# Specializing C
# an introduction to the principles behind
# C-Mix/II

Henning Makholm (henning@makholm.net)

August 3, 1999

# Contents


**1 Introduction** **4**

1.1 About this report . . . 5
1.2 Generating extensions . . . 6
1.2.1 The Futamura projections . . . 6
1.2.2 Binding-time analysis . . . 7
1.3 Acknowledgements . . . 8

**2 The gegen approach explained** **9**

2.1 Straight-line code . . . 9
2.1.1 Impossible binding-time annotations . . . 11
2.1.2 Binding-time annotations on expressions . . . 12
2.1.3 The `Code` type . . . 13
2.2 Control flow . . . 15
2.2.1 Static conditionals . . . 16
2.2.2 Dynamic conditionals and speculative specialization . . . 17
2.2.3 Residual code sharing . . . 19
2.2.4 Infinite specialization . . . 23
2.2.5 Dangers of speculative specialization . . . 24
2.2.6 When to use the pending list . . . 24
2.3 External function calls . . . 26
2.3.1 External calls and correctness . . . 27
2.3.2 “Under dynamic control” . . . 30
2.3.3 Arbitrary external functions . . . 31
2.3.4 The role of the BTA, revisited . . . 32
2.4 Functions . . . 33
2.4.1 Function specialization: basic principles . . . 34
2.4.2 Sharing residual functions . . . 38
2.4.3 Interplay between function specialization and calls to external functions . . . 40
2.4.4 Function inlining . . . 40
2.5 Global variables . . . 43
2.5.1 The unique-return-state requirement . . . 44
2.5.2 Implications for memoization . . . 46
2.5.3 Implications for function sharing . . . 46
2.5.4 Functions returning $\texttt{int}_s$ . . . 47
2.6 Partially static data . . . 48
2.6.1 A simple example: backed-up ints . . . 48
2.6.2 An implementation sketch . . . 50

2.6.3 Correctness of the transformation . . . 54
2.6.4 Lessons learned . . . 54

**3 Analysis of gegen requirements** **56**
3.1 Preliminaries . . . 57
3.1.1 Notation for Core C types . . . 58
3.1.2 Structure of the binding-time type system . . . 58
3.1.3 Model objects in the generating extension . . . 59
3.1.4 The signature of a binding-time type . . . 60
3.1.5 Faithful binding-time types . . . 61
3.1.6 Binding-time types of expressions . . . 61
3.2 Developing the type system . . . 62
3.2.1 Primitive and abstract types . . . 63
3.2.2 Enumerated types . . . 64
3.2.3 Data pointers . . . 65
3.2.4 Arrays . . . 71
3.2.5 Structs . . . 75
3.2.6 Unions . . . 80
3.2.7 Function pointers . . . 82
3.3 Summary of the binding-time type system . . . 85
3.3.1 Typing rules for expressions . . . 85
3.3.2 Typing rules for lifts . . . 90
3.3.3 Typing rules for statements . . . 90
3.4 Safety rules . . . 90
3.4.1 Pointer analysis . . . 90
3.4.2 Truly local variables . . . 91
3.4.3 The unique-return-state requirement . . . 92
3.4.4 Residual scope issues . . . 94

**4 Synthesis of a BTA algorithm** **97**
4.1 Overall strategy . . . 97
4.1.1 Constraint-based BTA . . . 98
4.1.2 Logic-based BTA . . . 99
4.1.3 Application to the full binding-time type system . . . 100
4.2 Design principles . . . 101
4.2.1 Input to the BTA . . . 101
4.2.2 Catalogue of propositions . . . 101
4.3 Arrow generation . . . 103
4.3.1 Forcing identity of binding-time types . . . 103
4.3.2 Ensuring that binding-time types are well-formed . . . 104
4.3.3 Arrow generation for lifts . . . 106
4.3.4 Forcing binding-time types to be faithful . . . 106
4.3.5 Arrow generation for expressions . . . 107
4.3.6 Arrow generation for statements . . . 114
4.3.7 Arrow generation for jumps . . . 117
4.3.8 Other arrows . . . 117
4.4 Finding the provable propositions . . . 119
4.4.1 Optimizations . . . 120
4.5 Computing signatures . . . 120
4.6 Efficiency of the BTA . . . 121

**5 The splitter phase: eliminating partially static binding-time types 123**
5.1 How to split types . . . . . . . . . . 123
5.2 How to split variables . . . . . . . . . . 125
5.3 How to split functions . . . . . . . . . . 125
5.4 How to split expressions . . . . . . . . . . 126
5.5 How to split statements . . . . . . . . . . 129

**6 Conclusion 133**
6.1 Contributions relative to Andersen (1994) . . . . . . . . . . 133
6.2 Problems not yet satisfactorily solved . . . . . . . . . . 135
6.3 Directions for further extension . . . . . . . . . . 135

**A Core C 137**
A.1 Programs . . . . . . . . . . 138
A.2 Declarations . . . . . . . . . . 140
A.2.1 Function declarations . . . . . . . . . . 140
A.2.2 Variable declarations . . . . . . . . . . 140
A.3 Initializers . . . . . . . . . . 141
A.4 Statements . . . . . . . . . . 141
A.4.1 Control statements . . . . . . . . . . 142
A.5 Expressions . . . . . . . . . . 142
A.5.1 Restrictions on expressions . . . . . . . . . . 143
A.5.2 Displaying expressions . . . . . . . . . . 143
A.6 Types . . . . . . . . . . 144
A.6.1 Structural sharing of type representation . . . . . . . . . . 145

**B Why not to specialize arrays into structures 147**
B.1 The pragmatic value is doubtful . . . . . . . . . . 147
B.2 Inventing new struct types should be avoided . . . . . . . . . . 147
B.3 Necessary pointer types are hard to construct . . . . . . . . . . 148

**C References 149**

**Index 151**

# Chapter 1

# Introduction

A **program specializer** is a software system that given a program $\mathrm{p}$ and (commonly) some of its input $d_1$ produces a new program $p_{\mathrm{res}}$ whose behavior on the remaining input is identical to that if the original program.

We can also express that in algebraic guise:

$$\forall d_1, d_2 : \quad [\![p]\!](d_1, d_2) = [\![\, [\![spec]\!](p, d_1)]\!](d_2) \tag{1.1}$$

where $[\![\cdot]\!]$ is the mapping that takes a program text to its meaning as a function from input to output.

The program $\mathrm{p}$ that is given to the specializer is called the **subject program**. The resulting program, called $p_{\mathrm{res}}$ in the verbal description and $[\![spec]\!](p, d_1)$ in the equation, is called the **residual program**[1].

It is not hard to construct a program specializer with the stated property. Basically, one can just hard-code $d_1$ into $\mathrm{p}$ and otherwise keep $\mathrm{p}$ unchanged. The real challenge is to employ the knowledge of $d_1$ to produce a $p_{\mathrm{res}}$ that runs (significantly) faster than $\mathrm{p}$.

The last few decades have seen considerable advances in this respect, and current techniques are often able to generate $p_{\mathrm{res}}$s that outperform the original $\mathrm{p}$ by factors ranging from 10–20% to several orders of magnitude. The general principle is to precompute the intermediate results in the computation of $[\![p]\!](d_1, d_2)$ and then quote those values in $p_{\mathrm{res}}$ rather than letting $p_{\mathrm{res}}$ carry out the computations that lead to them.

This process is known as **partial evaluation**, and the program specializer is also called a **partial evaluator**.

The case that partial evaluation is important other than an academic exercise has been made enough times that we shall not use space on it here. The sceptical reader is referred to, e.g., Jones (1996) or Consel and Danvy (1993).

#### C-Mix

C-Mix[2] is a specializer for programs written in the C programming language (Kernighan and Ritchie 1988; ANSI 1990). It represents the current state of the

[1] The term "residual program" is, strictly speaking, only completely proper when partial evaluation is used to specialize the subject program. The idea is that the residual program is what is left of the subject program when all the precomputeable computations have been removed.

[2] "mix" is a traditional name for partial evaluators.

art in source-to-source partial evaluation for real-world imperative languages. The system is freely available on the internet[3].

The development of C-Mix was started in 1991 by Andersen (1992). Since then various people at DIKU have been working on it. During 1997-1998 the system underwent a next-to-complete rewrite and is now called C-Mix$_{II}$.

Another partial evaluation system for C is the Tempo Specializer (Consel et al. 1996). A discussion of the primary conceptual differences between C-Mix and Tempo appears in Section 2.2 (page 15).

## 1.1 About this report

Hitherto, Andersen (1994) has been the principal reference for the theory behind C-Mix$_{II}$. However, a replacement for it is desirable for several reasons:

- The development activity in recent years has made it increasingly incorrect with respect to the actual implementation.
- Actual flaws in its logic have been uncovered. For example, its concept of "control dependence" (Andersen 1994, page 200), which is used to determine when a side effect on non-local static data is safe, has been found to err in the unsafe direction. The current C-Mix$_{II}$ implementation has corrected the error, but nothing about it has as yet appeared in print.
- Generally, it was never a particularly readable document.

This report aims at giving a self-contained description of the key principles behind the specialization process in the current implementation of C-Mix$_{II}$, with emphasis on the binding-time analysis. It could be used as an up-to-date replacement for Chapters 3 and 5 of Andersen (1994).

Due to time constraints the report does not cover the support for heap allocations in C-Mix$_{II}$.

In the main text of the report no particular attempt is made to differentiate between those ideas that come from Andersen (1994) and those that were invented later. However, a summary of changes since Andersen (1994) appears in Section 6.1.

The intended audience includes future developers, academics looking for a description of the new techniques, and C-Mix$_{II}$ users who wish to understand in detail why the results of C-Mix$_{II}$ are as they are.

Previous knowledge of partial evaluation techniques is not necessary, but it is assumed that the reader knows the C language. From Chapter 3 the text will assume knowledge of the internal representation C-Mix$_{II}$ uses for C programs, called Core C. No description of Core C is generally available, so one is included as Appendix A.

### On terminology

This report uses the terms "static" and "dynamic" to refer to values and program constructions that are known or executed at specialization time, respectively in the specialized program. This is the standard terminology in partial evaluation literature.

[3] `http://www.diku.dk/research-groups/topps/activities/cmix/`

The user interface and end-user documentation for C-Mix/II use the words "spectime" and "residual" instead, so as not to confuse the average C programmer to whom `static` is a keyword which already has at least two subtly different meanings. There should be little danger of such confusion in this text: the `static` keyword is not mentioned at all, except here. Its various meanings in C has been reduced to simpler constructs as part of the translation to Core C (Appendix A).

## 1.2 Generating extensions

The idea behind **generating extensions** is to factor the program specialization process itself into phases.

Remember that the program specialization problem is to create $p_{res}$ given $p$ and $d_1$. We now divide this into two separate tasks. First, we somehow derive from $p$ an auxiliary program $p_{gen}$ called a **generating extension** for $p$. This is done without looking at $d_1$.

Now, when $p_{gen}$ is run with $d_1$ as its input, it produces the final $p_{res}$:

$$\forall d_1, d_2 : \quad [\![ [\![p_{gen}]\!](d_1) ]\!](d_2) = [\![p]\!](d_1, d_2) \tag{1.2}$$

The original reason for devising this rather complicated procedure was a hope than one could create a $p_{gen}$ that was faster and used less resources than a "monolithic" specializer that tried to arrive at $p_{res}$ in a single go. The construction of $p_{gen}$ itself might still be complex, but there could be a net gain if one wanted to specialize the same $p$ with respect to several different $d_1$s in succession.

### 1.2.1 The Futamura projections

Now, how do we arrive at $p_{gen}$ when we know $p$? If $p$ is well understood and one doesn't care too much about development time, $p_{gen}$ can be constructed by hand. However, in most cases this is not a practical option because of the inherent conceptual complexity of programs that generate other programs; and the difficulty of maintaining a hand-written generating extension in case $p$ changed.

Suppose instead that we already have a "monolithic" specializer $spec$. We then know how to produce $p_{res}$; obtaining $p_{gen}$ is only a matter of splitting the specialization process into  stages. It happens that this kind of "staging" is exactly what specialization is good for. This suggests that we might be able to use $spec$ to stage its own actions. And indeed, we find that we can let $p_{gen}$ be the output of running $[\![spec]\!](spec, p)$. One can see that this satisfies (1.2) by appying (1.1) twice:

$$\begin{aligned}
\forall d_1, d_2 : \quad [\![ [\![p_{gen}]\!](d_1) ]\!](d_2) &= [\![ [\![ [\![spec]\!](spec, p) ]\!](d_1) ]\!](d_2) \\
&= [\![ [\![ spec]\!](p, d_1) ]\!](d_2) \\
&= [\![p]\!](d_1, d_2)
\end{aligned}$$

Taking this idea a step further, we can even derive a program that reads $p$ and automatically constructs $p_{gen}$. This program has been known in the literature as *Cogen*, which stands for "COmpiler GENerator", because it turns out that

in some cases a generating extension can be used as a compiler. Because this is only one application of generating extensions, we prefer to call the tool **Gegen**, abbreviating "Generating Extension GENerator".

To produce Gegen, run $\mathtt{spec}$ with its own source code as both inputs, i.e., $[\![\mathtt{spec}]\!](\mathtt{spec}, \mathtt{spec})$. We can se that this yields a Gegen that works as expected, thus:

$$\begin{aligned} \forall p : \quad [\![\mathit{gegen}]\!](p) &= [\![\, [\![\mathtt{spec}]\!](\mathtt{spec}, \mathtt{spec})]\!](p) \\ &= [\![\mathtt{spec}]\!](\mathtt{spec}, p) \\ &= p_{\mathrm{gen}} \end{aligned}$$

This procedure for creating a Gegen was first proposed by Futamura (1971). Mind-boggling as it is at first sight, it should come as no surprise that it was easier said than done. Early attempts at realizing the **Futamura projections** invariably ended up with $p_{\mathrm{gen}}$s that used at least as much space and time as simply using the original $\mathtt{spec}$ to specialize the subject program.

The reason is that a program specializer is a rather complex piece of software. So, to be able to optimize itself, it has to be powerful enough to handle that level of complexity. Which itself tends to add complexity...

Eventually, however, the search for an efficiently self-applicable program specializer succeeded (Jones et al. 1989). Since then, a considerable amount of work has gone into generalizing its lessons to other programming languages than the very restricted one Jones et al. (1989) could handle.

### 1.2.2 Binding-time analysis

Still, there is no free lunch. Along the way it also became clear that for various technical reasons the Futamura projections are not a good approach to specializing strongly typed lanugages like C.

Fortunately, the research on the Futamura projections also produced a deeper understanding about the workings of a generating extension and its relationship the subject program, which *can* be generalized to C. Thus, today's knowledge makes it possible to write Gegen directly, without the detour of self-application. This is the approach used in C-Mix/II.

The most important of the ideas that grew out of the search for self-application and is still valuable in the "hand-written Gegen approach" is the idea of an **binding-time analysis** as an essential step in constructing $p_{\mathrm{gen}}$.

Recall that partial evaluation is about trying to precompute the outcome of certain of the subject program's computations, given limited information about its input. The task of a **binding-time analysis** (abbreviated BTA) is to determine in advance of the partial evaluation process *which* of the computations *can* be precomputed and how to do them. The BTA does not actually do the precomputation, so it does not need to know even the limited information about the input that the partial evaluator needs. The BTA *does*, however, need to know just *how* limited the information will be. The result of the BTA are **binding-time annotations** that tell the actual partial evaluation process what to do with each part of the subject program.

The focal point of this report is to describe what the BTA phase in C-Mix/II does and why it does it. We will start with a general introduction to Gegen

principles in Chapter 2 and then move on to C-Mix/II specifics from Chapter 3 onwards.

## 1.3 Acknowledgements

Much of the theory I present in this report was developed through discussions with Jens Peter Secher and Arne Glenstrup. Special thanks go to Arne, with whose design sketches for C-Mix/II I disagreed enough to start thinking about writing down my own idea about how to understand the system.

Peter Makholm, Sebastian Skalberg, and Jens Peter Secher, read drafts of this report and provided constructive feedback.

I am grateful for the continual encouragement from my supervisor Neil D. Jones. He read several drafts of the report, and without his ability to purposefully misunderstand everything misunderstandable, this report would be much more opaque than it is now.

And, of course, without Lars Ole Andersen there would not have been any C-Mix to write about.

I am indebted to the authors of the excellent free software I used to write this report, including GNU Emacs, TeX, LaTeX, XY-pic, and dvips.

# Chapter 2

# The gegen approach explained

The purpose of the chapter is to provide a gentle introduction to the fundamental gegen techniques without the complication of developing all of them for the full C language.

The techniques will be introduced informally; the reader is supposed to be able to extrapolate from the examples to how similar cases would be treated. The goal is not to provide an exact definition of the gegen transformation, but to give a general overview of what it does and how it works.

## 2.1 Straight-line code

As our very first, extremely simple, example language, consider the subset of C defined by this abstract grammar:

| | | |
|---:|:---:|:---|
| *program* | ::= | *heading* `{` *decls stmts return* `}` |
| *heading* | ::= | *type Id parlist* \| `void` *Id parlist* |
| *parlist* | ::= | `(` *params* `)` \| `( void )` |
| *params* | ::= | *decl* \| *decl* `,` *params* |
| *decls* | ::= | ε \| *decl* `;` *decls* |
| *decl* | ::= | *type Id* |
| *stmts* | ::= | ε \| *stmt stmts* |
| *stmt* | ::= | *Id* `=` *expr* `;` |
| *return* | ::= | `return` *expr* `;` \| `return ;` |
| *expr* | ::= | *Id* \| *Constant* \| *expr op expr* |
| *op* | ::= | `+` \| `-` \| `*` |
| *type* | ::= | `int` |

where ε means "empty string". That is: only one type (`int`)[1]; only one function; no control flow statements; and assignment is a statement, not an operator. The

[1] "`void`" in function headers specifies the *absence* of any types

```
int pgm_A(int x,int y) {
  int z ;
  y = y * 5 ;
  z = x + 1 ;
  return z - y ;
}
```

*Figure 2.1: A very simple program*

```
int_d pgm_A(int_s x,int_d y) {
  int_s z ;
  y = y * 5 ;
  z = x + 1 ;
  return z - y ;
}
```

*Figure 2.2:* `pgm_A` *(from Figure 2.1) binding-time annotated*

program's input are the arguments to the single function; the output is the value of the `return` expression.

An example program in this minimal language would be the one in Figure 2.1.

Now let us construct a generating extension for this program. Doing so requires us to have **binding-time annotations** for the program. For now, we shall just assume that some kind of oracle has supplied us with the proper annotations. In studying how the binding-time annotations are *used* we will learn enough about them to be able to construct the **binding-time analysis** that creates them.

What are binding-time annotations, then? Well, in our setting they are simply a refinement of the subject language's type system. Instead of a single type, `int`, the binding-time annotated program has two different types, $\texttt{int}_s$ and $\texttt{int}_d$. To **binding-time annotate** a program is to replace every `int` in the program with either $\texttt{int}_s$ or $\texttt{int}_d$.

One possible binding-time annotation of `pgm_A` is shown in Figure 2.2, and the generating extension that corresponded to that could be as shown in Figure 2.3. The output of the generating extension, if the `x` parameter was 42, is shown in Figure 2.4.

Spend some time contemplating the relationship between the binding-time annotated `pgm_A` and the generating extension. Some key points to observe at this stage are

- The values of "type" $\texttt{int}_s$ are computed in the generating extension. They do not appear in the residual program. We call this property being **static**, hence the "`s`" in "$\texttt{int}_s$".
- The values of "type" $\texttt{int}_d$ appear only in the residual program. In the generating extensions, $\texttt{int}_d$ variables appear as mere names, and their values are not known at the time the generating extension runs. This property is called being **dynamic**.

```
void pgm_A_gen(int x) {
  int z ;
  printf("int pgm_A_res(int y) {\n");
  printf("  y = y * 5 ;\n");
  z = x + 1 ;
  printf("  return %d - y ;\n",z);
  printf("}\n");
}
```

*Figure 2.3: The $\mathfrak{p}_{\text{gen}}$ that corresponds to Figure 2.2*

```
int pgm_A_res(int y) {
  y = y * 5 ;
  return 43 - y ;
}
```

*Figure 2.4: Output from the program in Figure 2.3*

- In the *return* statement, the static value of `z` gets embedded into the residual program as a literal constant. This is known as **lifting**[2] the value. When we think of binding-time annotations as a type system, a lift can be seen as a conversion from $\texttt{int}_\text{s}$ to $\texttt{int}_\text{d}$.
- The binding-time annotations on the input (i.e., the parameters) to the subject program decide which parts of the input are read by the generating extension and which parts are read by the residual program.
- The generating extension uses `printf`: it is not written in the simplified C language that the subject program (and, indeed, the residual program) adheres to. This was to be expected, because the generating extension must output a program text, which is somewhat difficult when `int` is your only type (Gödel numberings were considered a little too opaque to be used in an introductory exposition...).

### 2.1.1 Impossible binding-time annotations

Exercise for the reader: write down the generating extension corresponding to this alternative binding-time annotation of `pgm_A`:

```
int_d pgm_A(int_d x,int_s y) {
  int_s z ;
  y = y * 5 ;
  z = x + 1 ;
  return z - y ;
}
```

[2]Though the term “lift” is firmly established in the partial evaluation literature, it might seem somewhat counterintuitive, because “lifting” something means moving it from the specialization phase into the residual program—and graphical illustrations of the partial evaluation tend to picture the residual program *below* the specialization. The origins of this use of “lift” are to be found in *lattice theory* which is commonly used as the mathemathical foundation for binding-time analyses, though not (at least not explicitly) for the one we present in this report.

See what goes wrong? In the second statement we must set `z` to the sum of `x` and 1. But the value of `x` is not known until the residual program, yet the binding-time type of `z` specifies that its value must be known already in the generating extension.

Clearly that is impossible, which leads to the important realization that not every imaginable binding-time annotated program can be used for making a generating extension. Various constraints must be met, and the one that is at play here is that *even though there is a conversion—a "lift"—from* $\texttt{int}_s$ *to* $\texttt{int}_d$*, there is not a corresponding "drop" from* $\texttt{int}_d$ *to* $\texttt{int}_s$.

This is different from what one might be used to from C, where any conversion has a reverse conversion (though information may be lost, both conversions are *allowed* by the type checker). Nevertheless it is essential to the idea of binding-time annotations as a special type system that some type conversions are only one way.

### 2.1.2 Binding-time annotations on expressions

The binding-time annotations we've shown so far has only consisted of reinterpreting those types that are visible in the subject program. This is really an oversimplification. In fact, every expression in a C program has a type assigned by the type-checking phase of the compiler according to strict rules in the language definition[3].

In the C subset we're concerned about here, one could imagine a type checker that mindlessly decided that the type of every expression is be an `int`. A compiler probably wouldn't implement it explicitly, because its judgements are not especially enlightening. The important thing is that those "invisible" `int`s are *also* part of the binding-time annotations. That is, the binding-time annotation declares each expression to be either $\texttt{int}_s$ or $\texttt{int}_d$.

We are not going to systematically show the annotation on every expression; that would be notationally awkward. What we *are* going to do at the moment is to annotate operators with the same index as the type they operate on. Thus $+_s$ is the operator that adds two $\texttt{int}_s$s giving another $\texttt{int}_s$, and $+_d$ adds together two $\texttt{int}_d$s. Likewise for literal constants: $42_s$ is a constant of type $\texttt{int}_s$, etc.

With this notation we can write binding-time annotated programs such as the ones in Figure 2.5, which couldn't be known apart if we only had binding-time annotations on the visible types. (The difference is in the binding time of the constant `5`).

The example to the left corresponds to the generating extension we have already seen (Figure 2.3); the one on the right is shown in Figure 2.6.

Notice how the operator annotations allows one to infer the placement of lifts unambigously.

We can also see that there can be multiple binding-time annotations of the same subject program that all share the samme patterns for the program's input. In general we want the BTA to select the one with the largest possible amount of static computation, so the residual programs will need to do as little as possible.

[3] ANSI (1990). Well, in *real* real life, the type of a C expression may also depend subtly on the relations between the standard types offered by the compiler. One could argue that this property makes it theoretically impossible to do *any* nontrivial transformation on C programs in a completely generic, foolproof and sound way. We shall, however, tacitly sidestep that issue because it is really irrelevant to the Gegen techniques.

```
int_d pgm_A(int_s x,int_d y) {
  int_s z ;
  y = y *_d 5_d ;
  z = x +_s 1_s ;
  return z -_d y ;
}
```

```
int_d pgm_A(int_s x,int_d y) {
  int_s z ;
  y = y *_d 5_s ;
  z = x +_s 1_s ;
  return z -_d y ;
}
```

*Figure 2.5: More elaborate annotations of* `pgm_A`

```
void pgm_A_gen(int x) {
  int z ;
  printf("int pgm_A_res(int y) {\n");
  printf("  y = y * %d ;\n",5);
  z = x + 1 ;
  printf("  return %d - y ;\n",z);
  printf("}\n");
}
```

*Figure 2.6: Yet another generating extension for* `pgm_A`

### 2.1.3 The `Code` type

The generating extensions we have seen so far output the residual program directly to the standard output stream with `printf` calls. That technique is good for illustrating basic principles on very simple examples.

However, when things begin to become more complicated (notably, when we introduce function calls) it is no longer satisfactory. It becomes difficult to arrange for the generating extension to produce the different parts of the residual program in the same order they must have syntactically.

The solution is to let the generating extension produce an intermediate representation of the residual program, and keep that in memory until the specialization process is completed. At that point the parts can be collected in the right order and a pretty-printed output of the residual program can be output.

With this strategy a generating extension could look like the one in Figure 2.7. The low-level tasks of maintaining an internal representation of the generated code are delegated to an external support library that is linked into every generating extension. We call this library **Speclib**, which is short for "specialization library".

`speclib.h` contains the external interface to this library. It defines the abstract type `Code` which is used to represent pieces of the residual program. It also contains declarations for Speclib entry points such as

```
Code MakeName(char *expr);
Code Lift(int);
void BeginFunction(char *fmt,...);
void Emit(char *fmt,...);
void EndFunction(void);
```

which allow the construction of residual functions. The functions like `Emit` which actually emit code to the "current" residual function have a vaguely

```
#include "speclib.h"
void pgm_A_gen(int x) {
  int z ;
  Code y = MakeName("y");
  Code fun_name = MakeName("pgm_A_res");
  BeginFunction("int ?(int ?)",fun_name,y);
  Emit("? = ? * 5",y,y);
  z = x + 1 ;
  Emit("return ? - ?",Lift(z),y);
  EndFunction();
}
```

*Figure 2.7: The program in Figure 2.3 redone using Speclib*

| binding-time type | representation in $p_{gen}$ | representation in $p_{res}$ |
|---|---|---|
| $\texttt{int}_s$ | `int` | (vanishes) |
| $\texttt{int}_d$ | `Code` | `int` |

*Table 2.1: Representation of* $\texttt{int}_d$ *and* $\texttt{int}_s$ *in* $p_{res}$ *and* $p_{gen}$.

`printf`-like interface: the `fmt` argument may contain "`?`" characters which are replaced with the `Code` pieces that follow it, when the residual program is eventually written out.

