Strongly Typed Flow-Directed Representation Transformations (Extended Abstract)

Allyn Dimock
Harvard University

Robert Muller
Boston College

Franklyn Turbak
Wellesley College

J. B. Wells[*]
Glasgow University

Topic Areas: compilation, lambda calculus, intersection and union types, typed flow analysis, closure conversion, inlining

Abstract

We present a new framework for transforming data representations in a strongly typed intermediate language. Our method allows both value producers (sources) and value consumers (sinks) to support multiple representations, automatically inserting any required code. Specialized representations can be easily chosen for particular source/sink pairs. The framework is based on these techniques:

1. *Flow annotated types* encode the "flows-from" (source) and "flows-to" (sink) information of a flow graph.

2. *Intersection and union types* support (a) encoding precise flow information, (b) separating flow information so that transformations can be well typed, (c) automatically reorganizing flow paths to enable multiple representations.

As an instance of our framework, we provide a function representation transformation that encompasses both closure conversion and inlining. Our framework is adaptable to data other than functions.

1 Introduction

Typed intermediate languages [20, 19, 23, 30] support type-directed transformations while simultaneously increasing confidence in the correctness of such transformations. In this paper, we focus on *representation* transformations, i.e., those that arise in data type implementations. We consider representation transformations that transform all *sources* at which values are produced and all *sinks* at which they are consumed in a consistent manner. In functional programming languages, a particularly important representation transformation is *closure conversion*, which implements a function value as a *closure*, packaging the function code with the values of its free variables [17, 5, 23, 28, 18].

[*]This author's work was done at Boston University and was partially supported by NSF grants CCR–9113196 and CCR–9417382 and EPSRC grant GR/L 36963.

This paper appears in the *Proceedings of the 1997 ACM SIGPLAN Internation Conference on Functional Programming (ICFP'97)*, Amsterdam, The Netherlands, June 9–11, 1997.

By using *multiple representations* for a single data type, a compiler can choose more efficient representations by considering their uses. For example, *selective* closure conversion uses two function representation [28]: it can represent open functions (i.e., functions with free variables) as code/environment pairs, but in some cases can represent closed functions simply as code. The latter can be more efficient because it avoids packing the code into and unpacking it from a pair with a useless empty environment.

Although beneficial in an optimizing compiler, implementing multiple representations is challenging:

1. Global flow information is required to match up the sources and sinks that share a representation. Because there is growing recognition that such flow analyses are necessary for optimizing higher-order languages [24, 22, 13], this requirement is not too burdensome.

2. To distinguish between multiple representations, it is necessary to separate data flow paths. The "plumbing" of the program must be transformed in such a way that the meaning of the program is preserved.

In this paper, we address the plumbing problem while using existing flow analysis techniques [6]. We present a framework for representation transformation that supports multiple representations within a strongly typed language. The framework is both *type-directed* and *flow-directed* in the sense that it uses the types of terms and global flow information encoded in those types to determine how the terms are transformed. We illustrate our framework by demonstrating how multiple function representations can be used in the same program in a strongly typed manner.

Contributions of this Paper

- We show how flows-from and flows-to annotations on types can be used to pair up sources and sinks. We then use this information to make pairwise representation decisions. Flows-from annotations on types have been used in previous work [11, 6], but we are the first to use them in combination with flows-to annotations.

- We solve the plumbing problem for multiple representations using intersection and union types. We introduce virtual tuples (values of intersection type) into programs to refine flows-to information, and virtual variants (values of union type) to refine flows-from information. A later stage transforms some virtual tuples and variants to real tuples and variants to provide separate data flow paths for incompatible representations.

- We present a novel function representation transformation that subsumes both closure conversion and inlining as special cases.
  - We are the first to perform function transformations using multiple representations with multiple interfaces in a strongly typed language. Earlier approaches to typed closure conversion [10, 18, 19] have required all function representations to use the same application protocol. The only flow-based closure conversion work known to us that supports multiple application protocols is expressed in an untyped language [28].
  - Our transformation can inline functions along arbitrary flow paths, even open functions.[1]

This paper is organized as follows. Section 2 gives an overview of our framework. Section 3 presents the stages of our framework in more detail. Section 4 discusses related research. Section 5 discusses future work. Our intermediate language is defined in the Appendix.

2 Overview

Our representation transformation framework is depicted by the diagram in figure 1. The framework is a composition of well-typedness-preserving transformations on typings of terms in our intermediate language λ^{CIL}.[2] The Flow Separation and Splitting/Tagging stages preserve normal forms for closed terms at base type. Meaning preservation has not yet been proved of the Representation Transformation stage.

The modularity of our framework makes it possible to experiment with different approaches to representation transformations via mix-and-match parts. Our approach can work with many flow analysis algorithms and also allows great flexibility in making representation decisions. Given a flow analysis and a set of representation decisions consistent with the flow analysis, our algorithm performs the program transformations that implement the decisions.

Our language λ^{CIL} is an explicitly typed lambda calculus with product, sum, union, intersection and function types. Function (arrow) types, abstractions, and applications, are annotated with flow labels approximating the flow of functions from abstractions to applications. The Appendix formally defines λ^{CIL}, but we will informally introduce its features in the main text as needed.

We introduce the stages of our framework with the following example:[3]

$$\begin{array}{l}\textbf{let } f^{\mathrm{int}\to\mathrm{int}} = \lambda x^{\mathrm{int}}.x * 2 \\ \textbf{in let } g^{\mathrm{int}\to\mathrm{int}} = \lambda y^{\mathrm{int}}.y + a^{\mathrm{int}} \\ \quad \textbf{in } \times\big(f \mathbin{@} 5, (\textbf{if } b^{\mathrm{bool}} \textbf{ then } f \textbf{ else } g) \mathbin{@} 7\big)\end{array}$$

In this term there are two free variables: a and b. The closed function $(\lambda x^{\mathrm{int}}.x * 2)$ flows to two application sites, the second of which is also a sink for the open function $(\lambda y^{\mathrm{int}}.y + a^{\mathrm{int}})$. It is important that the flow properties of this simple term are merely examples of more complex flow patterns that arise in real programs. In order to stress that our framework can handle arbitrarily complex flow patterns, we will illustrate subsequent examples diagramatically with the abstractions and applications detached from most of their surrounding text.

The Flow Analysis stage computes an approximation to the flow of values between sources and sinks in the input term and encodes this via flow labels in the output typing. Figure 2 shows a possible result of flow analysis for the sample term.

In the diagram, each abstraction site $(\lambda^{l}_{\psi} x^{\sigma}.M)$ is annotated with a source label l and a set of sink labels ψ approximating the set of application sites that can consume λ^{l}. Source and sink labels are also used to annotate function types. If M has type τ in the above term, the type of the abstraction is $\sigma \xrightarrow[\psi]{\{l\}} \tau$. Each application site $(M \mathbin{@}^{\phi}_{k} N)$ is annotated with a sink label k and a set of source labels ϕ approximating the set of abstractions that can be consumed at $@_k$. If N has type σ, then such an abstraction is well-typed only if M has a type of the form $\sigma \xrightarrow[\{k\}]{\phi} \tau$.[4]

The Representation Choices stage chooses representations for the values that flow along each flow path from source to sink and supplies information about these choices to later stages. In the case of function values, there are a wide variety of representations to choose from. For simplicity, we consider only the following representations. An open function can be represented either as (1) a *closure* consisting of a code/environment pair, or (2) an environment, where the code has been inlined at the application sites that the function flows to. A closed function can use one of the open function representations (with a dummy environment), but it can also be represented as just a code pointer.

In many cases, the representations for each flow path can be chosen independently.[5] However, it is necessary to modify the flow graph with plumbing coercions that appropriately handle the flow of multiple representations. Multiple representations produced at a source must be packaged into a tuple at the source and later projected out. If multiple representations reach a sink, they must be injected into a variant earlier so that the sink can perform a case analysis on the variant tag. The examples will illustrate this.

The Representation Choices stage does not perform any plumbing coercions or make any representation changes, but

[1]Because we have not yet explored how to support *lightweight* closure conversion (where free variables whose values are available at the sink do not need to be included in closures) in our framework, the inlining we support is complementary to rather than a replacement for classical inlining.

[2]In λ^{CIL}, "C" stands for the Church Project (http://www.cs.bu.edu/groups/church/) and "IL" stands for "intermediate language". The purpose of the Church Project, named in honor of Alonzo Church, is to investigate the applications of intersection and union types in the compilation of strongly typed higher-order languages. $\lambda^{\mathrm{CIL}}_{\mathrm{DLO}}$ and $\lambda^{\mathrm{CIL}}_{\mathrm{DLS}}$ are subsets of λ^{CIL} defined in Section 3.

[3]Variables are explicitly annotated with types, applications are marked by "@", and (real) tuples are marked by "×". For readability, we omit types on bound variable occurrences when the binding is visible. We have omitted the flow labels λ^{CIL} requires; assume they are all 0. Also, we use base types (like int and bool) and constants and familiar operators for these types, even though they are not in the formal presentation.

[4]Explicit type coercions are required by λ^{CIL} when the flow labels on function types do not match. We omit these coercions from our example diagrams for readability.

[5]In certain closure representations, such as those with linked environments, the representation of one abstraction can depend on that for its enclosing abstractions. We have not yet dealt with the issue of multiple representations in combination with linked closures.

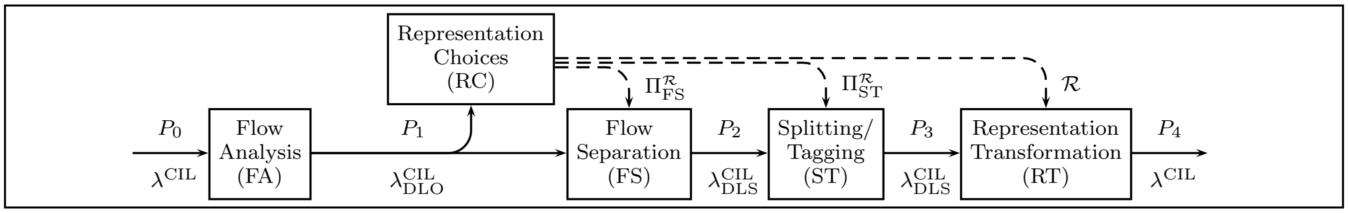
Representation Choices (RC)
P_0
λ^{CIL}
Flow Analysis (FA)
P_1
$\lambda^{\mathrm{CIL}}_{\mathrm{DLO}}$
$\Pi^{\mathcal{R}}_{\mathrm{FS}}$
Flow Separation (FS)
P_2
$\lambda^{\mathrm{CIL}}_{\mathrm{DLS}}$
$\Pi^{\mathcal{R}}_{\mathrm{ST}}$
Splitting/ Tagging (ST)
P_3
$\lambda^{\mathrm{CIL}}_{\mathrm{DLS}}$
$\mathcal{R}$
Representation Transformation (RT)
P_4
λ^{CIL}

Figure 1: Program transformation framework.

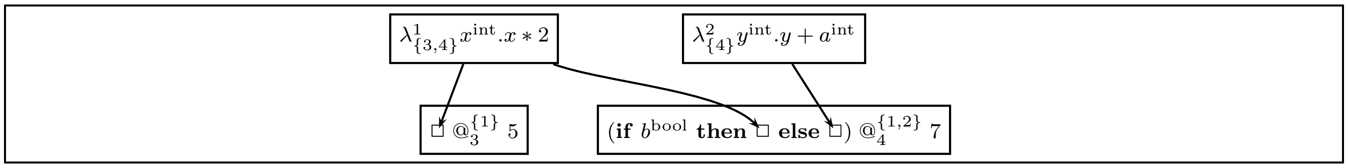
$\lambda^1_{\{3,4\}} x^{\mathrm{int}}.x * 2$
$\lambda^2_{\{4\}} y^{\mathrm{int}}.y + a^{\mathrm{int}}$
$\square\ @^{\{1\}}_3\ 5$
$(\mathbf{if}\ b^{\mathrm{bool}}\ \mathbf{then}\ \square\ \mathbf{else}\ \square)\ @^{\{1,2\}}_4\ 7$

Figure 2: Flow Analysis result.

it provides the required partitioning information to subsequent stages that implement the changes.

