash_{\mathrm{lm}} \mathrm{fix}^b\ x.e : \tau}$$

Figure 3: Typing rules of Lambdamix

$$\begin{array}{lrcl}
\text{semantic values} & \mathrm{Val} & = & \mathrm{Int} + \mathrm{RExpr} + (\mathrm{Val} \to \mathrm{Val}) \\
\text{environments} & \rho \in \mathrm{Env} & = & \mathrm{Var} \to \mathrm{Val}
\end{array}$$

$$\mathcal{S} : \mathrm{2Expr} \to \mathrm{Env} \to \mathrm{Val}$$

$$\begin{array}{lcl}
\mathcal{S}[\![k]\!] & = & \lambda\rho.\mathbf{int}(k) \\
\mathcal{S}[\![e_1 +^S e_2]\!] & = & \lambda\rho.\mathcal{S}[\![e_1]\!]\rho + \mathcal{S}[\![e_2]\!]\rho \\
\mathcal{S}[\![x]\!] & = & \lambda\rho.\rho(x) \\
\mathcal{S}[\![\lambda^S x.e]\!] & = & \lambda\rho.\lambda y.\mathcal{S}[\![e]\!]\rho[y/x] \\
\mathcal{S}[\![e_1 @^S e_2]\!] & = & \lambda\rho.(\mathcal{S}[\![e_1]\!]\rho)(\mathcal{S}[\![e_2]\!]\rho) \\
\mathcal{S}[\![\mathrm{if0}^S\ e_0\ e_1\ e_2]\!] & = & \lambda\rho.\mathbf{if0}\ (\mathcal{S}[\![e_0]\!]\rho)\ (\mathcal{S}[\![e_1]\!]\rho)\ (\mathcal{S}[\![e_2]\!]\rho) \\
\mathcal{S}[\![\mathrm{fix}^S\ x.e]\!] & = & \lambda\rho.\mathbf{fix}\ \lambda y.\mathcal{S}[\![e]\!]\rho[y/x] \\
\\
\mathcal{S}[\![\mathrm{lift}\ e]\!] & = & \lambda\rho.\mathbf{quote}(\mathcal{S}[\![e]\!]\rho) \\
\mathcal{S}[\![e_1 +^D e_2]\!] & = & \lambda\rho.\mathcal{S}[\![e_1]\!]\rho \underline{+} \mathcal{S}[\![e_2]\!]\rho \\
\mathcal{S}[\![\lambda^D x.e]\!] & = & \lambda\rho.\underline{\lambda} n.\mathcal{S}[\![e]\!]\rho[n/x] \\
\mathcal{S}[\![e_1 @^D e_2]\!] & = & \lambda\rho.\mathcal{S}[\![e_1]\!]\rho \underline{@} \mathcal{S}[\![E_2]\!]\rho \\
\mathcal{S}[\![\mathrm{if0}^D\ e_1\ e_2\ e_3]\!] & = & \lambda\rho.\underline{\mathrm{if0}}\ (\mathcal{S}[\![e_1]\!]\rho)\ (\mathcal{S}[\![e_2]\!]\rho)\ (\mathcal{S}[\![e_3]\!]\rho) \\
\mathcal{S}[\![\mathrm{fix}^D\ x.e]\!] & = & \lambda\rho.\underline{\mathrm{fix}}\ n.\mathcal{S}[\![e]\!]\rho[n/x]
\end{array}$$

Figure 4: Lambdamix specializer

The intuition is that $\mathcal{T}$ extracts the specialization-time parts whereas $\mathcal{E}$ extracts the run-time parts. $\mathcal{K}$ provides the "type" of the terms generated by $\mathcal{T}$, that is, $\Gamma \vdash_{tt} \mathcal{T}[\![e]\!]\varphi : \mathcal{K}[\![\tau]\!]$ for suitable Γ. Here $\Gamma \vdash_{tt} M : N$ is the typing judgement of the type theory with M and N ranging over its terms and Γ over its environments. For $\mathcal{E}$ the connection is somewhat more involved, since $\mathcal{T}$ does not always generate a type. It may also yield a constructor function, which must first be converted into a type by an auxiliary function **type**. Section 4 states the exact relations between Lambdamix type derivations and the translated terms.

Appendix C gives a summary of the target language.

3.1 Specialization-time integers

The encoding of specialization into type theory relies on creating specialization-time copies of the basic types "elevated" to kinds [4, 5]. For example, while int is the type of the run-time integers $0, 1, \ldots$, its elevated copy, the kind $\overline{\text{int}}$ with the types $\overline{0}, \overline{1}, \ldots$ as elements, models the compile-time integers. Technically, for each integer i, the type $\overline{i}$ has exactly one element $\top : \overline{i}$. Similarly, the primitive operator $e + e$ has an elevated copy, the binary type operator $.\overline{+}.$ (using infix notation) which comes with a set of associated δ rules: $\overline{i} \overline{+} \overline{j} \to \overline{i+j}$. Just like the value operator $+$ has type $\text{int} \to \text{int} \to \text{int}$, the type operator $\overline{+}$ has kind $\overline{\text{int}} \to \overline{\text{int}} \to \overline{\text{int}}$.

An alternative way of viewing the δ rules is to consider them as type equivalences so that the algebra of types is no longer a free term algebra, but rather a quotient algebra factored over the compatible closure of the δ rules. Such approaches are common in dealing with recursive types [6] and with record types [40].

For specialization-time integer expressions, all computation in the translated term takes place on the type level. However, we still need an expression of that type. This expression is never inspected, it is just there because we cannot have a type without an expression. Therefore, we introduce an expression $\top$ which can assume any type $\overline{i}$. The expression translation maps any constant and any specialization-time addition to this expression.

$$\begin{array}{lcl} \mathcal{E}[\![k]\!]\varphi & = & \top \\ \mathcal{E}[\![e_1 +^S e_2]\!]\varphi & = & \top \end{array}$$

The actual specialization-time computation takes place at the type level. The accompanying translation $\mathcal{T}$ from annotated expressions to terms at the type level takes care of that:

$$\begin{array}{lcl} \mathcal{T}[\![k]\!]\varphi & = & \overline{k} \\ \mathcal{T}[\![e_1 +^S e_2]\!]\varphi & = & \mathcal{T}[\![e_1]\!]\varphi \overline{+} \mathcal{T}[\![e_2]\!]\varphi \end{array}$$

Finally, there is a translation from annotated types to kinds of which we can deduce one case, so far.

$$\mathcal{K}[\![\text{int}^S]\!] \quad = \quad \overline{\text{int}}$$

For the minimal subset of expressions considered, we see that if $\vdash_{lm} e : \tau$ then $\vdash_{tt} \mathcal{T}[\![e]\!]\varphi : \mathcal{K}[\![\tau]\!]$, which is true in general as we will see.

It happens to be the case that $\vdash_{tt} \mathcal{E}[\![e]\!]\varphi : \mathcal{T}[\![e]\!]\varphi$. This is a special case of the general relation between a type derivation for an annotated term and the type derivation of its translation. See Sec. 4 for the general case.

3.2 Run-time integers

The kind structure for specialization-time integers provides for values $\top$, types $\overline{i}$, and the kind $\overline{\text{int}}$. For run-time integers there is an analogous structure, which comprises values i, the type int, and the singleton kind INT. The only member of INT is the type int.

In Lambdamix, the expression lift e converts specialization-time integers to run-time integers. The translation must transform an expression e of type $\overline{i} : \overline{\text{int}}$ to an expression of type int : INT. To this end we require a non-standard feature in the type theory: a constant **choose** : $\prod t : \overline{\text{int}} . t \to \text{int}$ which maps a one-element type of kind $\overline{\text{int}}$ and an expression of this type to the corresponding element of int. Hence, **choose** un-elevates these types.[2]

$$\begin{array}{lcl} \mathcal{E}[\![\text{lift } e]\!]\varphi & = & \textbf{choose } (\mathcal{T}[\![e]\!]\varphi)\ (\mathcal{E}[\![e]\!]\varphi) \\ \mathcal{T}[\![\text{lift } e]\!]\varphi & = & \text{int} \\ \mathcal{K}[\![\text{int}^D]\!] & = & \text{INT} \end{array}$$

The translation of the addition of run-time integers is straightforward.

$$\begin{array}{lcl} \mathcal{E}[\![e_1 +^D e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi + \mathcal{E}[\![e_2]\!]\varphi \\ \mathcal{T}[\![e_1 +^D e_2]\!]\varphi & = & \text{int} \end{array}$$

3.2.1 Example

Consider the expression lift $(17 +^S 4)$.