The reader may also note another feature that is different in this generating extension. Previously, residual identifiers such as "y" or "`pgm_A_res`" simply appeared as verbatim parts of the format strings. Now, instead we start by creating `Code` variables that are initialized with the names.

The reason is that when we add new features to the specialization process (in particular, when we introduce function inlining in Section 2.4.4) a single residual function might need to contain residual versions of several variables that had the same name in the subject program. We then need to rename one or more of them; this is the job of the `MakeName` function which is used to initialize the `y` and `fun_name` variables in the generating extension. Calling `MakeName` several times with the same argument will return *different* identifiers wrapped up as `Code`.

The fine art of creating names for the residual program[4] is known as **name management**. In this report we shall not go into detail about how `MakeName` works or which kinds of conflicts the name manager must avoid. The lesson to remember at this point is that an $\texttt{int}_d$ variable in some sense *does* exist in $p_{gen}$ even though it has no value to hold yet. Instead it exists as a `Code` variable containing the name of the corresponding residual variable. We can represent this in schematic form, as in Table 2.1.

One last word about Speclib. If you compare the generating extension texts

[4]And, in fact, for the generating extension. Say, if one of the subject program's variables has the same name as an entry point in Speclib it needs to be renamed lest it will shadow the Speclib function, most likely leading to a generating extension that can't be compiled.

above with the text of a genuine generating extension produced by C-Mix/II, you still won't see much similarity. That is because the interface details for Speclib are different in the real world than in this introduction. We deliberately chose to present an idealised interface here, so we avoid explaining the intricacies of the real Speclib yet. In particular, the name management issue is a lot more complex than we need to discuss in this report.

## 2.2 Control flow

The subject language we dealt with in the previous section was very restricted—restricted enough, in fact, to be completely useless. What is needed, of course, is the ability to make decisions about control flow at run time. We will now describe how to create generating extensions when the language includes control flow constructs.

We can choose between two basic principles here. One would be to require that the control flow be specified with structured control constructs. The other is the exact opposite: it is to unfold all the structured control constructs into raw `goto`s and forget about the original structure.

Superficially, the former choice appears to be a dead end when our goal is to specialize C, because C has unrestricted `goto`s within each function. However, there are algorithms which can re-express any nonstructured control flow using structured primitives, at the cost of possibly introducing some "flag" variables (Erosa and Hendren 1994; Ammarguellat 1992). Knowing that the program has a strict structure enables analysis and specialization techniques that would not be possible with structure-less programs. This avenue has been followed by the Tempo Specializer (Consel et al. 1996), and empirical results show that it can lead to considerable gains in the speed of the specialization process—enough that "run-time code generation" that produces specialized machine code in fractions of a second is practical.

On the other hand, the very techniques that are made possible by insisting on structured control flow also have negative effects on the *strength* of the specialization. That is, the generating extensions may run fast, but the residual programs are not always as well specialized as they could have been.

The C-Mix/II development has focused on applications such as domain-specific languages that turn out to be quite badly suited for the "structured control flow" approach. Thus C-Mix/II immediately translates the subject program into a "core" language with only conditional and unconditional `goto`s. In this report we assume that this translation has already been done, so we modify our example language as follows:

| | | |
|---:|---|---|
| *program* | ::= | *heading* `{` *decls blocks* `}` |
| *blocks* | ::= | *block blocks* \| *block* |
| *block* | ::= | *label* `:` *stmts jump* |
| *jump* | ::= | *return* \| *goto* \| *conditional* |
| *conditional* | ::= | `if (` *expr* `)` *goto* `else` *goto* |
| *goto* | ::= | `goto` *label* `;` |

```
int_d pgm_B(int_s x,int_s y, int_d z) {
START: y = y *_s y +_s 1_s ;
       z = z +_d 1_d ;
       if ( x ==_s y ) goto EQUAL; else goto NONEQUAL;
EQUAL: x = 100_s /_s x ;
       z = z +_d x ;
       return z ;
NONEQUAL: x = x +_s 1_s ;
          return z -_d x ;
}
```

*Figure 2.8: A program with a static conditional*

```
void pgm_B_gen(int x, int y) {
  Code z = MakeName("z");
  Code fun_name = NameNake("pgm_B_res");
  BeginFunction("int ?(int ?)",fun_name,z);
 START:
  y = y * y + 1 ;
  Emit("? = ? + 1",z,z);
  if ( x == y ) goto EQUAL; else goto NONEQUAL;
 EQUAL:
  x = 100 / x ;
  Emit("? = ? + ?",z,z,Lift(x));
  Emit("return ?",z);
  goto done ;
 NONEQUAL:
  x = x + 1 ;
  Emit("return ? - ?",z,Lift(x));
  goto done ;
 done:
  EndFunction();
}
```

*Figure 2.9: Generating extension for the progam in Figure 2.8*

(The “missing” nonterminals are as on page 9). The body of a function is now a set of **basic block**s, each consisting of a piece of straight-line code and a *jump* that defines what happens when the block has been executed. Execution of the program starts at the beginning of the first block.

### 2.2.1 Static conditionals

Consider the binding-time annotated program in Figure 2.8. We can handle this with the techniques we already have: Because the condition expression is of type $\texttt{int}_s$ we can compute its value in the generating extension and select at that time what to do. The generating extension in Figure 2.9 would produce programs like

```
int_d pgm_B(int_s x,int_d y, int_d z) {
START: y = y *_d y +_d 1_s ;
       z = z +_d 1_s ;
       if ( x ==_d y ) goto EQUAL; else goto NONEQUAL;
EQUAL: x = 100_s /_s x ;
       z = z +_d x ;
       return z ;
NONEQUAL: x = x +_s 1_s ;
          return z -_d x ;
}
```

*Figure 2.10: A program with a dynamic conditional*

```
int pgm_B_res(int z) {
  z = z + 1 ;
  z = z + 10 ;
  return z ;
}
```

```
int pgm_B_res(int z) {
  z = z + 1 ;
  return z - 4 ;
}
```

corresponding to the inputs $\mathtt{x} = 10, \mathtt{y} = 3$ and $\mathtt{x} = 3, \mathtt{y} = 10$.

### 2.2.2 Dynamic conditionals and speculative specialization

It is not as easy to decide what to do about conditionals where the governing expression is an $\mathtt{int}_d$. Consider, e.g., the alternative binding-time annotation of `pgm_B` shown in Figure 2.10. With $\mathtt{x} = 20$ we would expect to get a residual program something like

```
int pgm_B_res(int y, int z) {
START: y = y * y + 1 ;
       z = z + 1 ;
       if( 20 == y ) goto EQUAL; else goto NONEQUAL;
EQUAL: z = z + 5 ;
       return z ;
NONEQUAL: return z - 21 ;
}
```

Now, how could a generating extension be capable of that? Several interconnecting issues arise.

First, it is clear that the generating extension cannot simply stick to tracing a single execution path. When it comes to the conditional it has to continue with tracing *both* branches of the program, generating residual code for either of them.

This could be implemented by having the generating extension maintain a set—commonly called the **pending list**—of those residual program blocks that *should* be specialized into residual code but *have* not yet been. The generating extension would then repetitively pull out a block from the pending list, specialize it (which might result in new blocks being added to the pending list), and terminate when the list ends up empty.

In this particular example, the pending list initially consists solely of `START`. The first iteration of the **pending loop** removes `START` from the list and specializes that block. At the end, when the conditional is reached, `EQUAL` and `NONEQUAL` gets added to the pending list. The second iteration removes `EQUAL` and does not add anything. Lastly `NONEQUAL` is removed from the pending list; after this the pending list is empty and we know the residual program to be complete.

Second, one must ask: where do the constants `5` and `21` in the residual come from. Clearly, they are the value of `x` after the execution of

`x = 100 / x ;` respectively `x = x + 1 ;`

But if we just blindly executed these statements, guided by the pending list, `x` would be 5 when we specialized `NONEQUAL` and the residual program would end up containing `6` instead of the correct `21`.

Out of these considerations comes the realization that the pending list must contain not only names of basic blocks but also the values of all the static variables at entry to the block. We say that the values get **memoized**. When the pending loop gets a block from the pending list it must also restore all of the static variables to their memoized values.

What is going on, then, is that we execute the static parts of `EQUAL` and `NONEQUAL` **speculatively**—that is, we execute though we do not know for sure whether they would have been executed or not in a run of the subject program.

In modern high-performance microprocessors, speculative execution is commonly used to avoid stalling the entire pipeline when the condition of a conditional jump arrives late. What we are doing here is essentially the same thing, but on a grander scale: our speculative execution is not merely ahead by a couple of clock cycles but need to presuppose entire execution paths of the program, because in our context the condition arrives *really* late.

Now we can try to synthesize the points about the pending list and speculative specialization into a concrete generating extension. This generating extension, shown in Figure 2.11, makes use of a language addition in the C compiler we use: a special unary `&&` operator can take the address of a label, and later in the same function one can `goto` the resulting pointer. The behavior could easily be simulated in standard C with a `switch` statement, replacing the labels by distinct integers.

Though the details of the pending list administration have been hidden inside helper functions from Speclib, the overall structure should be clear from this example. The only thing that maybe deserves special notice is the labels for residual basic blocks are generated by the `PutPending` function which stores them in the pending list in addition to returning them to the caller. When the `GetPending` retrieves the pending item, it is responsible for emitting the label to the residual program.

The most striking feature of the example is the nearly total absense of memoization code. All of the memoization is handled “behind the curtain” by Speclib calls. The only visible artifact is the call to `RegisterStaticVar` which tells Speclib that any entry in the current pending list must include a memoized value for `x`. The various calls to `PutPending` and `GetPending` then take care of the necessary saves and restores.

```
void pgm_B_gen(int x) {
  Code y = MakeName("y");
  Code z = MakeName("z");
  Code fun_name = MakeName("pgm_B_res");
  BeginFunction("int ?(int ?,int ?)",fun_name,y,z);
  NewPendingList();
  RegisterStaticVar(&x,sizeof x);
  PutPending(&&START);
 pending_loop:
  if ( AnyPending() ) goto *GetPending();
  DisposePendingList();
  EndFunction();
  return;
 START:
  Emit("? = ? * ? + ?",y,y,y,Lift(1));
  Emit("? = ? + ?",z,z,Lift(1));
  Emit("if( ? == ? ) goto ?; else goto ?;",Lift(x),y,
       PutPending(&&EQUAL),PutPending(&&NONEQUAL));
  goto pending_loop ;
 EQUAL:
  x = 100 / x ;
  Emit("? = ? + ?",z,z,Lift(x));
  Emit("return ?",z);
  goto pending_loop ;
 NONEQUAL:
  x = x + 1 ;
  Emit("return ? - ?",z,Lift(x));
  goto pending_loop ;
}
```

*Figure 2.11: Generating extension for the program in Figure 2.10*

The following subsections deal with things that can go wrong when using the pending list technique. Basically there are two risks:

- The pending loop only exits when the pending loop becomes empty. Will that ever happen? If not, we encounter **infinite specialization**. Section 2.2.3 describes modifications to the pending list technique that can sometimes prevent infinite specialization from happening. They do not solve the problem entirely; Section 2.2.4 describes how manual intervention in the binding-time analysis can still be necessary.
- A portion of the subject program may be speculatively specialized under static conditions that never arise in practise. In some cases that may cause the generating to malfunction silently. Section 2.2.5 describes the problem in more detail and warns that we have no good solution to it.

### 2.2.3 Residual code sharing

Consider the annotated program pgm_C in Figure 2.12. This program consists

```
int_d pgm_C(int_d d) {
       int_s i ;
       int_d j ;
START: i = 1_s ;
       goto I_BEG ;
I_BEG: j = 1_d ;
       goto J_BEG ;
J_BEG: d = d +_d i *_d j ;
       j = j +_d 1_d ;
       if ( j <=_d 2 ) goto J_BEG; else goto J_END;
J_END: i = i +_s 1_s ;
       if ( i <=_s 2 ) goto I_BEG; else goto I_END;
I_END: return d ;
}
```

*Figure 2.12: A program with loops*

of two nested loops, the inner one controlled by the dynamic variable `j`; the outer one controlled by the static variable `i`.

Constructing a generating extension for `pgm_C` does not need any new skills. Figure 2.13 shows what we get (the perhaps odd-looking combination of pending list manipulation and direct jumps between labels will be explained in Section 2.2.6). Now, try running this generating extenstion "by hand". Do you encounter an infinte loop?

If you strick to the description of pending list management we've seen until now, you *should* never reach the end of the pending loop. We can even say precisely what causes it: the label `J_BEG`, once put into the pending loop, can never get removed. Or rather, each tine `GetPending()` returns `J_BEG` another `J_BEG` gets added to the pending list as part of the actions for `J_BEG`.

If you think further about it, the pending list technique as it has been described previously does not allow the residual program to contain loops *at all*, because each call to `PutPending` is supposed to return a *fresh* residual label.

Obviously, the solution is to allow `PutPending` to not always construct a new label but sometimes reuse an earlier label and not—somewhat contrarily to its name—actually put anything on the pending list. We'll have to be careful, though, about when to invent a new label and when to reuse an earlier one (and then, which one).

It is immediately clear that it won't do any good to have `PutPending(&&FOO)` return a label that was generated earlier by `PutPending(&&BAR)`: if the static labels do not match up, we'll be completely of out hope of conforming to what the subject program does.

However, that alone is not enough to guarantee correctness. If we imagined a `PutPending` that reused old labels whenever there were one which matched the static label, we would get a residual program like the one in Figure 2.14. The first iteration of the static loop is perfectly fine here, but as $p_{gen}$ gets to the second iteration where $\texttt{i} = 2$ it tries to reuse the residual use which has already been generated, but which is specialized to $\texttt{i} = 1$! Not only does this cause the summation in `d` to be wrong, it also cause $p_{gen}$ to finish off the function without

```
void pgm_C_gen(void) {
  Code d = MakeName("d");
  int i ;
  Code j = MakeName("j");
  Code fun_name = MakeName("pgm_C_res");
  BeginFunction("int ?(int ?)",fun_name,d);
  NewPendingList();
  RegisterStaticVar(&i,sizeof i);
  LocalVar("int ?",j);
  PutPending(&&START);
 pending_loop:
  if ( AnyPending() ) goto *GetPending();
  DisposePendingList();
  EndFunction();
  return;
 START:
  i = 1 ;
  goto I_BEG ;
 I_BEG:
  Emit("? = 1",j);
  Emit("goto ?",PutPending(&&J_BEG));
  goto pending_loop ;
 J_BEG:
  Emit("? = ? + ? * j",d,d,Lift(i),j);
  Emit("? = ? + 1",j,j);
  Emit("if( ? <= 2 ) goto ?; else goto ?",j,
       PutPending(&&J_BEG),PutPending(&&J_END));
  goto pending_loop ;
 J_END:
  i = i + 1 ;
  if ( i <= 2 ) goto I_BEG; else goto I_END;
 I_END:
  Emit("return ?",d);
  goto pending_loop ;
}
```

*Figure 2.13: Generating extension for* `pgm_C` *(from Figure 2.12)*

```
int pgm_C_res_bad(int d) {
  int j ;
 L1: /* GetPending: goto &&START, restore i to (garbage) */
  j = 1 ;
  goto L2 ;
 L2: /* GetPending: goto &&J_BEG, restore i to 1 */
  d = d + 1 * j ;
  j = j + 1 ;
  if ( j <= 2 ) goto L2; else goto L3 ;
 L3: /* GetPending: goto &&J_END, restore i is 1 */
  j = 1 ;
  goto L2 ;
/* pending list is now empty */
}
```

*Figure 2.14: An incorrect specialization of the program in Figure 2.12*

ever emitting a return statement. The pending loop simply quits when there are no loose ends left—and, having created a residual perpetual loop, there are indeed no loose ends in Figure 2.14.

It seems clear that we have to avoid sharing residual labels between sections of residual code where the same of the subject program's basic blocks should be specialized with respect to *different* values of the static variables. But how can we do that.

Enter memoization. Recall (from page 18) that a pending list entry contains not only a label into the $p_{gen}$ text and the residual label that was returned by `PutPending`: there is also back-ups of all of the static variables in the program. The `GetPending` call restores the static variables to the values they had when `PutPending` was called.

We can use this for controlling whether `PutPending` should reuse an old entry: When considering the old pending-list entry, it is not enough for the $p_{gen}$ labels to match, the memoized values must match also. (One notes that we need to keep old pending-list entries for comparison after they have been "removed" by `GetPending`; `PutPending` needs to consider entries that have already been processed as well as those that are currently pending).

Is this enough to make sure that code sharing will not result in incorrectly specialized programs? It turns out that it is. Do not despair if you do not feel convinced; this matter is at the very heart of partial evaluation and is known for being notoriously hard to grasp. Give it the thought it deserves. Try to construct counterexamples and see how they sort themselves out anyway.

The formally minded reader might wish to consult Jones et al. (1993, Sections 4.4.1ff) where the same basic idea is presented in a more formal setting, providing the building blocks for a mathematical argument that the technique works.

If everything else fails you'll have to take my word for it.

You might find it comforting to know that this is the height of intellectual sophistication in this text. Matters may become more complex further on, but they should not become harder to understand.

```
int_d pgm_D(int_d limit, int_d factor) {
  int_s i ;
  int_d sum ;
START: sum = 0_s ;
       i = 1_s ;
       goto BODY ;
BODY: sum = sum +_d i *_d factor ;
      i = i +_s 1_s ;
      if ( i <=_d limit ) goto BODY; else goto END;
END: return sum ;
}
```

*Figure 2.15: A program that specializes infinitely*

### 2.2.4 Infinite specialization

Having just made the bold assertion that everything works if only `PutPending` checks the memoized static state before re-using a label, I'll have to retreat a little. Or rather, to clarify: The residual program will be faithful to the subject program, *if we get any complete residual program at all*.

Namely, there is still the risk that $p_{gen}$ may loop perpetually. E.g. if we try to specialize the program in Figure 2.15, $p_{gen}$ will try to specialize the `BODY` for `i` $= 1, 2, 3, \ldots$ and so on indefinitely—because the value of `limit` is unknown it won't (and *can't* ever) know where to stop.

Now, the complex strategy for residual code sharing we developed in the previous section was supposedly motivated by a $p_{gen}$ that went into an infinite loop. Here we find that this can still happen. Did we get anywhere at all?

The answer of course is, yes we did. The residual code sharing means that we *can* specialize more programs than we could before. We *can* produce residual programs with loops. The fact that residual code sharing does not solve all of the world's problems does not mean that it is useless.

That still leaves us with the problem of what to do with `pgm_D`. A little thought reveals that the problem here is that we're asking for the impossible. We want an algoritm that can sum the first $\ell$ natural numbers (times some constant factor) and still has no counter variable in it. The binding-time annotations in Figure 2.15 are simply ill-conceived, though they are not direct type errors.

To fix the problem, we need to re-annotate the program so that `i` becomes an `int`$_d$ instead of an `int`$_s$. Without any `int`$_s$s involved the residual program is not going to be much different from what we started with, but it is going to be there and be correct, which is what we'll have to settle for with this input.

We have no automatic way of identifying this situation. The binding-time analysis in C-Mix$_{/\mathrm{II}}$ may well decide on precisely the annotations in Figure 2.15 unless it is told otherwise. The human user has to observe that $p_{gen}$ loops, figure out why, and manually instruct C-Mix$_{/\mathrm{II}}$ that he wants `i` to be dynamic.

### 2.2.5 Dangers of speculative specialization

Consider what would happen in the generating extension from Figure 2.11 (on page 19 if we ran it with x = 0.

It would compute along happily but as it tried to specialize the `EQUAL` block it would try to divide by zero, which could lead to all sorts of mischief. Possibly the execution of the generating extension would simply abort, so we'd never get any residual program. This means a problem in one branch prevents us from getting a residual program that works correctly even in the branch where there is no problem.

Furthermore it is (to a human reader) clear as day that when `x` is zero, the `EQUAL` branch is only taken when the square of something plus one is zero, i.e., never. What irony! The original `pgm_B` *never* tries to divide by zero for any input combination, yet our generating extension fails because it tries to do it anyway.

The important message that needs to be delivered here is that *we have found no good strategy for avoiding this kind of problems*. They can and do appear in the C-Mix/II we're shipping.

If division by zero was the only problem it would be easy, of cause. One could simply prepend every static division with a test on the divisor and, if zero, emit some residual code that provokes a program halt and return to the pending loop. Or, alternatively, one could simply forbid static divisions totally.

However, the C language has a creative infinity of possible ways to behave badly, many of which are impossible to detect just by looking at the code. If one wanted to guard oneself against every possible form of pointer abuse, the generating extensions would be so heavy with security checks that it would need gigantic resources. And, worse, it would be impossible to interface the generating extension to unspecialized, separately compiled program modules.

Fortunately, though, our experience is that problems such as this one rarely seems to occur in practise.

### 2.2.6 When to use the pending list

If you look again at Figure 2.13 (on page 21) you'll notice that sometimes a "`goto X;`" in the subject program is simply translated to

```
goto X ;
```

while at other times the more elaborate

```
Emit("goto ?",PutPending(&&X));
goto pending_loop ;
```

is used. How come?

First, using the pending list is rather heavy on memory and time usage while runing $p_{gen}$: We need space to store memoized values, and it takes time to compare the static states with earlier pending-list entries, as well as to memoize values and later restore them again. Directly jumping around between $p_{gen}$ blocks is much faster, so we should prefer doing that, all things being equal.

On the other hand, using the pending list also gives us the opportunity to save some specialisation effort and produce a less bloated residual program, if we succeed in sharing the target label between multiple residual jumps. And, of

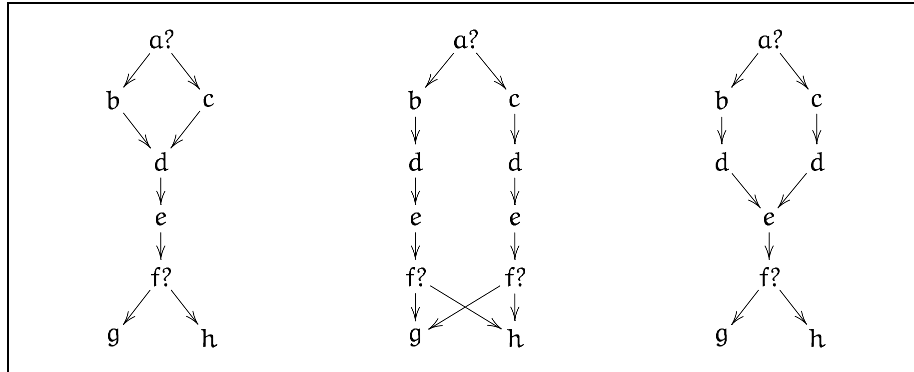



*Figure 2.16: Flow graphs that examplify why case (2) in the transition compression strategy is needed.*

course, at least one of the jumps in a dynamic conditional *must* use the pending list; that is what it was originally conceived for.

These considerations often tend to point in different directions (see Jones et al. (1993, Section 4.4.4) for a general discussion of this *transtition compression problem*). Luckily, the precise choice we make affects neither the correctness of the residual program nor the danger than $p_{gen}$ may loop indefinitely. The only things at stake are the efficiency of $p_{gen}$ and the number of times $p_{res}$ will contain identical pieces of specialized code.

The compromise we feel is reasonable for C-Mix$_{II}$ is this: A jump gets handled through the pending list if the target label

1. is also the target of some dynamic conditional (either the “then” or the “else” branch), or
2. is the start of a basic block that ends with a dynamic conditional *and* the basic block is itself reachable from some dynamic conditional.

It is easy to convince onself that case (1) is a good idea. There is no reason why it wouldn’t work to manage just the conditional jumps to a given label via the pending list and let other jumps to it be direct jumps. We simply think it is more useful to try to share code when we reach a point that might be in the pending list.

For an example of why case (2) is needed, look at Figure 2.16. We are trying the specialize the program to the left, with nothing dynamic. If only labels b, c, g, and h are handled through the pending list we arrive on the residual program shown in the middle of the figure. This flow graph cannot be expressed without explicit `goto`s which makes the residual program dificult to read. Case (2) in the above strategy specifies that e should be handled through the pending list too, giving the residual program to the right. This program *can* be expressed without explicit `goto`s. (Case (2) does not guarantee that this is *always* the case, but it does repair a great many of the situations where a `goto`-free residual program could reasonably be expected in practise).

```
void pgm_E(int_s prompt) {
  int_s count ;
  int_d sum ;
  int_d data ;
START: write_s(prompt) ;
       count = read_s() ;
       sum = 0_s ;
       goto LOOP ;
LOOP: if ( count <=_s 0_s ) goto END; else goto BODY;
BODY: data = read_d() ;
      sum = sum +_d data ;
      count = count -_s 1_s ;
      goto LOOP ;
END: write_d(sum) ;
     return ;
}
```

*Figure 2.17: A program that does I/O*

C-Mix/II also provides an option to select the brute-force strategy of passing *every* jump through the pending list[5]. Our experience is that this causes $p_{gen}$ to be unacceptably slow, so the option is only provided for the possible benefit of hard-core users that really want to squeeze every bit of code sharing out of $p_{res}$.

## 2.3 External function calls

Real-life programs do not receive all of their input as function parameters, such as the simple programs we have seen do. They also do not restrict themselves to providing output as a single `int` return value.

We need to extend our generating extensions to cover programs that interact with the outside world as they run. For the sake of the example we extend our example language with `read` and `write` primitives thus:

$$\begin{array}{rcl} \textit{stmt} & ::= & \textit{Id} = \textit{expr} \ ; \\ & | & \textit{Id} = \texttt{read ( ) ;} \\ & | & \texttt{write (} \ \textit{expr} \ \texttt{) ;} \end{array}$$

where we assume a single channel for input and a single channel for output.

The question is now, what should we do about `read` and `write` in $p_{gen}$? The two obvious methods is to either let $p_{gen}$ do the input and output or have it emit `read` and `write` statements to $p_{res}$. We'll denote these by $\texttt{read}_s$, $\texttt{write}_s$, $\texttt{read}_d$ and $\texttt{write}_d$.

Using this syntax we can write program such as the one shown in Figure 2.17. The program first "prompts" the user by writing its `prompt` argument to the output channel. Then it reads a number $n$ from the input channel, and after that $n$ more numbers whose sum it writes to the output channel.

[5]Except jumps where we can *prove* that code sharing is not an issue, because they cannot be reached from a dynamic conditional, even indirectly.

```
void pgm_E_gen(int prompt) {
  int count ;
  Code sum = MakeName("sum");
  Code data = MakeName("data");
  Code fun_name = MakeName("pgm_E_res");
  BeginFunction("int ?(void)",fun_name);
  LocalVar("int ?",sum) ;
  LocalVar("int ?",data) ;
  NewPendingList();
  RegisterStaticVar(&count,sizeof count);
  goto START ;
 pending_loop:
  if ( AnyPending() ) goto *GetPending();
  DisposePendingList();
  EndFunction();
  return;
 START:
  write(prompt) ;
  count = read() ;
  Emit("? = ?",sum,Lift(0)) ;
  goto LOOP ;
LOOP:
  if ( count <= 0 ) goto END; else goto BODY;
BODY:
  Emit("? = read()",data) ;
  Emit("? = ? + ?",sum,sum,data) ;
  count = count - 1 ;
  goto LOOP ;
END:
  Emit("write(?)",sum) ;
  Emit("return") ;
  goto pending_loop;
}
```

*Figure 2.18: Generating extension for the program in Figure 2.17*

The binding-time annotations encode that we want the count of summands to be static input and the summands themselves to be dynamic. Naturally we only want the sum to be output by $p_{res}$, but the initial prompt should appear when $p_{gen}$ expects its part of the input.