The first such stage is Flow Separation, which introduces potential plumbing coercions wherever a function type will need to be transformed into multiple representation types. Flow Separation is guided by $\Pi^{\mathcal{R}}_{\mathrm{FS}}$, a partitioning of flow paths according to the transformed representation type chosen for them. Figure 3 shows the result of Flow Separation on our example in the case where all three flow paths will be given different representation types.[6] The abstraction occurrence $\lambda^1_{\{3,4\}}$ is transformed into a *virtual tuple* (a value of intersection type) containing two abstraction occurrences $\lambda^1_{\{3\}}$ and $\lambda^1_{\{4\}}$. Intuitively, a virtual tuple is a compile-time value of intersection type containing virtual copies of a term that differ only in their types. All of the values inside a virtual tuple share the same run-time representation, and no space needs to be allocated for the virtual tuple at run-time. Similarly, the application occurrence $@^{\{1,2\}}_4$ is transformed into a *virtual case* expression that dispatches on the tag of a virtual variant (a value of union type) to one of two application occurrences $@^{\{1\}}_4$ or $@^{\{2\}}_4$.

The Splitting/Tagging stage reifies some of the virtual plumbing coercions into real plumbing coercions by changing some intersections ($\wedge$) to products ($\times$) and some unions ($\vee$) to sums ($+$). It is guided by $\Pi^{\mathcal{R}}_{\mathrm{ST}}$, a partitioning of flow paths according to the run-time code which will implement the transformed representation types. Consider figure 3 and suppose that the $\{{}^1_3\}$ partition is represented by a code pointer and $\{{}^1_4, {}^2_4\}$ is represented by a closure. Then Splitting/Tagging would transform $\bigwedge\big(\lambda^1_{\{3\}} x^{\mathrm{int}}.x{*}2,\ \lambda^1_{\{4\}} x^{\mathrm{int}}.x{*}2\big)$ into $\times\big(\lambda^1_{\{3\}} x^{\mathrm{int}}.x{*}2,\ \lambda^1_{\{4\}} x^{\mathrm{int}}.x{*}2\big)$ because different code will be executed at run-time to create the two values. The associated $\pi^{\wedge}_i$ terms would also be transformed to $\pi^{\times}_i$. Even though the run-time code constructing the environments for the two closures would differ, the run-time code invoking them would the same, so the $\mathbf{case}^{\vee}$ would *not* be transformed to $\mathbf{case}^{+}$.

[6]Choosing the same representation method (e.g., flat closures) for different functions of the same type can result in different representation types. The closure representation and environment representation expose the types of the function's free variables, thus leading to different transformed types even when the pre-transformation types are the same. These types are only superficially different, because the closures are invoked in the same way. We combine such superficially different types using union types. For some function representations (but not all), existential types can serve a similar purpose [18].

As another example of Splitting/Tagging, suppose that $\{{}^1_3, {}^1_4\}$ are represented by code pointers and $\{{}^2_4\}$ is represented by a closure. Then Splitting/Tagging would transform the instance of $\mathbf{case}^{\vee}$ and $\mathbf{in}^{\vee}_i$ to $\mathbf{case}^{+}$ and $\mathbf{in}^{+}_i$ because a run-time discrimination will be required to choose between $@^{\{1\}}_4$ and $@^{\{2\}}_4$ which must be implemented by different run-time code. (In this case, Flow Separation would not have needed to introduce a virtual tuple at $\lambda^1_{\{3,4\}}$.)

Together, Flow Separation and Splitting/Tagging install the correct plumbing for the final Representation Transformation stage. This stage is controlled by a representation map $\mathcal{R}$ supplied by Representation Choices. The map $\mathcal{R}$ specifies consistent transformations on types, source subterms producing values, and sink subterms consuming these values. Figures 4–6 show the output of Representation Transformation on our sample term for these three sets of representation choices:

Figure	${}^1_3$ rep.	${}^1_4$ rep.	${}^2_4$ rep.
4	code	code/env	code/env
5	code	code	code/env
6	code	code	env (inlining)

Figures 4, 5, and 6 illustrate, respectively, the splitting of a value at a source, discrimination on a tagged value at a sink, and inlining an open function.[7] The input abstraction $(\lambda^2_{\{4\}} y^{\mathrm{int}}.y{+}a^{\mathrm{int}})$ has been transformed to an environment containing its free variable, while its code has been inlined at the application site $@^{\{2\}}_4$. Because a code pointer representation can reach site $@^{\{1\}}_4$, the two incompatible values are injected into a variant.

3 Transformation Framework

3.1 Preliminary Definitions

We use our typed intermediate language λ^{CIL}, which is defined in the Appendix. For this paper, we will add the fol-

[7]Figures 4, 5, and 6 have been simplified by standard local optimizations. The subterm "$(\pi^{\times}_1\ h)\ \ @^{\{0\}}_0\ \ \times\big((\pi^{\times}_2\ h),7\big)$" in the first branch of the case in figure 4 is actually "$\mathbf{let}\ x^{\sigma 1} = h\ \mathbf{in}\ (\pi^{\times}_1\ x)\ @^{\{0\}}_0\ \times\big((\pi^{\times}_2\ x),7\big)$". A similar simplification has been made for the second case branch in figures 4 and 5. See footnote 8 for an explanation of the flow label 0. In figure 6, the subterm "$7{+}(\pi^{\times}_1\ h)$" in the second case branch is actually "$\mathbf{let}\ y^{\times[\times[\mathrm{int}],\mathrm{int}]} = \times(h,7)\ \mathbf{in}\ (\pi^{\times}_2\ y){+}(\pi^{\times}_1\ \pi^{\times}_1\ y)$".

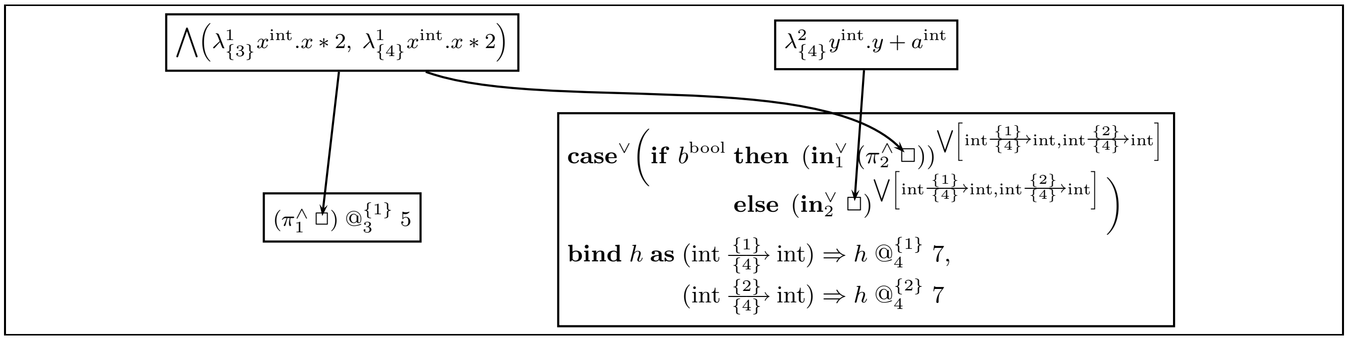
$\bigwedge\left(\lambda^1_{\{3\}} x^{\text{int}}.x * 2,\ \lambda^1_{\{4\}} x^{\text{int}}.x * 2\right)$
$\lambda^2_{\{4\}} y^{\text{int}}.y + a^{\text{int}}$
$(\pi^\wedge_1\ \square)\ @^{\{1\}}_3\ 5$
$\textbf{case}^\vee\Big(\textbf{if}\ b^{\text{bool}}\ \textbf{then}\ (\textbf{in}^\vee_1\ (\pi^\wedge_2\ \square))^{\bigvee\left[\text{int} \xrightarrow[\{4\}]{\{1\}} \text{int}, \text{int} \xrightarrow[\{4\}]{\{2\}} \text{int}\right]}$
$\textbf{else}\ (\textbf{in}^\vee_2\ \square)^{\bigvee\left[\text{int} \xrightarrow[\{4\}]{\{1\}} \text{int}, \text{int} \xrightarrow[\{4\}]{\{2\}} \text{int}\right]}\Big)$
$\textbf{bind}\ h\ \textbf{as}\ (\text{int} \xrightarrow[\{4\}]{\{1\}} \text{int}) \Rightarrow h\ @^{\{1\}}_4\ 7,$
$(\text{int} \xrightarrow[\{4\}]{\{2\}} \text{int}) \Rightarrow h\ @^{\{2\}}_4\ 7$

Figure 3: Flow Separation result.

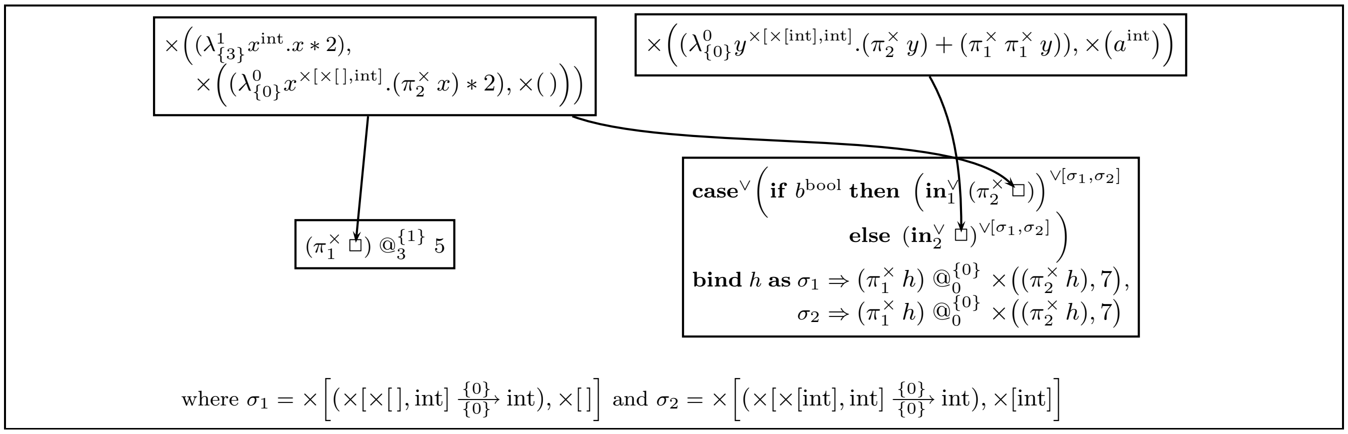
$\times\Big((\lambda^1_{\{3\}} x^{\text{int}}.x * 2),$
$\times\Big((\lambda^0_{\{0\}} x^{\times[\times[\,],\text{int}]}.(\pi^\times_2\ x) * 2), \times(\,)\Big)\Big)$
$\times\Big((\lambda^0_{\{0\}} y^{\times[\times[\text{int}],\text{int}]}.(\pi^\times_2\ y) + (\pi^\times_1\ \pi^\times_1\ y)), \times(a^{\text{int}})\Big)$
$(\pi^\times_1\ \square)\ @^{\{1\}}_3\ 5$
$\textbf{case}^\vee\Big(\textbf{if}\ b^{\text{bool}}\ \textbf{then}\ \big(\textbf{in}^\vee_1\ (\pi^\times_2\ \square)\big)^{\vee[\sigma_1,\sigma_2]}$
$\textbf{else}\ (\textbf{in}^\vee_2\ \square)^{\vee[\sigma_1,\sigma_2]}\Big)$
$\textbf{bind}\ h\ \textbf{as}\ \sigma_1 \Rightarrow (\pi^\times_1\ h)\ @^{\{0\}}_0\ \times\big((\pi^\times_2\ h), 7\big),$
$\sigma_2 \Rightarrow (\pi^\times_1\ h)\ @^{\{0\}}_0\ \times\big((\pi^\times_2\ h), 7\big)$
where $\sigma_1 = \times\Big[(\times[\times[\,], \text{int}] \xrightarrow[\{0\}]{\{0\}} \text{int}), \times[\,]\Big]$ and $\sigma_2 = \times\Big[(\times[\times[\text{int}], \text{int}] \xrightarrow[\{0\}]{\{0\}} \text{int}), \times[\text{int}]\Big]$

Figure 4: Representation Transformation result illustrating splitting.