1. For 17, we have $\mathcal{E}[\![17]\!]\varphi = \top$ and $\mathcal{T}[\![17]\!]\varphi = \overline{17}$.
2. For 4, we have $\mathcal{E}[\![4]\!]\varphi = \top$ and $\mathcal{T}[\![4]\!]\varphi = \overline{4}$.
3. For $e \equiv 17 +^S 4$, we have $\mathcal{E}[\![e]\!]\varphi = \top$ and $\mathcal{T}[\![e]\!]\varphi = \overline{17} \overline{+} \overline{4}$ by items 1 and 2. The last term is type-correct (or rather well-kinded) since $\overline{+} : \overline{\text{int}} \to \overline{\text{int}} \to \overline{\text{int}}$.
4. By the conversion rule for types

$$\frac{\Gamma \vdash_{tt} \top : \overline{17} \overline{+} \overline{4} \quad \Gamma \vdash_{tt} \overline{21} : \overline{\text{int}} \quad \overline{17} \overline{+} \overline{4} =_\delta \overline{21}}{\Gamma \vdash_{tt} \top : \overline{21}}$$

5. For $e_2 \equiv \text{lift } (17 +^S 4)$, we have $\mathcal{E}[\![e_2]\!]\varphi = \textbf{choose}\ \overline{21}\ \top$ and $\mathcal{T}[\![e_2]\!]\varphi = \text{int}$.

The extraction function (to be defined in Sec. 3.7) maps the type derivation of **choose** $\overline{21}$ $\top$ to the specialized expression 21.

3.3 Specialization-time functions

The specializer executes specialization-time functions. Consequently, in the translated term, we expect the type checker to reduce specialization-time β-redexes, too. Although the type checker cannot perform a reduction at the term level, it can perform substitution into the types as part of the application rule:

$$\frac{\Gamma \vdash_{tt} M : \prod x : A . B \quad \Gamma \vdash_{tt} N : A}{\Gamma \vdash_{tt} MN : B\{x := N\}}$$

Hence, the translation of specialization-time functions involves dependent product types. The following example gives additional evidence. Consider the expression

$$(\lambda^S f.\lambda^S x.f @^S (f @^S x)) @^S (\lambda^S y.y +^S 1) @^S 7$$

[2] The operator **choose** is related to Girard's 0 operator of type $\prod \alpha : * . \alpha$, which can be used together with Girard's J operator of type $\prod \alpha : * . \prod \beta : * . \alpha \to \beta$ to construct a diverging term in System F [18]. In our calculus, this problem does not arise since **choose** is suitably restricted to the one-element types corresponding to elevated integers.

$$
\begin{array}{lcl}
\mathcal{E}[\![\lambda^S x.e : \tau_2 \xrightarrow{S} \tau_1]\!]\varphi & = & \lambda t : \mathcal{K}[\![\tau_2]\!].\lambda x : \mathbf{type}(\tau_2)(t).\mathcal{E}[\![e]\!]\varphi[x \mapsto t] \\
\mathcal{E}[\![e_1 @^S e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi(\mathcal{T}[\![e_2]\!]\varphi)(\mathcal{E}[\![e_2]\!]\varphi) \\
\mathcal{E}[\![x]\!]\varphi & = & x
\end{array}
$$

Figure 5: Translation to expressions: specialization-time functions

$$
\begin{array}{lcl}
\mathbf{type} : \prod \tau : 2\mathrm{Type}\,.\,\mathcal{K}[\![\tau]\!] \to \mathrm{TYPE} & & \\
\mathbf{type}(\mathrm{int}^S)(f : \overline{\mathrm{int}}) & = & f \\
\mathbf{type}(\mathrm{int}^D)(f : \mathrm{INT}) & = & \mathrm{int} \\
\mathbf{type}(\tau_2 \xrightarrow{S} \tau_1)(f : \mathcal{K}[\![\tau_2]\!] \to \mathcal{K}[\![\tau_1]\!]) & = & \prod t : \mathcal{K}[\![\tau_2]\!]\,.\,\mathbf{type}(\tau_2)(t) \to \mathbf{type}(\tau_1)(ft)
\end{array}
$$

Figure 6: Mapping constructors to types

Recalling that all specialization-time computation should happen at the type level, we intuitively expect for the translation of f which is bound to $\lambda^S y.y +^S 1$ a type like $\overline{n} \to \overline{n+1}$ for some suitable n. However, f has two uses, one at type $\overline{7} \to \overline{8}$, and the other at type $\overline{8} \to \overline{9}$. Hence, f must have a polymorphic type which abstracts over all possible n. The dependent product type $\prod t : \overline{\mathrm{int}}\,.\,t \to t \,\overline{+}\, \overline{1}$ does the trick.

To provide a term of such a dependent product type, a translated specialization-time lambda expression must take *two* parameters, the first one being a type-level parameter that holds all the specialization-time information and the second one being a value parameter of an associated type containing the run-time information. In the example, the specialization-time information is in $t : \overline{\mathrm{int}}$ and the value parameter has type t. Consequently, the translation of a specialization-time application must provide an additional parameter, too. Figure 5 shows the translation.

The translation of the specialization-time lambda refers to a function **type**. $\mathbf{type}(\tau)(t)$ converts the constructor t of kind $\mathcal{K}[\![\tau]\!]$ to a type. It is necessary for two reasons: Firstly, the annotation of x on the right side of Fig. 5 must be a type, i.e., a term of kind TYPE. Secondly, in general, t may not be a type, but it can be a constructor function. The latter case arises when the type τ is a specialization-time function type $\tau_2 \xrightarrow{S} \tau_1$. In this case, t's kind is defined by

$$\mathcal{K}[\![\tau_2 \xrightarrow{S} \tau_1]\!] \quad = \quad \mathcal{K}[\![\tau_2]\!] \to \mathcal{K}[\![\tau_1]\!]$$

and the type translation produces the corresponding expressions at the type level:

$$
\begin{array}{lcl}
\mathcal{T}[\![\lambda^S x.e : \tau_2 \xrightarrow{S} \tau_1]\!]\varphi & = & \lambda t : \mathcal{K}[\![\tau_2]\!].\mathcal{T}[\![e]\!]\varphi[x \mapsto t] \\
\mathcal{T}[\![e_1 @^S e_2]\!]\varphi & = & \mathcal{T}[\![e_1]\!]\varphi(\mathcal{T}[\![e_2]\!]\varphi) \\
\mathcal{T}[\![x]\!]\varphi & = & \varphi(x)
\end{array}
$$

That is, a specialization-time function becomes a constructor abstraction and a specialization-time application becomes a constructor application.

In the example, the task of the **type** function is to map a constructor f of kind $\overline{\mathrm{int}} \to \overline{\mathrm{int}}$ to the type $\prod t : \overline{\mathrm{int}}\,.\,t \to t \,\overline{+}\, \overline{1}$. The solution is to abstract the parameter $t : \overline{\mathrm{int}}$ and to apply f to it to produce the result type. Figure 6 gives the details. It is trivial if its constructor argument is already a type.

3.3.1 Example

In the running example, the translation produces for

$$\lambda^S y.y +^S 1 : \mathrm{int}^S \xrightarrow{S} \mathrm{int}^S$$

a term of type $\prod t : \overline{\mathrm{int}}\,.\,t \to t \,\overline{+}\, \overline{1}$.

Formally, we get for $f = \lambda^S y.y +^S 1$ and for $\tau = \mathrm{int}^S \xrightarrow{S} \mathrm{int}^S$:

$$\mathcal{E}[\![f]\!]\varphi = \lambda t : \overline{\mathrm{int}}.\lambda x : t.\top$$

$$\mathcal{T}[\![f]\!]\varphi = \lambda t : \overline{\mathrm{int}}.t \,\overline{+}\, \overline{1}$$

$$
\begin{array}{l}
\mathbf{type}(\tau)(\mathcal{T}[\![f]\!]\varphi) \\
= \prod t : \mathcal{K}[\![\mathrm{int}^S]\!]\,.\,\mathbf{type}(\mathrm{int}^S)(t) \to \mathbf{type}(\mathrm{int}^S)((\lambda t : \overline{\mathrm{int}}.t \,\overline{+}\, \overline{1})t) \\
= \prod t : \overline{\mathrm{int}}\,.\,t \to (\lambda t : \overline{\mathrm{int}}.t \,\overline{+}\, \overline{1})t \\
= \prod t : \overline{\mathrm{int}}\,.\,t \to t \,\overline{+}\, \overline{1}
\end{array}
$$

3.4 Run-time functions

Just as the translation maps a specialization-time function to a function with a polymorphic type level argument, the translation of a run-time function remains monomorphic. This intuition is straightforward to carry through.