The generating extension that accomplishes this has no surprises. It can be seen in Figure 2.18. If we ran it and answered 4 to the prompt we would get the residual program in Figure 2.19.

### 2.3.1 External calls and correctness

In the preceeding example everything went well. Is it as simple as that, always? Regrettably no.

```
void pgm_E_res(void) {
  int sum ;
  int data ;
  sum = 0 ;
  data = read() ;
  sum = sum + data ;
  data = read() ;
  sum = sum + data ;
  data = read() ;
  sum = sum + data ;
  data = read() ;
  sum = sum + data ;
  write(sum) ;
  return ;
}
```

*Figure 2.19: The program in Figure 2.17 specialized to 4*

The bad news of this section is that some binding-time annotations are asking for trouble. They create problems we have no way of recovering from.

The good news is that everything will be fine as long as the annotated program fits the following two rules:

a) *specialized code cannot be shared if a static* `read` *is in any way reachable from the label.*

b) *static* `read` *operations must not be allowed to be specialized speculatively*.

These rules *can* be reliably checked as part of the binding-time analysis. C-Mix/II will not simply go ahead and produce an incorrect program but will halt with an error message instead, telling the user to revise his expectations.

To see what might go wrong, consider the version of `pgm_E` shown in Figure 2.20. Here, the number of summands is dynamic but the `read()`s for the summands themselves are static. Intuitively, expecting $p_{gen}$ to be able to read in some number of integers without ever telling it how many precisely, is bound to create problems.

So it does, but which form of problems exactly do we encounter? One would probably expect something like a $p_{gen}$ that kept reading in numbers while trying to construct a $p_{res}$ that output the sum of an appropriate initial sequence of $p_{gen}$'s (infinitely long) input stream.

Try it. Construct a $p_{gen}$ according to the principles we've seen so far. Then "run" it[6] and see what happens. Use the Fibonacci sequence (1, 1, 2, 3, 5, 8,...) for the static inputs.

If you did everything by the book, you didn't get a $p_{gen}$ that tried to read infinitely many summands—after the program had succesfully read the first one or two summands, speclib would decide that it would be safe to share some

[6] You'll have to do that by hand, because you don't have a speclib that actually works as described here

```
void pgm_E(int_d prompt) {
  int_d count ;
  int_d sum ;
  int_s data ;
START: write_d(prompt) ;
       count = read_d() ;
       sum = 0_d ;
       goto LOOP ;
LOOP: if ( count <=_d 0_d ) goto END; else goto BODY;
BODY: data = read_s() ;
      sum = sum +_d data ;
      count = count -_d 1_d ;
      goto LOOP ;
END: write_d(sum) ;
     return ;
}
```

*Figure 2.20: Exercise: what happens now?*

residual code, and you'd end up with a residual program very like the one in Figure 2.21.[7]

Now, the interesting fact is not that things go wrong. After all, what we're asking is clearly unreasonable. The interesting fact is that we *do* get a complete, syntactically correct, residual program which does something when we run it. Only, of course, it does the *wrong* thing: certainly it does not compute the sum of the first $n$ terms in the Fibonacci sequence.

Compare that to the blatant assertation on page 23 that whenever we get a residual program at all it is going to behave correctly.

Clearly there's something here that does not quite fit together. We'll have to try to save face by inventing some condition that will tag Figure 2.20 as illegal. After all we did add some new features since page 23, so we ought to be entitled to put some conditions on their use.

So we have to ask ourselves: what is basically wrong in Figure 2.20 (apart from the fact that it doesn't work)?

Most people's first quick answer would be something about `read`$_d$ operations coming before `read`$_s$. On further examination, however, this idea does not give us anything. It turns out to be fully possible to get into trouble without mixing `read`$_d$ and `read`$_s$ in any suspect way. For example, we could replace

```
count = read_d() ;
```

with

```
count = read_s() ;
```

lifting the read value into `count` as soon as it has been read.

What actually failed is the code sharing stategy. It should not have allowed the different versions of the `BODY` to be shared. Why? Because the `BODY` contains

[7]It is also possible to get a shorter residual program, if the random garbage that the `data` variable starts out with happens to be `1`.

```
void pgm_E_res(int prompt) {
  int count ;
  int sum ;
    write(prompt) ;
    count = read() ;
    sum = 0 ;
    if ( count <= 0 ) goto L1; else goto L2;
L2: sum = sum + 1 ;
    count = count - 1 ;
    if ( count <= 0 ) goto L3; else goto L4;
L4: sum = sum + 1 ;
    count = count - 1 ;
    if ( count <= 0 ) goto L3; else goto L4;
L3: write(sum);
    return ;
L1: write(sum);
    return ;
}
```

*Figure 2.21: Answer to Exercise 2.20*

a static `read`, so we can't be sure the specialization will progress in the same way each time we do it. And, come to think about it, it would be equally bad to allow "indirect" sharing, by sharing the residual label of a block that ends with a jump to `BODY`.

Thus rule (a):

a) *specialized code cannot be shared if a static* `read` *is in any way reachable from the label.*

This happens to "solve" the problem at hand in the sense that $p_{gen}$ will now become an infinite loop that reads in more and more numbers but never gets to finish a residual program. At least the page 23 assertation is saved for this time.

Now, why do we need the additional rule:

b) *static* `read` *operations must not be allowed to be specialized speculatively*.

Imagine that the `END` label in Figure 2.20 $read_s$ a number which it subtracts from the sum before writing it out. Hand tracing the actions of $p_{gen}$, we might still succeed, but when $p_{gen}$ gets compiled and run on a computer, the poor user would have a very hard time figuring out when to input what—the order in wihch $p_{gen}$ does the various $\texttt{read}_s$s depends subtly on how `GetPending` chooses which item to take out of the pending list; and even if we defined *that* firmly, the resulting order would by no means be obvious from a study of the annotated subject program. If the source of $read_s$ numbers was not a terminal user but a disk file, all bets would be completely off.

### 2.3.2 "Under dynamic control"

When enforcing rule (b) of the previous section, we need to know whether a particular statement in the program risks being specialized speculatively. This

```
   if_d ( ... ) goto A; else goto B;
A: /* do something here */
   goto C;
B: /* do something else here */
   goto C;
C: /* common actions */
```

*Figure 2.22: Label* `C` *might be specialized speculatively.*

is a generalization over all possible specialization processes, and is theoretically undecidable. We have to resort to approximations.

Of course we want a "safe" approximation in the sense that if it says something can *not* be specialized speculatively it really can't. Saying that something may be specialized speculatively when in practise it is not, is not dangerous, but we do not want it to happen too often.

The basic idea is to assume that any code that is reachable from a dynamic conditional might be specialized speculatively. We call this being **under dynamic control**. Even though the two branches of the conditional "flows together" shortly after, as on Figure 2.22, we must still consider `C` to be under dynamic control, because the static effects of `A` and `B` might be different enough to prevent sharing of the subsequent blocks.[8]

With this definition of dynamic control, it turns out that when we enforce condition (b) of the previous section, we get (a) for free. With (b) no label that can reach any static `read` is allowed to be a target of a dynamic conditional. But by the rules of Section 2.2.6 this means that such a label is not handled through the pending list, so there is no risk of it being shared.

### 2.3.3 Arbitrary external functions

When one tries to generalize the rules of Section 2.3.1 to other external calls than `read`, one needs a little support from the user.

A good example of why this is needed is `write`. The rules only mentioned `read`; how should `write` be treated? The safest choice would be to impose the same restrictions on $\texttt{write}_s$ as on $\texttt{read}_s$. The values we read could very well be the environment's response to the values we write—and then it would be important that the prompts appeared in the right order.

On the other hand, one could also imagine programs where it didn't matter if the output got a little mixed up. The output could be mere progress messages, and the user might not care that $p_{gen}$ wrote out, "`now taking course A; now taking course B`" as long as $p_{res}$ chose either course A or course B and kept its mouth shut about it.

This suggests that the user has to specify how the environment responds to the data we write out. If the rest of the execution depends on what value was written, or on whether a value was written at all, we have to treat that $\texttt{write}_s$ as cautiously as a $\texttt{read}_s$. However, if it is irrelevant whether a piece of output

[8]And then we're even forgetting that in the default case `C` never gets a chance to be shared at all because it is not the target of a dynamic conditional. The point is that even if `C` *was* handled through the pending list it could not be sure of sharing, so it could have to be speculatively specialized.

gets delivered or doesn't get delivered, or gets delivered twice, we can treat the $\texttt{write}_s$ as any other static action.

Now, to to generalize to arbitrary external functions: *Dynamic* external function calls are always safe, and are what C-Mix/II will give you if you do not specify otherwise. If you want to have something happen in $p_{gen}$ instead, you have to specify whether the call is

**sensitive**, meaning that it is dangerous to duplicate or eliminate it during the specialization process. A call falls in this class if it may alter some important state property that the pending list mechanism cannot see when it memoizes and compares states.

`read` is a prominent member of this category: it alters the property of where in the input stream we are, and the pending list mechanism cannot memoize that.

`write` is only sometimes here: The question of "what has been written to the output stream yet" *is* a state property that is invisible to the pending list, but it is only sometimes that it is important.

**benign**, those we can freely duplicate or eliminate. As we have argued, some calls to `write` belong here. Another, more intuitive, example is the `sin` and `log` from the math library: there is no reason to treat a sine or logarithm calculation as more hazardous than a multiplication[9].

Other examples include external function calls that query some kind of external database, as long as the contents of the database does not change while $p_{gen}$ runs.

The difference between the two kinds of calls is only in whether the binding-time annotated program can be accepted: The sensitive calls must not appear in code that is under dynamic control, while the benign ones can appear anywhere.

Once the binding-time annotated program has been accepted, however, the two kinds of calls are treated in the same way. For now, that is. In Section 2.4.3 we shall see differences arise.

Notice that the real C-Mix/II allows a more fine-grained specification of the behavior of static external calls than the mere division into sensitive and benign ones. That serves to improve the precision of auxiliary analyses such as the pointer analysis but is not particularly relevant to the topics discussed in this report.

### 2.3.4 The role of the BTA, revisited

Now is a good time to review the task of the binding-time analysis. We still don't know enough to construct it, but we're beginning to get a feel for what the problems are.

To back up a little: The binding-time analysis takes as input a program without annotations, and produces "suitable" binding-time annotations for it by decorating the types that appear in the program. We know that there can be several possible annotated versions of the program. How does the analysis select one?

[9] And then again there is. If one gives a negative argument to `log`, it is supposed to set `errno` to `EDOM`. Most real programs seems to silently ignore that possibility, but the programs that do check for it might still be confused if the `log` operation occurs speculatively.

For pragmatic reasons C-Mix/II ignores this by default and treats most of the math functions as benign. The user may, however, specify that individual calls should be treated with more care.

```
int_d pgm_F(int_d x) {
S: if ( x==0 ) goto A; else goto B;
A: x = read_s() ;
   return x ;
B: return x - 1;
}
```

*Figure 2.23: An impossible task for the BTA*

The basic idea is to select the annotation with the fewest possible "$\ldots_d$"s and the most possible "$\ldots_s$"s. It would seem there is an easy solution to that: Simply make everything static. Indeed, an annotation with everything static passes every condition we have stipulated so far.

What we've forgot here is that the user might want some of the input to be dynamic, whether the input is given as parameters to the "goal" function or `read` in while the program runs. In fact, the user is *expected* to specify which parts of the program's I/O behaviour are static and which parts are dynamic—and the BTA is supposed to find a set of annotations that matches that.

It would be nice to say that the BTA can always do that. In some cases, however, it turns out to be impossible. If the user requests the program in Figure 2.23 analyzed, there's no way the given annotations can be completed. The condition `x==0` *cannot* be anything but dynamic, and this means that block `A` is *necessarily* under dynamic control, conflicting with the rule that a $\texttt{read}_s$ must not be that. The best we can hope for in such a situation is for the BTA to emit an error message and give up.

In practise, what the BTA does is to compute a "maximally static" set of binding-time annotations without regard to the rules of Section 2.3.1 Afterwards it checks whether its result matches the rules.

If some rules are broken, the original problem formulation was impossible. All of the conditions in the program have already been made as static as they can, meaning that blocks that are still under dynamic control are that of necessity. Though one *could* imagine reannotating the offending call as "$\ldots_d$" instead of "$\ldots_s$" that would be against the user's reqests which are supposed to be explicit about the binding time of any sensitive call.

## 2.4 Functions

In this section we'll extend our example language with user-defined functions with or without return values and calls to them:

| | | |
|---:|---|---|
| *program* | ::= | *functions goalfun* |
| *functions* | ::= | ε \| *function functions* |
| *function* | ::= | *heading* `{` *decls blocks* `}` |
| *goalfun* | ::= | *heading* `{` *decls stmts return* `}` |
| *stmt* | ::= | . . . \| *Id* `=` *Id* `(` *exprs* `)` `;` \| *Id* `(` *exprs* `)` `;` |
| *exprs* | ::= | *expr* \| *exprs* `,` *expr* |

Programs now consist of a number of “internal” function definitions and the definition of a **goal function**. The disinction between the goal function and the other functions is nice to have technically: the goal function is the only one that is called “from the outside” of the program, and we will also assume that the goal function is never called from inside the program. Furthermore, the body of the goal function has an especially simple structure which means that it will not need a pending list.

In C-Mix/II, the goal function is generated “behind the scenes” when the user specifies which function he wants to specialize: the specializer directive

```
goal: foobar specializes baz(?,$1)
```

causes C-Mix/II to create an implicit goal function resembling

```
int_d foobar(int_d p1, int_s p2) {
  int r ;
  r = baz(p1,p2) ;
  return r ;
}
```

Thus the user of C-Mix/II does not need to worry about the restrictions on the shape of the goal function, because he does not write the goal function himself.

There are no global variables in the language yet. We'll add them in Section 2.5.

### 2.4.1 Function specialization: basic principles

Figure 2.24 illustrates the basic behaviour of the specializer on programs with internal functions. One function in the subject program can have multiple specialized counterparts in the residual program. This way the residual functions are able to respect the different values of the static parameter `b` even though `b` is not explicitly represented in the residual program. Each call statement in the subject program becomes a call to a corresponding specialized function in $p_{res}$.

Figure 2.25 shows how a generating extension that accomplishes this could look. The different return types of `f` and `g` lead to slightly different $p_{gen}$ idioms being used. Some points are, however, valid regardless of return type.

- Each function in the subject program leads to similarly named function in the generating extension. We call this a **generating function**. The generating function is called when a function call expression must be specialized. Its parameters are the $p_{gen}$ represenation of the parameters in the function call expression.
- Internally, the generating function is equipped with the entire pending list machinery we've already developed.
- For a dynamic parameter (such as `f`'s `a` parameter), the generating function has two `Code` variables that both in some sense represents the parameter. `a_expr` contains the *actual* parameter used in the call expression that must be specialized. The plain `a` contains the name of *formal* parameter in the residual function that is being generated. Understanding this difference is important: it stresses that the division of labor between caller and callee in $p_{gen}$ is subtly different from an “intuitive” understanding of how the subject program works.

```
int_d f(int_d a, int_s b) {
ST: a = a *_d b ;
    return a ;
}
void h(int_s p) {
  int_d q ;
ST: q = read_d() ;
    write_d(p +_d q);
    return ;
}
int_d pgm_G(int_s x, int_d y) {
  int_d z ;
  z = f(y, x) ;
  y = f(y +_d 1_d , x -_s 1_s ) ;
  h(42) ;
  return z + y ;
}
```

```
int f_5(int a) {
  a = a * 5 ;
  return a ;
}
void h_42(void) {
  int q ;
  q = read() ;
  write(42 + q);
  return ;
}
int f_4(int a) {
  a = a * 4 ;
  return a ;
}
int pgm_G_res(int y) {
  int z ;
  z = f_5(y) ;
  y = f_4(y+1) ;
  h_42() ;
  return z + y ;
}
```

*Figure 2.24: A program with functions and its residual for* `x==5`

```
Code f(Code a_expr, int b) {
  Code a = MakeName("a");
  Code fun_name = MakeName("f");
  BeginFunction("int ?(int ?)",fun_name,a);
  NewPendingList();
  RegisterStaticVar(&b,sizeof b);
  goto ST;
 pending_loop: if ( AnyPending() ) goto *GetPending();
               DisposePendingList();
               EndFunction();
               return MakeCode("?(?)",fun_name,a_expr);
 ST: Emit("? = ? * ?",a,a,Lift(b)) ;
     Emit("return ?",a);
     goto pending_loop:
}
void h(int p) {
  Code q = MakeName("q");
  Code fun_name = MakeName("h");
  BeginFunction("int ?(void)",fun_name);
  LocalVar("int ?",q);
  NewPendingList();
  RegisterStaticVar(&q,sizeof q);
  goto ST;
 pending_loop: if ( AnyPending() ) goto *GetPending();
               DisposePendingList();
               EndFunction();
               Emit("?()",fun_name);
               return ;
 ST: Emit("? = read()",q);
     Emit("write(? + ?)",Lift(p),q);
     Emit("return");
     goto pending_loop:
}
void pgm_G_gen(int x) {
  Code y = MakeName("y");
  Code z = MakeName("z");
  Code fun_name = MakeName("pgm_G_res");
  BeginFunction("int ?(int ?)",fun_name,y);
  Emit("int ?",z);
  Emit("? = ?",z, f(MakeCode("?",y),x) );
  Emit("? = ?",y, f(MakeCode("? + 1",y),x-1) );
  h(42);
  Emit("return ? + ?",z,y);
  EndFunction();
}
```

*Figure 2.25: Generating extension for* `pgm_G`

- While $p_{gen}$ runs, the calls to the generating function take place while the specialization of calling function has not yet been finished. Speclib uses the `BeginFunction` and `EndFunction` calls to maintain a stack of "functions under construction" and directs each `Emit` to the topmost of these.

- The simpler structure of the goal function lets `pgm_G_gen` use simpler constructions than for the genenerating functions: it does not need a pending list, and it does not return anything to its caller (which is one reason we do not allow the goal function to be called from within the residual progam).

The difference between `f` which returns an $\texttt{int}_d$ and `g` which returns `void` surfaces, not surprisingly, in the difference between the how the generating functions return:

- Functions returning $\texttt{int}_d$: The generating function returns a `Code` containing a call expression that can be used in the residual function that is being generated. Note that the returned piece of code is *not* simply the function name ("`f_5`")[10]; nor the entire residual statement ("`z=f_5(y)`") put precisely the call expression: "`f_5(y)`".

  This division of labor between the caller and the callee may seem strange; it has been designed to allow the generating function to inline its residual code without changes in the call interface of the generating function. We'll see more about inlining in a couple of pages.

- Functions returning `void`: It is still the generating function's job to construct a call expression. This call expression is not, however, returned to the caller but emitted directly as a call statement into the caller's residual code. Note how this is accomplished by calling `Emit` *after* `EndFunction`.

  There are several reasons for this. First, returning the call expression and having the caller emit it itself would be going to be awkward when we introduce inlining.

  Second, we are going to need a scheme like this for functions that return $\texttt{int}_s$. In that case the return value of the generating function is going to be occupied by the actual `int` that is returned.

  Third, letting the generating function return `void` means that if a function has no $\texttt{int}_d$ parameters and does not return $\texttt{ind}_d$, its generating function will have the exact same heaing as the subject function. From the *caller*'s viewpoint, there is no difference between calling a non-dynamic generating function and calling an external function. This subtle symmetry is going to be useful when we extend the technique to the function pointers of full C (in Section 3.2.7).

- Functions returning $\texttt{int}_s$: Basically, the behavior with respect to residual code is identical to functions returning `void`—only the generating function returns an `int` which is the actual return value.

  Figure 2.25 does not show an example of this; that is because returning $\texttt{int}_s$ from a function requires some considerations that will be easier to understand after having introduced global variables. Thus we shall have more to say about functions returning $\texttt{int}_s$ in Section 2.5.4.

These rules are summarized in Table 2.2

[10] as was the case in early versions of C-Mix/II

| subject fcn. returns | generating fcn. returns | residual fcn. returns |
|---|---|---|
| `int`$_s$ | `int` | `void` |
| `int`$_d$ | `Code` | `int` |
| `void` | `void` | `void` |

*Table 2.2: The rules for the return type of generating functions. The similarity with the Table 2.1 on page 14 is by design!*

The reader should also observe that the parameter type lists for the generating function and the generated residual function can be obtained by "filtering" the binding-time annotated parameter types through the table on page 14.

### 2.4.2 Sharing residual functions

When the generating functions are written similarly to the one in Figure 2.25, a new residual function is generated each time a function call expression gets specialized. This is unsafe and wasteful.

*Unsafe* because if the subject function is recursive it is very likely that the generating function will be stuck in an infinite recursion. Often the situation has a natural solution of generating a recursive residual function.

*Wasteful* because it means that the residual program contains several identical functions rather than calling the same function from two different places. This needless duplication propagates to the functions that are called from the two identical functions, and so on.

These are two sides of the same general problem: that this strategy exclusively produces residual program whose **call graph**s are tree-shaped (i.e., no function is called from two different places, which also means that no recursion is possible).

Basically this is the same problem we had in Section 2.2.3 with respect to the flow graphs of residual problems. Then we had to invent sharing of code inside the residual function. Now, our solution is similar: we must share entire residual functions. That is, instead of creating a new residual function each time a generating function is called, sometimes we have to reuse the name of a residual function we have already begun to create.

This means that we need to decide on when to share names. The analysis is similar enough to the local code sharing case that we will not repeat it in detail. To summarise, we can share an old function name if

- it is the name of a specialized version of the same subject function, and
- the values of the static parameters that were specialized into the function match those we have now.

Thus each generator function will have its own "pool" of reuse candidates, each being a pair of a `Code` holding the residual function name, and memoized values for the function parameters. Notice that the values we need to memoize are not exactly the same as the ones we memoize in the pending list[11]: local

[11]Noting the similarity between local code sharing and function sharing, the reader may wonder if not the sharing of functions needs some counterpart to the pending list we use for keeping track

```
struct FunctionPool f_copies =  NULL, "f" ;
/* Speclib defines an appropriate struct FunctionPool */
Code f(Code a_expr, int b) {
  Code a = MakeName("a");
  Code fun_name ;
  if ( PutFunction(&f_copies,&fun_name,b) )
    return MakeCode("?(?)",fun_name,a_expr);
  BeginFunction("int ?(int ?)",fun_name,a);
  NewPendingList();
  RegisterStaticVar(&b,sizeof b);
  goto ST;
 pending_loop:
  if ( AnyPending() ) goto *GetPending();
  DisposePendingList();
  EndFunction();
  return MakeCode("?(?)",fun_name,a_expr);
 ST:
  Emit("? = ? * ?",a,a,Lift(b)) ;
  Emit("return ?",a);
  goto pending_loop:
}
```

*Figure 2.26: Sharing-aware generating function*

variables that are not parameters have no place in the function memoization. Anyway, at the call point they start out as garbage, and one garbage value is as good as another[12].

Figure 2.26 shows how this might look in practise (compare Figure 2.25): Before the generating functions begins to generate a residual function, it checks if a suitable function has already been made. If so, it immediately constructs a call of the reused function and returns to its caller.

As usual, most of the complexity is hidden in a Speclib function: the one that is called `PutFunction` in Figure 2.26[13]. Its name is meant as a hint that what it does is similar to the memoization part of `PutPending`'s work: It first tries to find a reuseable function, and if it succeeds it returns the name of that (in the `fun_name` variable). If no reuse is possible, it constructs a new name to return, and puts an entry of it into the reuse pool. The fact that this happens before the actual specialization of the function is performed also means that the new reuse entry will be available if the specialization should contain a recursive call (though the one shown here does not).

---

of local code sharing. In some sense, the "reuse pool" fits that role. However the "reuse pool" does not govern which functions we *still need* to begin specializing; we get that for free from the call-and-return discipline in $p_{gen}$.

[12]Programmers who expect special kinds of garbage in their unitialized local variables should learn how to use the `static` keyword in C, or they deserve to get wrong results from handing their programs to an unsuspecting partial evaluator.

[13]The reader should excuse us from side-stepping the issue of how `PutFunction` knows how many parameters to memoize. The notation in the example would not make sense in the real world; but using a technique that actualy worked would make the example harder to follow for no good reason.

Note that the `a_expr` parameter to the generating function is not memoized. This is safe because it does not affect the residual function that is actually generated: it is only used when constructing the return value from the generating function—and it is used in the same way whether or not the call is shared.

#### Infinite specialization by recursion

Recall from page 23 that local code sharing allowed some generating extensions to produce a residual program with loops instead of going into an infinite loop, but did not eliminate all possibility of infinite loops.

Likewise, while function sharing may allow some generating extensions to produce recursive residual programs, there is still a risk that $p_{gen}$ might get stuck in an infinite recursion if new values for the static parameters keep turning up.

Like for local code sharing, this can only be prevented by the user realizing the situation and specifically forcing the culprit parameter to be dynamic. The BTA, simply trying to make everything as dynamic as the rules allow, does not do this by itself.

### 2.4.3 Interplay between function specialization and calls to external functions

In Section 2.3.1 we developed conditions for when "sensitive" calls to external functions should be allowed. We need to specify what those rules should look like in the presence of user-defined functions.

The only rule that turned out to need enforcement in the no-functions case was that *sensitive calls to external functions may not happen under dynamic control*. This still holds. We only need to specify that when we have functions, the entire body of a function must be considered under dynamic control if *any* call to the function is itself under dynamic control.

However, there was another rule for sensitive calls to external functions: their effects must not be allowed to be shared. Previously it happened that this condition was ensured for free once one ruled out dynamic control. Not so with function calls: even if a program contains no dynamic conditionals at all, there might still be a potential for function sharing. So we need to be explicit about not to share residual functions if sensitive calls were performed while specializing them.

In C-Mix/II we take the approach of deciding this while constructing $p_{gen}$: A function is marked as **unshareable** if it contains any sensitive call, or a call to a function that contains such a call, and so on.

Such a function could be treated as in Figure 2.25 (on page 36. However, as we shall see shortly there is a smarter way of handling it.

### 2.4.4 Function inlining

Unshareable functions have a unique property: each residual version of them is called from exactly one point in $p_{res}$. This means that there is really no reason not to integrate their bodies in the caller's definitions rather than let them be a separate functions. On the contrary, it would making $p_{res}$ run a tad faster because we would be able to avoid function entry and exit code.

The reasons to make some code a function of its own is to make the program easier to understand for human readers and to save space by not duplicating the same code. The former point is not important for a partial evaluator; the very point of partial evaluation is to trade maintainability and ease of understanding for speed. And in the case of unshareable functions we know the latter point does not apply in the residual program.

Thus, for unshareable functions we simply go ahead and insert their residual code into the calling function instead of generating a call. The calling conventions for generating functions have been designed to make this possible without the caller being aware of it.