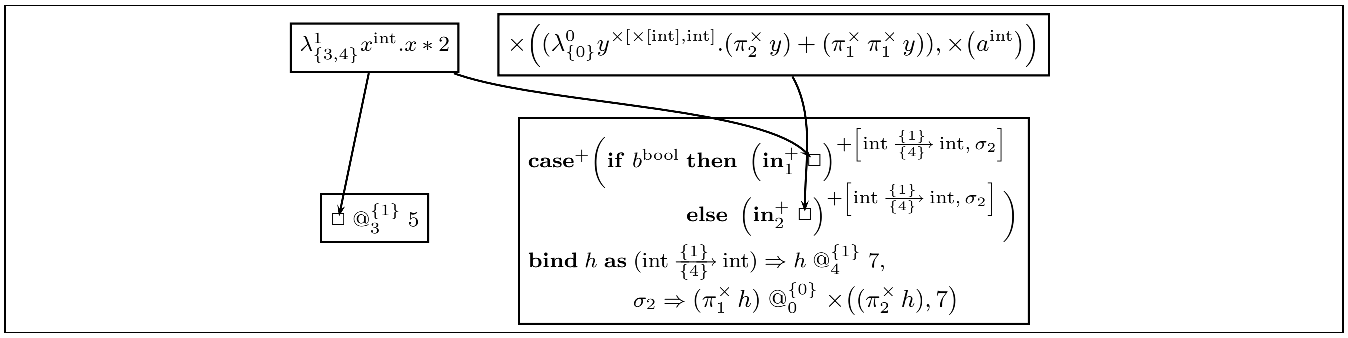
$\lambda^1_{\{3,4\}} x^{\text{int}}.x * 2$
$\times\Big((\lambda^0_{\{0\}} y^{\times[\times[\text{int}],\text{int}]}.(\pi^\times_2\ y) + (\pi^\times_1\ \pi^\times_1\ y)), \times(a^{\text{int}})\Big)$
$\square\ @^{\{1\}}_3\ 5$
$\textbf{case}^+\Big(\textbf{if}\ b^{\text{bool}}\ \textbf{then}\ \big(\textbf{in}^+_1\ \square\big)^{+\left[\text{int} \xrightarrow[\{4\}]{\{1\}} \text{int}, \sigma_2\right]}$
$\textbf{else}\ \big(\textbf{in}^+_2\ \square\big)^{+\left[\text{int} \xrightarrow[\{4\}]{\{1\}} \text{int}, \sigma_2\right]}\Big)$
$\textbf{bind}\ h\ \textbf{as}\ (\text{int} \xrightarrow[\{4\}]{\{1\}} \text{int}) \Rightarrow h\ @^{\{1\}}_4\ 7,$
$\sigma_2 \Rightarrow (\pi^\times_1\ h)\ @^{\{0\}}_0\ \times\big((\pi^\times_2\ h), 7\big)$

Figure 5: Representation Transformation result illustrating tagging.

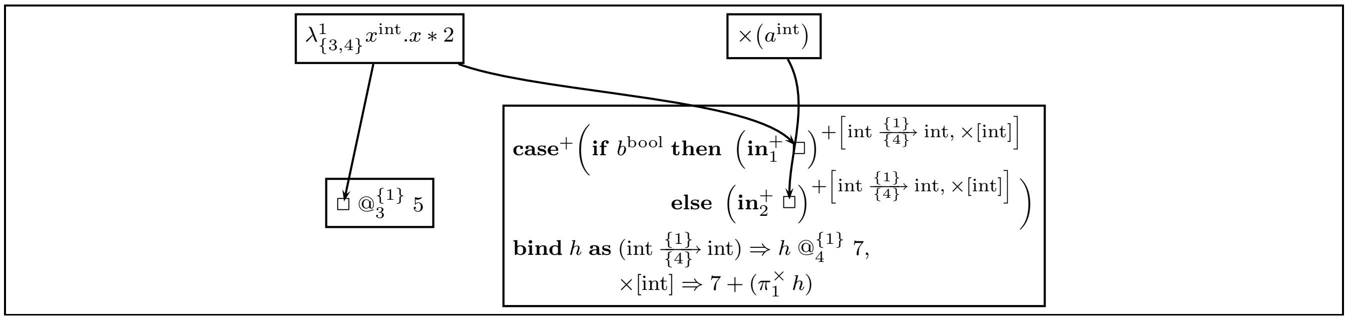
$\lambda^1_{\{3,4\}} x^{\text{int}}.x * 2$
$\times(a^{\text{int}})$
$\square\ @^{\{1\}}_3\ 5$
$\textbf{case}^+\Big(\textbf{if}\ b^{\text{bool}}\ \textbf{then}\ \big(\textbf{in}^+_1\ \square\big)^{+\left[\text{int} \xrightarrow[\{4\}]{\{1\}} \text{int}, \times[\text{int}]\right]}$
$\textbf{else}\ \big(\textbf{in}^+_2\ \square\big)^{+\left[\text{int} \xrightarrow[\{4\}]{\{1\}} \text{int}, \times[\text{int}]\right]}\Big)$
$\textbf{bind}\ h\ \textbf{as}\ (\text{int} \xrightarrow[\{4\}]{\{1\}} \text{int}) \Rightarrow h\ @^{\{1\}}_4\ 7,$
$\times[\text{int}] \Rightarrow 7 + (\pi^\times_1\ h)$

Figure 6: Representation Transformation result illustrating inlining.

lowing definitions and requirements.

We require that every flow label only appears on an arrow at one type. As an example of a type violating this condition, in the type $(\text{int}\xrightarrow[2]{1}\text{int})\xrightarrow[4]{1}(\text{bool}\xrightarrow[2]{3}\text{bool})$ the label 1 corresponds to the two distinct types $(\text{int}\xrightarrow[2]{1}\text{int})$ and $(\text{int}\xrightarrow[2]{1}\text{int})\xrightarrow[4]{1}(\text{bool}\xrightarrow[2]{3}\text{bool})$ and the label 2 corresponds to the two distinct types $(\text{int}\xrightarrow[2]{1}\text{int})$ and $(\text{bool}\xrightarrow[2]{3}\text{bool})$.

Definition 3.1 (Type/Label Consistent). Given some syntactic entity X, define these sets:

$$\begin{aligned}
\check{X} &= \{\,\tau \mid \tau \trianglelefteq X\,\}\\
\check{X}^i &= \{\sigma \xrightarrow[\{0\}]{\{0\}} \tau \mid \sigma \xrightarrow[\psi]{\phi\cup\{i\}} \tau \in \check{X}\}\\
\check{X}_i &= \{\sigma \xrightarrow[\{0\}]{\{0\}} \tau \mid \sigma \xrightarrow[\psi\cup\{i\}]{\phi} \tau \in \check{X}\}
\end{aligned}$$

Then X is type/label consistent (TLC) if and only if $|\check{X}^i| + |\check{X}_i| \leq 1$ for all i. □

Since our framework will use flow labels for transforming subterms, the labels embedded within a term must be sufficiently distinctly chosen. A term M has a *label* (at the top level) if and only if M is an abstraction or an application. An abstraction $(\lambda^i_\psi x^\tau.M)$ has label i (the top, or *source* label) and an application $(M \mathbin{@}^\phi_j N)$ has label j (the bottom, or *sink* label). Given a set of terms $\mathbf{M} = \{M_1, \ldots, M_n\}$ where the label of each M_i is l_i, the set $\mathbf{M}$ is *distinctly labelled* if and only if $l_i = l_j$ implies $i = j$. Given a term M, the *subterms of* M are the members of the set $\{\,N \mid N \trianglelefteq M\,\}$. Given a term M, the *subterm occurrences of* M are the members of the set $\{\,N^{(p)} \mid N \trianglelefteq M \text{ at position } p\,\}$ where $N^{(p)}$ pairs the subterm N with the position p (an unspecified notion) at which it occurs in M, so the set has one copy of the subterm for each position at which it occurs.

Definition 3.2 (Distinctly Labelled).
1. A term M is *distinctly labelled by occurrence* (DLO) if and only if the set of subterm occurrences of M is distinctly labelled.
2. A term M is *distinctly labelled by subterm* (DLS) if and only if the set of subterms of M is distinctly labelled. □

Definition 3.3 ($\lambda^{\text{CIL}}_{\text{DLO}}$ and $\lambda^{\text{CIL}}_{\text{DLS}}$). A judgement "$A \vdash M : \tau$" is derivable in the language subset $\lambda^{\text{CIL}}_{\text{DLO}}$ (respectively $\lambda^{\text{CIL}}_{\text{DLS}}$) if and only if $A \vdash_{\lambda^{\text{CIL}}} M : \tau$ via $\mathcal{D}$ and the term M is DLO (respectively DLS) and the typing $\mathcal{D}$ is TLC. □

3.2 Algorithm

The overall algorithm $\mathcal{F}$ implementing our representation transformation framework proceeds as follows (see also figure 1):

$$\begin{aligned}
\mathcal{F}(P_0) = {}&\textbf{let } P_1 = \text{FA}(P_0)\\
&\quad\textbf{in let } \langle\mathcal{R}, \Pi^{\mathcal{R}}_{\text{FS}}, \Pi^{\mathcal{R}}_{\text{ST}}\rangle = \text{RC}(P_1)\\
&\quad\quad\textbf{in let } P_2 = \text{FS}(P_1, \Pi^{\mathcal{R}}_{\text{FS}})\\
&\quad\quad\quad\textbf{in let } P_3 = \text{ST}(P_2, \Pi^{\mathcal{R}}_{\text{ST}})\\
&\quad\quad\quad\quad\textbf{in let } P_4 = \text{RT}(P_3, \mathcal{R})\\
&\quad\quad\quad\quad\quad\textbf{in } P_4
\end{aligned}$$

The algorithm $\mathcal{F}$ uses sub-algorithms FA, RC, FS, ST, and RT, which are described in sections 3.3, 3.4, 3.5, 3.6, and 3.7, respectively.

3.3 Flow Analysis (FA)

In the literature, a *flow analysis* (sometimes called a *closure analysis*) is any of a class of analyses that either relate abstraction occurrences (function definitions) to application occurrences (function call sites), or relate abstractions which may be called to the abstractions from whose bodies they are called [3, 6, 11, 12, 24]. In our intermediate language λ^{CIL}, as in the work of Heintze [11] and Banerjee [6], flow information is encoded as annotations on arrow types.

The Flow Analysis (FA) stage takes the input program P_0, performs a flow analysis on P_0 and encodes the results of this analysis in type annotatations on its output P_1. Rather than providing an algorithm to implement this stage, we merely specify minimum requirements that an algorithm must satisfy to work with our framework.

Definition 3.4 (Label Erasure). The label erasure of any syntactic entity X, denoted $\langle X\rangle$, is obtained by replacing all labels on types, abstractions, or applications in X by the label 0. □

Definition 3.5 (Flow Analysis Requirements). A function FA from typed terms in λ^{CIL} to typed terms in $\lambda^{\text{CIL}}_{\text{DLO}}$ is a flow analysis suitable for use in our framework if and only if the following conditions hold. If $A \vdash_{\lambda^{\text{CIL}}} M : \tau$ via $\mathcal{D}$ and $M' = \text{FA}(M)$, then there exist A', τ', and $\mathcal{D}'$ such that

1. $A' \vdash_{\lambda^{\text{CIL}}_{\text{DLO}}} M' : \tau'$ via $\mathcal{D}'$.
2. $|M'| \equiv |M|$ (meaning preservation).
3. $\langle A'\rangle = \langle A\rangle$ and $\langle\tau'\rangle = \langle\tau\rangle$.

□

There is a flow analysis for λ^{CIL} that is as accurate as the 0CFA of Shivers [24]. Due to our deliberately restricted subtyping relation, such a flow analysis must introduce virtual tuples and variants to achieve sufficient accuracy.

3.4 Representation Choices (RC)

The Representation Choices (RC) stage takes as input a typed program P_1 and computes a *representation map* $\mathcal{R} = \langle\mathcal{R}^\lambda, \mathcal{R}^@, \mathcal{R}^{\text{var}}, \mathcal{R}^\to\rangle$ and partitioning functions $\Pi^{\mathcal{R}}_{\text{FS}}$ and $\Pi^{\mathcal{R}}_{\text{ST}}$, i.e.,

$$\text{RC}(P_1) = \langle\mathcal{R}, \Pi^{\mathcal{R}}_{\text{FS}}, \Pi^{\mathcal{R}}_{\text{ST}}\rangle$$

The map components are used by the Representation Transformation stage to perform transformations on the program's syntax tree. The component $\mathcal{R}^\lambda$ outputs code implementing each occurrence of an abstraction in the source term, $\mathcal{R}^@$ outputs code implementing each occurrence of an application in the source term, $\mathcal{R}^{\text{var}}$ outputs a function from variables to code implementing variable lookup, and $\mathcal{R}^\to$ maps function types to the types of the corresponding function implementations.