The translation of run-time function types to kinds is just a constant function.

$$\mathcal{K}[\![\tau_2 \xrightarrow{D} \tau_1]\!] \quad = \quad \mathrm{TYPE}.$$

The well-formedness criterion reveals that the mapping from two-level types with top annotation D is straightforward. If $A \vdash_{\mathrm{Tm}} \lambda^D x.e : \tau_2 \xrightarrow{D} \tau_1$ it follows that $\tau_2 \xrightarrow{D} \tau_1$ wft, which means that all annotations in this type are D. Hence, there is exactly one constructor $\mathcal{C}[\![\tau_2]\!]$ with $\mathcal{C}[\![\tau_2]\!] : \mathcal{K}[\![\tau_2]\!]$:

$$
\begin{array}{lcl}
\mathcal{C}[\![\mathrm{int}^D]\!] & = & \mathrm{int} \\
\mathcal{C}[\![\tau_2 \xrightarrow{D} \tau_1]\!] & = & \mathcal{C}[\![\tau_2]\!] \to \mathcal{C}[\![\tau_1]\!]
\end{array}
$$

The translation to expressions and to constructors is straightforward, as promised.

$$\mathcal{E}[\![\lambda^D x.e : \tau_2 \xrightarrow{D} \tau_1]\!]\varphi \quad = \quad \lambda x : \mathcal{C}[\![\tau_2]\!].\mathcal{E}[\![e]\!]\varphi[x \mapsto \mathcal{C}[\![\tau_2]\!]]$$

$$\mathcal{T}[\![\lambda^D x.e : \tau_2 \xrightarrow{D} \tau_1]\!]\varphi \quad = \quad \mathcal{C}[\![\tau_2 \xrightarrow{D} \tau_1]\!]$$

We extend the conversion function from constructors to types accordingly.

$$\mathbf{type}(\tau_2 \xrightarrow{D} \tau_1)(c : \mathrm{TYPE}) \quad = \quad c$$

$$
\begin{array}{rcl}
\mathcal{T}[\![\mathrm{fix}^S\, x.e : \tau]\!]\varphi & = & \mu\, c : \mathcal{K}[\![\tau]\!].\mathcal{T}[\![e]\!]\varphi[x \mapsto c] \\
\mathcal{E}[\![\mathrm{fix}^S\, x.e : \tau]\!]\varphi & = & \mu\, x : \mathbf{type}(\tau)(\mathcal{T}[\![\mathrm{fix}^S\, x.e]\!]\varphi).\mathcal{E}[\![e]\!]\varphi[x \mapsto \mathcal{T}[\![\mathrm{fix}^S\, x.e]\!]\varphi]
\end{array}
$$

Figure 7: Translation of specialization-time fixpoints

The translation of run-time application follows immediately from the translation of the lambda abstraction.

$$
\begin{array}{rcl}
\mathcal{E}[\![e_1 @^D e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi(\mathcal{E}[\![e_2]\!]\varphi) \\
\mathcal{T}[\![e_1 @^D e_2 : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!]
\end{array}
$$

3.5 Conditionals

The conditional does not introduce new problems. We express the type of a specialization-time conditional using a conditional in the type expression and reduce it during type checking using the conversion rule. To this end, there is a conditional expression $\overline{\mathrm{if0}}\, T\; M\; N$ at the type level that reduces to M if T is the type $\overline{0}$ and to N, otherwise. This is a δ rule at the type level just as for elevated addition.

$$
\frac{\Gamma \vdash_{tt} T : \overline{\mathrm{int}} \quad \Gamma \vdash_{tt} M : A \quad \Gamma \vdash_{tt} N : B}{\Gamma \vdash_{tt} \overline{\mathrm{if0}}\, T\; M\; N : \overline{\mathrm{if0}}\, T\; A\; B}
$$

Using $\overline{\mathrm{if0}}\, T\; M\; N$ it is straightforward to define the translations.

$$
\begin{array}{rcl}
\mathcal{E}[\![\mathrm{if0}^S\, e_1\, e_2\, e_3]\!]\varphi & = & \overline{\mathrm{if0}}\, (\mathcal{T}[\![e_1]\!]\varphi)\, (\mathcal{E}[\![e_2]\!]\varphi)\, (\mathcal{E}[\![e_3]\!]\varphi) \\
\mathcal{T}[\![\mathrm{if0}^S\, e_1\, e_2\, e_3]\!]\varphi & = & \overline{\mathrm{if0}}\, (\mathcal{T}[\![e_1]\!]\varphi)\, (\mathcal{T}[\![e_2]\!]\varphi)\, (\mathcal{T}[\![e_3]\!]\varphi)
\end{array}
$$

The translations for the run-time conditional are obvious, thanks to the well-formedness criterion (b-if) which states that the branches of a run-time conditional must have run-time types, too.

$$
\begin{array}{rcl}
\mathcal{E}[\![\mathrm{if0}^D\, e_1\, e_2\, e_3]\!]\varphi & = & \mathrm{if0}\, (\mathcal{E}[\![e_1]\!]\varphi)\, (\mathcal{E}[\![e_2]\!]\varphi)\, (\mathcal{E}[\![e_3]\!]\varphi) \\
\mathcal{T}[\![\mathrm{if0}^D\, e_1\, e_2\, e_3 : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!]
\end{array}
$$

3.6 Fixpoints

Thus far, all of our translated terms live within a strongly normalizing type theory. If we wish to translate the fixpoint combinator, we give up strong normalization and require a partial type theory instead [10]. However, if we only admit fixpoint operators at run-time, all specialization-time computations remain strongly normalizing, the extraction function is terminating, and type checking remains decidable.

The translation of the run-time constructs is straightforward.

$$
\begin{array}{rcl}
\mathcal{E}[\![\mathrm{fix}^D\, x.e : \tau]\!]\varphi & = & \mathrm{fix}\, \lambda x : \mathcal{C}[\![\tau]\!].e \\
\mathcal{T}[\![\mathrm{fix}^D\, x.e : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!]
\end{array}
$$

Specialization-time fixpoints are slightly more involved. The basic insight is that the type transformation must take the fixpoint of a constructor function. Once that is understood, the expression translation follows immediately. $\mu \ldots$ is the fixpoint operator that works on constructors. Figure 7 shows the definition.

3.6.1 Example

As an example, consider a contrived identity function at specialization time.

$$
\mathrm{fix}^S\, i.\lambda^S n.\mathrm{if0}^S\, n\, 0\, (2 +^S i @^S (n -^S 1))
$$

1. $(\lambda t : K.\lambda x : B.M)TN \to M\{t, x := T, N\}$ where K : kind
2. $(\lambda t : K.M)T \to M\{t := T\}$ where K : kind
3. $\overline{\mathrm{if0}}\, \overline{0}\, M\, N \to M$
4. $\overline{\mathrm{if0}}\, \overline{n}\, M\, N \to N$ if $n \neq 0$
5. $\mathbf{choose}\ \overline{n}\ e \to n$
6. $\mu\, t : K.M \to M\{t := \mu\, t : K.M\}$

Figure 8: Extraction

It has type $\tau = \mathrm{int}^S \xrightarrow{S} \mathrm{int}^S$ and its type translation yields

$$
\mu\, i : \overline{\mathrm{int}} \to \overline{\mathrm{int}}.\lambda t : \overline{\mathrm{int}}.\overline{\mathrm{if0}}\, t\, \overline{0}\, (\overline{2} \mathbin{\overline{+}} i(t \mathbin{\overline{-}} \overline{1}))
$$

There is nothing to do for this function at run time, hence the expression translation yields an uninteresting term.

$$
\mu\, i : \mathbf{type}(\tau)(\mathcal{T}[\![\mathrm{fix}^S\, x.e]\!]\varphi).\lambda t : \overline{\mathrm{int}}.\lambda n : t.\overline{\mathrm{if0}}\, t \top \top
$$

3.7 Extraction

The extraction function is easy enough to describe: reduce all type-indexed constructs. The remaining expression is the result of specialization. More formally, extraction performs the reductions in Fig. 8 exhaustively in the translated expression.

4 Formal properties

There are two basic results. First, we have to check whether the translation of a well-formed expression type-checks in the type theory. Second, we have to check that the extraction function simulates static reduction in Lambdamix.

To establish the typing property, we need to map Lambdamix assumptions to assumptions of the type theory. This translation requires the annotated $x :^b \tau$ pairs. The function $A \rightsquigarrow \Gamma; \varphi$ defined in Fig. 9 maps a Lambdamix assumption A to an assumption Γ in the type theory and to a suitable environment φ for the translation functions. The symbol $\diamond$ stands for the empty assumption and φ_0 for the empty environment.