Figure 2.27 show how an inlining version of the generating function in Figure 2.25 would look. Some points to observe are

- The generating function does *not* call `BeginFunction` and `EndFunction`. This means that the `Emits` in it end up in the "current function" that is already current when it is called. Which, elegantly, performs the inlining.
- The function has the usual pending list construction which means that its contribution the the caller's flow graph can be complex. Returning from the inlined function gets converted to unconditional jumps to a label the generating function emits just before returning to the caller. Thus the caller can continue emitting code that will be executed after the call.
- The dynamic parameter still has "double representation": `a_expr` which holds the actual parameter expression and `a` which is now the name of a local variable that replaces the formal parameter.

  We cannot simply substitute `a_expr` into every occurence of `a` in the subject function, because a formal parameter is passed by value and should be allowed to change without affecting variables in the caller. Indeed it would not make sense at all to substitute an expression like "`y+1`" into the left side of an assignment.

  Another reason for transfering the value of `a_expr` to `a` as soon as possible is that when we introduce global variables the value of `a_expr` might change during the execution of the function, which must not affect the value of the formal parameter.
- The intuitive way to handle a return from the inlined function would be to simply return the expression in the `return` statement from the generating function. That is,

  ```
  return MakeCode("?",a);
  ```

  Remember, though, that the inlined function can be arbitrarily complex and contain several return statements with different expressions. Thus we need another local variable `return_val` that communicates the return value to the calling code in $p_{res}$. Then the generating function returns the name of that variable.[14]
- Local variables are declared with the Speclib call `LocalVar` rather than `Emit`. In this way Speclib can collect all of the local declarations at the

[14]In the real C-Mix$_{II}$, if the function contains no dynamic conditionals the entire pending list mechanism is optimized away when constructing $p_{gen}$, and the "intuitive" return translation is used instead.

```
Code f(Code a_expr, int b) {
  Code a = MakeName("a");
  Code return_val = MakeName("f") ;
  Code end_label ;
  NewPendingList();
  RegisterStaticVar(&b,sizeof b);
  LocalVar("int ?",a);
  LocalVar("int ?",return_val);
  end_label = MakeLabel() ;
  Emit("? = ?",a,a_expr);
  goto ST;
 pending_loop:
  if ( AnyPending() ) goto *GetPending();
  DisposePendingList();
  Emit("?:",end_label);
  return return_val ;
 ST:
  Emit("? = ? * ?",a,a,Lift(b)) ;
  Emit("? = ?",return_val,a);
  Emit("goto ?",end_label);
  goto pending_loop:
}
```

```
void h_42(void) { (as in Figure 2.24) }
int pgm_G_res(int y) {
  int z ;
  int a_5 ;
  int f_5 ;
  int a_4 ;
  int f_4 ;
  a_5 = y ;
  a_5 = a_5 * 5 ;
  f_5 = a_5 ;
  goto L1 ;
 L1: ;
  z = f_5 ;
  a_4 = y + 1 ;
  a_4 = a_4 * 4 ;
  f_4 = a_4 ;
  goto L2 '
 L2: ;
  y = f_4 ;
  h_42() ;
  return z + y ;
}
```

*Figure 2.27: A generating function that inlines its function, and the residual program that results from this*

front of the function, even when some of them result from inlined functions.

- The example also illustrates the need for name management for the local variables in `pgm_G_res`.

### Other uses of inlining

Imagine specializing a subject program where some functions only manipulate static values. Granted, it is hard to imagine that being useful until we allow global variables or static return values, but once we do that (which is to say in a couple of pages from now) the situation will not be uncommon.

If the functions do not involve sensitive calls they will not be flagged as unshareable—which means that by default they are subject to the standard function sharing discipline. This means that although the static computations will be done in $p_{gen}$ and disappear from $p_{res}$ they will still leave their trace in $p_{res}$ in the form of calls to completely empty functions such as

```
void foo(void) {
}
```

This is something of a luxury problem, because an empty function is quite certainly faster to call than a function that does something. Still, a lot of empty functions may seriously impair the clarity of the residual program[15] and also its efficiency.

C-Mix/II therefore contains heuristics to identify functions which are likely to become "trivial" when specialized. These functions are artificially marked as unshareable before constructing the generating extension, which means they will be inlined at specialization time. (When an empty function gets inlined it disappears completely).

Recursive functions are exempt from these heuristics, lest we introduce an infinite recursion in $p_{gen}$ that could have been avoided.

## 2.5 Global variables

With user-defined function in place it seems natural to introduce global variables:

$$program \quad ::= \quad decls\ functions\ goalfun$$

For the imperative programmer, a global variable is the plainest of features. Supporting them in a partial evaluator, however, proves to introduce a surprisingly large number of complications.

Once one moves to real C with pointers, whatever this section says about global variables holds for local variables that, through pointers, are accessed from a function other than the one they "belong" to (see Section 3.4.2 for a explanation of exactly what this means).

[15] Though it does not appear to be necessary to look at the machine-generated residual program—how many actually read the output from their parser generator?—it turns out that examining residual code is of great help when one tries to put partial evaluation to practical use.

```
ints g ;
intd foo(intd d) {
L1: g = 42s ;
    if ( d ==d 0 ) goto L2; else goto L3;
L2: g = g +s 1s ;
    goto L3 ;
L3: return d ;
}
void pgm_H(intd d) {
  d = foo(d);
  writed(g +d d);
  return ;
}
```

*Figure 2.28: A program with a statically ambiguous return state*

### 2.5.1 The unique-return-state requirement

Consider the program in Figure 2.28 How should a specialized version look? Do not inline `foo`!

The problem here is that the value of the global variable `g` upon return from `foo` depends on the value of `foos` argument, which is not known until $p_{res}$ runs.

Ons possible solution would be to create a residual program such as the one in Figure 2.29. An auxillary global variable `statecode` is used to keep track of which global state applies when `foo` returns. After each call, the caller uses the `statecode` to decide where to go next.

A more philosophical amount of what happened could be that `foo`'s return statement may be speculatively specialized, and so the speculation propagates back to the caller through the `statecode` construction.

There is nothing intrinsically wrong or unsound in this. C-Mix/II just doesn't do it. What C-Mix/II in fact does is to forbid the construction. The binding-time annotations should follow the rule that

- *The values of the static global variables upon exit from a function must be uniquely given as a function of their values before the call, and the static parameters to the function (and the result of the static external calls in the function).*

This is the **unique-return-state requirement**.

The unique-return-state requirement is phrased desciptively, and it is not immediately clear how to enforce it in practise. We use this more operational constraint to enforce it:

- *Non-local side effects under dynamic control must not appear to static variables.*

Here, a **non-local side effect** is an assignment to a global variable. In Section 3.4.3 we shall extend this definition to cover assignments to any variable that does not reside in the current stack frame.[16]

[16] In the simplified language we are working with at the moment the only variables one *can* assign to are global variables and variables in the current stack frame.

```
int statecode ;
int foo(int d) {
  if ( d == 0 ) {
    statecode = 0;
    return d ;
  } else {
    statecode = 1 ;
    return d ;
  }
}
void pgm_H_res(int d) {
  d = foo(d) ;
  switch(statecode) {
  case 0:
    write(43+d);
    return ;
  case 1:
    write(42+d);
    return ;
  }
}
```

*Figure 2.29: A possible residual version of* `pgm_H`

### Box 2.1—WOULD NON-UNIQUE RETURN STATES BE A GOOD THING?

It is debatable—and debated—whether the unique-return-state requirement ought to be removed or not. The argument against it is of course that it might have enabled some important specialization further on to allow the global variable to stay static. This might be genuinely useful in practical applications of C-Mix$_{/II}$.

The arguments *for* the unique-return-state requirement are more subtle:

- It is easier to implement C-Mix$_{/II}$ with it. OK, this might not be so subtle after all.
- Returning from a function with polyvariant static end state *adds* labor to $p_{res}$ that was not present in the subject program: setting the `statecode` indicator before the return and testing it afterwards. Saving a few assignments to the global variable might result in a burden on multiple levels of returns. In fact the statecode construction is the first construction we have seen that might result in a $p_{res}$ that runs *slower* than the subject program (given an appropriately idealized measure of running time).
- The unique-return-state requirement is useful for the user as a way to control the spread of dynamic control. Without it, dynamic control is a highly infectous property. Once it arises somewhere deep down in a low-level function it propagates all the way up the call stack, deploying statecode checks as it goes. *With* the unique-return-state requirement, the C-Mix$_{/II}$ user can encapsulate the dynamic control in a function and know that it stays within it.

Contrarily to, e.g., the requirement that static sensitive calls to an external function may not be under dynamic control, there *is* an automatic defense against the unique-return-state requirement. One simply forces the global variables in question to be dynamic.

In the example in Figure 2.28, the increment of `g` is under dynamic control. Thus C-Mix$_{II}$ would, in sheer self-defense, force `g` to be an $\texttt{int}_\texttt{d}$. Once the variable that depends upon dynamic conditions is itself dynamic, everything is well.

We shall have more to say about enforcement of the unique-return-state requirement in Section 3.4.3.

### 2.5.2 Implications for memoization

Specializing a function produces a residual function where not only the values of the static parameters may be built in, but also the values of the static global variables. So when memoizing a function variant for deciding code sharing one must also take into account the static globals' values.[17]

One might think that we would also need this when memoizing for the pending list. The unique-return-state requirement unexpectedly comes to our rescue, however. Once we first use the pending list, we'll be about to specialize a block that is the target of a dynamically conditional jump (see page 25). This means that we'll be under dynamic control, and the *only* way to get out of that is to return from the function—at which point the pending list is forgotten. Now, the unique-return-state requirement essentially says that nobody is allowed to change the values of global variables as long as we're under dynamic control. Which means that the global variables cannot change as long as the pending list is alive, which means that the pending list does not need to care about them. (Whew! Did you follow that?).

### 2.5.3 Implications for function sharing

Consider a function that (directly or indirectly) modifies static global variables. The first time the function is called in a given static configuration, a description of the configuration is saved for later sharing. Then the function is specialized which causes some of the static globals to be changed. Fine so far.

Now, what happens the next time the function is called with the same static configuration? The configuration is found in the function's reuse pool, so the generating function simply recycles the previously constructed residual function and returns immediately. What happens to the global variables? Nothing?

That would be wrong—if some set of static side effects happened the first time we traced the execution of the function, surely the same side effects would happen the next time the subject program goes through the same drill once again. Thus if we simply specialized the code after the second call using the unchanged globals, we could be deviating from the subject program's actual behavior.

[17]When specializing the full C language, other function's local variables fall under the same considerations as we describe for global variables here, if they may be reached via pointers. Thus function call memoization becomes a rather complicated issue. This report completely avoids going into details about how that is resolved. Andersen (1996) describes the strategy currently followed by C-Mix$_{II}$. Though it is not fully up to date it about the details is not all obsolete either.

The answer, of course, is to restore the static side effects when sharing the function. An item in the function's reuse pool now consists of, (a) the name of a residual function, (b) the function's *entry state* memoized, (c) the function's *exit state* memoized. The exit state only records changes to the global variables, because the static parameters (which *are* included in the entry state) are lost when the function returns, anyway.

This can be handled without much change outside Speclib. The restore is easy: when `PutFunction` finds a matching reuse entry, it should restore its exit state before returning.

The more complex question is how to *create* the exit state. Obviously it should be a snapshot of the static global variables when the function is about to return, which implies that the generating function should collect it about the same time it calls `EndFunction`. This is fine, too, but naturally it also means that the reuse item will not be complete before the residual function has been fully specialized. Away goes the ability to create recursive residual functions?

Miraculously no. It turns out that if a residual call matches a saved entry state while the residual function is still being specialized, it is safe to reuse the function *without restoring any end state!*. How can that be? Well, either the function is going to do global static side effects or it is not. If it is not, everything is well and it really does not need restoration. On the other hand, if the function *does* any global static side effects, the only way it would be allowed to do so is by not being under dynamic control. This means that the recursive call in it cannot be under dynamic control either. And the fact that entry in a given static state leads to a recursive call in the *same* static state without being under dynamic control means that an infinite recursion is inevitable—not as an artifact of the specialization process, but as real behavior of the subject program, and the residual one too. Thus when $p_{gen}$ runs, the call to the residual function *will never return*, so it does not matter that we risk emitting wrongly-specialized code to follow it.[18]

### 2.5.4 Functions returning $\mathtt{int}_s$

The reason we deferred the full treatment of functions with static return values until now is that we wanted to have the unique-return-state requirement stated first. This is because the unique-return-state requirement also applies to static return values: a static return value must depend on nothing but the static entry conditions of the function.

In practise this means that none of the function's return statements may be under **local dynamic control**. A return statement is under local dynamic control if it is reachable from a dynamic conditional in the *same* function. This is different from our normal concept of dynamic control which is inherited from function call sites to entire functions.

[18]This is not a correct representation of what the current C-Mix/II does. It actually does not share functions that might perform global static side-effects until they are complete. And "functions that might perform global static side-effects" are taken to be precisely those functions that are never called under dynamic control. Then the above argument is used to justify that the lack of early sharing for these functions will only be a problem if there is an infinite recursions under purely static control. C-Mix/II then invokes the partial evaluator's traditional right to behave badly if the subject program does so under non-dynamic control, and feels OK to enter an infinite recursion if that is the result.

Functions with static return values are currently always unshareable. There are no deep reasons for that, other than that the code in Speclib that handles sharing of functions does not know how to communicate return values to the shared instances. This introduces no additional risk of infinite recursion in $p_{gen}$: clearly it would be impossible to share a function returning $\texttt{int}_s$ until it is fully specialized, because the return value is not known until then.

Functions with static return values are thus always inlined. The return value is handled naturally by being returned from the generating function.

## 2.6 Partially static data

From the very first time we met binding-time annotations in Figure 2.2, they have consisted of separating all the `int` types in the program into

*static* $\texttt{int}_s$s. In $p_{gen}$ they appear as perfectly ordinary `int`s. In $p_{res}$ they disappear.

*dynamic* $\texttt{int}_d$s. In $p_{gen}$ they are handled symbolically as `Code`. In $p_{res}$ they are perfectly ordinary `int`s.

The dualism of static and dynamic has served us well, and the reader may have gotten the impression that it represents a necessary, absolute truth; that it is damn well *self-evident* that these two, and only these two, are the choices we have.

The purpose of this section—as perhaps its title implies—is to prove that impression wrong.

Before we do so, however, we'll have to announce a slight shift in the text's relation to reality. The techniques that have been described in the preceding sections all have direct counterparts in C-Mix/II. Every effect mentioned so far can be recreated in the real world by giving C-Mix/II essentially the same input as in the examples here. Of course, one has to allow for the slight differences between the real Speclib interface and the idealised one we have used for the examples. And the real C-Mix/II is sometimes smart enough to optimize away a pending loop that we have shown as present in an example in order to avoid confusion. But the essentials of the Gegen process are as we have stated them here.

We would like to continue along that vein while introducing partially static data. Unfortunately that is not conveniently possible. In the real world, partially static data only arises out of the use of structs. And explaining how one handles structs would take us deeper into technical subtleties than we ought to go just for being able to present the (rather simple) basic idea.

Thus, conditions force us to present partially static data by the means of a purely hypothetical feature that has been designed solely for this purpose:

### 2.6.1 A simple example: backed-up ints

Static data are known in $p_{gen}$ and, unless lifted, lost in $p_{res}$. Dynamic data are not known in $p_{gen}$ but known in $p_{res}$. Imagine a third possibility between these extremes: data that are known in $p_{gen}$ and also known in $p_{res}$.

| binding-time type | representation in $p_{gen}$ | representation in $p_{res}$ |
|---|---|---|
| $\texttt{int}_s$ | `int` | (vanishes) |
| $\texttt{int}_d$ | `Code` | `int` |
| $\texttt{int}_b$ | `Code` + `int` | `int` |

*Table 2.3: Type representation table covering* $\texttt{int}_b$.

If the data are to be known in $p_{gen}$ it means that the computations that lead to changes to it must happen in $p_{gen}$, too. However, we might keep a "back-up copy" of the data in $p_{res}$ by letting $p_{gen}$ emit code to keep the residual back-up up-to-date each time the "master version" in $p_{res}$ changes.

It might be hard to imagine what possible benefits this could yield, because there seems to be no good reasons that $p_{res}$ would ever need to *use* the back-up copy. Imagine, however that one had an array of integer variables whose values were statically computable and changed during the program's execution. If then, once in a while, the program needed to *read* from the array with a dynamic index, the back-ups might be of some use.

Granted, this example is somewhat contrived, which is why C-Mix/II does not support backed-up ints. Still, a similar idea will be used to handle partially static structures, so let us assume we want our Gegen transformation to handle them.

First we need a binding-time annotation for requesting the back-ups to be made. Call it $\texttt{int}_b$.

How should an $\texttt{int}_b$ be repesentented in $p_{gen}$? The back-up copy in $p_{res}$ needs a name, and in $p_{gen}$ the name needs a `Code` variable to be stored in. However, the "master copy" also needs an `int` variable in $p_{res}$. Thus *one* $\texttt{int}_b$ variable in the subject program becomes *two* variables in $p_{gen}$. We can extend Table 2.1 from page 14 to cover $\texttt{int}_b$, producing Table 2.3.

We also have to decide about which lifts are possible for $\texttt{int}_b$:

- From $\texttt{int}_b$ to $\texttt{int}_s$: simple. Just ignore the back-up.
- From $\texttt{int}_b$ to $\texttt{int}_d$: also simple. Just ignore the master.
- From $\texttt{int}_s$ to $\texttt{int}_b$: no problems here either. The $\texttt{int}_s$ can be immediately used as a master copy, and we can make a back-up copy from it using the standard $\texttt{int}_s$-to-$\texttt{int}_d$ lift.
- From $\texttt{int}_d$ to $\texttt{int}_b$: impossible. The $\texttt{int}_d$'s value is only known in $p_{res}$, so there is no way to create the "master" side of a corresponding $\texttt{int}_b$.

Note how the everyday meaning of "lift" gradually ceases to make sense here: one can "lift" an $\texttt{int}_s$ to an $\texttt{int}_b$ and then "lift" that $\texttt{int}_b$ back to the same $\texttt{int}_s$ that we started out with. It is probably best just to think of "lift" as a time-honored term for "conversion between two binding-time annotated versions of the same subject type".

It is not immediately clear how to define arithmetic on $\texttt{int}_b$s in an obviously right way. We choose not to do arithmetic on them at all: if needed, it works fine to lift to $\texttt{int}_s$ and do arithmetic there, then lift back into $\texttt{int}_b$.

### 2.6.2 An implementation sketch

How do we extend our gegen principles to handle $\mathtt{int}_b$? It would seem that we would need to start all over and go through pages 10–48, working out how the various constructions should respond to $\mathtt{int}_b$ and possibly having to invent new conditions and techniques along the way.

Wait, don't give up yet! It turns out that there is a better way. Consider again $\mathtt{int}_b$ row in Table 2.3. Doesn't it look suspiciously like a formal sum of the representations of $\mathtt{int}_s$ and $\mathtt{int}_d$? It does. Might we be able to get easily around $\mathtt{int}_b$ by simply replacing each $\mathtt{int}_b$ with an $\mathtt{int}_s$ and an $\mathtt{int}_d$? We can.

In other words, when we have an annotated subject program mentioning $\mathtt{int}_b$ we can just pass that program through a simple preprocessing phase that transforms each $\mathtt{int}_b$ variable into an $\mathtt{int}_s$ variable and an $\mathtt{int}_d$ variable and rewrites the code that uses the variable accordingly. This preprocessing leaves us with an annotated program that has *no* $\mathtt{int}_b$s in it, so we can use our tried and tested Gegen techniques on that.

Thus all we have to do is to develop the preprocessing transformation. That turns out to be significantly less complex than integrating $\mathtt{int}_b$ support directly into the Gegen algorithm.

The preprocessing transformation is defined in Figures 2.30 and 2.31. The rewriting rules define recursively the transformation mapping $\{\!| \cdot |\!\}$. At some places the definition is by case analysis on the type of an expression. It is assumed that lifts are made explicit before doing the transformation: the notation "$\mathsf{Lift}_\alpha^\beta(e)$" means that the expression $e$ has type $\mathtt{int}_\alpha$ and gets lifted to $\mathtt{int}_\beta$.

The basic idea behind the expression transformation is this

- any expression of type $\mathtt{int}_s$ or $\mathtt{int}_d$ is mapped to "itself" (except if it contains $\mathtt{int}_b$ manipulations) by the $\{\!|e|\!\}^\circ$ mapping.
- an expression of type $\mathtt{int}_b$ can be transformed into expressions evaluating to either its static component or its dynamic component. This is done by the $\{\!| \cdot |\!\}^{\mathbf{s}}$ and $\{\!| \cdot |\!\}^{\mathbf{d}}$ mappings. Sometimes $\{\!|e|\!\}^{\mathbf{s}}$ and $\{\!|e|\!\}^{\mathbf{d}}$ for the same $e$ are both used. This might cause code to be duplicated; luckily a closer inspection reveals that the transformed program contains at most two copies of any original expression, and that no dynamic computations are ever duplicated.

The translation of simple assignment statements contains a small subtlety: when assigning to a partially static variable we assign to the dynamic part before assigning to the static part. This means that, say,

$$\mathtt{b} = \mathsf{Lift}_s^b(\mathsf{Lift}_b^s(\mathtt{b}) +_s 1_s) \ ;$$

is translated to

$$\mathtt{b} = \mathsf{Lift}_s^d(\mathtt{b_master} +_s 1_s) \ ; \ \mathtt{b_master} = \mathtt{b_master} +_s 1_s \ ;$$

rather than

$$\mathtt{b_master} = \mathtt{b_master} +_s 1_s \ ; \ \mathtt{b} = \mathsf{Lift}_s^d(\mathtt{b_master} +_s 1_s) \ ;$$

which would store a wrong value in the back-up copy. Assigning to the dynamic part first is safe: that way the second assignment is going to take place in $p_{gen}$, so there can be absolutely no risk that the value assigned depends on the dynamic

**For declarations:**

$$
\begin{array}{rcl}
\{\!|\texttt{int}_s\ x;|\!\} & \Longrightarrow & \texttt{int}_s\ x; \\
\{\!|\texttt{int}_d\ x;|\!\} & \Longrightarrow & \texttt{int}_d\ x; \\
\{\!|\texttt{int}_b\ x;|\!\} & \Longrightarrow & \texttt{int}_d\ x;\ \ \texttt{int}_s\ \texttt{x_master};
\end{array}
$$

**For functions:**

$$
\begin{array}{rcl}
\{\!|\texttt{int}_s\ \texttt{f(p...)}\ \ \{\ \texttt{d...}\ \texttt{b...}\ \}|\!\} & \Longrightarrow & \texttt{int}_s\ \texttt{f(}\{\!|\texttt{p}|\!\}\texttt{...)}\ \ \{\ \{\!|\texttt{d}|\!\}\ldots\{\!|\texttt{b}|\!\}\ldots\ \} \\
\{\!|\texttt{int}_d\ \texttt{f(p...)}\ \ \{\ \texttt{d...}\ \texttt{b...}\ \}|\!\} & \Longrightarrow & \texttt{int}_d\ \texttt{f(}\{\!|\texttt{p}|\!\}\texttt{...)}\ \ \{\ \{\!|\texttt{d}|\!\}\ldots\{\!|\texttt{b}|\!\}\ldots\ \} \\
\{\!|\texttt{int}_b\ \texttt{f(p...)}\ \ \{\ \texttt{d...}\ \texttt{b...}\ \}|\!\} & \Longrightarrow & \texttt{int}_s\ \texttt{f_return}; \\
 & & \texttt{int}_d\ \texttt{f(}\{\!|\texttt{p}|\!\}\texttt{...)}\ \ \{\ \{\!|\texttt{d}|\!\}\ldots\{\!|\texttt{b}|\!\}\ldots\ \} \\
\{\!|\texttt{void}\ \texttt{f(p...)}\ \ \{\ \texttt{d...}\ \texttt{b...}\ \}|\!\} & \Longrightarrow & \texttt{void}\ \texttt{f(}\{\!|\texttt{p}|\!\}\texttt{...)}\ \ \{\ \{\!|\texttt{d}|\!\}\ldots\{\!|\texttt{b}|\!\}\ldots\ \}
\end{array}
$$

**For control structure:**

$$
\begin{array}{rcl}
\{\!|\ell\texttt{:s...}\ \texttt{j}|\!\} & \Longrightarrow & \ell\texttt{:}\{\!|\texttt{s}|\!\}\ldots\{\!|\texttt{j}|\!\} \\
\{\!|\texttt{goto}\ \ell;|\!\} & \Longrightarrow & \texttt{goto}\ \ell; \\
\{\!|\texttt{if ( e )}\ \ell_1\ \texttt{else}\ \ell_2;|\!\} & \Longrightarrow & \texttt{if (}\ \{\!|\texttt{e}|\!\}^{\circ}\ \texttt{)}\ \ell_1\ \texttt{else}\ \ell_2; \\
\{\!|\texttt{return e : int}_s;|\!\} & \Longrightarrow & \texttt{return}\ \{\!|\texttt{e}|\!\}^{\circ}\ ; \\
\{\!|\texttt{return e : int}_d;|\!\} & \Longrightarrow & \texttt{return}\ \{\!|\texttt{e}|\!\}^{\circ}\ ; \\
\{\!|\texttt{return e : int}_b;|\!\} & \Longrightarrow & \texttt{f_return}\ = \{\!|\texttt{e}|\!\}^{\mathbf{s}}\ ;\ \ \texttt{return}\ \{\!|\texttt{e}|\!\}^{\mathbf{d}}\ ; \\
\{\!|\texttt{return;}|\!\} & \Longrightarrow & \texttt{return;}
\end{array}
$$

**For statements:**

$$
\begin{array}{rcl}
\{\!|x = \texttt{e : int}_s;|\!\} & \Longrightarrow & x = \{\!|\texttt{e}|\!\}^{\circ}\ ; \\
\{\!|x = \texttt{e : int}_d;|\!\} & \Longrightarrow & x = \{\!|\texttt{e}|\!\}^{\circ}\ ; \\
\{\!|x = \texttt{e : int}_b;|\!\} & \Longrightarrow & x = \{\!|\texttt{e}|\!\}^{\mathbf{d}}\ ;\ \texttt{x_master}\ = \{\!|\texttt{e}|\!\}^{\mathbf{s}}\ ; \\
\{\!|x = \texttt{f(e...) : int}_s;|\!\} & \Longrightarrow & x = \texttt{f(}\{\!|\texttt{e}|\!\}\texttt{...);} \\
\{\!|x = \texttt{f(e...) : int}_d;|\!\} & \Longrightarrow & x = \texttt{f(}\{\!|\texttt{e}|\!\}\texttt{...);} \\
\{\!|x = \texttt{f(e...) : int}_b;|\!\} & \Longrightarrow & x = \texttt{f(}\{\!|\texttt{e}|\!\}\texttt{...);}\ \texttt{x_master}\ = \texttt{f_return;} \\
\{\!|\texttt{f(e...);}|\!\} & \Longrightarrow & \texttt{f(}\{\!|\texttt{e}|\!\}\texttt{...);}
\end{array}
$$

**For call arguments:**

$$
\begin{array}{rcl}
\{\!|\texttt{e : int}_s|\!\} & \Longrightarrow & \{\!|\texttt{e}|\!\}^{\circ} \\
\{\!|\texttt{e : int}_d|\!\} & \Longrightarrow & \{\!|\texttt{e}|\!\}^{\circ} \\
\{\!|\texttt{e : int}_b|\!\} & \Longrightarrow & \{\!|\texttt{e}|\!\}^{\mathbf{d}}\ , \{\!|\texttt{e}|\!\}^{\mathbf{s}}
\end{array}
$$

*Figure 2.30:* $\texttt{int}_b$ *eliminating transformation, part 1*

**For expressions:**

$$
\begin{array}{rcl}
\{\!|x|\!\}^{\circ} & \Longrightarrow & x \\
\{\!|c|\!\}^{\circ} & \Longrightarrow & c \\
\{\!|e_1 \circ e_2|\!\}^{\circ} & \Longrightarrow & \{\!|e_1|\!\}^{\circ} \circ \{\!|e_2|\!\}^{\circ} \\
\\
\{\!|\mathsf{Lift}_{b}^{s}(e)|\!\}^{\circ} & \Longrightarrow & \{\!|e|\!\}^{\mathbf{s}} \\
\{\!|\mathsf{Lift}_{s}^{d}(e)|\!\}^{\circ} & \Longrightarrow & \mathsf{Lift}_{s}^{d}(\{\!|e|\!\}^{\circ}) \\
\{\!|\mathsf{Lift}_{b}^{d}(e)|\!\}^{\circ} & \Longrightarrow & \{\!|e|\!\}^{\mathbf{d}} \\
\\
\{\!|x|\!\}^{\mathbf{d}} & \Longrightarrow & x \\
\{\!|\mathsf{Lift}_{s}^{b}(e)|\!\}^{\mathbf{d}} & \Longrightarrow & \mathsf{Lift}_{s}^{d}(\{\!|e|\!\}^{\circ}) \\
\\
\{\!|x|\!\}^{\mathbf{s}} & \Longrightarrow & \texttt{x_master} \\
\{\!|\mathsf{Lift}_{s}^{b}(e)|\!\}^{\mathbf{s}} & \Longrightarrow & \{\!|e|\!\}^{\circ}
\end{array}
$$

*Figure 2.31:* $\texttt{int}_{b}$ *eliminating transformation, part 2*

assignment that happened "before" but will only cause something to actually change in $p_{res}$.