The partitioning functions $\Pi^{\mathcal{R}}_{\text{FS}}$ and $\Pi^{\mathcal{R}}_{\text{ST}}$ are used in the Flow Separation and Splitting/Tagging stages to prepare for the Representation Transformation stage. The Flow Separation transformation is guided by $\Pi^{\mathcal{R}}_{\text{FS}}$ in separating types, abstractions, and applications which either (1) need different types or (2) actually need different code. The Splitting/Tagging transformation is guided by $\Pi^{\mathcal{R}}_{\text{ST}}$ in changing virtual tuples and variants into real tuples and variants to allow possibility (2).

In the following we use these textual abbreviations (i.e., free variables in the expansions are captured):

$$
\begin{array}{llll}
x = \mathrm{name}(\mathrm{bv}(^{\phi}_{\psi}, N)) & \tau \ = \mathrm{type}(\mathrm{bv}(^{\phi}_{\psi}, N)) & x_i = \mathrm{name}(\mathrm{fv}_i(^{\phi}_{\psi}, N)) & \tau_i = \mathrm{type}(\mathrm{fv}_i(^{\phi}_{\psi}, N)) \\
n = \#\mathrm{fv}(^{\phi}_{\psi}, N) & \tau' = [\![\tau]\!]_{\mathcal{R},N} & \rho \ = \mathcal{E}^{\rightarrow}(^{\phi}_{\psi}, N) & \rho' = \times[\rho, \tau']
\end{array}
$$

The function "$[\![\,\cdot\,]\!]_{\cdot,\cdot}$" is defined in figure 9. The representation map $\mathcal{R}$ will be what this figure is defining. The parameter P_1 is the entire program as input to the RC stage.

Representation Choice Constraints

Any total predicates $\mathcal{R}^{\mathrm{code?}}$ and $\mathcal{R}^{\mathrm{env?}}$ may be chosen as long as:

$$\forall\,^{l}_{k}.\ ((\#\mathrm{fv}(^{l}_{k}, P_1) = 0) \text{ or } \mathcal{R}^{\mathrm{code?}}(^{l}_{k}) \text{ or } \mathcal{R}^{\mathrm{env?}}(^{l}_{k}))$$
$$\forall\,^{l}_{k}.\ (\text{not inlinedin?}^{*}(^{l}_{k}, ^{l}_{k}, P_1))$$

$$
\begin{array}{lcl}
\text{inlinedin?}(^{l}_{k}, ^{l'}_{k'}, N) & \iff & \exists\, M, M', \phi, k.\ ((M \ @_{k}^{\{l\}\cup\phi}\ M') \trianglelefteq \mathrm{body}(^{l'}_{k'}, N) \text{ and not } \mathcal{R}^{\mathrm{code?}}(^{l}_{k})) \\
\text{inlinedin?}^{*}(^{l_1}_{k_1}, ^{l_n}_{k_n}, N) & \iff & \exists\,^{l_2}_{k_2}, \ldots, ^{l_{n-1}}_{k_{n-1}}.\ (\text{inlinedin?}(^{l_1}_{k_1}, ^{l_2}_{k_2}, N) \text{ and } \cdots \text{ and inlinedin?}(^{l_{n-1}}_{k_{n-1}}, ^{l_n}_{k_n}, N))
\end{array}
$$

Representation Implementation

The functions $\langle \mathcal{R}^{\lambda}, \mathcal{R}^{@}, \mathcal{R}^{\mathrm{var}}, \mathcal{R}^{\rightarrow} \rangle$ depend on the value of $\langle \mathcal{R}^{\mathrm{code?}}(^{\phi}_{\psi}), \mathcal{R}^{\mathrm{env?}}(^{\phi}_{\psi}) \rangle$ as follows:

	$\mathcal{R}^{\rightarrow}(^{\phi}_{\psi}, \sigma_1, \sigma_2, N)$	$\mathcal{R}^{\lambda}(^{\phi}_{\psi}, E, M, N)$	$\mathcal{R}^{@}(^{\phi}_{\psi}, M_1, M_2, M_3, N)$	$\mathcal{R}^{\mathrm{var}}(^{\phi}_{\psi}, E, N)$
$\langle \mathrm{true}, \mathrm{true} \rangle$	$\mathcal{C}^{\rightarrow}(^{\phi}_{\psi}, \sigma_1, \sigma_2, N)$	$\mathcal{C}^{\lambda}(^{\phi}_{\psi}, E, M, N)$	$\mathcal{C}^{@}(M_1, M_2)$	$\mathcal{C}^{\mathrm{var}}(^{\phi}_{\psi}, N)$
$\langle \mathrm{true}, \mathrm{false} \rangle$	$\sigma_1 \xrightarrow[\psi]{\phi} \sigma_2$	$\lambda_{\psi}^{\min \phi} x^{\tau'}.M$	$M_1 \ @^{\phi}_{\min \psi}\ M_2$	$E[x^{\tau} \mapsto x^{\tau'}]$
$\langle \mathrm{false}, \mathrm{true} \rangle$	$\mathcal{E}^{\rightarrow}(^{\phi}_{\psi}, N)$	$\mathcal{E}^{\lambda}(^{\phi}_{\psi}, E, N)$	$\mathcal{E}^{@}(^{\phi}_{\psi}, M_1, M_2, M_3, N)$	$\mathcal{E}^{\mathrm{var}}(^{\phi}_{\psi}, N)$
$\langle \mathrm{false}, \mathrm{false} \rangle$	$\times[]$	$\times()$	$\textbf{let } x^{\tau'} = M_2 \textbf{ in } M_3$	$\{x^{\tau} \mapsto x^{\tau'}\}$

The M_3 parameter of $\mathcal{R}^{@}$ will be a function body to inline.
The N parameter of each function will be the entire program P_3 as input to the RT stage.

Closure Implementation

$$
\begin{array}{ll}
\mathcal{C}^{\rightarrow}(^{\phi}_{\psi}, \sigma_1, \sigma_2, N) & = \times\Big[(\times[\rho, \sigma_1] \xrightarrow[\{0\}]{\{0\}} \sigma_2), \rho\Big] \\
\mathcal{C}^{\lambda}(^{\phi}_{\psi}, E, M, N) & = \times\Big((\lambda^{0}_{\{0\}} x^{\rho'}.M), \mathcal{E}^{\lambda}(^{\phi}_{\psi}, E, N)\Big) \\
\mathcal{C}^{@}(M_1^{\sigma}, M_2) & = \textbf{let } y^{\sigma} = M_1 \textbf{ in } (\pi_1^{\times}\ y^{\sigma})\ @_0^{\{0\}}\ \times\Big((\pi_2^{\times}\ y^{\sigma}), M_2\Big) \quad \text{where } y \text{ is fresh} \\
\mathcal{C}^{\mathrm{var}}(^{\phi}_{\psi}, N) & = \mathcal{E}^{\mathrm{var}}(^{\phi}_{\psi}, N)
\end{array}
$$

Environment Implementation

$$
\begin{array}{ll}
\mathcal{E}^{\rightarrow}(^{\phi}_{\psi}, N) & = \times\big[[\![\tau_1]\!]_{\mathcal{R},N}, \ldots, [\![\tau_n]\!]_{\mathcal{R},N}\big] \\
\mathcal{E}^{\lambda}(^{\phi}_{\psi}, E, N) & = \times\big(E(x_1^{\tau_1}), \ldots, E(x_n^{\tau_n})\big) \\
\mathcal{E}^{@}(^{\phi}_{\psi}, M_1^{\sigma}, M_2, M_3, N) & = \textbf{let } x^{\times[\sigma,\tau']} = \times(M_1, M_2) \textbf{ in } M_3 \\
\mathcal{E}^{\mathrm{var}}(^{\phi}_{\psi}, N) & = \{x^{\tau} \mapsto \pi_2^{\times}\ x^{\rho'}\} \cup \{{x_i}^{\tau_i} \mapsto \pi_i^{\times}\ \pi_1^{\times}\ x^{\rho'} \mid 1 \le i \le n\}
\end{array}
$$

Flow Partitioning Functions

Each partitioning function maps a flow path to a string identifying a partition.
Strings are concatenated via ":".

$$
\Pi^{\mathcal{R}}_{\mathrm{FS}}(^{l}_{k}) = \begin{cases} \mathcal{R}^{\mathrm{env?}}(^{l}_{k}) : \mathcal{R}^{\mathrm{code?}}(^{l}_{k}) : l & \text{if } \#\mathrm{fv}(^{l}_{k}, P_1) \ge 1 \text{ or not } \mathcal{R}^{\mathrm{code?}}(^{l}_{k}), \\ \mathcal{R}^{\mathrm{env?}}(^{l}_{k}) : \mathrm{true} & \text{otherwise.} \end{cases}
$$

$$
\Pi^{\mathcal{R}}_{\mathrm{ST}} = \langle \Pi^{\mathcal{R},\lambda}_{\mathrm{ST}}, \Pi^{\mathcal{R},@}_{\mathrm{ST}} \rangle \quad \text{where} \quad
\begin{array}{l}
\Pi^{\mathcal{R},\lambda}_{\mathrm{ST}}(^{l}_{k}) = \mathcal{R}^{\mathrm{env?}}(^{l}_{k}) : \mathcal{R}^{\mathrm{code?}}(^{l}_{k}) \\
\Pi^{\mathcal{R},@}_{\mathrm{ST}}(^{l}_{k}) = \begin{cases} \mathcal{R}^{\mathrm{env?}}(^{l}_{k}) : \mathrm{false} : l & \text{if not } \mathcal{R}^{\mathrm{code?}}(^{l}_{k}), \\ \mathcal{R}^{\mathrm{env?}}(^{l}_{k}) : \mathrm{true} & \text{otherwise.} \end{cases}
\end{array}
$$

Figure 7: Example representation map with corresponding flow partitions.

The algorithm implementing the RC stage will generally need to to inspect the entire input program in generating the representation map $\mathcal{R}$ and the partitioning functions $\Pi^{\mathcal{R}}_{\mathrm{FS}}$ and $\Pi^{\mathcal{R}}_{\mathrm{ST}}$. In addition, the components $\langle \mathcal{R}^{\lambda}, \mathcal{R}^{@}, \mathcal{R}^{\mathrm{var}}, \mathcal{R}^{\rightarrow} \rangle$ of the representation map itself will need access to global program information. For this reason, each component will have an extra parameter which will be supplied with the entire program as an argument in the Representation Transformation stage.

Definition 3.6 (Flow Paths and Bundles). Given $l, k \in \mathbf{Label}$, the pair $\genfrac{}{}{0pt}{3}{l}{k}$ is a *flow path.* Given $\phi, \psi \subset \mathbf{Label}$, the pair $\genfrac{}{}{0pt}{3}{\phi}{\psi}$ is a *flow bundle*. We write $\genfrac{}{}{0pt}{3}{l}{k} \in \genfrac{}{}{0pt}{3}{\phi}{\psi}$ to denote that $l \in \phi$ and $k \in \psi$. □

The presence of a flow path $\genfrac{}{}{0pt}{3}{l}{k}$ in a program indicates that the Flow Analysis stage has guessed that the source labelled l might be able to flow to the sink labelled k. A flow bundle $\genfrac{}{}{0pt}{3}{\phi}{\psi}$ can be seen as the collection of the flow paths $\{\genfrac{}{}{0pt}{3}{l}{k} \mid \genfrac{}{}{0pt}{3}{l}{k} \in \genfrac{}{}{0pt}{3}{\phi}{\psi}\}$.