The following lemmas are both proved by induction on the derivation of $A \vdash_{\overline{lm}} e : \tau$.

Lemma 3 *Suppose $A \vdash_{\overline{lm}} e : \tau$ and $A \rightsquigarrow \Gamma; \varphi$. Then $\Gamma \vdash_{tt} \mathcal{T}[\![e]\!]\varphi : \mathcal{K}[\![\tau]\!]$.*

Lemma 4 *Suppose $A \vdash_{\overline{lm}} e : \tau$ and $A \rightsquigarrow \Gamma; \varphi$. Then $\Gamma \vdash_{tt} \mathcal{E}[\![e]\!]\varphi : \mathbf{type}(\tau)(\mathcal{T}[\![e]\!]\varphi)$.*

Theorem 1 *Suppose $e \in$ 2Expr closed, $\tau \in$ 2Type, and $\vdash_{\overline{lm}} e : \tau$. Then $\vdash_{tt} \mathcal{E}[\![e]\!]\varphi_0 : \mathbf{type}(\tau)(\mathcal{T}[\![e]\!]\varphi_0)$.*

$$\diamond \rightsquigarrow \diamond; \varphi_0$$

$$\frac{A \rightsquigarrow \Gamma; \varphi}{A, x :^D \tau \rightsquigarrow \Gamma, x : \mathcal{C}[\![\tau]\!]; \varphi[x \mapsto \mathcal{C}[\![\tau]\!]]}$$

$$\frac{A \rightsquigarrow \Gamma; \varphi}{A, x :^S \tau \rightsquigarrow \Gamma, t : \mathcal{K}[\![\tau]\!], x : \mathbf{type}(\tau)(t); \varphi[x \mapsto t]}$$

Figure 9: Mapping between assumptions

Proof Immediate from Lemma 4. □

Running Lambdamix on an annotated expression is equivalent to type checking the translated expression and applying the extraction function.

Theorem 2 *Suppose* $e \in$ 2Expr *closed,* $\tau \in$ 2Type*, and* $\vdash_m e : \tau$*. Let* e' *be the result of the extraction transformation from Sec. 3.7 applied to* $\mathcal{E}[\![e]\!]\varphi_0$*.*

Then $\mathcal{S}[\![e]\!]\rho_0 = e'$ *up to* α*-conversion.*

Proof (sketch) Using the alternative formulation of Lambdamix in terms of reductions (see Sec. 2.2), we show that reduction in the type theory restricted to the rules from Sec. 3.7 simulates static reduction in Lambdamix.

More precisely, if $e_1 \Rightarrow e_2$ in Lambdamix then $\mathcal{E}[\![e_1]\!]\varphi_0 \Rightarrow_{tt} \mathcal{E}[\![e_2]\!]\varphi_0$. Therefore, if $e_1 \overset{*}{\Rightarrow} e_2$ in Lambdamix then $\mathcal{E}[\![e_1]\!]\varphi_0 \overset{*}{\Rightarrow}_{tt} \mathcal{E}[\![e_2]\!]\varphi_0$, by induction on the length of the reduction sequence. It is easy to see that if e_2 is a normal form then so is $\mathcal{E}[\![e_2]\!]\varphi_0$. Using the Lemma 1, $\mathcal{S}[\![e_1]\!]\rho_0 = \mathcal{S}[\![e_2]\!]\rho_0$ in particular.

Using the correctness of the binding-time analysis [38], since $\vdash_m e_1 : \tau$ so does $\vdash_m e_2 : \tau$. Furthermore, if e_2 is a normal form then it consists solely of run-time constructs.

Finally, induction on e_2 yields that $\mathcal{S}[\![e_2]\!]\rho_0 = \mathcal{E}[\![e_2]\!]\varphi_0$, as desired. □

5 Partial evaluation and the phase distinction

There is a deep relation between specialization and the "phase distinction" in type theory, and our work simulates one with the other. We demonstrate that Cardelli's original intuition of the phase distinction [4] was too restrictive, whereas later work on module languages by Harper and others [24] provides a perfect match. This latter work provides the starting point for extending Lambdamix to embody a module language, too.

5.1 The phase distinction — originally

Cardelli [4] argued for structuring a type system such that arbitrary computations might be performed at compile-time during type-checking while retaining a strict distinction between compile-time and run-time. He proposes [4, page 5]:

> **Phase Distinction Requirement**
> If A is a compile-time term and B is a subterm of A, then B must also be a compile-time term.

This requirement is too restrictive to model specialization. With Lambdamix, compile-time terms may have run-time subterms and vice versa. As examples, consider the following Lambdamix terms:

- $(\lambda^S x.x)@^S(\lambda^D y.y)$, a compile-time term with a run-time subterm;
- $\lambda^D x.(\lambda^S y.y)@^S x$, a run-time term with a compile-time subterm.

Instead of the phase distinction requirement, Lambdamix imposes the well-formedness criterion which restricts a compile-time term embedded into a run-time term to have a *run-time type*, which means that every component value of a run-time value is also a run-time value.

5.2 The phase distinction and modules

Our translation from Lambdamix to type theory corresponds closely to the translation of a higher-order module calculus into a structure calculus with explicit phase separation from earlier work by Harper, Mitchell, and Moggi [24][3] (hereafter HMM, for short). Since their work is concerned with making the phase separation of the module language explicit, their translation considers all lambda abstractions (functors) and applications (functor applications) as compile-time constructs. They do not consider the run-time constructs at all.

Looking at the translation of compile-time constructs [24, Table 7], the cases for variables (s), lambda abstraction ($\lambda s : S.M$) and application ($M_1 M_2$) correspond exactly to our definitions for specialization-time functions (see Sec. 3.3). In these cases, the type translation $\mathcal{T}$ yields their compile-time part and the expression translation $\mathcal{E}$ yields their run-time part. Furthermore, our function **type** gives the same result as the run-time part of their translation for dependent products (the type of a functor: $\prod s : S_1 . S_2$). Our function $\mathcal{K}$ provides the compile-time part.

While HMM does not consider compile-time integers and conditionals, a recent paper by Crary, Harper, and Puri considers fixpoints in the guise of recursive modules [12]. Their work includes a phase-splitting transformation (Fig. 4 of [12]) which is identical (modulo syntax) to our treatment of the specialization-time fixpoint (cf. Fig. 7).

The close connection leads to the question if we can specify the elaboration of the module language using Lambdamix or a suitable extension thereof. To this end, we consider a simple example, freely borrowing ML-style syntax.

```
signature M =
  sig
    type t
    val v : t * t
  end

functor IdM (structure m : M) =
  struct
    type t = m.t
    val  v = m.v
  end
```

In HMM's module calculus (which is based on MacQueen's ideas [31]) the signature `M` is modeled as a dependent sum type $\sum t : * . t \times t$, the elements of which are structures like $\langle \text{int}, \langle 3, 4\rangle\rangle$ or $\langle \text{bool}, \langle \text{true}, \text{false}\rangle\rangle$. The functor `IdM` is expressed as a dependent function $\lambda m : \texttt{M}.\langle \text{fst}(m), \text{snd}(m)\rangle$ of type $\prod m : \texttt{M}.\texttt{M}$.

There are the following correspondences:

functor	specialization-time function
signature	partially static dependent sum type (a two-level type)
structure	partially static pair
type declaration	specialization-time value
value declaration	run-time value

[3] This was suggested to the author by Moggi in September 1998.

Consequently, the annotated versions of the terms of the above example read:

- signature `M`: $\sum^S t : * . t \times^D t$;
- typical structure (element): $\langle \mathrm{int}, \langle \mathrm{lift}\ 3, \mathrm{lift}\ 4\rangle^D\rangle^S$;
- functor `IdM`: $\lambda^S m : \mathtt{M}.\langle \mathrm{fst}^S(m), \mathrm{snd}^S(m)\rangle^S$;
- the functor's type: $\prod^S m : \mathtt{M}.\mathtt{M}$

The annotation ^{S} of the dependent sum's type means that the specializer can access the first and second component of the underlying pair. Thus, the annotated dependent sum generalizes partially static pairs. In the same way, the specialization-time dependent product generalizes ordinary specialization-time functions.

5.3 The phase distinction — extended

Consequently, it is now easy to extend Lambdamix to embrace a module calculus. The intention is to regard the types manipulated by the module calculus as specialization-time values. Since they are actually types of pure run-time values their embedding into 2Type must annotate all type constructors with ^{D}.