At first sight, the transformation of function calls and returns may seem unnecessarily complicated. Why not simply pass the static parts of any $\texttt{int}_{b}$ arguments and return values, and let the function make its own back-ups of the parameters, and the caller maintain the back-up of the destination variable itself?

Indeed, this would be possible as long as we're talking backed-up ints. Remember, though, that the sole purpose of backed-up ints is to introduce the thoughts behind the *real* partially static data support in C-Mix$_{II}$. The fact that the dynamic part of an $\texttt{int}_{b}$ can be reconstructed when we know the static part is only an accident due to the simplicity of the example. It is not going to hold true in the real world, so we might as well avoid depending too much on it now.

The transformation rules have been carefully designed to isolate knowledge about the exact relationship between the static and dynamic parts of a partially static variable as much as possible. The only rules that know what an $\texttt{int}_{b}$ in fact is are the four ones concerned with lifts to and from $\texttt{int}_{b}$.

With this design goal in mind, the logic behind the function call transformation should be easier to follow. Parameters pose no probem: we split a partially static parameter into a dynamic and a static part (here, the order is arbitrary) and split argument expressions accordingly.

Function returns are more of a problem, because the transformed function must have only one return type: we cannot return both the static and the dynamic parts through the normal function return mechanism. Our solution is to generate a global variable for holding one part of the return value. The last thing the function does before actually returning is to set this variable; it is read

```
int_b foo(int_s s) {                  int_s foo_return ;
  return s ;                          int_d foo(int_s s) {
                                        foo_return = s ;
}                                       return s ;
int_d bar(int_d d) {                  }
  int_b q ;                           int_d bar(int_d d) {
                                        int_d q ;
  if( d ) goto A;                       int_s q_master ;
          else goto B;                  if( d ) goto A;
A: q = foo(42);                                 else goto B;
                                      A: q = foo(42);
   return d+q ;                          q_master = foo_return ;
B: q = foo(117);                         return d+q ;
                                      B: q = foo(117);
   return d+q ;                          q_master = foo_return ;
}                                        return d+q ;
                                      }
```

*Figure 2.32: The* $\texttt{int}_b$ *elimintation may introduce an insignificant breach of the unique-return-state requirement.*

by the caller immediately after the function has returned. Because we treat function calls as statements instead of expressions, there is no risk of getting the return values from several different calls mixed up.

However, our choice of passing the *static* part through the global variable, involves creating a static global side effect in the function, which is known (page 44) to be hazardous in some contexts. Why not do it the other way around, that is, store the *dynamic* part in s global variable instead? The reasons are—

- Returning a value from a function may be assumed to be more efficient than storing it intermediately in a global variable. Because we want as efficient a $\mathfrak{p}_{res}$ as possible, it makes sense to use the efficient mode for the part that appears in $\mathfrak{p}_{res}$.
- Our method makes it simple for the translated call statement to do its side effects in the right sequence. This does not matter in the C subset we're currently looking at, but when left-hand sides of statements get more complicated it becomes important to assign the dynamic part first here, as well. Consider, e.g.,

  ```
  a[a[0]] = foo() ;
  ```

  If `a[0]` $=$ `0` initially and the master copy of the array element were updated before the back-up was assigned from an intermediate global, we would risk the *wrong* back-up being updated.

Instead we will—as a special exception to the usual rules about non-local side effects under dynamic control—allow the autogenerated intermediate return variable to be modified under dynamic control, as long as the modification is not under *local* dynamic control.

Then some functions may not fully meet the unique-return-state requirement with respect to the return variable. For example, in Figure 2.32, the value

of `foo_return` when `bar` returns depends on which branch of the dynamic `if` was taken. This is safe because `foo_return` is only used for anything immediately after a call to `foo`, so its value when `bar` returns is irrelevant to the specialization process.

### 2.6.3 Correctness of the transformation

This is all fine, but can we be sure it works? There are two things we could fear going wrong when using the preprocessing transformation.

First, it might be that the transformation in some way changed the meaning, ignoring the binding-time annotations, of the program. We have seen that updating the master and the back-up in the wrong order would have gotten us into trouble—and merely looking at the faulty set of transformation rules would not necessarily reveal that there was any problem with them. Can we be sure there are no more risks hiding?

Second, the transformation might produce an annotated program that violate the rules for a well-annotated subject program, so that it would be rejected by the Gegen process or result in an erroneous $p_{gen}$. It might be that due to some mistake in the expression tranformation the transformed program contained an assignment of an $\mathtt{int}_d$ expression to an $\mathtt{int}_s$ variable. Or the transformation might introduce a (malign) non-local side effect under dynamic control.

One great virtue of the preprocessing approach to partially static data is that these two questions can be separated. Verifying the correctness of a Gegen transformation that went directly from partially static data to a completed $p_{gen}$ would be horribly complex and error-prone in itself. The use of a separate preprocessing phase supplies by itself a valid division of the task into manageable chunks.

Indeed, the semantic soundness of the transformation seems to be amenable to a mathematical proof. The nonacademic reader—if he is still following us—might appreciate the fact that we shall not have time to actually give such proof. The point is that one could construct it using standard techiques from formal programming language semantics.

Regarding the validity of the binding-time annotations of the transformed program, the *type correctness* issue is guaranteed by the transformation if the input program is type correct. This should be mathematically provable, too.

Fulfilling the restrictions about what can happen under dynamic control is more a matter of imposing similar conditions about the use of $\mathtt{int}_b$ in the input, however. Thus if we were to make a specializer supporting $\mathtt{int}_b$ we would need rules like

- *A side effect on a global* $\mathtt{int}_b$ *may not happen under dynamic control.*
- *A function returning* $\mathtt{int}_b$ *may not return under local dynamic control.*

and so on.

### 2.6.4 Lessons learned

As said at the beginning, the constructions in this section are hypothetical and not actually supported by C-Mix$_{/II}$. The reader may wish a concise statement about what we did actually learn.

The first lesson is the simple fact that “static” and “dynamic” do not in themselves constitute the entire range of possibilities how a partial evaluator could treat a value. No matter how trivial this fact seems to be it deserves to be emphasized. The most important questions when designing a partial evaluator *are*, “what should happen at specialization time? What should happen in the residual progam?” Thus it is all too easy to be trapped by the assumption that once one has devised a “static” way of handling a feature and a “dynamic” way of handling it, the useful possibilities for handling that feature have been exhausted.

The second valuable lesson is the general idea of using a separate transformation step between the binding-time analysis and the actual specialization. Such a step makes it possible to transparently synthesize complex binding-time types out of simpler ones: the BTA needs not be aware that these types are synthesized, and the specialization process needs not know they ever existed. This idea is not limited to partially static data. For example, some ways of supporting function pointers (see $\mathsf{FunPtr}_{\mathrm{dn}}$ in Box 3.7) lend themselves naturally to being implemented as a rewriting step.

The third outcome of our thought experiment is the concrete framework for a transformation that we presented in Figures 2.30 and 2.31. When one disregards the specific rules for dealing with lifts, the framework can be used completely generally for deriving transformations for data that somehow split into multiple components. The actual splitter phase in Chapter 5 builds directly on the principles presented here.

# Chapter 3

# Analysis of gegen requirements

In this chapter the techniques from Chapter 2 will be extended to cover the full Core C language that is C-Mix/II's internal representation of C programs. A description of Core C can be found in Appendix A. The reader is encouraged to consult this description, because the notation we use for types and expressions in this chapter has few points in common with the notation used in C program sources.

The primary difference between the final form of Chapter 2's example language and Core C is that Core C has a richer type system and operators to manipulate values of compound types. Because the hard parts about memoization and speculative specialization have already been adequately covered[1], we can stick to considering how simple assignments and expressions involving compond types should be treated.

We keep the idea from Chapter 2 that binding-time annotations consist of attaching descriptive suffixes to types of the original program's declarations and expressions. The matter gets more complex, of course, because in Core C types can be made up of other types, as in "Pointer(Enum)". There will be several ways of decorating Pointer and several ways of decorating Enum—but only in some combinations do they fit meaningfully together.

### Goals of this chapter

First, to develop possible **specialization variants** for each type constructor.

Second, to derive and describe the conditions for a set of binding-time annotations to be acceptable. These can be divided into

- **typing rules** which rule out, e.g., attempts to create a "dynamic pointer to static int" or to use a dynamic index expression to select an element from a statically indexed array. The common task of the typing rules is to make sure that the generating extension will itself be a legal program at all.

[1]It should be noted that the presence of pointers creates special problems for memoization, too. This report is not going to discuss these problems in detail except what is necessary for developing the binding-time analysis. The reader is referred to Andersen (1996) for that.

The typing rules are all local: they specify how a binding-time type may be assembled, and the relationship between the types meeting at an operator.

- **safety rules** such as the prohibition against doing side efffects on non-local static values under dynamic control. Their task is to make sure that the *residual program* becomes a legal program that behaves as specified by the subject program.

  The safety rules commonly depend on global properties of the subject program. They rely on the results of auxiliary analyses such as a pointer analysis.

The specialization variants and the typing rules will be developed in parallel, guided by the Core C type system. Due to the global nature of the safety rules they presented separately in Sections 3.4.3 and 3.4.4.

**Order of presentation**

We'll have to apologize in advance for the perhaps inconvenient order in which topics are presented in the following analysis. The matter at hand simply *is* complicated and full of circular references: some features of our treatment of data pointers, for examples, are best motivated and illuminated with examples involving structs. On the other hand, developing a theory for structs involves considering how the decisions will mix with the decisions for pointers.

One could solve that by deciding that pointers and structs together are a tightly integrated subject that will have to be treated together. But just as valid arguments could be made for lumping together pointers and arrays, and structs and unions, *et cetera ad absurdum*. The net effect would be that, after deciding to do everyting in parallel we would still need to solve the original problem of how to present this parallel reasoning to a reader who must, after all, read the text in *some* order.

In short, it appears that there is no absolutely convenient way of presenting the analysis. We have chosen to employ the principle that if we cannot be *didactically* consistent, we can at least be *formally* consistent.

Thus our strategy will be to first postulate (in Section 3.1) some abstract facts about the binding-time type system we are going to develop. Then (in Section 3.2) we can develop its individual components without resorting to forward references in the argument that the techniques we describe do actually work.

We shall ruthlessly expect the entire binding-time type system to be known in the examples that motivate and illuminate our choices. The reader might have to read the chapter more than once in order to appreciate it fully.

## 3.1 Preliminaries

This section defines some concepts that are useful in the following discussion of the full binding-time type system.

### 3.1.1 Notation for Core C types

We will use the following simplified notation for Core C types:

$$
\begin{array}{lcll}
\tau & ::= & \mathsf{Abstract} & \\
 & | & \mathsf{Primitive} & \\
 & | & \mathsf{Enum} & \\
 & | & \mathsf{Array}(\tau, e) & (e \text{ is the length expression}) \\
 & | & \mathsf{Pointer}(\tau) & \\
 & | & \mathsf{Struct}(\tau_1, ..., \tau_n) & \\
 & | & \mathsf{Union}(\tau_1, ..., \tau_n) & \\
 & | & \mathsf{FunPtr}(\tau_1, ..., \tau_n \to \tau) &
\end{array}
$$

The simplifications amount to ignoring type qualifiers, ignoring the need to declare structs, unions, and enums, and collapsing the FunPtr and Fun constructions of Core C.

The notation leaves no way of writing down recursive types, but it is to be understood that they may exist, e.g., as solutions to the equation

$$\tau_{42} = \mathsf{Struct}(\mathsf{Pointer}(\tau_{42}), \mathsf{Primitive})$$

The keywords Abstract, Primitive,... FunPtr are called **type constructor**s.

### 3.1.2 Structure of the binding-time type system

A **binding-time type** is a Core C types in which each constructor has been decorated with a suffix that selects a **specialization variant**. Each type constructor has its own set of possible suffixes. In Section 3.2 it will be described just which specialization variants are possible for each type constructor. The specialization variant is named by the combination of a constructor and and the suffix.

For example, "$..._{\mathrm{r}}$" and "$..._{\mathrm{d}}$" are among the possible suffixes for Pointer(we shall define what they mean in Section 3.2.3), and "$..._{\mathrm{d}}$" is also possible for Primitive. Thus "$\mathsf{Pointer}_{\mathrm{r}}$", "$\mathsf{Pointer}_{\mathrm{d}}$" and "$\mathsf{Primitive}_{\mathrm{d}}$" are names of specialization variants. It has no formal significance that $\mathsf{Pointer}_{\mathrm{d}}$ and $\mathsf{Primitive}_{\mathrm{d}}$ both have a "$..._{\mathrm{d}}$" in them, except as a mnemonic hint that they may have some properties in common.

Not every combination of specialization variants forms a valid binding-time type. $\mathsf{Pointer}_{\mathrm{r}}(\mathsf{Primitive}_{\mathrm{d}})$ and $\mathsf{Pointer}_{\mathrm{d}}(\mathsf{Primitive}_{\mathrm{d}})$ happen to be **well-formed** binding-time types. So is $\mathsf{Pointer}_{\mathrm{r}}(\mathsf{Pointer}_{\mathrm{d}}(\mathsf{Primitive}_{\mathrm{d}}))$.

But $\mathsf{Pointer}_{\mathrm{d}}(\mathsf{Pointer}_{\mathrm{r}}(\mathsf{Primitive}_{\mathrm{d}}))$ is *not* well-formed: the binding-time type $\mathsf{Pointer}_{\mathrm{r}}(\mathsf{Primitive}_{\mathrm{d}})$ is not a valid argument to the specialization variant $\mathsf{Pointer}_{\mathrm{d}}$. As we develop the binding-time type system we shall also define the conditions that determine whether a binding-time type is well-formed.

The **erasure** of a binding-time type is the ordinary Core C type that results from removing all of the binding-time annotations.

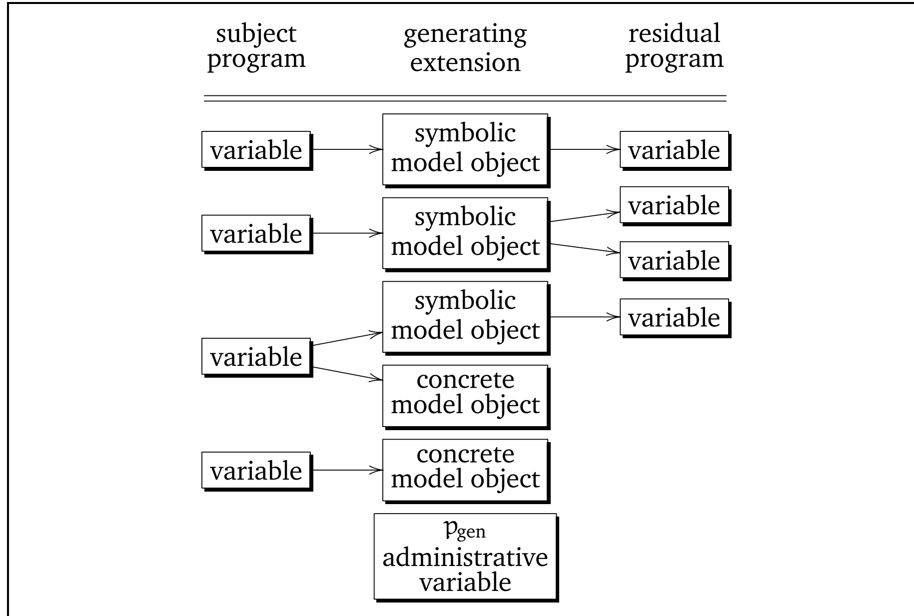



*Figure 3.1: The role of model objects. All model objects are variables in $p_{gen}$, but not all of $p_{gen}$'s variables are model objects. The model objects are those that correspond to variables in the subject program. The symbolic model objects represent variables in $p_{res}$, whereas the concrete model objects leave no mark there.*

### 3.1.3 Model objects in the generating extension

An object[2] in the subject program is modelled by one or more objects in $p_{gen}$. By design, each of these **model object**s must be either

**symbolic** where the model object contains the names of one or more objects in the residual program. This kind of model object is never changed once it has been properly initialized. Thus they need not be memoized. Or,

**concrete** which means that it contains actual data that correspond to the (static) value of the subject object. When something is assigned to the object in the subject program, the entire contents of the model object are replaced. Concrete model objects need to be memoized (e.g., their values should be stored in pending list entries and reuse-pool items as described in Sections 2.2.2 and 2.4.2).[3]

There must not be any model objects that is both symbolic and concrete—i.e., where assignment to the subject object involves replacing parts of the model object only.

[2]"Object" is the term used by the C standard to mean the place in memory where a value of some type is stored, e.g., a variable, a piece of heap-allocated memory, or member of an array or struct object. This report adopts that usage, which is not to be confused with the "objects" in object-oriented programming".

[3]Concrete model objects contain static values, and symbolic model objects model dynamic objects. We do not want to call them "static model objects" and "dynamic model objects", because the latter would feel contradictory to the fact that both kinds of model objects actually exist in the generating extension rather than in the residual program.

This design choice makes it possible to implement assignments as block copies in the generating extension. If some partially static type $\tau$ was implemented in the generating extension as a struct with a static member and a dynamic member, and one made two arrays of 100 $\tau$s each, assigning one array's contents to another[4] would imply copying each static member separately, because the dynamic members must not be touched.

**Dynamic, static, and partially static**

We call a binding-time type

**dynamic** if it has only symbolic model objects;

**static** if it has only concrete model objects; and

**partially static** if it needs concrete model objects as well as symbolic model objects.

### 3.1.4 The signature of a binding-time type

The **signature** of a binding-time type $\tau$ describes the number and kinds of model objects that are used to model it in $p_{gen}$. The signature is a pair of two integers $(|\tau|_{\mathbf{Y}}, |\tau|_{\mathbf{O}})$.

$|\tau|_{\mathbf{Y}}$ encodes the use of symbolic model objects. There is never more than one of these, but it might be used in two different ways:

$|\tau|_{\mathbf{Y}} = 0$ means that $\tau$ has no symbolic model objects.

$|\tau|_{\mathbf{Y}} = 1$ means that $\tau$ has a single symbolic model object that corresponds to a single object in $p_{res}$.

$|\tau|_{\mathbf{Y}} = -1$ means that $\tau$ has a single symbolic model object that contains the names of multiple unrelated $p_{res}$ objects. The main example of this is a statically indexed array of dynamic data.

Observe that these values have been chosen such that $|\tau|_{\mathbf{Y}}^2$ is the number of actual symbolic model objects in $p_{gen}$.

$|\tau|_{\mathbf{O}}$ is more straightforward: it is simply the number of concrete model objects that are used to model a single $\tau$. For most types $|\tau|_{\mathbf{O}}$ is 0 or 1, but $|\tau|_{\mathbf{O}} = 2$ arises for pointers to partially static data; they are modelled by a pointer to each of the partially static object's model objects. $|\tau|_{\mathbf{O}}$ is never greater than 2.

Of the 9 possible combinations of $|\tau|_{\mathbf{O}}$ and $|\tau|_{\mathbf{Y}}$, only the 6 ones where $|\tau|_{\mathbf{Y}}^2 + |\tau|_{\mathbf{O}}$ is 1 or 2 are actually used. Examples of binding-time types with each of these signatures are:

| | $\lvert\tau\rvert_{\mathbf{O}} = 0$ | $\lvert\tau\rvert_{\mathbf{O}} = 1$ | $\lvert\tau\rvert_{\mathbf{O}} = 2$ |
|---|---|---|---|
| $\lvert\tau\rvert_{\mathbf{Y}} = 0$ | | $\mathsf{Primitive}_s$ | $\mathsf{Pointer}_s(\tau_T)$ |
| $\lvert\tau\rvert_{\mathbf{Y}} = 1$ | $\mathsf{Primitive}_d$ | $\tau_T$ | |
| $\lvert\tau\rvert_{\mathbf{Y}} = -1$ | $\mathsf{Array}_s(\mathsf{Primitive}_d)$ | $\mathsf{Array}_s(\tau_T)$ | |

where $\tau_T$ is a partially static type such as $\mathsf{Struct}_p(\mathsf{Primitive}_d, \mathsf{Primitive}_s)$.

Observe that a binding-time type $\tau$ is *static* if $|\tau|_{\mathbf{Y}} = 0$, *dynamic* if $|\tau|_{\mathbf{O}} = 0$ and *partially static* if $|\tau|_{\mathbf{O}}$ and $|\tau|_{\mathbf{Y}}$ are both nonzero.

[4] One cannot directly assign arrays to one another in C, but it is possible by wrapping them into structures.

**Basic and compound binding-time types**

Some of the binding-time types selected by the BTA are not supported in the final step $p_{gen}$ construction phase of C-Mix/II. They are eliminated in an intermediate transformation phase called the **splitter phase** which reduces operations on them to operations on actually supported types.

We call binding-time types that we allow to be used in the splitter's output **basic** binding-time types. They have the common property that they have only one model object each. That is, their signature is always $(1,0)$, $(-1,0)$, or $(0,1)$.

Binding-time types that are not basic are called **compound**. They include all partially static binding-time types, as well as some other types that are not partially static, such as pointers to partially static types.

The splitter phase, akin to the transformation we described in Section 2.6.2, is developed and specified in Chapter 5.

Actually it is not implemented in C-Mix/II yet; instead the current binding-time analysis has been restricted to produce only basic binding-time types. Implementing it is high on our to-do list, however, so by the time this report reaches a wider audience than C-Mix/II project members and the external evaluator, implementation of the described compound binding-time types should be well on its way.

### 3.1.5 Faithful binding-time types

A **faithful** binding-time type is one whose representation in either the generating extension or the residual program is completely equal to its erasure. We talk about *statically faithful* or *dynamically faithful* binding-time types, according to which of the two possibilities is the case.

Faithful types are needed when communicating with the world outside of the specialization process. For example, the arguments to an external function had better be faithful. The erasure of the binding-time type is the type the argument had in the declaration that C-Mix/II read in originally, so that is the type the external code is going to expect to get. It cannot handle a pointer to `Code` if it expects a pointer to `int`.

Faithfulness is a simple syntactical property of binding-time types. Along with the development of specialization variants for each type constructor we shall also explain how to form and recognize faithful binding-time types.

In formulas we use $\mathbb{F}$ for the set of all faithful binding-time types. $\mathbb{F}_d$ and $\mathbb{F}_s$ are the sets of dynamically and statically faithful binding-time types, respectively.

A binding-time type can be dynamic without being faithful. An example is a dynamic pointer to a partially static type, as described on page 70.

### 3.1.6 Binding-time types of expressions

The preceding discussion of binding-time types has focused on the use of binding-time types to declare variables. We shall briefly describe the role of binding time annotations on the types of expressions.

The handling of an expression depends on the signature of its binding-time type $\tau$:

$(1, 0)$: A residual expression is constructed whose type is the same as the type of the $p_{res}$ variable that would represent a variable annotated to have binding-time type $\tau$.

$(-1, 0)$: The typing rules should guarantee that no expressions have binding-time types with this signature.

$(0, 1)$: The expression is represented by an expression in $p_{gen}$ whose type is that of $\tau$s (concrete) model object.

$(1, 1), (-1, 1), (0, 2)$: These signatures are only used for compound binding-time types; they are reduced by the splitter phase in a manner similar to the transformation presented in Section 2.6.2. Generally the expression is mapped into one or two expressions with basic binding-time types.

## 3.2 Developing the type system

With basic terminology and concepts such as the signature of a binding-time type defined, we can go on to systematically developing appropriate variants of each type constructor.

Each subsection will be organized according to this general agenda:

- Describe **basic (specialization) variants**, the ones needed to build basic binding-time types.

  Typically the basic variants are called $\mathsf{Someconstructor}_d$ and $\mathsf{Someconstructor}_s$. An important consideration when selecting them is that it should be possible to annotate any type to be faithful, statically as well as dynamically.
- Investigate the possible lifts between the basic variants.
- Develop methods for handling the operations particular to the type constructor using the basic variants.
- Develop variants for use in compound binding-time types.

  Compound variants can either be simple generalizations of basic variants (in which case they share the name of the basic variant; an example is $\mathsf{Pointer}_s$) or separately named (an example of this is $\mathsf{Struct}_p$).

  The handling of lifts and operators for compound variants can usually be extrapolated from the behaviour of the basic variants they reduce to. It is usually summarized, but only treated extensively where the extrapolation is nonobvious.

Examples will be given for each constructor of how the $p_{gen}$ and $p_{res}$ representation of different binding-time types using that constructor look. Many of the examples will involve an example partially static type "$\tau_T$" which is a structure containing a dynamic `int` member called `bar` and a static `int` member called `baz`.

The full intricacies of $\tau_T$ will be explained in Section 3.2.5; for now the reader will just need to know that the presence of $\tau_T$ in the annotated subject program causes two structs to be declared in $p_{gen}$ and one struct to be declared in $p_{res}$, as shown in Figure 3.2. `struct Ty` is its symbolic model object type. Its `this` member names a `struct Tr` in the residual program. The `bar` member of `struct Ty` always contains the same name as the `this` member, but suffixed

| Annotated subject type | Text in $p_{gen}$ | Text in $p_{res}$ |
|---|---|---|
| struct$_p$ T {<br>  int$_d$ bar ;<br>  int$_s$ baz ;<br>} | struct Ty {<br>  Code this ;<br>  Code bar ;<br>};<br>struct To {<br>  int baz ;<br>}; | struct Tr {<br>  int bar ;<br>}; |

*Figure 3.2: Examples of a partially static type $\tau_T$.*

with "`.bar`" such that it names the `bar` member of the residual struct. `struct To` is the concrete model object for $\tau_T$.