Definition 3.7. For any term N in DLS (distinctly labelled by subterm) form, if $M \equiv (\lambda^{l}_{\psi} x^{\tau}.M') \trianglelefteq N$, $k \in \psi$, and ${x_1}^{\tau_1}, \ldots, {x_n}^{\tau_n}$ are the free variables of M in order of first occurrence from left to right, then the following functions are defined:

$$\begin{array}{ll}
\mathrm{bv}(\genfrac{}{}{0pt}{}{l}{k}, N) & = x^{\tau} \\
\#\mathrm{fv}(\genfrac{}{}{0pt}{}{l}{k}, N) & = n \\
\mathrm{fv}_i(\genfrac{}{}{0pt}{}{l}{k}, N) & = {x_i}^{\tau_i} \quad \text{if } 1 \leq i \leq n \\
\mathrm{body}(\genfrac{}{}{0pt}{}{l}{k}, N) & = M' \\
\mathrm{name}(y^{\sigma}) & = y \\
\mathrm{type}(y^{\sigma}) & = \sigma
\end{array}$$

□

Convention 3.8. Any total function f on flow paths $\genfrac{}{}{0pt}{3}{l}{k}$ is automatically extended to a partial function on flow path bundles $\genfrac{}{}{0pt}{3}{\phi}{\psi}$ as follows:

$$f\left(\genfrac{}{}{0pt}{}{\phi}{\psi}\right) = \begin{cases} f(\genfrac{}{}{0pt}{}{\min\phi}{\min\psi}) & \text{if } \forall \genfrac{}{}{0pt}{}{l}{k} \in \genfrac{}{}{0pt}{}{\phi}{\psi}.\ f(\genfrac{}{}{0pt}{}{\min\phi}{\min\psi}) = f(\genfrac{}{}{0pt}{}{l}{k}), \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

□

In figure 7 we give an example implementation of the RC algorithm. The algorithm specified in figure 7 can produce a representation map that can perform both closure conversion and inlining. To choose function representations, this representation map uses predicates $\mathcal{R}^{\mathrm{code?}}$ and $\mathcal{R}^{\mathrm{env?}}$, which may be arbitrarily chosen total predicates on flow paths, except that an abstraction with free variables must be given some way to carry the values of those free variables and that inlining loops must be avoided. The concrete representations of closures and environments for this example are abstracted in $\mathcal{C} = \langle \mathcal{C}^{\lambda}, \mathcal{C}^{@}, \mathcal{C}^{\mathrm{var}}, \mathcal{C}^{\rightarrow} \rangle$[8] and $\mathcal{E} = \langle \mathcal{E}^{\lambda}, \mathcal{E}^{@}, \mathcal{E}^{\mathrm{var}}, \mathcal{E}^{\rightarrow} \rangle$[9]. The representation map $\mathcal{R}$ and partitioning functions $\Pi^{\mathcal{R}}_{\mathrm{FS}}$ and $\Pi^{\mathcal{R}}_{\mathrm{ST}}$ given in figure 7 are specifically designed to work together; if $\mathcal{R}$ were altered then $\Pi^{\mathcal{R}}_{\mathrm{FS}}$ and $\Pi^{\mathcal{R}}_{\mathrm{ST}}$ might need to change in a corresponding way.

[8]Due to our deliberately restricted subtyping rule, this example uses 0 for flow labels in closures, thus losing flow information in the output. This problem can be solved by adding flow labels to all data types.

[9]This definition is simplified from what will actually work. In order for a closure referencing a μ-bound variable to be a value in λ^{CIL}, the code storing the μ-bound variable value in the closure must be thunkified and the corresponding variable-access code in the function body must force the thunk. This problem seems to represent a flaw in the formulation of λ^{CIL}. Using a λ-graph-based calculus might avoid this problem. The problem does not occur if μ-bindings are restricted to the form **rec** $x^{\tau}.V$ and closure-passing style is used.

3.5 Flow Separation (FS)

The Flow Separation (FS) stage takes as input a typed program P_1 and a flow path partitioning function $\Pi^{\mathcal{R}}_{\mathrm{FS}}$ (supplied by Representation Choices to determine which flow paths can coexist in the same flow path bundles) and returns a transformed typed program P_2. We implement this stage with the FS algorithm, which applies the transformation in figure 8 to the input P_1 using $\Pi^{\mathcal{R}}_{\mathrm{FS}}$ for Π in the figure to obtain the result P_2.[10] More precisely, if we let $\overline{M}$ be an abbreviation for $\mathrm{FS}(M, \Pi^{\mathcal{R}}_{\mathrm{FS}})$, then

$$P_2 \equiv \mathrm{FS}(P_1, \Pi^{\mathcal{R}}_{\mathrm{FS}}) \equiv \overline{P_1}$$

Definition 3.9. If Π is a function on flow paths (source/sink flow label pairs), then a typing $\mathcal{D}$ *respects* Π if $\Pi(\genfrac{}{}{0pt}{3}{\phi}{\psi})$ is defined for every flow bundle $\genfrac{}{}{0pt}{3}{\phi}{\psi}$ (source/sink flow label set pairs) occurring in $\mathcal{D}$ (see convention 3.8). □

The type transformation $\overline{\tau}$ preserves the structure of all types except arrow types, which it transforms to unions of intersections of arrows in a way that respects $\Pi^{\mathcal{R}}_{\mathrm{FS}}$. The term transformation $\overline{M}$ preserves structure except at abstractions, applications, and coercions. An abstraction occurrence is transformed into a virtual variant containing a virtual tuple whose slots contain different type annotations of the same abstraction which are suitable for the different call sites to which the original abstraction flowed. An application occurrence is transformed into a virtual case expression which dispatches on the virtual tags of the abstractions that could reach the application's function (left) slot. Each alternative in the case expression is a virtual copy of the original application. The transformation of a **coerce** expression results in a term that converts a virtual variant of virtual tuples into another virtual variant of virtual tuples.

The type erasure of an abstraction remains the same, while the type erasure of an application is β-expanded from $(M \mathbin{@} N)$ to $((\lambda f.f \mathbin{@} N) \mathbin{@} M)$, since the type erasure of a virtual **case** expression is a β-redex. The transformation could take advantage of pre-existing β-redexes at the cost of a more complex specification. The type erasure of a **coerce** expression is β-expanded from M to $((\lambda f.f) \mathbin{@} M)$, again a result of our formulation of virtual $\mathbf{case}^{\vee}$ expressions and potentially simplifyable in the same way as for applications.

The Flow Separation transformation in figure 8 introduces unneeded singleton virtual variants and virtual tuples, which could be avoided by a more complex specification. In the worst case, Flow Separation may be provided with a partitioning function that assigns every flow path to a distinct partition and this would cause an at-least-quadratic expansion in size of the program representation. We expect that typical partitionings will be coarser-grained functions that avoid this worst-case expansion.

Theorem 3.10 (Flow Separation Correctness). *If*

1. $A \vdash_{\lambda^{\mathrm{CIL}}_{\mathrm{DLO}}} M : \tau$.
2. $\mathrm{t\text{-}nf}(M)$.
3. $\overline{M} \equiv \mathrm{FS}(M, \Pi)$.

then there exists a $\mathcal{D}$ *such that*

1. $\overline{A} \vdash_{\lambda^{\mathrm{CIL}}_{\mathrm{DLS}}} \overline{M} : \overline{\tau}$ *via* $\mathcal{D}$.

[10]Figure 8 is simplified to avoid a subtlety of our parallel term formulation. For correctness, for every **coerce**, we must pretend there is a (possibly useless) **coerce** in each corresponding parallel position so that the type erasure of parallel subterms is changed compatibly.

Flow Separation for Types (non-trivial case only)

The function Π mentioned below is a parameter to the algorithm.

$$\overline{\sigma \xrightarrow[\psi]{\phi} \tau} = \bigvee_{\phi_i \text{ in } \phi_1,\dots,\phi_n} \left[\bigwedge_{\psi_{i,j} \text{ in } \psi_{i,1},\dots,\psi_{i,m(i)}} \left[\overline{\sigma} \xrightarrow[\psi_{i,j}]{\phi_i} \overline{\tau} \right] \right]$$

where $\phi_1, \dots, \phi_n$ partitions ϕ so that $\Pi(\substack{\phi_i \\ \{k\}})$ is defined for $k \in \psi$ and $1 \leq i \leq n$,
$\psi_{i,1}, \dots, \psi_{i,m(i)}$ partitions ψ so that $\Pi(\substack{\{l\} \\ \psi_{i,m(i)}})$ is defined for $l \in \phi_i$ and $1 \leq j \leq m(i)$,
$(\min \phi_i) < (\min \phi_{i+1})$ for $1 \leq i < n$,
$(\min \psi_{i,j}) < (\min \psi_{i,j+1})$ for $1 \leq i \leq n$ and $1 \leq j < m(i)$,
and each partitioning chooses the largest possible partitions.

Flow Separation for Terms (non-trivial cases only)

$$\overline{(\lambda^l_\psi x^\sigma . M^\tau)^\rho} \equiv \left(\mathbf{in}^\vee_1 \ \bigwedge_{j=1}^m \left(\lambda^l_{\psi_j} x^{\overline{\sigma}} . \overline{M} \right) \right)^{\overline{\rho}} \quad \text{where } \overline{\rho} = \bigvee \left[\bigwedge_{j=1}^m \left[\overline{\sigma} \xrightarrow[\psi_j]{\{l\}} \overline{\tau} \right] \right]$$

$$\overline{(M^\rho \mathbin{@}^\phi_k N^\sigma)^\tau} \equiv (\mathbf{case}^\vee \ \overline{M} \ \mathbf{bind} \ f \ \mathbf{in} \ \dots, \ \rho_i \Rightarrow (\pi^\wedge_1 \ f^{\rho_i}) \mathbin{@}^{\phi_i}_k \ \overline{N}, \ \dots)$$

where f is fresh, $1 \leq i \leq n$, $\overline{\rho} = \vee_{i=1}^n [\rho_i]$, and $\rho_i = \bigwedge \left[\overline{\sigma} \xrightarrow[\{k\}]{\phi_i} \overline{\tau} \right]$

$$\overline{(\mathbf{coerce}\,(\rho, \rho')\, M)} \equiv \Big(\mathbf{case}^\vee \ \overline{M} \ \mathbf{bind} \ f \ \mathbf{in}$$

$$\dots, \ \rho_i \Rightarrow \left(\mathbf{in}^\vee_{r(i)} \ \bigwedge_{j=1}^{m'(r(i))} \left(\mathbf{coerce} \left(\rho_{i,q(i,j)}, \rho'_{r(i),j} \right) \pi^\wedge_{q(i,j)} \ f^{\rho_i} \right) \right)^{\overline{\rho'}}, \ \dots \Big)$$

where f is fresh, $1 \leq i \leq n$,
$\overline{\rho} = \vee_{i=1}^n [\rho_i], \quad \rho_i = \wedge_{j=1}^{m(i)} [\rho_{i,j}], \qquad \rho_{i,j} = \overline{\sigma} \xrightarrow[\psi_{i,j}]{\phi_i} \overline{\tau},$
$\overline{\rho'} = \bigvee_{i=1}^{n'} [\rho'_i], \ \ \rho'_i = \bigwedge_{j=1}^{m'(i)} \left[\rho'_{i,j} \right], \ \ \rho'_{i,j} = \overline{\sigma} \xrightarrow[\psi'_{i,j}]{\phi'_i} \overline{\tau},$
and $\phi_i \subseteq \phi'_{r(i)}$ and $\psi_{i,q(i,j)} \supseteq \psi'_{r(i),j}$ for $1 \leq i \leq n$ and $1 \leq j \leq m'(i)$

All other cases are purely structural, e.g., $\overline{\pi^P_i \ M} = \pi^P_i \ \overline{M}$ and $\overline{x^\tau} = x^{\overline{\tau}}$.

Figure 8: Flow Separation transformation.

2. *$\mathcal{D}$ respects* Π.

3. *If $\tau = o$, M is closed, $M \xrightarrow{\text{nf}}_{\text{r}} N$, and $\overline{M} \xrightarrow{\text{nf}}_{\text{r}} N'$, then $N \equiv N'$ (meaning preservation).*

□

3.6 Splitting/Tagging (ST)

The Representation Choices stage can (1) specify for an individual function that it has multiple closure creation (and variable access) methods for different call sites to which it can flow, and (2) specify for an individual call site that it must handle multiple function calling conventions for different functions which can flow to it. When this has happened, the Flow Separation stage will have used the features of our intermediate language λ^{CIL} in creating multiple parallel versions of the original function or call site which are connected by a virtual tuple $\wedge(N_1, \dots, N_m)$ or a virtual $\mathbf{case}^\vee$ expression ($\mathbf{case}^\vee \ M \ \mathbf{bind} \ x \ \mathbf{in} \ \sigma_1 \Rightarrow N_1, \dots, \sigma_m \Rightarrow N_m$). In these terms, the *parallel* subterms $N_1, \dots, N_m$ represent different type annotations of the *same* program phrase. If the type erasures of N_i and N_j for $i \neq j$ are changed incompatibly, the result will be ill typed. We make it possible for this to happen by (1) *splitting* some function definitions and having the different versions go to different call sites and (2) *tagging* (injecting into sum type) some function definitions and having call sites discriminate on the tags. The Splitting/Tagging (ST) stage does this by changing some occurrences of "$\wedge$" and "$\vee$" into "$\times$" and "$+$", respectively.