Figure 11 shows the extended syntax. It adds the introduction and elimination of pairs and also the types int and $e \to e$ to the previous set of expressions. The set of types now encompasses annotated versions of dependent sums and products, as well as the kind $*$ (the "type" of types) and the variable x. The function Top extracts the top-level annotation, again. Its definition on dependent sum and product types is obvious. A variable x is assumed to have annotation D. The embedding of the types at the expression level into 2Type motivates this choice. Once an application rule substitutes an expression-level type e for x in $\tau \in$ 2Type, it will carry the annotation D everywhere. Since $\mathrm{Top}(x\{x := e\}) = \mathrm{Top}(e) = D$ it must be that $\mathrm{Top}(x) = D$, too. Similar reasoning leads to $\mathrm{Top}(*) = D$.

Figure 12 defines additional well-formedness rules for Extended Lambdamix. The well-formedness judgement is now $A \vdash \tau : b$ for a type assumption A, a two-level type τ, and a binding time b. Obviously, $A \vdash \tau : b$ implies $\mathrm{Top}(\tau) = b$.

Any type variable x as well as $*$ are well-formed 2Type expressions of binding time D. A dependent sum is well-formed, if either the top-level binding times work out in the usual way or if the bound variable is a type (i.e., a run-time value) and the binding time of the second component is greater than or equal to the binding time of the sum constructor. Well-formedness for dependent product types works in the same way. Specializing the rules for the module level constructs yields the derived rules in Fig. 10.

$$\frac{A\{x :^b *\} \vdash \tau_2 : b_2 \quad b \leq b_2}{A \vdash \sum^b x : * . \tau_2 : b}$$

$$\frac{A\{x :^b *\} \vdash \tau_2 : b_2 \quad b \leq b_2}{A \vdash \prod^b x : * . \tau_2 : b}$$

$$\frac{A \text{ wft}}{A\{x : *\} \text{ wft}}$$

Figure 10: Derived well-formedness rules

Finally, Figure 13 shows the typing rules for the new expressions. The rules (b-int) and (b-fun) prescribe the type $*$ for the expressions int and $e_1 \to e_2$ (that is, they are types at the expression level), provided that e_1 and e_2 also have type $*$. The rules (b-pair), (b-fst), and (b-snd) are just the usual elimination rules for weak sums, augmented with annotations and well-formedness as appropriate. The rules (b-dabs) and (b-dapp) generalize rules (b-abs) and (b-app) from Fig. 3.

The operational semantics does not change at all. In fact, existing implementations of specializers can be used with this extended well-formedness criterion.

5.4 Extended translation

Extending the translation is simple, now using HMM as a guideline. We need to provide expression and type translations for each of the new expressions as well as kind translations for the new two-level types.

For the new expressions denoting types, this is particularly easy, since these types denote run-time values.

$$\begin{array}{lcl}
\mathcal{E}[\![\mathrm{int}]\!]\varphi &=& \mathrm{int}\\
\mathcal{E}[\![e_1 \to e_2]\!]\varphi &=& \mathcal{E}[\![e_1]\!]\varphi \to \mathcal{E}[\![e_2]\!]\varphi\\
\mathcal{T}[\![\mathrm{int}]\!]\varphi &=& \mathrm{TYPE}\\
\mathcal{T}[\![e_1 \to e_2]\!]\varphi &=& \mathrm{TYPE}\\
\mathcal{K}[\![*]\!] &=& \mathrm{kind}
\end{array}$$

Function types are now generalized to dependent product types. However, the expression and type translation of abstraction and application does not change with respect to Sec. 3.3 and Sec. 3.4. The only changes occur in the $\mathcal{K}$, $\mathcal{C}$, and **type** functions (see Fig. 14).

$$\begin{array}{lcl}
\mathcal{K}[\![\prod^D x : \tau_2 . \tau_1]\!] &=& \mathrm{TYPE}\\
\mathcal{K}[\![\prod^S x : \tau_2 . \tau_1]\!] &=& \prod x : \mathcal{K}[\![\tau_2]\!] . \mathcal{K}[\![\tau_1]\!]\\
\mathcal{C}[\![\prod^D x : \tau_1 . \tau_2]\!] &=& \prod x : \mathcal{C}[\![\tau_1]\!] . \mathcal{C}[\![\tau_2]\!]
\end{array}$$

Since pairs were not part of the language before, we introduce all the translations directly for the more general dependent sum types. The run-time case is entirely trivial and the translation to constructors can fall back to the $\mathcal{C}$ function defined earlier, again by exploiting the well-formedness criterion.

$$\begin{array}{lcl}
\mathcal{E}[\![\langle e_1, e_2\rangle^D]\!]\varphi &=& \langle \mathcal{E}[\![e_1]\!]\varphi, \mathcal{E}[\![e_2]\!]\varphi\rangle\\
\mathcal{E}[\![\mathrm{fst}^D(e)]\!]\varphi &=& \mathrm{fst}(\mathcal{E}[\![e]\!]\varphi)\\
\mathcal{E}[\![\mathrm{snd}^D(e)]\!]\varphi &=& \mathrm{snd}(\mathcal{E}[\![e]\!]\varphi)\\
\mathcal{T}[\![\langle e_1, e_2\rangle^D : \sum^D x : \tau_1 . \tau_2]\!]\varphi &=& \mathcal{C}[\![\sum^D x : \tau_1 . \tau_2]\!]\\
\mathcal{T}[\![\mathrm{fst}^D(e) : \tau]\!]\varphi &=& \mathcal{C}[\![\tau]\!]\\
\mathcal{T}[\![\mathrm{snd}^D(e) : \tau]\!]\varphi &=& \mathcal{C}[\![\tau]\!]\\
\mathcal{C}[\![\sum^D x : \tau_1 . \tau_2]\!] &=& \sum x : \mathcal{C}[\![\tau_1]\!] . \mathcal{C}[\![\tau_2]\!]\\
\mathcal{K}[\![\sum^D x : \tau_1 . \tau_2]\!] &=& \mathrm{TYPE}
\end{array}$$

The case for specialization-time pairs is not much more involved. In fact, most of the translation is identical, since even if the information is provided in the type level, there must still be expressions of these types present.

$$\begin{array}{lcl}
\mathcal{E}[\![\langle e_1, e_2\rangle^S]\!]\varphi &=& \langle \mathcal{E}[\![e_1]\!]\varphi, \mathcal{E}[\![e_2]\!]\varphi\rangle\\
\mathcal{E}[\![\mathrm{fst}^S(e)]\!]\varphi &=& \mathrm{fst}(\mathcal{E}[\![e]\!]\varphi)\\
\mathcal{E}[\![\mathrm{snd}^S(e)]\!]\varphi &=& \mathrm{snd}(\mathcal{E}[\![e]\!]\varphi)\\
\mathcal{T}[\![\langle e_1, e_2\rangle^S]\!]\varphi &=& \langle \mathcal{T}[\![e_1]\!]\varphi, \mathcal{T}[\![e_2]\!]\varphi\rangle\\
\mathcal{T}[\![\mathrm{fst}^S(e)]\!]\varphi &=& \mathrm{fst}(\mathcal{T}[\![e]\!]\varphi)\\
\mathcal{T}[\![\mathrm{snd}^S(e)]\!]\varphi &=& \mathrm{snd}(\mathcal{T}[\![e]\!]\varphi)\\
\mathcal{K}[\![\sum^S x : \tau_1 . \tau_2]\!] &=& \sum x : \mathcal{K}[\![\tau_1]\!] . \mathcal{K}[\![\tau_2]\!]
\end{array}$$

$$
\begin{array}{llll}
\text{expressions} & e & ::= & \ldots \mid \langle e,e\rangle^b \mid \mathrm{fst}^b(e) \mid \mathrm{snd}^b(e) \mid \mathrm{int} \mid e \to e \\
\text{types} & \tau & ::= & \ldots \mid \sum^b x:\tau\,.\,\tau \mid \prod^b x:\tau\,.\,\tau \mid x \mid * \\
\text{top function} & & & \mathrm{Top}(\sum^b x:\tau_1\,.\,\tau_2) = b \\
 & & & \mathrm{Top}(\prod^b x:\tau_1\,.\,\tau_2) = b \\
 & & & \mathrm{Top}(x) = D \\
 & & & \mathrm{Top}(*) = D
\end{array}
$$

Figure 11: Syntax of Extended Lambdamix

$$
\frac{A \text{ wft}}{A \vdash \mathrm{int}^b : b} \qquad \frac{A \text{ wft} \quad x :^b * \text{ in } A}{A \vdash x : D} \qquad \frac{A \text{ wft}}{A \vdash * : D}
$$

$$
\frac{A \vdash \tau_1 : b_1 \quad A\{x :^b \tau_1\} \vdash \tau_2 : b_2 \quad b \le b_1 \quad b \le b_2}{A \vdash \sum^b x : \tau_1\,.\,\tau_2 : b}
$$

$$
\frac{A \vdash \tau_1 : b_1 \quad A\{x :^b \tau_1\} \vdash \tau_2 : b_2 \quad b \le b_1 \quad b \le b_2}{A \vdash \prod^b x : \tau_1\,.\,\tau_2 : b}
$$

$$
\diamond \text{ wft} \qquad \frac{A \text{ wft} \quad A \vdash \tau : b_1 \quad b \le b_1}{A\{x :^b \tau\} \text{ wft}}
$$