### 3.2.1 Primitive and abstract types

Core C's Primitive type is used for C's built-in types such as `int`, `unsigned short`, or `float`. Abstract is used for `void` and for types explicitly declared as "abstract", such as "`FILE`" if the program includes the `<stdio.h>` header.

The common feature for these types is that as far as C-Mix$_{II}$ is concerned they have no internal structure and their values have no particular meaning: they are atomic. Primitive types have the special property that there is a C source syntax for constants of those types. This means that it is possible to lift values of Primitive types from $p_{gen}$ into $p_{res}$.

The two variants of `int` we saw in Chapter 2 work for Primitive types in general:

Primitive$_d$: Model each Primitive in $p_{gen}$ by a `Code` that contains the name of a $p_{res}$ Primitive which contains the actual value. The signature of a Primitive$_d$ is $(1, 0)$.

Primitive$_s$: Model each Primitive in $p_{gen}$ by itself. Thus the signature of a Primitive$_s$ is $(0, 1)$.

Both variants are faithful, Primitive$_d$ dynamically and Primitive$_d$ statically.

The choices for Abstract are Abstract$_s$ and Abstract$_d$, exactly as for Primitive.

#### Lifts

One can lift from Primitive$_s$ to Primitive$_d$; this is the prototypical lift also discussed in Chapter 2. This is the only lift possible for Primitive.

Because Abstract types have no (known) constant syntax they cannot be lifted: what should one write in $p_{res}$ at the position of a lift?

#### Operators

The operators related especially to Primitive are the Binary and Unary operators. Not all of the Primitive types can meaningfully be used with each particular operator, but at the time of the binding-time analysis, consistent usage has been

| foo annotated as | Text in $p_{gen}$ | Text in $p_{res}$ |
|---|---|---|
| $\mathsf{Primitive_d}$ | `Code foo ;` | `int foo ;` |
| $\mathsf{Primitive_s}$ | `int foo ;` | |
| $\mathsf{Abstract_d}$ | `Code foo ;` | `FILE foo ;` |
| $\mathsf{Abstract_s}$ | `FILE foo ;` | |
| $\mathsf{Enum_d}$ | `Code foo ;` | `enum T {...};`<br>`enum T foo ;` |
| $\mathsf{Enum_s}$ | `enum T {...};`<br>`enum T foo ;` | |

*Figure 3.3: Examples of Gegen treatment of primitive, abstract, and enum types*

enforced by a normal C type checker. Also, as part of the translation to Core C it has been made sure that the short-circuiting behaviour of `&&` and `||` has been converted to explicit control flow where relevant.

Thus we can treat these operators completely uniformly: each can either be dynamic (work on $\mathsf{Primitive_d}$ arguments and give a $\mathsf{Primitive_d}$ result) or static (work on $\mathsf{Primitive_s}$ arguments and give a $\mathsf{Primitive_s}$ result).

There are no operators that relate especially to Abstract types. They can be pointed to, assigned between variables and passed to and from functions. All of this is handled the same way as for every other type.

**Compound variants**

There are no compound variants of Primitive and Abstract.

The "backed-up ints" we discussed in Section 2.6 was a thought experiment only, and we do not think it useful enough to warrant implementation in the real C-Mix/II. There would not be any fundamental problem in adding a "$\mathsf{Primitive_b}$" to the binding-time type system, though.

### 3.2.2 Enumerated types

In C, an `enum` type is really just an alias for an appropriate (selected by the compiler) integral type. The main reason it has its own constructor in Core C is that a link to the `enum` type declaration has to be preserved in order to be able to declare variables of the type.

The possible variants are thus completely parallel to those of Primitive: one can have $\mathsf{Enum_s}$ and $\mathsf{Enum_d}$.

There is one caveat, however: the `enum` declaration may specify explicit `int` values for one or more of the enumeration constants. If the expression that specifies one of these values is a $\mathsf{Primitive_d}$ rather than $\mathsf{Primitive_s}$[5] the `enum`

[5]This could be the case, e.g., if the expression contains a Sizeof, because `sizeof` expressions are considered to be a sign of "dirty tricks" in the code and always residualized.

type cannot be declared properly in $\mathfrak{p}_{gen}$, so the only choice for that particular enumeration becomes $\mathsf{Enum}_d$.

$\mathsf{Enum}_d$ is always possible even when the explicit-value expressions are $\mathsf{Primitive}_s$s: these are lifted to $\mathsf{Primitive}_d$s when constructing the `enum` declaration for $\mathfrak{p}_{res}$.

**Lifts**

We do not allow lifts for Enum types (but see Box 3.1).

**Operators**

The only "operator" that is particularly relevant to Enum is EnumConst.

`enum`s are such low-class citizens in C that not even their own symbolic constants have `enum` type according to the rules in the standard: they have type `int` (ANSI 1990, clause 6.1.3.3)!

Nevertheless, that `int` could be a $\mathsf{Primitive}_s$ or a $\mathsf{Primitive}_d$, corresponding to the appearance of the symbolic constant as an identifier in $\mathfrak{p}_{gen}$ or in $\mathfrak{p}_{res}$. The former choice is only possible when the `enum` can be declared in $\mathfrak{p}_{gen}$ at all, that is, when $\mathsf{Enum}_s$ would be possible.

**Compound variants**

There are no compound variants of Enum.

### 3.2.3 Data pointers

(Function pointers are treated separately, see Section 3.2.7).

The basic variants are $\mathsf{Pointer}_d$, $\mathsf{Pointer}_s$, and $\mathsf{Pointer}_r$:

$\mathsf{Pointer}_d(\tau)$: In $\mathfrak{p}_{gen}$ there will be a `Code` containing the name of a pointer in $\mathfrak{p}_{res}$ that points to $\tau$'s residual representation. This is only possible if $\tau$ *has* a single residual representation that can be pointed to, so the signature of $\tau$ must be $(1, 0)$.

The signature of $\mathsf{Pointer}_d(\tau)$ is also $(1, 0)$. It is a dynamically faithful type if $\tau$ is.

$\mathsf{Pointer}_s(\tau)$: Model the pointer in $\mathfrak{p}_{gen}$ simply as a pointer to $\tau$'s model object. That model object can be either symbolic or concrete; the pointer itself is always a concrete model object.

> Box 3.1—POSSIBLE EXTENSION:
> LIFTS FOR ENUMS
>
> In theory there is nothing that prevents an `enum` from being lifted as if it were an `int`. The Right Thing to do, though, would clearly be to make sure that when an $\mathsf{Enum}_s$ was lifted the proper symbolic constant was written into $\mathfrak{p}_{res}$.
>
> Nobody has yet come around to writing the code to support that kind of lifting, so for the time being C-Mix/II does not support lifts for Enum types. It would be easy to add approprate support for it without affecting much else of the system.

| `foo` annotated as | Text in $p_{gen}$ | Text in $p_{res}$ |
|---|---|---|
| $\mathsf{Pointer}_d(\mathsf{Primitive}_d)$ | `Code foo ;` | `int *foo ;` |
| $\mathsf{Pointer}_s(\mathsf{Primitive}_s)$ | `int *foo ;` | |
| $\mathsf{Pointer}_s(\mathsf{Primitive}_d)$ | `Code *foo ;` | |
| $\mathsf{Pointer}_d(\tau_T)$ | `Code foo ;` | `struct Td *foo ;` |
| $\mathsf{Pointer}_p(\tau_T)$ | `Code fooA ;` `struct To *fooB ;` | `struct Td *fooA ;` |
| $\mathsf{Pointer}_s(\tau_T)$ | `struct Ty *fooA ;` `struct To *fooB ;` | |
| $\mathsf{Pointer}_s(\mathsf{Pointer}_s(\tau_T))$ | `struct Ty **fooA ;` `struct To **fooB ;` | |
| $\mathsf{Pointer}_s(\mathsf{Pointer}_d(\tau_T))$ | `Code *foo ;` | |
| $\mathsf{Pointer}_d(\mathsf{Pointer}_d(\tau_T))$ | `Code foo ;` | `struct Tr **foo ;` |

*Figure 3.4: Examples of Gegen treatment of pointers.* $\mathsf{Pointer}_r$ *can be subsituted for* $\mathsf{Pointer}_s$ *in all of the examples.*

Thus the signature of $\mathsf{Pointer}_s(\tau)$ is $(0, 1)$. It is a statically faithful type and if $\tau$ is.

$\mathsf{Pointer}_r(\tau)$: This is the same as $\mathsf{Pointer}_s$ except that it does not allow pointer arithmetic and is never considered faithful. Box 3.2 explains why this variant is necessary.

**Lifts**

Lifting $\mathsf{Pointer}_d$ to any of the two static variants is of course impossible.

One may lift $\mathsf{Pointer}_r(\tau)$ to $\mathsf{Pointer}_s(\tau)$ at any time; because the representations are identical nothing actually happens in $p_{gen}$.

It takes more care to lift a $\mathsf{Pointer}_r(\tau)$ to $\mathsf{Pointer}_d(\tau)$. Naturally, $\mathsf{Pointer}_d(\tau)$ needs to be well-formed. If it is, it means that $\tau$ has a symbolic model object that contains the name of the residual representation of the object the pointer points to. Through the $\mathsf{Pointer}_r$ we can get access to this name, and if we prefix it with the "`&`" operator we get a residual expression that evaluates to just the pointer to $\tau$'s residual part that is the residual form of $\mathsf{Pointer}_d(\tau)$.

Notice that just where we find the name to use depends on $\tau$. For example, if $\tau$ is $\mathsf{Primitive}_d$, its symbolic model object is a `Code` which itself is the name we need. But if $\tau$ is the $\tau_T$ type we use in the example figures, the symbolic model object is a `struct Ty` whose `this` member is the `Code` that contains the desired name. These differences can be resolved at Gegen time, however.

The pointer we try to lift may be `NULL`. We have to check for this as a special case and, if so, insert a null pointer constant into $p_{res}$.

### Box 3.2—RATIONALE FOR $\mathsf{Pointer}_r$

The reason for discriminating between $\mathsf{Pointer}_s$ and $\mathsf{Pointer}_r$ has to do with pointer arithmetic. The reader may be excused for thinking that pointer arithmetic is primarily a concern in connection with arrays. However, it happens that the hardest problems with pointer arithmetic arise when there is no array involved.

The problem is that the language standard specifies (ANSI 1990, clause 6.3.6, 4th paragraph of the "semantics" section) that a pointer to *any object*, even objects that are not array elements, may be treated like a pointer to the first element of a one-element array. In particular, it is allowed to move the pointer "one past the last (here: only) element of the array" and back again. This means that code like this is legal C and should result in accessing `foo`:

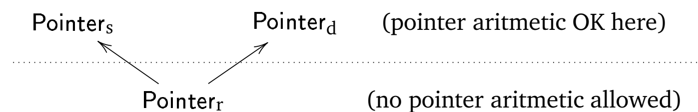
```
int foo, *p, *q ;
p = &foo + 1 ;
q = p ;
*(q-1) = 42 ;
```

Now, suppose we did not have $\mathsf{Pointer}_r$. It would then be well-typed to let `foo` be a $\mathsf{Primitive}_d$, `p` be a $\mathsf{Pointer}_s(\mathsf{Primitive}_d)$, and `q` be a $\mathsf{Pointer}_d(\mathsf{Primitive}_d)$.

Thus `p` would—in $p_{gen}$—point one past the `Code` that contained the name of `foo`'s residual. In the second assignment we would need to be able to lift that pointer into a residual expression that happened to be a pointer one past `foo`'s residual.

Inventing a lifting mechanism that can do *that* and also work in all imagineable variants of the situation seems to be a decidedly non-trivial task. It would probably need to introduce unpleasant discontinuities in the mapping from binding-time types to $p_{gen}$ representation and in any case add significant complexity to the $p_{gen}$ construction.

The solution we chose instead is to make sure that the example is not well-annotated, so that the BTA will chose to perform the lift before the first pointer arithmetic instead. The basic idea is to only allow a static pointer to be lifted into $\mathsf{Pointer}_d$ if we know that no pointer arithmetic has been performed on it:




A pointer starts out as a $\mathsf{Pointer}_r$. It can be manipulated and stored as such in $p_{res}$ as long as no pointer arithmetic is performed on it.

Before being subject to pointer arithmetic, though, it has to be lifted to either $\mathsf{Pointer}_d$ or $\mathsf{Pointer}_s$ (the "lift" from $\mathsf{Pointer}_r$ to $\mathsf{Pointer}_s$ simply consists of calling the same static pointer something else). Once the pointer arrives at $\mathsf{Pointer}_d$ or $\mathsf{Pointer}_s$, pointer arithmetic is allowed, but no further lifting is possible.

A more serious problem is that the name we use to lift the pointer may not be in scope at the point in the residual program where we insert it. We thus only allow pointer lifts when we can be sure that the pointed-to object is in scope. That is not a typing rule but a safety rule (see page 57), so we'll defer a detailed investigation until Section 3.4.4.

It is not generally allowed to lift $\mathsf{Pointer}_s$ to $\mathsf{Pointer}_d$, because a $\mathsf{Pointer}_s$ may have been subject to pointer arithmetic and so does not necessarily point to a model object at all.

However, some operations on $\mathsf{Pointer}_s(\tau)$ with a dynamic $\tau$ (the main example being when we want to read or write to the pointed-to value) are most conveniently expressed as giving a "special permission" to lift it to $\mathsf{Pointer}_d$ nevertheless and then using that for the operation.

We would prefer not to actually call this case of lifting by special permission a "lift", because it would ruin the general idea that lifts can happen anywhere in an expression. Instead we invent the term **carry**[6] to describe the conversion that may only happpen at these specific places.

We still need to make sure that the pointer we carry has not been moved to point behind an object instead of to it. We can do that by only allowing carries in very specific contexts: Carries are allowed only for the immediate operands on the DeRef, Struct, and Array Core C operators, and for the "target" expression in a Core C statement.

In these contexts, the translation from C to Core C only uses pointer expressions that represent lvalue expressions in the subject program. The C standard only allows the programmer to use lvalues that actually point to objects, so if we assume that the subject program is valid according to the C standard we can infer that an carried pointer always points to an object even if pointer arithmetic has been performed on it.

One special case is if a Member operator is the translation of C's "->" operator. In this case the operand does not originate in an lvalue, but the standard nevertheless requires that it (qua an operand to "->" points to an object).

### Pointer creation

The fundamental pointer creation operator is $\mathsf{Var}(\tau, d)$: taking the address of the variable d. This is easy to implement.

For $\mathsf{Var}(\mathsf{Pointer}_r(\tau), d)$ or $\mathsf{Var}(\mathsf{Pointer}_s(\tau), d)$ one simply takes the address of d's model object(s).

For $\mathsf{Var}(\mathsf{Pointer}_d(\tau), d)$ one finds the residual name in the symbolic model object for d and prepends an "&" to it. This is the same as would happen if one first computed the address as a $\mathsf{Pointer}_r$ and then lifted it to $\mathsf{Pointer}_d$—which is also how it is actually implemented i Gegen.

Another form of pointer creation is the Null operator, which simply creates a null pointer constant. We allow it to create any well-formed Pointer type.

The only Pointer variant that it is not totally obvious that Null should be able to create is $\mathsf{Pointer}_r$. But as we know how to lift a null pointer, that is no problem either.

[6]This word is inspired by the diagram in Box 3.2. Note that proper lifts actually moves the type upwards in the diagram, whereas $\mathsf{Pointer}_s$ and $\mathsf{Pointer}_d$ are at the same level. The closest physical equivalent to lifting something without changing its altitude must be to *carry* it somewhere.

| Pointer type | $\|\tau\|_Y$ | $\|\tau\|_O$ | How to access through it |
|---|---|---|---|
| $\mathsf{Pointer}_d(\tau)$ | 1 | 0 | Trivial: one just applies the "*" operator to the residual $\mathsf{Pointer}_d$ expression to get a residual lval expression that can be written to or read from. |
| $\mathsf{Pointer}_{r/s}(\tau)$ | 1 | 0 | We allow the pointer to be carried into a $\mathsf{Pointer}_d$ and access the pointed-to value using the standard method for $\mathsf{Pointer}_d$s. |
| $\mathsf{Pointer}_{r/s}(\tau)$ | $-1$ | 0 | Impossible: it is of no use to access the model object itself, and there's no single object to access dynamically. |
| $\mathsf{Pointer}_{r/s}(\tau)$ | 0 | 1 | Also trivial. The model object that the pointer points to is precisely what we want to read or write. |

*Table 3.1: Access through different basic pointer binding-time types*

### Pointer use

There are two fundamental ways to eventually use a pointer

- Reading the value pointed to by the pointer, using the Core C DeRef operator.
- Writing a new value to the object pointed to. In Core C this only happens when the pointer is the value of the "target" expression of a statement.

These two uses are similar enough to warrant common treatment, and we shall describe both as **access through** the pointer.

Table 3.1 discusses how to access through differently annotated pointer types.

### Pointer conversion operators

The Array and Member operators are used to convert pointers to aggregates into pointers to component types. They are described in Sections 3.2.4 on arrays and 3.2.5 on structs, respectively.

### Pointer arithmetic

It is fairly obvious how to implement the PtrArith operator for $\mathsf{Pointer}_d$ and $\mathsf{Pointer}_s$: We simply do arithmetic on the pointer's residual in the $\mathsf{Pointer}_d$ case or on the model object (which is itself a pointer) in the $\mathsf{Pointer}_s$ case. The integral operand must be a $\mathsf{Primitive}_d$ respectively $\mathsf{Primitive}_s$, but only the latter is a real restriction because we can always arrive at at $\mathsf{Primitive}_d$ by lifting.

It is even easier with $\mathsf{Pointer}_r$: Here, pointer arithmetic is not allowed at all.

### Pointer comparison

The Core C PtrCmp operator is used for comparing two pointers for equality or find out their relative difference if both point into the same array (this includes

subtraction of pointers as well as inequalities involving pointers).

The two pointers must, after possible lifting, have the same binding-time type. The result is $\mathsf{Primitive}_d$ if the pointers are $\mathsf{Pointer}_d$, and $\mathsf{Primitive}_s$ if they are $\mathsf{Pointer}_r$ or $\mathsf{Pointer}_s$.

### Compound variants of Pointer

When pointing to something which has only one model object, the basic variants of Pointer fully suffice. Now we'll study ways to point to things with two model objects. There are generalizations of the basic variants and a new variant, $\mathsf{Pointer}_p$, which is only used in compound binding-time types.

$\mathsf{Pointer}_d(\tau)$ can be generalized to the case where $\tau$ has signature $(1, 1)$, simply as a single $\mathsf{Pointer}_d$ that points to the dynamic part of $\tau$.

Access through such a generalized $\mathsf{Pointer}_d$ is not possible: because we have lost track of the static part of $\tau$ we can only read a partial value which cannot be used for anything sensible.

The Array and DeRef operators are still available, however; thus dynamic subobjects of the pointed-to objects *can* be accessed. Pointer arithmetic and comparison are also possible.

$\mathsf{Pointer}_p(\tau)$ is a **partially static pointer**, only possible for $\tau$s with signature $(1, 1)$. It consists of a $\mathsf{Pointer}_s$ to the static part of $\tau$ and a $\mathsf{Pointer}_d$ to the dynamic part of $\tau$. The signature of a $\mathsf{Pointer}_p$ is thus itself $(1, 1)$.

A $\mathsf{Pointer}_p$ can be lifted at any time to a $\mathsf{Pointer}_d$ simply by throwing the $\mathsf{Pointer}_s$ half away. This may be neccessary if one needs to do pointer arithmetic with dynamic offsets or to store the pointer in a variable that must be dynamic (because of a non-local side effect under dynamic control).

Pointer arithmetic is possible with a $\mathsf{Primitive}_s$ offset; the operation is done on each part of the pointer separately, with the offset lifted to $\mathsf{Primitive}_d$ for the $\mathsf{Pointer}_d$ part.

Two $\mathsf{Pointer}_p$s can be compared by comparing their theie static parts, producing a $\mathsf{Primitive}_s$ result. Similarly a $\mathsf{Pointer}_p$ can be compared to a $\mathsf{Pointer}_s$.

$\mathsf{Pointer}_s(\tau)$ and $\mathsf{Pointer}_r(\tau)$ can be generalized to any $\tau$s with two model objects simply by using two static pointers in parallel. Then the signature of the pointer type becomes $(0, 2)$.

When $\tau$ has signature $(1, 1)$ and hence $\mathsf{Pointer}_d$ and $\mathsf{Pointer}_p$ are also possible, a generalized $\mathsf{Pointer}_r$ can be lifted to one of them by lifting the pointer that points to the dynamic part. It is required that the pointer can only point to variables in scope.

The similar conversion from $\mathsf{Pointer}_s$ is a carry, just like in the basic case.

Table 3.2 discusses when access through the compound Pointer binding-time types can be done.

Note when comparing the possibilities for $\mathsf{Pointer}_s$ (and $\mathsf{Pointer}_r$) in Tables 3.1 and 3.2 that it happens that the $|\tau|_Y$ coordinate alone fully determines when access through a $\mathsf{Pointer}_s$ is possible.

| Pointer type | $\lvert\tau\rvert_Y$ | $\lvert\tau\rvert_O$ | How to access through it |
|---|---|---|---|
| $\mathsf{Pointer}_d(\tau)$ | 1 | 1 | Impossible. |
| $\mathsf{Pointer}_p(\tau)$ | 1 | 1 | We can simply access each part of the pointed-to object by the two parts of the $\mathsf{Pointer}_p$. |
| $\mathsf{Pointer}_{s/r}$ | 1 | 1 | Allowed if known to point to objects in scope. Implemented with a carry to $\mathsf{Pointer}_p$. |
| $\mathsf{Pointer}_{s/r}$ | $-1$ | 1 | Never allowed, for the same reasons that $(-1,0)$ is disallowed in Table 3.1. |
| $\mathsf{Pointer}_{s/r}$ | 0 | 2 | Always allowed, done pairwise. |

*Table 3.2: Access through different compound pointer binding-time types*

| Ptr-to-$\tau$ variant | $\mathsf{Pointer}_s$ | $\mathsf{Pointer}_r$ | $\mathsf{Pointer}_p$ | $\mathsf{Pointer}_d$ |
|---|---|---|---|---|
| Possible when | always | always | $\lvert\tau\rvert_Y = 1$ $\lvert\tau\rvert_O = 1$ | $\lvert\tau\rvert_Y = 1$ |
| Signature | $(0, \lvert\tau\rvert_Y^2 + \lvert\tau\rvert_O)$ | $(0, \lvert\tau\rvert_Y^2 + \lvert\tau\rvert_O)$ | $(1,1)$ | $(1,0)$ |
| Lift to $\mathsf{Pointer}_r$ | | always | no | no |
| Lift to $\mathsf{Pointer}_s$ | no | | no | no |
| Lift to $\mathsf{Pointer}_p$ | is a carry | when in scope | | no |
| Lift to $\mathsf{Pointer}_d$ | is a carry | when in scope | always | |
| Create with Var | yes | yes | yes | yes |
| Create with Null | yes | yes | yes | yes |
| Acess through | when $\lvert\tau\rvert_Y = 0$ or via carry | when $\lvert\tau\rvert_Y = 0$ or via lift | always | when $\lvert\tau\rvert_O = 0$ |
| Ptr arithmetic | $\mathsf{Primitive}_s$ | no | $\mathsf{Primitive}_s$ | $\mathsf{Primitive}_d$ |

*Table 3.3: Summary of* Pointer *variants*

**Summary**

Table 3.3 shows a summary of the attributes of the different Pointer variants.

### 3.2.4 Arrays

The most immediately occuring idea is simply to apply the array type construction at either the $p_{res}$ or the $p_{gen}$ stage:

$\mathsf{Array}_d(\tau, e)$, called a **dynamically indexed array**. It is implemented in $p_{res}$ simply by an array of $\tau$'s residual representation. In $p_{gen}$ there is a single `Code` containing the name of this array. There are no model objects for the element type $\tau$ itself in $p_{gen}$.

For this to be possible, $\tau$ must have signature $(1,0)$: the residual of each $\tau$ must be single objects, so they can be used as elements in the array.

As can be gathered from this description, $\mathsf{Array}(\tau,e)$ itself also has signature $(1,0)$. It is a dynamically faithful type if and only if $\tau$ is.

$\mathsf{Array}_s(\tau, e)$, called a **statically indexed array**. It is modelled in $p_{gen}$ simply as an array of model objects for $\tau$. If $\tau$ is static this is simple; the signature of $\mathsf{Array}_s(\tau, e)$ is then $(0, 1)$. But if $\tau$ is dynamic the consequence is that $p_{res}$ contains several independent instances of $\tau$'s residual. Thus the signature of $\mathsf{Array}_d(\tau, e)$ is $(-1, 0)$ if $\tau$'s signature is either $(-1, 0)$ or $(1, 0)$.

$\mathsf{Array}_s(\tau)$ is a statically faithful type if $\tau$ is.

For dynamic element types, using $\mathsf{Array}_s$ is to be preferred over $\mathsf{Array}_d$ if possible. Consider, e.g., the possible residual program fragments

```
int x[3] ;                        int x0, x1, x2;
/* ... */                         /* ... */
x[0] = x[1] + x[2] ;              x0 = x1 + x2 ;
```

We expect that programs like the right one (specialized with $\mathsf{Array}_s$) to be more efficient than the left one (specialized with $\mathsf{Array}_d$).[7]

The most immediate difference is that index calculations have been eliminated in the right version. However, because the indices actually used are constants (if they were not, $\mathsf{Array}_s$ would not be possible at all) the compiler can be expected to precompute the addresses of the array elements just as well has had they been separate variables.

The *real* advantage of the $\mathsf{Array}_s$ approach is that the use of separate variables makes it easier for the compiler to assess the risk of aliasing between the (former) array elements. Indeed, an optimizing compiler would most probably try to keep `x0`, `x1`, and `x2` in registers, while we know of no compiler that dares try to assign registers to genuine array elements.

**Lifts**

C's rules about "pointer decay" has the effect that in practise no *expressions* in a Core C program can have type Array(...). The purpose of lifts is to be able to use an expression of one binding-time type in contexts where a "sibling" binding-time type is expected, so lifts for Arrays are a non-question.

[7] We do know examples of compilers that in some cases generate *faster* code for sequential code that references a large array with constant indices than for equivalent code that uses many single variables. We suspect that the register allocator gets overloaded when too many variables are used, and fails to degrade gracefully.

Box 3.3—IMPOSSIBLE EXTENSION:
STATICALLY SPLIT ARRAYS THAT BECOME STRUCTS

A problem with $\mathsf{Array}_s$ is that even if the element type is dynamic a $\mathsf{Pointer}_d$ cannot point to the entire array, because there is no single object in $p_{res}$ to point to. It appears that this restriction could be avoided if $\mathsf{Array}_s$ of dynamic values was implemented by collecting the separate element residues in a struct.