Definition 3.11 (Parallel). Distinct subterm occurrences $N_i, N_j \lhd M$ are *parallel*, written "$N_i \parallel N_j$ in M", if and only if $M \equiv Cp[N_1, \dots, N_m]$ for some parallel context Cp and $i, j \in \{1, \dots, m\}$ and $i \neq j$. □

Definition 3.12. If Π is a function from flow paths to partition identifiers, then for some term M,

1. The parallel abstractions of M are *legal with respect to* Π if whenever $(\lambda^i_{\psi_1} x_1{}^{\sigma_1}.N_1) \| (\lambda^j_{\psi_2} x_2{}^{\sigma_2}.N_2)$ in M then $\Pi(\substack{i \\ \psi_1}) = \Pi(\substack{j \\ \psi_2})$. (The undefined result is *not* equal to itself.)

2. The parallel applications of M are *legal with respect to* Π if whenever $(N_1 \mathbin{@}^{\phi_1}_i N'_1) \| (N_2 \mathbin{@}^{\phi_2}_j N'_2)$ in M then $\Pi(\substack{\phi_1 \\ i}) = \Pi(\substack{\phi_2 \\ j})$.

□

Definition 3.13 (Joins). A subterm occurrence $J \trianglelefteq M$ *joins* distinct subterm occurrences $N, N' \lhd J$ when J is either a virtual tuple $\wedge(N_1, \dots, N_k)$ or a virtual $\mathbf{case}^\vee$ expression

$$(\mathbf{case}^\vee \ M \ \mathbf{bind} \ x \ \mathbf{in} \ \sigma_1 \Rightarrow N_1, \dots, \sigma_k \Rightarrow N_k)$$

and $N \trianglelefteq N_i$ and $N' \trianglelefteq N_j$ where $i \neq j$. □

The Splitting/Tagging (ST) stage takes as input a program P_2 and a pair of flow path partitionings $\Pi^{\mathcal{R}}_{\text{ST}} = \langle \Pi^{\mathcal{R},\lambda}_{\text{ST}}, \Pi^{\mathcal{R},@}_{\text{ST}} \rangle$ (supplied by Representation Choices to determine which functions and call sites will have compatible transformations) and returns as output the appropriately transformed typed program P_3. We provide the following ST algorithm to implement this stage. The ST algorithm inspects the derivation $\mathcal{D}$ which types P_2, transforms $\mathcal{D}$ into

$\mathcal{D}'$ as specified below, and returns as output the transformed typed program P_3 typed via $\mathcal{D}'$, i.e.,

$$P_3 = \mathrm{ST}(P_2, \langle \Pi_{\mathrm{ST}}^{\mathcal{R},\lambda}, \Pi_{\mathrm{ST}}^{\mathcal{R},@} \rangle)$$

In the algorithm below, $\langle \Pi^\lambda, \Pi^@ \rangle$ are the parameters which will be instantiated to $\langle \Pi_{\mathrm{ST}}^{\mathcal{R},\lambda}, \Pi_{\mathrm{ST}}^{\mathcal{R},@} \rangle$.

1. For every pair of parallel abstraction occurrences $M_1 \parallel M_2$ in P_2 that are not legal with respect to Π^λ, find the subterm occurrence $J \trianglelefteq P_2$ that joins M_1 and M_2. Mark J's type constructor symbol as either $\bar{\wedge}$ or $\bar{\vee}$. Do similarly for every pair of parallel application occurrences $M_1' \parallel M_2'$ in P_2 which are not legal with respect to $\Pi^@$.

2. Propagate the markings on $\bar{\wedge}$ and $\bar{\vee}$. When a marked $\bar{\wedge}$ or $\bar{\vee}$ is matched against an unmarked $\wedge$ or $\vee$ in $\mathcal{D}$ by the typing rules, add marks to the unmarked symbols.

3. When there is a subterm $M \trianglelefteq P_2$ which is a marked $\wedge, \vee$-introduction or elimination term and there is a distinct subterm M' such that $M \parallel M'$ in P_2, find the subterm $J \trianglelefteq P_2$ which joins M and M' and mark J.

4. Repeat steps 2 and 3 until they do nothing.

5. Change every marked $\bar{\wedge}$ into $\times$ and every marked $\bar{\vee}$ into $+$.

This algorithm can sometimes split excessively, but common-subexpression elimination can compensate.

Theorem 3.14 (Splitting/Tagging Correctness). *If*

1. $A \vdash_{\lambda^{\mathrm{CIL}}_{\mathrm{DLS}}} M : \tau$ *via* $\mathcal{D}$.

2. t-nf(M).

3. *The typing* $\mathcal{D}$ *respects* Π^λ *and* $\Pi^@$.

4. $M' \equiv \mathrm{ST}(M, \langle \Pi^\lambda, \Pi^@ \rangle)$.

then there exist A', τ', *and* $\mathcal{D}'$ *such that*

1. $A' \vdash_{\lambda^{\mathrm{CIL}}_{\mathrm{DLS}}} M' : \tau'$ *via* $\mathcal{D}'$.

2. *The parallel abstractions of* M' *are legal with respect to* Π^λ *and the parallel applications of* M' *are legal with respect to* $\Pi^@$.

3. *For any* Π', *if* $\mathcal{D}$ *respects* Π' *then* $\mathcal{D}'$ *respects* Π'.

4. *If* τ *does not mention* $\wedge$ *or* $\vee$, *then* $\tau = \tau'$.

5. *If* $\tau = o$, M *is closed,* $M \xrightarrow{\mathrm{nf}}_{\mathrm{r}} N$, *and* $M' \xrightarrow{\mathrm{nf}}_{\mathrm{r}} N'$, *then* $N \equiv N'$ *(meaning preservation).*

□

3.7 Representation Transformation (RT)

The Representation Transformation (RT) stage takes as input a typed program P_3 and a representation map $\mathcal{R}$ and then traverses the syntax tree of P_3 transforming abstractions, applications, variable occurrences and function types as specified by $\mathcal{R}$ to produce the resulting typed program P_4. We implement this stage with the RT algorithm defined in figure 9. More precisely,

$$P_4 = \mathrm{RT}(P_3, \mathcal{R}) = [\![P_3]\!]_{\mathcal{R}, E_{\mathrm{start}}, P_3}$$

where E_{start} is the function such that $E_{\mathrm{start}}(x^\tau) = x^{[\![\tau]\!]_{\mathcal{R},P_3}}$. Note that this implementation of the RT algorithm works with the example implementation of the RC algorithm in figure 7. A substantially different implementation of RC might require corresponding changes to RT.

The transformation on types proceeds structurally except for function types. Because the types of free variables of an abstraction are exposed in the transformed type of the transformed abstraction, some transformed types can be infinite. These types will always have a finite representation. An implementation of the RT algorithm is required to detect when this happens and tie off the recursion by inserting a μ-binding. The transformation on terms is defined on the structure of terms. In the application case, $[\![\cdot]\!]_{\mathcal{R},\cdot,\cdot}$ provides to $\mathcal{R}^@$ not only the transformations of the application's syntactic components, but sometimes also provides the transformation of a function body to inline in place of the operator. To avoid non-termination, the output of the RC algorithm must avoid specifying inlining loops.

Theorem 3.15 (Well-Definedness of RC and RT). *If*

1. $A \vdash_{\lambda^{\mathrm{CIL}}_{\mathrm{DLO}}} M : \tau$

2. $\langle \mathcal{R}, \Pi_{\mathrm{FS}}^{\mathcal{R}}, \Pi_{\mathrm{ST}}^{\mathcal{R}} \rangle = \mathrm{RC}(M)$ *(using RC from figure 7).*

3. $A' \vdash_{\lambda^{\mathrm{CIL}}_{\mathrm{DLS}}} M' : \tau'$ *where* $M' \equiv \mathrm{ST}(\mathrm{FS}(M, \Pi_{\mathrm{FS}}^{\mathcal{R}}), \Pi_{\mathrm{ST}}^{\mathcal{R}})$.

then $[\![A']\!]_{\mathcal{R},M'} \vdash_{\lambda^{\mathrm{CIL}}} [\![M']\!]_{\mathcal{R},E_{\mathrm{start}},M'} : [\![\tau']\!]_{\mathcal{R},M'}$. □

We have not yet proved that applying the RT algorithm preserves the meaning of the program.

4 Related Work

General research into the use of intersection types which has influenced us includes the work of Van Bakel [27] and Jim [14, 15]. Relevant research on both intersection and union types includes the work by Pierce [21], Aiken, Wimmers, and Lakshman [3], Trifonov, Smith, and Eifrig [26, 9], and Barbanera, Dezani-Ciancaglini, and de'Liguoro [7]. Of the above, only Pierce considers intersection and union types in an explicitly typed language. Even that is somewhat distant from our work because Pierce includes a general subtyping relation on intersection and union types which we deliberately avoid.

Flow-types were first named by Heintze in [11], and are also used by Banerjee in [6]. We differ from the above by including sink labels as well as source labels.

Our closure conversion is most closely related to Wand and Steckler's work on selective and lightweight closure conversion [28]. They give a 0CFA-based algorithm in the untyped setting and prove its correctness. Their algorithm can avoid closure creation when all the functions flowing to a particular call site are closed. In addition, the *lightweight* optimization uses an escape analysis to determine when the value of a free variable is available at the call site. Their algorithm restricts a function to a single representation and requires all functions flowing to a particular application site to observe the same application protocol.

Hannan [10] describes the annotations on types to perform the optimizations of [28] in a typed system. He specifies a conversion, but does not supply an algorithm. He does not handle multiple representations for a single function or mixed calling conventions at a single call site.

Minamide, Morrisett and Harper present an algorithm for typed closure conversion [18, 19]. They use purely local transformations to perform closure conversion. Their

Representation Transformation for Types (non-trivial case only)

$$[\![\sigma \xrightarrow[\psi]{\phi} \tau]\!]_{\mathcal{R},N} = \mathcal{R}^{\rightarrow}(\substack{\phi \\ \psi}, [\![\sigma]\!]_{\mathcal{R},N}, [\![\tau]\!]_{\mathcal{R},N}, N)$$

All other cases are purely structural, e.g., $[\![Q[\tau_1, \ldots, \tau_n]]\!]_{\mathcal{R},N} = Q\big[[\![\tau_1]\!]_{\mathcal{R},N}, \ldots, [\![\tau_n]\!]_{\mathcal{R},N}\big]$.

Representation Transformation for Terms (non-trivial cases only)

$$\begin{array}{ll}
[\![x^\tau]\!]_{\mathcal{R},E,N} & \equiv E(x^\tau) \\
[\![\lambda^l_\psi x^\sigma.M]\!]_{\mathcal{R},E,N} & \equiv \mathcal{R}^{\lambda}(\substack{\{l\} \\ \psi}, E, [\![M]\!]_{\mathcal{R},\mathcal{R}^{\mathrm{var}}(\substack{\{l\} \\ \psi}, E, N), N}, N) \\
[\![M_1 \mathbin{@}^\phi_k M_2]\!]_{\mathcal{R},E,N} & \equiv \mathcal{R}^{@}(\substack{\phi \\ \{k\}}, [\![M_1]\!]_{\mathcal{R},E,N}, [\![M_2]\!]_{\mathcal{R},E,N}, M_3, N) \\
 & \quad \text{where } M_3 = \begin{cases} [\![\mathrm{body}(\substack{\phi \\ \{k\}}, N)]\!]_{\mathcal{R},\mathcal{R}^{\mathrm{var}}(\substack{\phi \\ \{k\}}, E, N), N} & \text{if not } \mathcal{R}^{\mathrm{code?}}(\substack{\phi \\ k}), \\ c & \text{otherwise.} \end{cases} \\
[\![\mathbf{rec}\ x^\tau.M]\!]_{\mathcal{R},E,N} & \equiv \mathbf{rec}\ x^{[\![\tau]\!]_{\mathcal{R},N}}.[\![M]\!]_{\mathcal{R},E[x^\tau \mapsto x^{[\![\tau]\!]_{\mathcal{R},N}}],N} \\
[\![\mathbf{let}\ x^\tau = M_1\ \mathbf{in}\ M_2]\!]_{\mathcal{R},E,N} & \equiv \mathbf{let}\ x^{[\![\tau]\!]_{\mathcal{R},N}} = [\![M_1]\!]_{\mathcal{R},E,N}\ \mathbf{in}\ [\![M_2]\!]_{\mathcal{R},E[x^\tau \mapsto x^{[\![\tau]\!]_{\mathcal{R},N}}],N} \\
[\![\mathbf{case}^S\ M\ \mathbf{bind}\ x\ \mathbf{in} \ldots, \tau_i \Rightarrow M_i, \ldots]\!]_{\mathcal{R},E,N} & \equiv \mathbf{case}^S\ [\![M]\!]_{\mathcal{R},E,N}\ \mathbf{bind}\ x\ \mathbf{in} \ldots, [\![\tau_i]\!]_{\mathcal{R},N} \Rightarrow [\![M_i]\!]_{\mathcal{R},E_i,N}, \ldots \\
 & \quad \text{where } E_i = E[x^{\tau_i} \mapsto x^{[\![\tau_i]\!]_{\mathcal{R},N}}]
\end{array}$$

All other cases are purely structural.