Figure 12: Well-formedness rules for types and type assumptions of Extended Lambdamix

$$
(\text{b-int})\ \frac{A \text{ wft}}{A \vdash_{\mathrm{m}} \mathrm{int} : *}
$$

$$
(\text{b-fun})\ \frac{A \vdash_{\mathrm{m}} e_1 : * \quad A \vdash_{\mathrm{m}} e_2 : *}{A \vdash_{\mathrm{m}} e_1 \to e_2 : *}
$$

$$
(\text{b-pair})\ \frac{A \vdash_{\mathrm{m}} e_1 : \tau_1 \quad A\{x :^b \tau_1\} \vdash_{\mathrm{m}} e_2 : \tau_2 \quad A \vdash \sum^b x : \tau_1\,.\,\tau_2 : b}{A \vdash_{\mathrm{m}} \langle e_1, e_2\rangle^b : \sum^b x : \tau_1\,.\,\tau_2}
$$

$$
(\text{b-fst})\ \frac{A \vdash_{\mathrm{m}} e : \sum^b x : \tau_1\,.\,\tau_2}{A \vdash_{\mathrm{m}} \mathrm{fst}^b(e) : \tau_1}
$$

$$
(\text{b-snd})\ \frac{A \vdash_{\mathrm{m}} e : \sum^b x : \tau_1\,.\,\tau_2}{A \vdash_{\mathrm{m}} \mathrm{snd}^b(e) : \tau_2\{x := \mathrm{fst}^b(e)\}}
$$

$$
(\text{b-dabs})\ \frac{A\{x :^b \tau_2\} \vdash_{\mathrm{m}} e : \tau_1 \quad A \vdash \prod^b x : \tau_2\,.\,\tau_1 : b}{A \vdash_{\mathrm{m}} \lambda^b x.e : \prod^b x : \tau_2\,.\,\tau_1}
$$

$$
(\text{b-dapp})\ \frac{A \vdash_{\mathrm{m}} e_1 : \prod^b x : \tau_2\,.\,\tau_1 \quad A \vdash_{\mathrm{m}} e_2 : \tau_2}{A \vdash_{\mathrm{m}} e_1 @^b e_2 : \tau_1\{x := e_2\}}
$$

Figure 13: Typing rules of Extended Lambdamix

$$
\begin{array}{lcl}
\mathbf{type}(\prod^D x : \tau_2\,.\,\tau_1)(c : \mathrm{TYPE}) & = & c \\
\mathbf{type}(\prod^S x : \tau_2\,.\,\tau_1)(c : \prod x : \mathcal{K}[\![\tau_2]\!]\,.\,\mathcal{K}[\![\tau_1]\!]) & = & \prod t : \mathcal{K}[\![\tau_2]\!]\,.\,\prod x : \mathbf{type}(\tau_2)(t)\,.\,\mathbf{type}(\tau_1)(ct) \\
\mathbf{type}(\sum^D x : \tau_2\,.\,\tau_1)(c : \mathrm{TYPE}) & = & c \\
\mathbf{type}(\sum^S x : \tau_2\,.\,\tau_1)(c : \sum x : \mathcal{K}[\![\tau_2]\!]\,.\,\mathcal{K}[\![\tau_1]\!]) & = & \sum x : \mathcal{K}[\![\tau_2]\!]\,.\,\mathbf{type}(\tau_1)(\mathrm{snd}(c)) \\
\mathbf{type}(*)(c : \mathrm{kind}) & = & c \\
\mathbf{type}(x)(c : \mathrm{TYPE}) & = & c
\end{array}
$$

Figure 14: Extended **type** function

This translation differs from HMM. In HMM, every structure is a compile-time object with the first component a type (a compile-time object) and the second component a run-time value. Their translation takes advantage of this fact and maps pairs as follows:

$$\begin{array}{rcl}
\mathcal{E}[\![\langle e_1, e_2\rangle^{HMM}]\!]\varphi & = & \mathcal{E}[\![e_2]\!]\varphi \\
\mathcal{T}[\![\langle e_1, e_2\rangle^{HMM}]\!]\varphi & = & \mathcal{T}[\![e_1]\!]\varphi \\
\mathcal{E}[\![\mathrm{snd}^{HMM}(e)]\!]\varphi & = & \mathcal{E}[\![e]\!]\varphi \\
\mathcal{T}[\![\mathrm{fst}^{HMM}(e)]\!]\varphi & = & \mathcal{T}[\![e]\!]\varphi
\end{array}$$

with the remaining cases undefined.

Figure 14 shows the definition of the **type** function, extended to the new constructs.

The extended typing rules of the target language are well-known and therefore omitted.

5.5 Discussion

We claim that this kind of calculus may give rise to more powerful module languages. In addition to being able to perform type substitution (which is really the heart of HMM's module language), such a module calculus can perform arbitrary computations at compile/specialization time. The two-level annotations could be chosen in such a way that the specialization-time computation is sure to terminate, for example, by ruling out the specialization-time fixpoint operator or by replacing it with a terminating iteration construct, e.g., primitive recursion.

Another degree of flexibility comes with the ability to also express run-time modules and functors. This facility is interesting in concert with a model that distinguishes between compile-time, link-time, and run-time computation. Here, we might want to leave some module elaborations to link-time or even to run-time. Extending our calculus to a multi-level calculus [19] might yield a more accurate picture in this case.

5.6 Further work

An alternative idea that might be worth pursuing is to have a primitive type `type` of representations of types like `int`, `int -> int`, etc, as specialization-time data. In this scenario, the pair $\langle \mathtt{int}, 5\rangle$ would have type $\sum \mathtt{t} : \mathtt{type} \,.\, \mathrm{Set}(\mathtt{t})$ where "Set" maps the representation `t` of a type to the type itself. Similar constructs are well-known in type theories with universes [37].

A more general approach might allow for annotated types in the module language instead of restricting the types that are manipulated at specialization-time to types of run-time values. Furthermore, the extended well-formedness judgement of Fig. 13 suggests that b plays the role of a kind and that the judgement should be revised to $A \vdash \tau : *^b$. Alas, we leave that for further work.

6 Related work

Type theory finds many of its uses as a basis for formal reasoning about mathematics but also to analyze and verify programs. It is the basis of a number of automatic and semi-automatic systems for automated deduction and proof construction [17, 8, 9, 37]. Some systems support sophisticated program-extraction techniques that apply to completed proofs. These extraction mechanisms are geared to remove the "logical" parts from the proofs while leaving the algorithmic parts intact. For example, Paulin-Mohring's extraction framework for the Calculus of Constructions [39] removes all applications to and abstractions over objects of type Prop. This is different from our Lambdamix extraction function which removes the specialization-time information by reducing away all type manipulation.

Type theory has attracted some attention as a semantic metalanguage for defining the semantics of programming languages. Reynolds [41] gives a type-theoretic interpretation of Idealized Algol. Harper and Mitchell [23] do the same for a fragment of Standard ML. Harper and Stone [25] define a type theoretic semantics for Standard ML. Crary [11] defines the semantics of a higher-order kernel programming language with recursion, records, and modules using Nuprl [8]. There is no other work that defines the semantics of specialization using type theory. The closest is probably Davies's and Pfenning's work [16, 15] that connects modal and temporal logics with specialization.

There are also a number of investigations of the phase distinction in type theory [4, 5]. The idea of duplicating certain values at the type level (and their types at the kind level) is inspired from this work. Cardelli [5] calls these types and kinds *lifted* versions of the original values and types.

Harper, Mitchell, and Moggi [24] have specified the phase distinction in the ML module language extended with higher-order modules. They define and prove correct a staging transformation that separates the compile-time parts of a module from the run-time parts. Although this transformation is motivated from category theory, similar transformations are well-known in the partial evaluation community [33]. Moggi [34] has strengthened the connection between module languages and the two-level languages of the Nielsons [36] and Lambdamix [22] by showing that both have similar semantic models in terms of indexed categories. Our work may be seen as a further clue to the close connections between these two areas. Specifically our $\mathcal{E}$ and $\mathcal{T}$ are closely connected to the compile-time and run-time components defined in Harper, Mitchell, and Moggi's work [24]. Hence, the latter work is also an important contribution to partial evaluation.