However, there are good arguments that this would be technically very hard. Those arguments are rather long and complex and not very relevant to the main course of the report, so we only make them in an appendix (Appendix B)

Box 3.4—POSSIBLE EXTENSION: DYNAMICALLY INDEXED ARRAY WITH MODEL OBJECTS FOR EACH ELEMENT

$\text{Array}_{d}$ has the downside that the elements cannot be pointed to by static pointers, because there are no model objects representing the individual elements. One might imagine an alternative $\text{Array}_{dy}$ which had the same residual representation as $\text{Array}_{d}$ but where the symbolic model object also included an array of model objects for the elements. The "name" parts of these element model objects could contain something like "`array_name[42]`".

The main problem with this approach is that reaping its benefit would involve allowing to lift a pointer to an element after arithmetic has been done on it. This would be no problem as long as that pointer actually did point to an array element—but extending the type system for pointers to check this would probably make the binding-time analysis of pointer types rather more complex than it is already.

One would also have to find a workable solution to the problem of determining type equivalence in Core C (which will be mentioned in Section B.2), so that we can generated the right number of declarations for the symbolic model objects for $\text{Array}_{dy}$, which would need to be custom-made structs.

Those problems can probably all be solved, but we have decided not to do so until evidence that $\text{Array}_{dy}$ would be truly valuable in real-world situations is found.

| `foo` annotated as | Text in $p_{gen}$ | Text in $p_{res}$ |
|---|---|---|
| $\text{Array}_{d}(\text{Primitive}_{d}, 3)$ | `Code foo ;` | `int foo[3];` |
| $\text{Array}_{s}(\text{Primitive}_{s}, 3)$ | `int foo[3] ;` | |
| $\text{Array}_{s}(\text{Primitive}_{d}, 3)$ | `Code foo[3] ;` | `int foo0 ;`<br>`int foo1 ;`<br>`int foo2 ;` |
| $\text{Array}_{s}(\tau_{T}, 3)$ | `struct Ty fooA[3]`<br>`struct To fooB[3]` | `struct Tr fooA0 ;`<br>`struct Tr fooA1 ;`<br>`struct Tr fooA2 ;` |
| $\text{Array}_{p}(\tau_{T}, 3)$ | `Code fooA ;`<br>`struct To fooB[3] ;` | `struct Tr fooA[3] ;` |
| The variants below are not implemented in C-Mix/II | | |
| $\text{Array}_{dy}(\text{Primitive}_{d}, 3)$<br>(see Box 3.4) | `struct adHoc {`<br>`  Code this ;`<br>`  Code elts[3] ;`<br>`} foo ;` | `int foo[3];` |
| $\text{Array}_{struct}(\text{Primitive}_{d}, 3)$<br>(see Box 3.3<br>and Appendix B) | `struct adHoc {`<br>`  Code this ;`<br>`  Code elts[3] ;`<br>`} foo ;` | `struct adHoc {`<br>`  int elt0 ;`<br>`  int elt1 ;`<br>`  int elt2 ;`<br>`} foo ;` |

*Figure 3.5: Examples of Gegen treatment of arrays*

#### The Array operator

The unary Array operator in Core C is used to convert a pointer to an array into a pointer to its first element.

This may appear unfamiliar to C programmers, because in C the corresponding operation is invisible: you get a pointer to the first element of an array by mentioning its name, syntactically pretending to read the value of the array variable. Core C simply makes this "pointer decay" explicit. The operand of the Array operator is a pointer because Core C always reduces lvalue expressions in C to explicit pointer manipulation.

When implementing this operator we must make sure that array indexing using pointer arithmetic is going to work. The variants already developed for Pointer provide an excellent tool for this: by specifying the result of the operator as either $\mathsf{Pointer}_s$ or $\mathsf{Pointer}_d$ we can control which kind of pointer arithmetic is allowed[8].

If the operand is:

$\mathsf{Pointer}_d(\mathsf{Array}_d(\tau, e))$: The residue of the operand in $p_{res}$ points to an array; we want to derive a pointer to the array's first element. This is achieved by emitting C code to $p_{res}$ as if trying to read the value pointed to by the operand. The result is, naturally, $\mathsf{Pointer}_d(\tau)$.

$\mathsf{Pointer}_s(\mathsf{Array}_d(\tau, e))$: Carry the operand to $\mathsf{Pointer}_d$ and do as above.

$\mathsf{Pointer}_r(\mathsf{Array}_d(\tau, e))$: Lift the operand to $\mathsf{Pointer}_d$ and do as above.

$\mathsf{Pointer}_s(\mathsf{Array}_s(\tau, e))$: The result of the operation must be $\mathsf{Pointer}_s(\tau)$ because even if $\tau$ is dynamic we cannot allow attempts to use dynamic pointer arithmetic. The operation must convert the pointer to the array of $\tau$ model object into a pointer to the first of the model objects. Syntactically, in $p_{gen}$ this is the same as pretending to read the value pointed to by the operand.

$\mathsf{Pointer}_r(\mathsf{Array}_s(\tau, e))$: Lift the operand to $\mathsf{Pointer}_s$ and do as above.

This exhaust the operand type possibilities for basic binding-time types.

#### Array indexing

In Core C, array indexing is reduced to the Array operator just covered and the PtrArith operator described on page 69

#### Compound variants of Array

When the element type needs two model objects the strategy is to use two parallel arrays, one for each kind of model objects.

We always have the choice of letting both of these arrays be $\mathsf{Array}_s$s. This is the natural generalization of the basic $\mathsf{Array}_s$:

$\mathsf{Array}_s(\tau, e)$: When $\tau$ has two model objects, use an independent $\mathsf{Array}_s$ for each model object. The signature of the array is the same as $\tau$'s signature except that $|\mathsf{Array}_s(\tau, e)|_{\mathbf{Y}} = -1$ when $|\tau|_{\mathbf{Y}} = 1$.

[8]When the result of the Array operator is a $\mathsf{Pointer}_s$ it cannot be lifted into $\mathsf{Pointer}_d$ except by carrying. And the contexts where a carry is allowed all gurantee that the result of the carry is going to be "consumed" immediately: there is no possibility of doing dynamic pointer arithmetic directly on the resulting pointer.

| Array-of-$\tau$ variant | $\mathsf{Array}_s$ | $\mathsf{Array}_p$ | $\mathsf{Array}_d$ |
|---|---|---|---|
| Possible when | always | $\lvert\tau\rvert_{\mathbf{Y}} = 1$<br>$\lvert\tau\rvert_{\mathbf{O}} = 1$ | $\lvert\tau\rvert_{\mathbf{Y}} = 1$<br>$\lvert\tau\rvert_{\mathbf{O}} = 0$ |
| Signature | $(-\lvert\tau\rvert^2_{\mathbf{Y}}, \lvert\tau\rvert_{\mathbf{O}})$ | $(1, 1)$ | $(1, 0)$ |
| Array operator works on | $\mathsf{Pointer}_s$ | $\mathsf{Pointer}_p$ | $\mathsf{Pointer}_d$ |
| Array operator produces | $\mathsf{Pointer}_s$ | $\mathsf{Pointer}_p$ | $\mathsf{Pointer}_d$ |

*Table 3.4: Summary of* Array *variants*

The Array operator works on $\mathsf{Pointer}_s$ to the array and produces a $\mathsf{Pointer}_s$ to the first element.

When the signature of the element type is $(1, 1)$ we can also use an $\mathsf{Array}_d$ for the dynamic part of the type. We still need to have an $\mathsf{Array}_s$ for the static part of the type. The natural result of the Array operator in this case is a $\mathsf{Pointer}_p$, so let us call this variant $\mathsf{Array}_p$:

$\mathsf{Array}_p(\tau, e)$: Only possible when the signature of $\tau$ is $(1, 1)$. Model it as an $\mathsf{Array}_d$ of $\tau$'s dynamic part and an $\mathsf{Array}_s$ of $\tau$'s static part. The signature of $\mathsf{Array}_p(\tau, e)$ is $(1, 1)$ itself.

The Array operator works on a $\mathsf{Pointer}_p$ to the array (which may be produced by carrying a $\mathsf{Pointer}_s$) and produces a $\mathsf{Pointer}_p$ to the first element.

**Summary**

Table 3.4 shows a summary of the attributes of the different Array variants.

### 3.2.5 Structs

The two basic variants of Struct we implement are

$\mathsf{Struct}_d(\tau_1, \ldots, \tau_n)$: Create a $p_{res}$ struct containing the residual of each of the $\tau_i$s. Naturally this is only reasonable if no $\tau_i$ has a concrete model object. So the signatures of each $\tau_i$ must be $(1, 0)$ or $(-1, 0)$. In the latter case $\tau_i$ has several independent residuals but we simply let each of them be a member of the residual struct.

We then need a symbolic model object. We could use a simple `Code` containing the name of the residual struct variable, but it turns out to be problem-free to include the symbolic model objects for each $\tau_i$. The name parts of those can be initialized with residual expressions containing something like "`struct_name.member_name`" which makes it possible to have $\mathsf{Pointer}_s$s to individual members of the struct.

The symbolic model object thus becomes a struct whose members are the symbolic model objects for the $\tau_i$s. We still need to store the name of the entire residual struct so that we can create a $\mathsf{Pointer}_d$ to it. Thus we extend the symbolic model struct with a `Code` member named `this` that contains the name of the entire structure.[9]

[9] In the C-Mix/II currently shipped this extra member is named `cmix` instead; once we change the implementation language of $p_{gen}$ from C++ to C we'll change the name to `this`.

The signature of $\mathsf{Struct}_d(\tau_1, ..., \tau_n)$ is $(1, 0)$. It is a dynamically faithful type if each $\tau_i$ is.

$\mathsf{Struct}_s(\tau_1, ..., \tau_n)$: Use a single model object which is a struct containing the model objects for the $\tau_i$s. Because a model object may not have concrete as well as symbolic properties we have to require that none of the $\tau_i$s have symbolic model objects. It does not matter, however if some $\tau_i$ has *two* concrete model objects—we just use two fields in the model struct.

Thus the condition for the type being well-formed is that $|\tau_i|_Y = 0$ for each $i$. The signature of $\mathsf{Struct}_s(\tau_1, ..., \tau_n)$ becomes $(0, 1)$. It is a statically faithful type if each of the $\tau_i$s is.

### Lifts

We do not allow lifts of structures.

In theory one could imagine lifting a $\mathsf{Struct}_s$ to a $\mathsf{Struct}_d$, along with lifting the individual members. However, in practise this is not easy to do, because C has no syntax for constant expression of struct type. One would have do declare an appropriate residual struct object and initialize it with the lifted values, and the name of that could then be used in place of the lifted expression.

We have not deemed the practical benefits of this to be great enough to implement it. Especially its interaction with partially static structures is confusing; there seems to be a risk of *adding* housekeeping operations to $p_{res}$ that were not present in the subject program.

### The Member operator

The Member in Core C operator converts a pointer to a Struct into a pointer to one of its members. This is the same as C's "->" operator, except that it yields a pointer to the member rather than an lvalue.

If the operand is

$\mathsf{Pointer}_d(\mathsf{Struct}_d(\tau_1, ..., \tau_n))$: We can simply do the equivalent of Member in the residual program. The result is $\mathsf{Pointer}_d(\tau_i)$.

$\mathsf{Pointer}_s(\mathsf{Struct}_d(\tau_1, ..., \tau_n))$: There are two valid ways to handle this; both are supported in the $p_{gen}$ generation phase.

One is to carry the pointer into the above case and get a $\mathsf{Pointer}_d(\tau_i)$ result.

Another, since we have model objects for the members as part of the model object for the entire struct, is to do a Member in the generating extension, producing a $\mathsf{Pointer}_r(\tau_i)$ instead.

$\mathsf{Pointer}_r(\mathsf{Struct}_d(\tau_1, ..., \tau_n))$: The analysis here is equivalent to that for $\mathsf{Pointer}_s$.

$\mathsf{Pointer}_s(\mathsf{Struct}_s(\tau_1, ..., \tau_n))$: This is easy, of course: we can simply do a Member operation in $p_{gen}$. The result is $\mathsf{Pointer}_r(\tau_i)$.

This exhausts the basic binding-time types that can be operands to Member at all.

| foo annotated as | Text in $p_{gen}$ | Text in $p_{res}$ |
|---|---|---|
| struct R<br>$= \mathsf{Struct_d}(\mathsf{Primitive_d},$<br>$\mathsf{Primitive_d})$ | struct R {<br>  Code this ;<br>  Code bar ;<br>  Code baz ;<br>} foo ; | struct R {<br>  int bar ;<br>  int baz ;<br>} foo ; |
| struct S<br>$= \mathsf{Struct_s}(\mathsf{Primitive_s},$<br>$\mathsf{Primitive_s})$ | struct S {<br>  int bar ;<br>  int baz ;<br>} foo ; | |
| struct T<br>$= \tau_T$<br>$= \mathsf{Struct_p}(\mathsf{Primitive_d},$<br>$\mathsf{Primitive_s})$ | struct Ty {<br>  Code this ;<br>  Code bar ;<br>} fooA ;<br>struct To {<br>  int baz ;<br>} fooB ; | struct Tr {<br>  int bar ;<br>} fooA ; |
| struct U<br>$= \mathsf{Struct_p}(\tau_T,$<br>$\tau_T)$ | struct Uy {<br>  Code this ;<br>  struct Ty bar ;<br>  struct Ty baz ;<br>} fooA ;<br>struct Uo {<br>  struct To bar ;<br>  struct To baz ;<br>} fooB ; | struct Ur {<br>  struct Tr bar ;<br>  struct Tr baz ;<br>} fooA ; |
| struct V<br>$= \mathsf{Struct_p}(\mathsf{Primitive_s},$<br>$\mathsf{Primitive_s})$ | struct Vy {<br>  Code this ;<br>} fooA ;<br>struct Vo {<br>  int bar ;<br>  int baz ;<br>} fooB ; | struct Vr {<br>  char dummy ;<br>} fooA ; |
| | The variant below is not implemented in C-Mix$_{II}$ | |
| struct W<br>$= \mathsf{Struct_{sy}}(\mathsf{Primitive_d},$<br>$\mathsf{Primitive_d})$<br>(see Box 3.5) | struct W {<br>  Code bar ;<br>  Code baz ;<br>} foo ; | int foobar ;<br>int foobaz ; |

*Figure 3.6: Examples of Gegen treatment of structs. Each of the example structs has two members, the first called* `bar` *and the second called* `baz`*. In real life the type and variable declarations in $p_{res}$ and $p_{gen}$ are separate; they have been collapsed here to conserve space.*

### Compound variants of Struct

If the model objects for the member types include concrete as well as symbolic ones, neither $\mathsf{Struct}_d$ nor $\mathsf{Struct}_s$ can be used. The natural idea here is:

$\mathsf{Struct}_p(\tau_1, ..., \tau_n)$: Use a combination of a $\mathsf{Struct}_d$ which contains all of the symbolic model objects, and a $\mathsf{Struct}_s$ which contains all of the concrete model objects. There are no restrictions on the $\tau_i$s, except that $|\tau_i|_O > 0$ for at least one $i$. The signature of $\mathsf{Struct}_p$ is $(1, 1)$.

$\mathsf{Struct}_p$ introduces a new range of possible operands to the Member operator:

- $\mathsf{Pointer}_d(\mathsf{Struct}_p(\tau_1, ..., \tau_n))$ (remember that a $\mathsf{Pointer}_d$ can point to a partially static type but only locates the residue of the dynamic part): The result is $\mathsf{Pointer}_d(\tau_i)$, and the operation is naturally possible exactly when that type is well-formed.
- $\mathsf{Pointer}_s(\mathsf{Struct}_p(\tau_1, ..., \tau_n))$: this naturally yields a $\mathsf{Pointer}_r(\tau_i)$. Depending on the signature of $\tau_i$ one or both of the static pointers that represent the operand may be actually used.
- $\mathsf{Pointer}_r(\mathsf{Struct}_p(\tau_1, ..., \tau_n))$: also yields $\mathsf{Pointer}_r(\tau_i)$. May be viewed as lifting to $\mathsf{Pointer}_s$ before the Member operation.
- $\mathsf{Pointer}_p(\mathsf{Struct}_p(\tau_1, ..., \tau_n))$: the result type depends on the signature of the $\tau_i$ that is selected:
  - $(1, 0)$: $\mathsf{Pointer}_d(\tau_i)$. We prefer to view this as lifting the operand to $\mathsf{Pointer}_d$ before the Member operation[10].
  - $(1, 1)$: $\mathsf{Pointer}_p(\tau_i)$.
  - $(0, 1)$ or $(0, 2)$: $\mathsf{Pointer}_r(\tau_i)$.
  - $(-1, 0)$ or $(-1, 1)$: impossible

$\mathsf{Struct}_p$ is the prototypical example of a partially static type. In fact, any type with more than one model object must contain a $\mathsf{Struct}_p$ somewhere.

Besides the ability to contain members of any signature, $\mathsf{Struct}_p$ has the advantage over $\mathsf{Struct}_s$ that a $\mathsf{Pointer}_d$ can point to it. In some cases this can be the only reason why a struct cannot be $\mathsf{Struct}_s$ (though it is quite rare: it only occurs when there is a pointer to the struct that for some reason needs to be dynamic yet is neither used to access any data in the struct). Because of this we allow $\mathsf{Struct}_p$s with completely static members; as C does not allow structs without any members we insert an dummy `char` member into the residual struct. See `struct V` in Figure 3.6 for an example of this.

### Summary

Table 3.5 shows a summary of the different Struct variants and the different ways to use Member.

[10]The reason is that in the binding-time analysis we let lifts be implicit. Allowing Member on an unlifted $\mathsf{Pointer}_p(\mathsf{Struct}_p(...))$ to give a $\mathsf{Pointer}_d$ result would have made the BTA results ambigous: is the direct Member operation or a lift followed by a $\mathsf{Pointer}_d$ Member operation meant?

### Box 3.5—POSSIBLE EXTENSION: STRUCT WITH DETACHED DYNAMIC MEMBERS

One may wonder whether there is a useful Struct variant where, like for $\mathsf{Array}_s$, the dynamic subobjects of a subject object becomes separate variables in $p_{res}$. There is, but C-Mix/II does not currently implement it.

We could call this variant $\mathsf{Struct}_{sy}(\tau_1, ..., \tau_n)$. It would require that each $\tau_i$ is dynamic, and would be modelled by a struct in $p_{gen}$ containing the (symbolic) model objects for the $\tau_i$s. In $p_{res}$ there would be a separate variable for each member of the struct. The signature of $\mathsf{Struct}_{sy}$ would be $(-1, 0)$.

The primary advantage of $\mathsf{Struct}_{sy}$ would be that the compiler that compiles the residual program might do better register allocation of the member residues if they were not lumped together in a struct. One might well imagine a C compiler that does not even try to keep struct members in registers.

It would be easy to extend the binding-time type system and binding-time analysis of C-Mix/II to select $\mathsf{Struct}_{sy}$ instead of $\mathsf{Struct}_d$ where it is not necessary (because of assignments or dynamic pointers) for it to be a single object in $p_{res}$. There would also not be any deep problem in adding support for $\mathsf{Struct}_{sy}$ to the final $p_{gen}$ construction phase (though it would be quite laborious, because the internal structure of the current code depends rather heavily on the fact that arrays are the only types that map to multiple separate objects in $p_{res}$).

The primary reason why $\mathsf{Struct}_{sy}$ is not included in the current C-Mix/II is that nobody thought about it while we wrote it. Andersen (1994, section 3.7.1) did describe a $\mathsf{Struct}_{sy}$-like construction he called "static struct with dynamic members", but it was never implemented in the prototype C-Mix. Since then focus shifted away from the idea and design notes written since ca. 1996 seem to imply using what Andersen (1994) calls the "narrowing" solution in his Example 3.29—which is the $\mathsf{Struct}_p$ variant we do implement.

$\mathsf{Struct}_{sy}$ remains a possible candidate for future extensions of C-Mix/II.

| $\mathsf{Struct}(\tau_1, ..., \tau_n)$ variant | $\mathsf{Struct}_s$ | $\mathsf{Struct}_p$ | $\mathsf{Struct}_d$ |
|---|---|---|---|
| Possible when | $\forall i : \lvert\tau_i\rvert_Y = 0$ | $\exists i : \lvert\tau_i\rvert_O > 0$ | $\forall i : \lvert\tau_i\rvert_O = 0$ |
| Signature | $(0, 1)$ | $(1, 1)$ | $(1, 0)$ |

| Member operand | result type | conditions on $\tau_i$ |
|---|---|---|
| $\mathsf{Pointer}_d(\mathsf{Struct}_d(\tau_1, ..., \tau_n))$ | $\mathsf{Pointer}_d(\tau_i)$ | $(\lvert\tau_i\rvert_Y = 1$ |
| $\mathsf{Pointer}_s(\mathsf{Struct}_d(\tau_1, ..., \tau_n))$ | $\mathsf{Pointer}_r(\tau_i)$ | none |
| $\mathsf{Pointer}_s(\mathsf{Struct}_s(\tau_1, ..., \tau_n))$ | $\mathsf{Pointer}_r(\tau_i)$ | none |
| $\mathsf{Pointer}_d(\mathsf{Struct}_p(\tau_1, ..., \tau_n))$ | $\mathsf{Pointer}_d(\tau_i)$ | $(\lvert\tau_i\rvert_Y = 1)$ |
| $\mathsf{Pointer}_s(\mathsf{Struct}_p(\tau_1, ..., \tau_n))$ | $\mathsf{Pointer}_r(\tau_i)$ | none |
| $\mathsf{Pointer}_p(\mathsf{Struct}_p(\tau_1, ..., \tau_n))$ | $\mathsf{Pointer}_p(\tau_i)$<br>$\mathsf{Pointer}_r(\tau_i)$ | $\lvert\tau_i\rvert_Y = \lvert\tau_i\rvert_O = 1$<br>$\lvert\tau_i\rvert_Y = 0$ |

*Table 3.5: Summary of* Struct *variants*

### 3.2.6 Unions

There are two main uses of C's `union` types. The first is to conserve memory by co-locating variables that are not used simultaneously. The second is to re-interpret the machine representation of one type as if it were some other one (typically one of them is a `char` array).

The latter use is not portable; C-Mix$_{II}$ regards it as a "dirty trick" and assumes that it is not done. We cannot check if this assumption actually holds in a given program: a useful approximative analysis would be very hard to design, and checking the correct behaviour at run time would be very time-consuming and interfere with the possibility of specifying faithful binding-time types.

A Union can be handled much like a Struct:

$\mathsf{Union}_d(\tau_1, ..., \tau_n)$: The same as $\mathsf{Struct}_d(\tau_1, ..., \tau_n)$, except that the residual type is a `union` instead of a `struct`. The type of the model object is still a `struct`, though: it would be wrong to try to superimpose the *names* of the residual `union`'s members on each other.

The signature of $\mathsf{Union}_d(\tau_1, ..., \tau_n)$ is $(1, 0)$. It is a dynamically faithful type if each $\tau_i$ is.

$\mathsf{Union}_s(\tau_1, ..., \tau_n)$: The same as $\mathsf{Struct}_s(\tau_1, ..., \tau_n)$, only the model object should be a `union` instead of a `struct`.

The signature of $\mathsf{Union}_s(\tau_1, ..., \tau_n)$ is $(0, 1)$. It is a statically faithful type if each $\tau_i$ is.

#### The first union rule

We need to take special measures to ensure that the following rule from the C standard (ANSI 1990, clause 6.2.2.3) works:

> If a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them. Two structures share a *common initial sequence* if corresponding members have compatible types (and for bit-fields, the same widths) for a sequence of one or more initial members.

In practical terms this means that a Union binding-time type is only well-formed if the binding-time types in such common initial sequences match. This is the **first union rule**; it guarantees that the common initial sequence will have the same representation in $p_{gen}$ and $p_{res}$, so that the same rule when applied to $p_{gen}$ and $p_{res}$ makes sure that the program behaves as expected.

#### The second union rule

Another issue concerns memoization. Without going into details about memoization strategies, one optimization we use involves approximating the set of "legally accessible" objects at $p_{gen}$'s run time by inspecting the values of accessible pointers to concrete model objects.

We do not want to have to inspect such pointers inside `union` members that may not be the ones used at a given time: the values of such pointers could be indeterminate and the results would be unpredictable. Because we also don't

want to add run-time information about which member is current, our decision is that

- *The target type of any pointer type found inside a union must be a dynamic binding-time type. The exception is pointers that are in an initial sequence that is common for all members of the union.*

This is the **second union rule**. Because of this restriction it is safe to do this "legally accessible objects" analysis as if the union always contained its first member.

We need only enforce the second union rule for union types that the subject program only uses to define objects. If the subject program only manipulates pointers to the union, the second union rule need not hold. Those pointers would have to either point nowhere or point to union objects defined outside of the subject program (e.g., the pointers could be returned from external function); the latter case is unsafe because the pointed-to objects cannot be memoized, but enforcing the union rule would only prevent users who know what they do from doing so, and only in special cases.

### The Member operator for unions

The Member operator is also used to select members from union types. The analysis in Section 3.2.5 can be reused unchanged, except writing Union instead of Struct.

### Compound variants of Union

The equivalent of $\mathsf{Struct}_p$, $\mathsf{Union}_p$, works just fine.The first and second union rules must still hold. When the first union rule is enforced, a common initial sequence gets split into a common initial sequence in each of the parts.

---

Box 3.6—RELAXING THE SECOND UNION RULE

It is tempting to rephrase the second union rule such that the affected pointer can also be a $\mathsf{Pointer}_d$ to partially static data. As far as memoization is concerned this would be perfectly safe.

However, we would lose the important property that there is a unique "most static" possible annotation, as witnessed by this program fragment:

```
struct T  { struct T *p ; };
struct TT { struct T a, b ; };
union  U  { struct TT tt ; } u ;
...
  u.tt.a.p = &u.tt.b ;
  u.tt.b.p = &u.tt.a ;
```

where either `u.tt.a.p` or `u.tt.b.p` could be allowed to be a $\mathsf{Pointer}_s$, but not both at the same time.

It would probably be nice to have a way for the user to specify that an alternative variant of the second union rule (where the pointer rather than the pointed-to data is forced to be dynamic) should be used in a given case. Presently C-Mix/II contains no such feature, however.

---

### 3.2.7 Function pointers

Formally, in C, there is only one "pointer" construct and a separate "function returning..." construct that can be combined with it.

In reality, the two concepts of a function and a pointer to it are not that separate. The "function returning..." is a very low-class citizen in C: basically the only thing one can do with it is to point a pointer to it.

The type rules for the language prevent any expression from having plain "function" type: as soon as one tries that—by mentioning the name of a function or trying to apply the "`*`" operator to a "pointer to function"—it *must* be immediately hidden behind an "`&`" operator, and an invisible one is generated if one is not already present. In the function-call expression the "function" must be an expression of type "pointer to function": the invisible "`&`" causes `foo(42)` to be rewritten as `(&foo)(42)`.

Such is at least the stated semantics of C, and neither we nor C-Mix/II shall be bothered by the fact that real-life compilers may generate better code for `foo(42)` than for `(&foo)(42)`.[11]

In Core C, the nicest choice would be to avoid this mess by having a single type constructor which corresponded to C's "pointer to function accepting this and returning that" construction. However, for technical reasons related to efficient implementation of the pointer analysis, Core C's type system does discriminate between the function and the pointer to it.