Figure 9: Representation Transformation.

framework supports multiple closure representations but it is restricted in that all representations share the same interface.

There are many studies of closure conversion which focus on the data structures used for variable lookup and the tradeoffs between sharing and time to lookup the value of a variable [17, 25, 4, 23]. Our framework abstracts out data structure representation issues.

Jagannathan and Wright discuss flow-directed inlining in an untyped system [13]. Their paper examines heuristics for selecting application sites for inlining. They do not discuss inlining open terms. Plevyak and Chien have experimented with flow directed inlining in an object oriented language [22].

5 Future Work

Although this paper focuses on the transformation of function representations, our framework can be extended to transformations on other data types. These extensions involve adding flow labels to all value producing and consuming forms. We expect to handle specialized tuple, variant, and inductive datatype representations.

We are implementing our framework and will experiment with various flow analysis algorithms and representation decision heuristics.

We will refine the example representation transformation to include analysis enabling lightweight closure conversion [28]. This allows variables available at the call site to be omitted from the environment of the closure. This will also improve inlining.

In this paper we have presented a simple algorithm for the Splitting/Tagging transformation. We plan to develop and implement a more efficient algorithm.

An important practical issue in compiling with types is controlling the size of the intermediate representations. Our current language, following the style of [16], duplicates terms when it duplicates types. Our language is convenient for specifying our framework, but for implementation a considerable size savings can be obtained by using a typed calculus with intersection and union types in the style of [29].

Finally, we plan to study the interaction of our current approach with separate compilation.

6 Acknowledgements

We would like to thank Mitch Wand, Will Clinger, Paul Steckler and several anonymous reviewers for their helpful comments and suggestions. We would also like to thank our fellow Church Project members for their support and encouragement, especially Assaf Kfoury and Glenn Holloway.

References

[1] ACM. *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, 1994.

[2] ACM. *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[3] Alexander S. Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In POPL '94 [1], pages 163–173.

[4] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[5] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Conf. Rec. 16th Ann. ACM Symp. Princ. of Prog. Langs.*, pages 293–302, 1989.

[6] Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. 1997 Int'l Conf. Functional Programming.* ACM Press, 1997.

[7] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Inform. & Comput.*, 119:202–230, 1995.

[8] H[enrik] P[ieter] Barendregt. Lambda calculi with types. In S[amson] Abramsky, Dov M. Gabbay, and T[homas] S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[9] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proc. 1995 Mathematical Foundations of Programming Semantics Conf.*, volume 1 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1995.

[10] John Hannan. Type systems for closure conversion. In *Workshop on Types for Program Analysis*, pages 48–62, 1995. The TPA '95 proceedings are DAIMI PB-493.

[11] Nevin Heintze. Control-flow analysis and type systems. In *Proc. 2nd Int'l Static Analysis Symp.*, volume 983 of *LNCS*, pages 189–206, 1995.

[12] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, pages 393–407. ACM, 1995.

[13] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *Proc. ACM SIGPLAN '96 Conf. Prog. Lang. Design & Impl.*, pages 193–205, 1996.

[14] Trevor Jim. Rank 2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, November 1995.

[15] Trevor Jim. What are principal typings and what are they good for? In POPL '96 [2].

[16] A. J. Kfoury and J. B. Wells. New notions of reduction and non-semantic proofs of β-strong normalization in typed λ-calculi. In *Proc. 10th Ann. IEEE Symp. Logic in Computer Sci.*, pages 311–321, 1995.

[17] David Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: An optimizing compiler for Scheme. In *Proc. SIGPLAN '86 Symp. Compiler Construction*, pages 219–233, 1986.

[18] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In POPL '96 [2].

[19] Greg Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, 1995.

[20] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: A technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conf.*, 1993.

[21] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[22] John Plevyak and Andrew A. Chien. Iterative flow analysis. Submitted, July 1995.

[23] Zhong Shao. *Compiling Standard ML for Efficient Execution on Modern Machines.* PhD thesis, Princeton University, 1994.

[24] Olin Shivers. *Control Flow Analysis of Higher Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

[25] Guy Steele. Rabbit: A compiler for Scheme. Technical Report MIT/AI-TR-474, Massachusetts Institute of Technology, 1978.

[26] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proc. 3rd Int'l Static Analysis Symp.*, pages 349–365, 1996.

[27] Steffen J. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems.* PhD thesis, Catholic University of Nijmegen, 1993.

[28] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In POPL '94 [1], pages 435–445.

[29] J. B. Wells. Intersection types revisited in the Church style. Manuscript, June 1996.

[30] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, pages 757–771, 1997. Superseded by [31].

[31] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Programming*, 200X. To appear. Supersedes [30].

Appendix: λ^{CIL}

This appendix presents the essential definitions for and theorems about λ^{CIL}. For a more thorough explanation and a discussion of design decisions, see [30], an extended version of which is available from http://www.cs.bu.edu/groups/church/reports/.

The statement $X \lhd Y$ means that the syntactic entity X occurs properly within the syntactic entity Y; $X \unlhd Y$ has the same meaning except X and Y may be the same.

A *simple* notion of reduction (n.o.r.) R is a pair $(\leadsto_R, \mathbf{C}_R)$ of a redex/contractum relation $\leadsto_R$ and a set of reduction contexts $\mathbf{C}_R$. The statement $M \leadsto_R N$ means M is an *R-redex* and N is the *R-contractum* of M. For a simple n.o.r., $M \longrightarrow_R N$ means M is transformed into N by contracting R-redexes in positions in M specified by an *R-reduction context*, i.e., there are a context $C \in \mathbf{C}_R$ with k holes and terms M_i and N_i for $i \in \{1, \dots, k\}$ such that $M \equiv C[M_1, \dots, M_k]$ and $N \equiv C[N_1, \dots, N_k]$ and $M_i \leadsto_R N_i$ for $i \in \{1, \dots, k\}$. A *composite* n.o.r. R is a rule composing reduction steps of simple n.o.r.'s; in this case $M \longrightarrow_R N$ means M and N are related by the rule. Writing "$\twoheadrightarrow_R$" denotes the transitive and reflexive closure of "$\longrightarrow_R$". A term M is in *normal form* with respect to R, written R-nf(M), when there is no term N such that $M \longrightarrow_R N$. The statement $M \xrightarrow{\text{nf}}_R N$ means $M \twoheadrightarrow_R N$ and R-nf(N).

Figure 10 shows the syntax and semantics of the untyped language $\lambda^{\text{CIL}}_{\text{ut}}$.

Theorem A.1 (Confluence of Untyped Reduction). *If $\hat{M} \twoheadrightarrow_{\text{ut}} \hat{N}_1$ and $\hat{M} \twoheadrightarrow_{\text{ut}} \hat{N}_2$, then there exists $\hat{M}'$ such that $\hat{N}_1 \twoheadrightarrow_{\text{ut}} \hat{M}'$ and $\hat{N}_2 \twoheadrightarrow_{\text{ut}} \hat{M}'$.* □

Figure 11 shows the syntax of our explicitly typed language λ^{CIL}.

Convention A.2 (Bound Variable Names). For convenience, we assume by α-conversion that parallel (cf. definition 3.11) variable bindings use the same variable name. □

Our definition of type equality on recursive types is standard [8]. We do not distinguish between equal recursive types in any context, so we have no rules for folding or unfolding recursive types. We assume free type variables do not occur (i.e., every type variable will be bound by μ) in typing derivations and terms.

Figure 12 gives the typing rules of λ^{CIL}. If A is a *type environment*, then $A, x{:}\tau$ denotes A extended to map x to type τ. The domain of definition of A is DomDef(A). A triple $A \vdash M : \tau$ is a *judgement.* A *derivation* $\mathcal{D}$ is a sequence of judgements, each obtained from the previous ones by one of the typing rules. We write "$A \vdash_{\lambda^{\text{CIL}}} M : \tau$ via $\mathcal{D}$" to mean $\mathcal{D}$ is valid in λ^{CIL} and $\mathcal{D}$ ends with $A \vdash M : \tau$. In this case, $\mathcal{D}$ is a *typing* for M and M is *well typed.* The statement $A \vdash_{\lambda^{\text{CIL}}} M : \tau$ means there exists some $\mathcal{D}$ such that $A \vdash_{\lambda^{\text{CIL}}} M : \tau$ via $\mathcal{D}$. For any subset $\lambda^{\text{CIL}}_{\text{X}}$ of λ^{CIL}, the statement $A \vdash_{\lambda^{\text{CIL}}_{\text{X}}} M : \tau$ means $A \vdash M : \tau$ is derivable in that subset. The notation M^τ asserts that M is well typed and has type τ.

Theorem A.3 (Uniqueness of Typings). *For $M \in$ **Term**, there is at most one type environment A and type τ such that* DomDef(A) = FV(M) *and $A \vdash_{\lambda^{\text{CIL}}} M : \tau$.* □

The call-by-value reduction rules for our typed language λ^{CIL} are in figure 13. We assume terms are always kept in t-normal form.

Untyped Syntax

$$
\begin{array}{rcl}
x, y, z \in \mathbf{Variable} & & \\
c \in \mathbf{Constant} & & \\
\hat{C} \in \mathbf{UntContext} & ::= & \square \mid c \mid x \mid \mathbf{rec}\ x.\hat{C} \mid \lambda x.\hat{C} \mid \hat{C}_1 \mathbin{@} \hat{C}_2 \\
& & \mid \times\Big(\hat{C}_1, \ldots, \hat{C}_n\Big) \mid \pi_i^{\times}\,\hat{C} \\
& & \mid \mathbf{in}_i^{+}\,\hat{C} \mid \mathbf{case}^{+}\,\hat{C}\ \mathbf{bind}\ x\ \mathbf{in}\ \hat{C}_1, \ldots, \hat{C}_n \\
\hat{M}, \hat{N} \in \mathbf{UntTerm} & = & \{\, \hat{C} \mid \square \not\trianglelefteq \hat{C} \,\} \\
\hat{V} \in \mathbf{UntValue} & ::= & c \mid \lambda x.\hat{M} \mid \times\Big(\hat{V}_1, \ldots, \hat{V}_n\Big) \mid \mathbf{in}_i^{+}\,\hat{V}
\end{array}
$$

Untyped Reduction

$$
\begin{array}{lll}
(\lambda x.\hat{M}) \mathbin{@} \hat{V} & \leadsto_{\mathrm{ut}} \hat{M}[x{:=}\hat{V}] & \\
\pi_i^{\times} \times\Big(\hat{V}_1, \ldots, \hat{V}_n\Big) & \leadsto_{\mathrm{ut}} \hat{V}_i & \text{if } 1 \le i \le n \\
\mathbf{case}^{+}\,(\mathbf{in}_i^{+}\hat{V})\ \mathbf{bind}\ x\ \mathbf{in}\ \hat{M}_1, \ldots, \hat{M}_n & \leadsto_{\mathrm{ut}} (\lambda x.\hat{M}_i) \mathbin{@} \hat{V} & \text{if } 1 \le i \le n \\
\mathbf{rec}\ x.\hat{M} & \leadsto_{\mathrm{ut}} \hat{M}[x{:=}(\mathbf{rec}\ x.\hat{M})] &
\end{array}
$$

Reduction contexts: $\mathbf{C}_{\mathrm{ut}} = \{\, \hat{C} \mid \hat{C} \in \mathbf{UntContext}$ and $\hat{C}$ has exactly one hole$\,\}$

Figure 10: Untyped language $\lambda_{\mathrm{ut}}^{\mathrm{CIL}}$.