In his thesis, Launchbury [30] explains a connection between partial evaluation and dependent types in his description of the type of a specializer. In his framework, the source program has type $\sum_{s\in S} D(s) \to R$, where S is the type of specialization-time values, $D(s)$ is the type of run-time values which depends on the particular value of $s \in S$, and R is the type of the result. Then the partial evaluator applied to the source program has a dependent product type: $\prod_{s\in S} D(s) \to R$. This typing seems to correspond to our translation of unfoldable (specialization-time) functions, where the additional type argument signifies the static part s and the value argument of type s only carries the dynamic part of the value.

Nielson and Nielson [36] investigate a two-level lambda calculus with slightly different properties, in particular the well-formedness criterion differs from the one considered here. It should in principle be possible to construct a type-theoretic semantics for their calculus along the same lines as done here, but we leave that for further investigation.

The whole approach is inspired by Hughes's work on type specialization [28]. Indeed, the translation to type theory presented in this paper is applicable to type specialization in a large degree. Unfortunately, the translation of run-time functions does not quite work out because of the function $\mathcal{C}[\![.]\!]$. This function is not well-defined for type specialization because type specialization does not require two-level types to be well-formed.

7 Conclusion

We have shown a type-theoretical semantics of Lambdamix specialization. The target language is a fairly powerful type theory with subkinding and some special operators. Exploiting the phase distinction in type theory, all specialization-time computation is done during type checking.

Our translation also extends to a dependently typed source language. Thus, it provides a foundation for specialization for dependently typed programming languages [1] and for module languages. Even the existing specializers should remain usable; it is only a question of supplying an appropriate binding-time analysis.

Acknowledgements

Many thanks to Gilles Barthe and John Hatcliff for discussions on specialization for languages with dependent types. Also, thanks to the referees for their comments.

References

[1] Lennart Augustsson. Cayenne—a language with dependent types. In Paul Hudak, editor, *Proc. International Conference on Functional Programming 1998*, Baltimore, USA, September 1998. ACM Press, New York.

[2] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[3] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Tim S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford Science Publications, 1992.

[4] Luca Cardelli. Phase distinctions in type theory. Unpublished Manuscript, January 1988.

[5] Luca Cardelli. Types for data oriented languages. In *Conf. on Extending Database Technology*, pages 1–15. Springer-Verlag, 1988. LNCS 303.

[6] Felica Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92:48–80, 1991.

[7] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[8] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall International, 1986.

[9] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[10] Karl Crary. Admissibility of fixpoint induction over partial types. In *CADE1998*, 1998.

[11] Karl Crary. Programming language semantics in foundational type theory (extended version). Technical Report CS TR98-1666, Cornell University, 1998.

[12] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? Available from http://www.cs.cmu.edu/ crary/papers/1998/recmod/recmod.ps.gz, October 20 1998.

[13] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Dagstuhl Seminar on Partial Evaluation 1996*, volume 1110 of *Lecture Notes in Computer Science*, Schloß Dagstuhl, Germany, February 1996. Springer-Verlag.

[14] Olivier Danvy and Kristoffer Rose. Higher-order rewriting and partial evaluation. In *Rewriting Techniques and Applications: 9th International Conference, RTA '98*, Lecture Notes in Computer Science, Tsukuba, Japan, April 1998. Springer-Verlag.

[15] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[16] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 258–270, St. Petersburg, Fla., January 1996. ACM Press.

[17] N. G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[18] Jean-Yves Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, Amsterdam, 1971.

[19] Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, July 1997.

[20] Carsten K. Gomard. Partial type inference for untyped functional programs. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 282–287, Nice, France, 1990. ACM Press.

[21] Carsten K. Gomard. A self-applicable partial evaluator for the lambda-calculus. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[22] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–70, January 1991.

[23] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[24] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990. ACM Press.

[25] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Robin Milner Festschrift.* MIT Press, 1998. (To appear).

[26] John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In Doaitse Swierstra and Manuel Hermenegildo, editors, *International*

Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95), volume 982 of *Lecture Notes in Computer Science*, pages 279–298, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[27] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.

[28] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Danvy et al. [13], pages 183–215.

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

[30] John Launchbury. *Projection Factorisations in Partial Evaluation*, volume 1 of *Distinguished Dissertations in Computer Science.* Cambridge University Press, 1991.

[31] David B. MacQueen. Using dependent types to express modular structure. In *Proc. 13th Annual ACM Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg, Florida, 1986. ACM.

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

[33] Torben Æ. Mogensen. Separating binding times in language specifications. In *Proc. Functional Programming Languages and Computer Architecture 1989*, pages 14–25, London, GB, 1989.

[34] Eugenio Moggi. A categorical account of two-level languages. In *Proc. Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference*, volume 6 of *Electronic Notes in Theoretical Computer Science*, Pittsburgh, PA, March 1997. Carnegie Mellon University, Elsevier Science BV. URL: http://www.elsevier.nl/locate/entcs/volume6.html.

[35] Eugenio Moggi. Functor categories and two-level languages. In M. Nivat and A. Arnold, editors, *Foundations of Software Science and Computation Structures, FoSSaCS'98*, Lecture Notes in Computer Science, Lisbon, Portugal, April 1998.

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

[37] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf Type Theory. An Introduction.* Oxford University Press, 1990.

[38] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–364, July 1993.

[39] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989. ACM Press.

[40] Didier Rémy. Extension of ML type system with a sorted equational theory on types. Rapports de Recherche 1766, INRIA, October 1992.

$$
\begin{array}{lcl}
\mathcal{T}[\![k]\!]\varphi & = & \overline{k} \\
\mathcal{T}[\![e_1 +^S e_2]\!]\varphi & = & \mathcal{T}[\![e_1]\!]\varphi \mathbin{\overline{+}} \mathcal{T}[\![e_2]\!]\varphi \\
\mathcal{T}[\![e_1 +^D e_2]\!]\varphi & = & \mathrm{int} \\
\mathcal{T}[\![\mathrm{lift}\ e]\!]\varphi & = & \mathrm{int} \\
\mathcal{T}[\![x]\!]\varphi & = & \varphi(x) \\
\mathcal{T}[\![\lambda^S x.e : \tau_2 \xrightarrow{S} \tau_1]\!]\varphi & = & \lambda t : \mathcal{K}[\![\tau_2]\!].\mathcal{T}[\![e]\!]\varphi[x \mapsto t] \\
\mathcal{T}[\![e_1 @^S e_2]\!]\varphi & = & \mathcal{T}[\![e_1]\!]\varphi(\mathcal{T}[\![e_2]\!]\varphi) \\
\mathcal{T}[\![\lambda^D x.e : \tau_2 \xrightarrow{D} \tau_1]\!]\varphi & = & \mathcal{C}[\![\tau_2 \xrightarrow{D} \tau_1]\!] \\
\mathcal{T}[\![e_1 @^D e_2 : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!] \\
\mathcal{T}[\![\mathrm{if0}^D\ e_1\, e_2\, e_3 : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!] \\
\mathcal{T}[\![\mathrm{if0}^S\ e_1\, e_2\, e_3]\!]\varphi & = & \overline{\mathrm{if0}}\ (\mathcal{T}[\![e_1]\!]\varphi)\ (\mathcal{T}[\![e_2]\!]\varphi)\ (\mathcal{T}[\![e_3]\!]\varphi) \\
\mathcal{T}[\![\mathrm{fix}^D\ x.e : \tau]\!]\varphi & = & \mathcal{C}[\![\tau]\!] \\
\mathcal{T}[\![\mathrm{fix}^S\ x.e : \tau]\!]\varphi & = & \mu\, c : \mathcal{K}[\![\tau]\!].\mathcal{T}[\![e]\!]\varphi[x \mapsto c]
\end{array}
$$

Figure 16: Translation to constructors

$$
\begin{array}{lcl}
\mathcal{K}[\![\mathrm{int}^S]\!] & = & \overline{\mathrm{int}} \\
\mathcal{K}[\![\mathrm{int}^D]\!] & = & \mathrm{INT} \\
\mathcal{K}[\![\tau_2 \xrightarrow{S} \tau_1]\!] & = & \mathcal{K}[\![\tau_2]\!] \to \mathcal{K}[\![\tau_1]\!] \\
\mathcal{K}[\![\tau_2 \xrightarrow{D} \tau_1]\!] & = & \mathrm{TYPE}
\end{array}
$$

Figure 17: Translation to kinds

[41] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, October 1981. North-Holland.

A Notation

$\mathrm{FV}(M)$ denotes the set of free variables of expression M. $M\{x := N\}$ denotes the result of the capture-avoiding substitution of N for x in M. $\rho[x \mapsto t]$ denotes the updating of a finite map (an environment).