Theorem A.4 (Subject Reduction). *If $M \longrightarrow_{\mathrm{r}} N$ and $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$, then $A \vdash_{\lambda^{\mathrm{CIL}}} N : \tau$.* □

Theorem A.5 (Typed/Untyped Reduction Correspondence). *If $A \vdash_{\lambda^{\mathrm{CIL}}} M : \tau$, then*

1. *If $M \longrightarrow_{\mathrm{r}} N$, then $|M| \longrightarrow_{\mathrm{ut}} |N|$.*
2. *If $|M| \longrightarrow_{\mathrm{ut}} \hat{N}$, then there exists a term N where $M \longrightarrow_{\mathrm{r}} N$ and $|N| \equiv \hat{N}$.*

□

Theorem A.6 (Confluence of Typed Reduction). *If $M \twoheadrightarrow_{\mathrm{r}} N_1$ and $M \twoheadrightarrow_{\mathrm{r}} N_2$, then there exist M_1' and M_2' such that $|M_1'| \equiv |M_2'|$ and $N_1 \twoheadrightarrow_{\mathrm{r}} M_1'$ and $N_2 \twoheadrightarrow_{\mathrm{r}} M_2'$.* □

Syntax Shared between Types and Terms

$$Q ::= P \mid S \qquad S ::= \vee \mid + \qquad P ::= \wedge \mid \times \qquad l, k \in \mathbf{Label} = \mathbb{N} \qquad \varnothing \neq \phi, \psi \subset \mathbf{Label}$$

Types

$$\begin{array}{rcl} \alpha & \in & \mathbf{TypeVariable} \\ \rho, \sigma, \tau & ::= & o \mid \upsilon_1 \xrightarrow[\psi]{\phi} \upsilon_2 \mid Q[\upsilon_1, \ldots, \upsilon_n] \mid \mu\alpha.\tau \\ \upsilon & ::= & \alpha \mid \tau \end{array}$$

Type Equality

$$\sigma = \tau \quad \text{iff} \quad U(\sigma) \text{ and } U(\tau)\text{, the infinite unfoldings of } \sigma \text{ and } \tau\text{, are identical}$$

Type-Annotated Contexts

$$\begin{array}{rcl} C \in \mathbf{Context} & ::= & \square \mid c \mid x^{\tau} \mid \mathbf{rec}\ x^{\tau}.C \mid \lambda^{l}_{\psi} x^{\tau}.C \mid C_1 \mathbin{@^{\phi}_{k}} C_2 \\ & \mid & P(C_1, \ldots, C_n) \mid \pi^{P}_{i}\, C \mid \mathbf{coerce}\,(\sigma, \tau)\, C \mid \mathbf{let}\ x^{\tau} = C_1\ \mathbf{in}\ C_2 \\ & \mid & \left(\mathbf{in}^{S}_{i}\ C\right)^{\tau} \mid \mathbf{case}^{S}\, C\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow C_1, \ldots, \tau_n \Rightarrow C_n \end{array}$$

Type Erasure (a partial function)

$$\begin{array}{llll} |\square| & \equiv \square & |c| & \equiv c \\ |x^{\tau}| & \equiv x & |\mathbf{rec}\ x^{\tau}.C| & \equiv \mathbf{rec}\ x.|C| \\ \left|\lambda^{l}_{\psi} x^{\tau}.C\right| & \equiv \lambda x.|C| & \left|C_1 \mathbin{@^{\phi}_{k}} C_2\right| & \equiv |C_1| \mathbin{@} |C_2| \\ |\times(C_1, \ldots, C_n)| & \equiv \times(|C_1|, \ldots, |C_n|) & |\mathbf{coerce}\,(\sigma, \tau)\, C| & \equiv |C| \\ \left|\pi^{\times}_{i}\, C\right| & \equiv \pi^{\times}_{i}\, |C| & |\pi^{\wedge}_{i}\, C| & \equiv |C| \\ \left|\left(\mathbf{in}^{+}_{i}\ C\right)^{\tau}\right| & \equiv \mathbf{in}^{+}_{i} |C| & |(\mathbf{in}^{\vee}_{i}\ C)^{\tau}| & \equiv |C| \end{array}$$

$$|\mathbf{let}\ x^{\tau} = C_1\ \mathbf{in}\ C_2| \equiv (\lambda x.|C_1|) \mathbin{@} |C_2|$$

$$\left|\mathbf{case}^{+}\, C\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow C_1, \ldots, \tau_n \Rightarrow C_n\right| \equiv \mathbf{case}^{+}\, |C|\ \mathbf{bind}\ x\ \mathbf{in}\ |C_1|, \ldots, |C_n|$$

$$|\mathbf{case}^{\vee}\, C\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow C_1, \ldots, \tau_n \Rightarrow C_n| \equiv \begin{cases} (\lambda x.|C_1|) \mathbin{@} |C| & \text{if } |C_1| \equiv \cdots \equiv |C_n|, \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

$$|\wedge(C_1, \ldots, C_n)| \equiv \begin{cases} |C_1| & \text{if } |C_1| \equiv \cdots \equiv |C_n|, \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

Type-Annotated Terms, Values, Parallel Contexts

$$\begin{array}{rl} M, N \in \mathbf{Term} & = \{\, C \mid \text{the type erasure } |C| \in \mathbf{UntTerm} \,\} \\ V \in \mathbf{Value} & = \{\, C \mid \text{the type erasure } |C| \in \mathbf{UntValue} \,\} \\ \mathit{Cp} \in \mathbf{ParallelContext} & = \{\, C \mid \text{the type erasure } |C| \text{ has exactly one hole} \,\} \end{array}$$

Syntactic Sugar for Examples

$$\text{bool} = +[\times[\,], \times[\,]] \qquad \text{true} \equiv \left(\mathbf{in}^{+}_{1}\ \times(\,)\right)^{\text{bool}} \qquad \text{false} \equiv \left(\mathbf{in}^{+}_{2}\ \times(\,)\right)^{\text{bool}}$$

$$(\mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3) \equiv \mathbf{case}^{+}\, M_1\ \mathbf{bind}\ x\ \mathbf{in}\ \times[\,] \Rightarrow M_2, \times[\,] \Rightarrow M_3 \quad \text{where } x \text{ is fresh}$$

Figure 11: Syntax of explicitly typed language λ^{CIL}.

$$\begin{array}{llll}
\text{(const)} & \dfrac{}{A \vdash c : o} & \text{(var)} & \dfrac{}{A, x{:}\tau \vdash x^{\tau} : \tau} \\[2ex]
(\to \text{elim}) & \dfrac{A \vdash M : \sigma \xrightarrow[\{k\}]{\phi} \tau;\ A \vdash N : \sigma}{A \vdash M\ @^{\phi}_{k}\ N : \tau} & (\to \text{intro}) & \dfrac{A, x{:}\sigma \vdash M : \tau}{A \vdash \lambda^{l}_{\psi} x^{\sigma}.M : \sigma \xrightarrow[\psi]{\{l\}} \tau} \\[2ex]
(\times \text{ intro}) & \dfrac{\forall^{n}_{i=1}.\ A \vdash M_i : \tau_i}{A \vdash \times(M_1, \ldots, M_n) : \times[\tau_1, \ldots, \tau_n]} & \text{(coerce)} & \dfrac{A \vdash M : \sigma;\ \sigma \leq \tau}{A \vdash \mathbf{coerce}\,(\sigma, \tau)\, M : \tau} \\[2ex]
(\wedge \text{ intro}) & \dfrac{\forall^{n}_{i=1}.\ A \vdash M_i : \tau_i;\ |M_1| \equiv \cdots \equiv |M_n|}{A \vdash \wedge(M_1, \ldots, M_n) : \wedge[\tau_1, \ldots, \tau_n]} & \text{(rec)} & \dfrac{A, x{:}\tau \vdash M : \tau}{A \vdash \mathbf{rec}\ x^{\tau}.M : \tau} \\[2ex]
(\times,\wedge \text{ elim}) & \dfrac{A \vdash M : P[\tau_1, \ldots, \tau_n];\ 1 \leq i \leq n}{A \vdash \pi^{P}_{i}\ M : \tau_i} & (\text{arrow-}\leq) & \dfrac{\phi \subseteq \phi';\ \psi' \subseteq \psi}{\sigma \xrightarrow[\psi]{\phi} \tau \leq \sigma \xrightarrow[\psi']{\phi'} \tau} \\[2ex]
(+,\vee \text{ intro}) & \dfrac{A \vdash M : \tau_i;\ 1 \leq i \leq n}{A \vdash \left(\mathbf{in}^{S}_{i}\ M\right)^{S[\tau_1, \ldots, \tau_n]} : S[\tau_1, \ldots, \tau_n]} & \text{(let)} & \dfrac{A, x{:}\sigma \vdash N : \tau;\ A \vdash M : \sigma}{A \vdash \mathbf{let}\ x^{\sigma} = M\ \mathbf{in}\ N : \tau} \\[2ex]
(+ \text{ elim}) & \dfrac{A \vdash M : +[\tau_1, \ldots, \tau_n];\ \forall^{n}_{i=1}.\ A, x{:}\tau_i \vdash M_i : \tau}{A \vdash \mathbf{case}^{+}\ M\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n : \tau} & & \\[2ex]
(\vee \text{ elim}) & \dfrac{A \vdash M : \vee[\tau_1, \ldots, \tau_n];\ \forall^{n}_{i=1}.\ A, x{:}\tau_i \vdash M_i : \tau;\ |M_1| \equiv \cdots \equiv |M_n|}{A \vdash \mathbf{case}^{\vee}\ M\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n : \tau} & &
\end{array}$$

Figure 12: Typing rules of explicitly typed language λ^{CIL}.

Main Notion of Reduction for Type-Annotated Terms

$$M \longrightarrow_{\mathrm{r}} N \quad \text{iff} \quad \exists M', N'.\ (M \xrightarrow{\mathrm{nf}}_{\mathrm{t}} M' \longrightarrow_{\mathrm{c}} N' \xrightarrow{\mathrm{nf}}_{\mathrm{t}} N)$$

Computation Reduction

$$\begin{array}{lll}
\mathbf{let}\ x^{\tau} = V\ \mathbf{in}\ M & \leadsto_{\mathrm{c}} M[x{:=}V] & \\
\pi^{\times}_{i}\ \times(V_1, \ldots, V_n) & \leadsto_{\mathrm{c}} V_i & \text{if } 1 \leq i \leq n \\
\mathbf{case}^{+}\ \left(\mathbf{in}^{+}_{i}\ V\right)^{\tau}\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n & \leadsto_{\mathrm{c}} \mathbf{let}\ x^{\tau_i} = V\ \mathbf{in}\ M_i & \text{if } 1 \leq i \leq n \\
\mathbf{rec}\ x^{\tau}.M & \leadsto_{\mathrm{c}} M[x{:=}(\mathbf{rec}\ x^{\tau}.M)] &
\end{array}$$

Reduction contexts: $\mathbf{C}_{\mathrm{c}} = \mathbf{ParallelContext}$

Type-Annotation-Simplification Reduction

$$\begin{array}{lll}
(\lambda^{l}_{\psi} x^{\tau}.N)\ @^{\phi}_{k}\ M & \leadsto_{\mathrm{t}} \mathbf{let}\ x^{\tau} = M\ \mathbf{in}\ N & \\
\pi^{\wedge}_{i}\ \wedge(M_1, \ldots, M_n) & \leadsto_{\mathrm{t}} M_i & \text{if } 1 \leq i \leq n \\
\mathbf{case}^{\vee}\ (\mathbf{in}^{\vee}_{i}\ N)^{\tau}\ \mathbf{bind}\ x\ \mathbf{in}\ \tau_1 \Rightarrow M_1, \ldots, \tau_n \Rightarrow M_n & \leadsto_{\mathrm{t}} \mathbf{let}\ x^{\tau_i} = N\ \mathbf{in}\ M_i & \text{if } 1 \leq i \leq n \\
(\mathbf{coerce}\,(\sigma, \tau)\ (\lambda^{l}_{\psi} x^{\rho}.M))\ @^{\phi}_{k}\ N & \leadsto_{\mathrm{t}} \mathbf{let}\ x^{\rho} = N\ \mathbf{in}\ M & \\
\mathbf{coerce}\,(\sigma_1, \tau)\ \ \mathbf{coerce}\,(\rho, \sigma_2)\ M & \leadsto_{\mathrm{t}} \mathbf{coerce}\,(\rho, \tau)\ M &
\end{array}$$

Reduction contexts: $\mathbf{C}_{\mathrm{t}} = \{\, C \mid C \in \mathbf{Context}$ and C has exactly one hole$\,\}$

Figure 13: Reduction rules of explicitly typed language λ^{CIL}.