B Summary of the translations

This section contains an overview of the translations defined in Sec. 3 of this work. Figure 15 contains the expression translation, Figure 16 contains the type translation, and Figure 17 contains the kind translation.

C Typing rules of the target language

The typing rules prove judgements of the following forms:

1. Assumption Γ is well-formed $\vdash_{\mathrm{tt}} \Gamma\ \mathit{env}$ (see Fig. 19) with the empty assumption $\diamond$.

2. In assumption Γ expression M is an N: $\Gamma \vdash_{\mathrm{tt}} M : N$ (see Fig. 21). The formation rules are parameterized over a set of rules (Fig. 20) in the style of pure type systems [2, 3].

As customary, we write $D \to E$ instead of $\prod x : D\,.\,E$ if $x \notin \mathrm{FV}(E)$.

The use of subkinding (see Fig. 20) merely avoids an explosion of the rule set. Without subkinding, we would have to replace $(\mathrm{TYPE}, \mathrm{TYPE}, \mathrm{TYPE})$ by $(s_1, s_2, \mathrm{TYPE})$ where $s_1, s_2 \in \{\overline{\mathrm{int}}, \mathrm{INT}, \mathrm{TYPE}\}$.

$$
\begin{array}{lcl}
\mathcal{E}[\![k]\!]\varphi & = & \top \\
\mathcal{E}[\![e_1 +^S e_2]\!]\varphi & = & \top \\
\mathcal{E}[\![e_1 +^D e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi + \mathcal{E}[\![e_2]\!]\varphi \\
\mathcal{E}[\![\text{lift}\, e]\!]\varphi & = & \mathbf{choose}\ (\mathcal{T}[\![e]\!]\varphi)\ (\mathcal{E}[\![e]\!]\varphi) \\
\mathcal{E}[\![x]\!]\varphi & = & x \\
\mathcal{E}[\![\lambda^S x.e : \tau_2 \xrightarrow{S} \tau_1]\!]\varphi & = & \lambda t : \mathcal{K}[\![\tau_2]\!].\lambda x : \mathbf{type}(\tau_2)(t).\mathcal{E}[\![e]\!]\varphi[x \mapsto t] \\
\mathcal{E}[\![e_1 @^S e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi(\mathcal{T}[\![e_2]\!]\varphi)(\mathcal{E}[\![e_2]\!]\varphi) \\
\mathcal{E}[\![\lambda^D x.e : \tau_2 \xrightarrow{D} \tau_1]\!]\varphi & = & \lambda x : \mathcal{C}[\![\tau_2]\!].\mathcal{E}[\![e]\!]\varphi[x \mapsto \mathcal{C}[\![\tau_2]\!]] \\
\mathcal{E}[\![e_1 @^D e_2]\!]\varphi & = & \mathcal{E}[\![e_1]\!]\varphi(\mathcal{E}[\![e_2]\!]\varphi) \\
\mathcal{E}[\![\text{if0}^D\, e_1\, e_2\, e_3]\!]\varphi & = & \text{if0}\ (\mathcal{E}[\![e_1]\!]\varphi)\ (\mathcal{E}[\![e_2]\!]\varphi)\ (\mathcal{E}[\![e_3]\!]\varphi) \\
\mathcal{E}[\![\text{if0}^S\, e_1\, e_2\, e_3]\!]\varphi & = & \overline{\text{if0}}\ (\mathcal{T}[\![e_1]\!]\varphi)\ (\mathcal{E}[\![e_2]\!]\varphi)\ (\mathcal{E}[\![e_3]\!]\varphi) \\
\mathcal{E}[\![\text{fix}^D\, x.e : \tau]\!]\varphi & = & \text{fix}\ \lambda x : \mathcal{C}[\![\tau]\!].e \\
\mathcal{E}[\![\text{fix}^S\, x.e : \tau]\!]\varphi & = & \mu\, \lambda x : \mathbf{type}(\tau)(\mathcal{T}[\![\text{fix}^S\, x.e]\!]\varphi).\mathcal{E}[\![e]\!]\varphi[x \mapsto \mathcal{T}[\![\text{fix}^S\, x.e]\!]\varphi].
\end{array}
$$

Figure 15: Translation to expressions

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathbf{type} : \prod \tau : 2\text{Type}\,.\,\mathcal{K}[\![\tau]\!] \to \text{TYPE}} \\
\mathbf{type}(\text{int}^S)(c : \overline{\text{int}}) & = & c \\
\mathbf{type}(\text{int}^D)(c : \text{INT}) & = & \text{int} \\
\mathbf{type}(\tau_2 \xrightarrow{S} \tau_1)(c : \mathcal{K}[\![\tau_2]\!] \to \mathcal{K}[\![\tau_1]\!]) & = & \prod t : \mathcal{K}[\![\tau_2]\!]\,.\,\mathbf{type}(\tau_2)(t) \to \mathbf{type}(\tau_1)(ct) \\
\mathbf{type}(\tau_2 \xrightarrow{D} \tau_1)(c : \text{TYPE}) & = & c
\end{array}
$$

Figure 18: Mapping constructors to types

$$\vdash_{\text{tt}} \diamond\ env$$

$$\frac{\vdash_{\text{tt}} \Gamma\ env \quad \Gamma \vdash_{\text{tt}} M : N \quad x \notin \Gamma}{\vdash_{\text{tt}} \Gamma, x : M\ env}$$

Figure 19: Environment construction

kinds	$\overline{\text{int}} : \text{kind}$ $\text{INT} : \text{kind}$ $\text{TYPE} : \text{kind}$
subkinding	$\overline{\text{int}} <: \text{TYPE}$ $\text{INT} <: \text{TYPE}$
constructors	$\bar{i} : \overline{\text{int}}$ *(for all integers i)* $\text{int} : \text{INT}$ $\to : \text{TYPE} \to \text{TYPE} \to \text{TYPE}$ $\overline{+} : \overline{\text{int}} \to \overline{\text{int}} \to \overline{\text{int}}$
values	$i : \text{int}$ *(for all integers i)* $\top : \bar{i}$ *(for all integers i)*
rule set	$(\text{TYPE}, \text{TYPE}, \text{TYPE})$ $(\text{kind}, \text{TYPE}, \text{TYPE})$ $(\text{kind}, \text{kind}, \text{kind})$

Figure 20: Typing axioms

$$\frac{\vdash_{\text{tt}} \Gamma\ env}{\Gamma \vdash_{\text{tt}} x : A}\quad x : A \text{ in } \Gamma$$

$$\frac{\vdash_{\text{tt}} \Gamma\ env}{\Gamma \vdash_{\text{tt}} A : B}\quad A : B \text{ is an axiom}$$

$$\frac{\Gamma \vdash_{\text{tt}} A : B \quad B <: C}{\Gamma \vdash_{\text{tt}} A : C}$$

$$\frac{\Gamma \vdash_{\text{tt}} A : s_1 \quad \Gamma, x : A \vdash_{\text{tt}} B : s_2}{\Gamma \vdash_{\text{tt}} \prod x : A\,.\,B : s_3}\quad (s_1, s_2, s_3) \text{ is a rule}$$

$$\frac{\Gamma, x : A \vdash_{\text{tt}} M : B \quad \Gamma \vdash_{\text{tt}} \prod x : A\,.\,B : s}{\Gamma \vdash_{\text{tt}} \lambda x : A.M : \prod x : A\,.\,B}$$

$$\frac{\Gamma \vdash_{\text{tt}} M : \prod x : A\,.\,B \quad \Gamma \vdash_{\text{tt}} N : A}{\Gamma \vdash_{\text{tt}} MN : B\{x := N\}}$$

$$\frac{\Gamma \vdash_{\text{tt}} M : A \quad \Gamma \vdash_{\text{tt}} A' : s \quad A =_{\beta\delta} A'}{\Gamma \vdash_{\text{tt}} M : A'}$$

$$\frac{\Gamma \vdash_{\text{tt}} A : \text{int} \quad \Gamma \vdash_{\text{tt}} M : B \quad \Gamma \vdash_{\text{tt}} N : B}{\Gamma \vdash_{\text{tt}} \text{if0}\ A\, M\, N : B}$$

$$\frac{\Gamma \vdash_{\text{tt}} T : \overline{\text{int}} \quad \Gamma \vdash_{\text{tt}} M : A \quad \Gamma \vdash_{\text{tt}} N : B}{\Gamma \vdash_{\text{tt}} \overline{\text{if0}}\, T\, M\, N : \overline{\text{if0}}\, T\, A\, B}$$

Figure 21: Typing rules