For the purposes of binding-time analysis, though, it is convenient to imagine having a single type constructor "$\mathsf{FunPtr}(\tau_1, ..., \tau_n \to \tau)$" meaning, "pointer to a function taking $\tau_1, ..., \tau_n$ arguments and returning $\tau$."

The basic variants are

$\mathsf{FunPtr}_d(\tau_1, ..., \tau_n \to \tau)$: Represented in $p_{res}$ as a pointer to a function taking the $\tau_i$s' residues as arguments and returning the residue of $\tau$. $\tau$ and the $\tau_i$s must all have signature $(1, 0)$. In $p_{gen}$ there is a `Code` containing the name of the residual pointer variable.

The main use of $\mathsf{FunPtr}_d$ is to allow $p_{gen}$ to contain calls to external functions in $p_{gen}$. This requires a dynamically faithful type, which $\mathsf{FunPtr}_d$ is if the $\tau_i$s and $\tau$ are dynamically faithful too.

$\mathsf{FunPtr}_s(\tau_1, ..., \tau_n \to \tau)$: This is represented as a function pointer in $p_{gen}$. The arguments and return value types are derived from $\tau_1, ..., \tau_n, \tau$ in the following way:

- A parameter or return type that has a *static* binding-time type is replaced by its concrete model object type.
- A parameter or return type that has a *dynamic* binding-time type is replaced by `Code`, no matter if that is not the type of its symbolic model object type.[12]

These rules generalize the conventions for how to call generating functions given on page 34ff. A parameter to the generating function that corresponds to static data models the subject program's parameter object. If

[11] The generating extensions or residual program generated by C-Mix/II never contains the construction `(&foo)(42)`; a FunAdr expression is rendered without a `&`.

[12] The only case of a symbolic model object that does not have type `Code` is currently that of a $\mathsf{Struct}_d$ (and $\mathsf{Array}_s$ which does not occur as a parameter or return type); but more cases may arise in the future.

it corresponds to dynamic data, the generating function's parameter does *not* model an object in the subject program: it carries a finished *expression*. Likewise for the return type.

From this discussion it is clear that a $\mathsf{FunPtr}_s$ can be used to call a generating function in $p_{gen}$. Observe, though, that if the $\tau_i$s and $\tau$ are all statically faithful, then the entire $\mathsf{FunPtr}_s$ is, too. This means that a $\mathsf{FunPtr}_s$ can also be used to refer to external functions that are linked together with $p_{gen}$.

The signature of a $\mathsf{FunPtr}_s$ is $(0, 1)$.

### Lifts

There can be no lifts between $\mathsf{FunPtr}_s$ and $\mathsf{FunPtr}_d$. (But see Box 3.7 for a discussion of how it might be allowed).

### Function pointer creation

The way to create a function pointer is to take the address of a function. If the function is a external function, this is represented as a Core C Const expression of FunPtr type. The expression can have either the statically faithful or the dynamically faithful version of the library function's type.

A pointer to a function for which C-Mix/II knows the body is created with a FunAdr expression[13]. This operator creates a $\mathsf{FunPtr}_s$ which points to the generator function.

There is no way to create a $\mathsf{FunPtr}_d$ that references a function for which we know the body. The point about these functions is that they are to be specialized, so there is generally no single corresponding version of them in $p_{res}$ that can be called at any time in the program.

Currently, the binding-time analysis may result in an annotated program that asks for a $\mathsf{FunPtr}_d$ to be created for a known function. In that case C-Mix/II halts with an error message before the $p_{gen}$ construction phase.

The only possible way to treat this case, except refusing to create a $p_{gen}$, seems to be to specialize the function so that $p_{res}$ will contain a version of it, but to do it so weakly that the specialized version does not depend on *any* static data. That means that if a $\mathsf{FunPtr}_d$ is required to reference a known function, all of the parameters must be forced to be dynamic, and so must any non-local variable that the function or any function it calls may access in any way.

The practical value of this in comparison with simply letting the user exclude the function's definition from the input to C-Mix/II (so that it becomes a external) is perceived to be small. For that reason there are no current plans to implement this facility.

See Box 3.7 for discussion of an alternative way of creating dynamic pointers to specializable functions.

[13]In Appendix A as well as in the current C-Mix/II source code, this operation is referred to as a variant of the Var operator. The two uses are, however, treated quite differently in most parts of C-Mix/II, so we plan to split the Var operator in two cases shortly. In this report we shall assume the split has already taken place.

## Box 3.7—POSSIBLE EXTENSION: LIFTABLE FUNCTION POINTER VARIANTS

It is desirable to make it possible to lift function pointers at least in certain cases. There are various proposals for how to do it, none of which has been implemented in C-Mix/II yet:

Lifting from $\mathsf{FunPtr_s}$ to $\mathsf{FunPtr_d}$: It has been suggested that if a $\mathsf{FunPtr_s}$ is known to point to one of a fixed set of external functions, it can be lifted by comparing it to each of the external functions' addresses, thereby selecting the right name to embed into $p_{res}$. The argument and return types would get coerced from statically faithful to dynamically faithful in the process.

The disadvantage of this is that it needs all of the external functions to be linked into $p_{gen}$ so that their addresses are known and can be used as tags—even if the functions are not actually called from $p_{gen}$. This seems wasteful.

$\mathsf{FunPtr_c}(\tau_1, ..., \tau_n \to \tau)$: A hypothetical basic variant: All the $\tau_i$s and $\tau$ must have signature $(1, 0)$. The pointer is represented in $p_{gen}$ as a `Code` containing a residual *constant* that evaluates to a function pointer in $p_{res}$. This `Code` is *not* a symbolic but a concrete model object, thus the signature of $\mathsf{FunPtr_c}$ is $(0, 1)$.

A $\mathsf{FunPtr_c}$ could be created in the same way as a $\mathsf{FunPtr_d}$, and could be lifted to $\mathsf{FunPtr_d}$ at any time. Its main advantage is that if the pointer was only lifted just before a call through it, the residual call statement would look like a direct call and presumably be more efficient than an indirect call.

It is thought, however, that the situations in which this advantage could be exploited are rare.

$\mathsf{FunPtr_{dn}}(\tau_1, ..., \tau_n \to \tau)$: A hypothetical compound variant, to be used when the function pointer is known to point to one of a fixed set of functions. The function pointer type is preprocessed into a $\mathsf{Primitive_d}$(`unsigned`) before the $p_{gen}$ construction. The `unsigned` value would contain a small integer that identified a function.

A call through a $\mathsf{FunPtr_{dn}}$ would be converted to something like

```
if ( fp == 0 ) x = f(y) ;
else if ( fp == 1 ) x = g(y) ;
else if ( fp == 4 ) x = h(y) ;
else x = k(y) ;
```

and the actual calls would be performed in $p_{gen}$.

This variant would also allow a dynamic function pointer to point to a specializable function.

A problem here is that that conditional structure introduces dynamic control, something that other calls through function pointers don't. It is not clear whether it would be easy to modify the binding-time analysis to take that into account.

$\mathsf{FunPtr_{sn}}(\tau_1, ..., \tau_n \to \tau)$: The static version of $\mathsf{FunPtr_{dn}}$, mapping to a $\mathsf{Primitive_s}$ instead of a $\mathsf{Primitive_d}$. It can be lifted to $\mathsf{FunPtr_{dn}}$ or called directly—in the latter case the call does *not* introduce dynamic control.

**Function calls**

All function calls in Core C take place through a function pointer.

If the function pointer is a $\mathsf{FunPtr}_d$, a call statement in $p_{res}$ is simply generated.

Matters are more complex when the function pointer is a $\mathsf{FunPtr}_s$. The description in Section 2.4.1 continues to be valid, and the only influence of the richer type system of Core C is that the function to call is selected by a function pointer expression instead of directly by name. In the common case where the function pointer expression is a FunAdr the result is hard to tell apart from what it would look like in Section 2.4.1.

Notice that the conventions for how to call a generating function have been designed so that if the function pointer's binding-time type happens to be faithful the call sequence collapses into a normal function call. Thus the same call statement representation can be used even when the pointer can point to either an external function or a generating function.

**Compound variants of FunPtr**

$\mathsf{FunPtr}_s$ can also be used if one of the arguments have more than one model object. In that case the splitter phase (Chapter 5) splits the offending argument into two separate arguments. If the return type has two model objects, one part of it is passed through a global variable, as outlined in Section 2.6.2.

## 3.3 Summary of the binding-time type system

In this section we present binding-time typing rules in a compact form for easy reference during the development of the binding-time analysis.

The typing rules are expressed as an inference system recursively defining the following judgements:

| Judgement | Meaning |
|---|---|
| $\Gamma \vdash e : \tau$ | Given variable types $\Gamma$, expression $e$ can have binding-time type $\tau$, either before or after lifting. |
| $\Gamma \vdash e :\!\!< \tau$ | Given variable types $\Gamma$, expression $e$ can have binding-time type $\tau$ after possible lifting or carrying. |
| $\vdash \tau_1 < \tau_2$ | It is possible to lift a value of binding-time type $\tau_1$ to $\tau_2$. |
| $\Gamma \vdash s.$ | Variable types $\Gamma$ allow statement $s$ to be typed. |

Table 3.6 summarizes the rules for assembling well-formed binding-time types. It is to be implicitly understood that each type that appears in the following rules must be well-formed.

### 3.3.1 Typing rules for expressions

In the typing rules that follow we ignore the type field of the Core C expression representation, as it would be notationally awkward to use it. The natural requirement that the binding-time type assigned to an expression should have the expression's original type as its erasure is implicit.

| Type | Well-formed if | | Signature | page |
|---|---|---|---|---|
| $\mathsf{Primitive_d}$ | always | | $(1,0)$ | 63 |
| $\mathsf{Primitive_s}$ | always | | $(0,1)$ | 63 |
| $\mathsf{Abstract_d}$ | always | | $(1,0)$ | 63 |
| $\mathsf{Abstract_s}$ | always | | $(0,1)$ | 63 |
| $\mathsf{Enum_d}$ | always | | $(1,0)$ | 64 |
| $\mathsf{Enum_s}$ | | [a] | $(0,1)$ | 64 |
| $\mathsf{Pointer_d}(\tau)$ | $\lvert\tau\rvert_{\mathbf{Y}} = 1$ | | $(1,0)$ | 65 |
| $\mathsf{Pointer_p}(\tau)$ | $(\lvert\tau\rvert_{\mathbf{Y}}, \lvert\tau\rvert_{\mathbf{O}}) = (1,1)$ | | $(1,1)$ | 70 |
| $\mathsf{Pointer_r}(\tau)$ | always | | $(0, \lvert\tau\rvert^2_{\mathbf{Y}} + \lvert\tau\rvert_{\mathbf{O}})$ | 66 |
| $\mathsf{Pointer_s}(\tau)$ | always | | $(0, \lvert\tau\rvert^2_{\mathbf{Y}} + \lvert\tau\rvert_{\mathbf{O}})$ | 65 |
| $\mathsf{Array_d}(\tau, e)$ | $(\lvert\tau\rvert_{\mathbf{Y}}, \lvert\tau\rvert_{\mathbf{O}}) = (1,0)$ | | $(1,0)$ | 71 |
| $\mathsf{Array_p}(\tau, e)$ | $(\lvert\tau\rvert_{\mathbf{Y}}, \lvert\tau\rvert_{\mathbf{O}}) = (1,1)$ | [b] | $(1,1)$ | 75 |
| $\mathsf{Array_s}(\tau, e)$ | | [b] | $(-\lvert\tau\rvert^2_{\mathbf{Y}}, \lvert\tau\rvert_{\mathbf{O}})$ | 72 |
| $\mathsf{Struct_d}(\tau_1, ..., \tau_n)$ | $\forall i : \lvert\tau_i\rvert_{\mathbf{O}} = 0$ | | $(1,0)$ | 75 |
| $\mathsf{Struct_p}(\tau_1, ..., \tau_n)$ | $\exists i : \lvert\tau_i\rvert_{\mathbf{O}} > 0$ | | $(1,1)$ | 78 |
| $\mathsf{Struct_s}(\tau_1, ..., \tau_n)$ | $\forall i : \lvert\tau_i\rvert_{\mathbf{Y}} = 0$ | | $(0,1)$ | 76 |
| $\mathsf{Union_d}(\tau_1, ..., \tau_n)$ | $\forall i : \lvert\tau_i\rvert_{\mathbf{O}} = 0$ | [c] | $(1,0)$ | 80 |
| $\mathsf{Union_p}(\tau_1, ..., \tau_n)$ | $\exists i : \lvert\tau_i\rvert_{\mathbf{O}} > 0$ | [c] | $(1,1)$ | 81 |
| $\mathsf{Union_s}(\tau_1, ..., \tau_n)$ | $\forall i : \lvert\tau_i\rvert_{\mathbf{Y}} = 0$ | [c] | $(0,1)$ | 80 |
| $\mathsf{FunPtr_d}(\tau_1, ..., \tau_n \to \tau_0)$ | $\forall i : (\lvert\tau_i\rvert_{\mathbf{Y}}, \lvert\tau_i\rvert_{\mathbf{O}}) = (1,0)$ | | $(1,0)$ | 82 |
| $\mathsf{FunPtr_s}(\tau_1, ..., \tau_n \to \tau_0)$ | always | | $(0,1)$ | 82 |

[a]: not possible if a constant specification has type $\mathsf{Primitive_d}$.
[b]: not possible if the length expression has type $\mathsf{Primitive_d}$.
[c]: the union rule(s) must be observed.

*Table 3.6: Supported specialization variants and rules for well-formedness*

### The Var operator

We explicitly allow any Pointer variant to be created, as long as the binding-time type is well-formed. Though $\mathsf{Pointer_d}$ can usually be obtained by lifting a $\mathsf{Pointer_r}$, the lift might not always be allowed as a lift[14].

$$\frac{}{\Gamma \vdash \mathsf{Var}(x) : \mathsf{Pointer_d}(\Gamma(x))} \qquad \frac{}{\Gamma \vdash \mathsf{Var}(x) : \mathsf{Pointer_p}(\Gamma(x))}$$

$$\frac{}{\Gamma \vdash \mathsf{Var}(x) : \mathsf{Pointer_r}(\Gamma(x))} \qquad \frac{}{\Gamma \vdash \mathsf{Var}(x) : \mathsf{Pointer_s}(\Gamma(x))}$$

### The FunAdr operator

We assume that the type of the function is given by the type environment $\Gamma$. It must be a $\mathsf{FunPtr_s}$; $\mathsf{FunPtr_d}$s can only point to external functions, for which

[14]Pointers can only be lifted when they are sure to be in scope. The pointer created by Var *is* in scope, but if it is a (non-truly) local variable in a recursive function that cannot always be inferred from the points-to set alone. We do not like to let the possibility of lift depend on the structure of the expression to be lifted (because it already depends on the context, and it would be inelegant for it to depend on both at the same time).

$\mathsf{Const}$ operators are used.

$$\frac{}{\Gamma \vdash \mathsf{FunAdr}(f) : \Gamma(f) = \mathsf{FunPtr}_s(\ldots)}$$

**The $\mathsf{EnumConst}$ operator**

$$\frac{}{\Gamma \vdash \mathsf{EnumConst}(\eta) : \mathsf{Primitive}_d}$$

$$\frac{}{\Gamma \vdash \mathsf{EnumConst}(\eta) : \mathsf{Primitive}_s}\ \eta \text{ has no } \mathsf{Primitive}_d \text{ constants}$$

**The $\mathsf{Const}$ operator**

The $\mathsf{Const}$ operator of Core C is not only used for numerical constants but also for "magic words" like `stderr`. `stderr` should be an rvalue expression of type "`FILE*`", so the translation to Core C makes it (with the help of a C-Mix/II-specific `<stdio.h>` header) into a $\mathsf{Const}$ expression of type $\mathsf{Pointer}(\mathsf{Abstract})$. When we include `<stdio.h>` in $p_{gen}$ and $p_{res}$, "`stderr`" becomes a suitable constant of type "`FILE*`". The binding-time annotations that match this are precisely the faithful ones.

$$\frac{}{\Gamma \vdash \mathsf{Const} : \tau}\ \tau \in \mathbb{F}$$

**The $\mathsf{Null}$ operator**

There are no restrictions of the binding-time type of the created pointer The implicit requirement that it matches the original type takes care of its being a pointer at all.

$$\frac{}{\Gamma \vdash \mathsf{Null} : \tau}$$

**The $\mathsf{Unary}$ operator**

$$\frac{\Gamma \vdash e : \mathsf{Primitive}_d}{\Gamma \vdash \mathsf{Unary}(e) : \mathsf{Primitive}_d} \qquad \frac{\Gamma \vdash e : \mathsf{Primitive}_s}{\Gamma \vdash \mathsf{Unary}(e) : \mathsf{Primitive}_s}$$

**The $\mathsf{PtrArith}$ operator**

$$\frac{\Gamma \vdash e_1 : \mathsf{Pointer}_d(\tau) \quad \Gamma \vdash e_2 : \mathsf{Primitive}_d}{\Gamma \vdash \mathsf{PtrArith}(e_1, \circ, e_2) : \mathsf{Pointer}_d(\tau)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Pointer}_p(\tau) \quad \Gamma \vdash e_2 : \mathsf{Primitive}_s}{\Gamma \vdash \mathsf{PtrArith}(e_1, \circ, e_2) : \mathsf{Pointer}_p(\tau)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Pointer}_s(\tau) \quad \Gamma \vdash e_2 : \mathsf{Primitive}_s}{\Gamma \vdash \mathsf{PtrArith}(e_1, \circ, e_2) : \mathsf{Pointer}_s(\tau)}$$

**The** PtrCmp **operator**

$$\frac{\Gamma \vdash e_1 : \mathsf{Pointer_d}(\tau) \quad \Gamma \vdash e_2 : \mathsf{Pointer_d}(\tau)}{\Gamma \vdash \mathsf{PtrCmp}(e_1, \circ, e_2) : \mathsf{Primitive_d}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Pointer_s}(\tau) \quad \Gamma \vdash e_2 : \mathsf{Pointer_s}(\tau)}{\Gamma \vdash \mathsf{PtrCmp}(e_1, \circ, e_2) : \mathsf{Primitive_s}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Pointer_p}(\tau) \quad \Gamma \vdash e_2 : \mathsf{Pointer_p}(\tau)}{\Gamma \vdash \mathsf{PtrCmp}(e_1, \circ, e_2) : \mathsf{Primitive_s}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Pointer_p}(\tau) \quad \Gamma \vdash e_2 : \mathsf{Pointer_s}(\tau)}{\Gamma \vdash \mathsf{PtrCmp}(e_1, \circ, e_2) : \mathsf{Primitive_s}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Pointer_s}(\tau) \quad \Gamma \vdash e_2 : \mathsf{Pointer_p}(\tau)}{\Gamma \vdash \mathsf{PtrCmp}(e_1, \circ, e_2) : \mathsf{Primitive_s}}$$

**The** Binary **operator**

$$\frac{\Gamma \vdash e_1 : \mathsf{Primitive_d} \quad \Gamma \vdash e_2 : \mathsf{Primitive_d}}{\Gamma \vdash \mathsf{Binary}(e_1, \circ, e_2) : \mathsf{Primitive_d}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Primitive_s} \quad \Gamma \vdash e_2 : \mathsf{Primitive_s}}{\Gamma \vdash \mathsf{Binary}(e_1, \circ, e_2) : \mathsf{Primitive_s}}$$

**The** Member **operator**

$$\frac{\Gamma \vdash e \precsim \mathsf{Pointer_d}(\mathsf{Struct_d}(\tau_1, ..., \tau_n))}{\Gamma \vdash \mathsf{Member}(e, m) : \mathsf{Pointer_d}(\tau_m)}$$

$$\frac{\Gamma \vdash e \precsim \mathsf{Pointer_d}(\mathsf{Struct_p}(\tau_1, ..., \tau_n))}{\Gamma \vdash \mathsf{Member}(e, m) : \mathsf{Pointer_d}(\tau_m)}$$

$$\frac{\Gamma \vdash e \precsim \mathsf{Pointer_p}(\mathsf{Struct_p}(\tau_1, ..., \tau_n))}{\Gamma \vdash \mathsf{Member}(e, m) : \mathsf{Pointer_p}(\tau_m)}$$

$$\frac{\Gamma \vdash e \precsim \mathsf{Pointer_p}(\mathsf{Struct_p}(\tau_1, ..., \tau_n))}{\Gamma \vdash \mathsf{Member}(e, m) : \mathsf{Pointer_r}(\tau_m)}\ |\tau_m|_Y = 0$$

$$\frac{\Gamma \vdash e \precsim \mathsf{Pointer_s}(\mathsf{Struct_d}(\tau_1, ..., \tau_n))}{\Gamma \vdash \mathsf{Member}(e, m) : \mathsf{Pointer_r}(\tau_m)}$$

$$\frac{\Gamma \vdash e \precsim \mathsf{Pointer_s}(\mathsf{Struct_p}(\tau_1, ..., \tau_n))}{\Gamma \vdash \mathsf{Member}(e, m) : \mathsf{Pointer_r}(\tau_m)}$$

$$\frac{\Gamma \vdash e \precsim \mathsf{Pointer_s}(\mathsf{Struct_s}(\tau_1, ..., \tau_n))}{\Gamma \vdash \mathsf{Member}(e, m) : \mathsf{Pointer_r}(\tau_m)}$$

... plus the same rules with Struct replaced by Union.

**The** Array **operator**

$$\frac{\Gamma \vdash e \mathrel{\dot{\prec}} \mathsf{Pointer}_{d}(\mathsf{Array}_{d}(\tau, e))}{\Gamma \vdash \mathsf{Array}(e) : \mathsf{Pointer}_{d}(\tau)} \qquad \frac{\Gamma \vdash e \mathrel{\dot{\prec}} \mathsf{Pointer}_{p}(\mathsf{Array}_{p}(\tau, e))}{\Gamma \vdash \mathsf{Array}(e) : \mathsf{Pointer}_{p}(\tau)}$$

$$\frac{\Gamma \vdash e \mathrel{\dot{\prec}} \mathsf{Pointer}_{s}(\mathsf{Array}_{s}(\tau, e))}{\Gamma \vdash \mathsf{Array}(e) : \mathsf{Pointer}_{s}(\tau)}$$

### The DeRef operator

$$\frac{\Gamma \vdash e \mathrel{\dot{\prec}} \mathsf{Pointer}_{d}(\tau)}{\Gamma \vdash \mathsf{DeRef}(e) : \tau}\, |\tau|_{O} = 0 \qquad \frac{\Gamma \vdash e \mathrel{\dot{\prec}} \mathsf{Pointer}_{p}(\tau)}{\Gamma \vdash \mathsf{DeRef}(e) : \tau}$$

$$\frac{\Gamma \vdash e \mathrel{\dot{\prec}} \mathsf{Pointer}_{s}(\tau)}{\Gamma \vdash \mathsf{DeRef}(e) : \tau}\, |\tau|_{Y} = 0$$

### The Cast operator

Casts are conversions between different types. The translation into Core C has divided the occurences of Cast into *benign* ones which are cast between different basic types and casts between structurally equivalent pointer tyles, and *malign* ones which are everything else.

In the benign case the result should simply have the same binding-time type as the operand (ignoring the insignificant differences between the underlying C types). Everything malign is expected to be "dirty" so we copy them unchanged into $p_{res}$ by requiring that the operand an the result are both dynamically faithful. [|textbf]

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Cast}_{benign}(e) : \tau} \qquad \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash \mathsf{Cast}_{malign}(e) : \tau}\, \tau \in \mathbb{F}_{d} \wedge \tau' \in \mathbb{F}_{d}$$

### The Sizeof operator

Some occurences of `sizeof` in the subject program are recognized by the Core C translation to be parts of idioms for heap allocation or array size computations. These are converted to clean Core C constructs. The rest of them are translated into the Sizeof[15] operator.

As with malign Casts we always expect these remaining Sizeofs to be signs on "dirty tricks" being played; we take the size in $p_{res}$ of the dynamically faithful version of the operand type.

$$\frac{}{\Gamma \vdash \mathsf{Sizeof}(\tau) : \mathsf{Primitive}_{d}}\, \tau \in \mathbb{F}_{d}$$

[15] Presently the C-Mix$_{II}$ implementation supports two differnent Sizeof operators: SizeofExpr and SizeofType. SizeofExpr has been found to make it difficult or in some cases even impossible to print $p_{gen}$ and $p_{res}$ with correct lexical scoping, so in future versions of C-Mix$_{II}$ the `sizeof` keyword applied to an expression will be translated to the size of the type that C-Mix$_{II}$'s type checker finds for the argument expression.

### 3.3.2 Typing rules for lifts

$$\frac{\Gamma \vdash e : \tau \quad \vdash \tau < \tau'}{\Gamma \vdash e : \tau'} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \mathrel{:\!\prec} \tau}$$

$$\frac{\Gamma \vdash e : \mathsf{Pointer}_s(\tau) \quad \vdash \mathsf{Pointer}_r(\tau) < \tau'}{\Gamma \vdash e \mathrel{:\!\prec} \tau'} \qquad \frac{}{\vdash \tau < \tau}$$

$$\frac{}{\vdash \mathsf{Primitive}_s < \mathsf{Primitive}_d} \qquad \frac{}{\vdash \mathsf{Pointer}_p(\tau) < \mathsf{Pointer}_d(\tau)}$$

$$\frac{}{\vdash \mathsf{Pointer}_r(\tau) < \mathsf{Pointer}_p(\tau)}\,[*] \qquad \frac{}{\vdash \mathsf{Pointer}_r(\tau) < \mathsf{Pointer}_d(\tau)}\,[*]$$

The two last rules—marked "[*]"—can only be used when the pointer to be lifted is known only to point to objects in scope at the time of the lift.

### 3.3.3 Typing rules for statements

$$\frac{\Gamma \vdash e' \mathrel{:\!\prec} \mathsf{Pointer}_d(\tau) \quad \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Assign}(e', e).}\ |\tau|_O = 0$$

$$\frac{\Gamma \vdash e' \mathrel{:\!\prec} \mathsf{Pointer}_p(\tau) \quad \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Assign}(e', e).}$$

$$\frac{\Gamma \vdash e' \mathrel{:\!\prec} \mathsf{Pointer}_s(\tau) \quad \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Assign}(e', e).}\ |\tau|_Y = 0$$

$$\frac{\Gamma \vdash e : \mathsf{FunPtr}(\tau_1, \ldots, \tau_n \to \tau) \quad \forall i : \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \mathsf{Call}(-, e, e_1, \ldots, e_n).}$$

$$\frac{\Gamma \vdash e' : \mathsf{Pointer}_d(\tau') \quad \Gamma \vdash e : \mathsf{FunPtr}(\tau_1, \ldots, \tau_n \to \tau) \quad \vdash \tau < \tau' \quad \forall i : \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \mathsf{Call}(e', e, e_1, \ldots, e_n).}\ |\tau|_O = 0$$

$$\frac{\Gamma \vdash e' : \mathsf{Pointer}_p(\tau) \quad \Gamma \vdash e : \mathsf{FunPtr}(\tau_1, \ldots, \tau_n \to \tau) \quad \vdash \tau < \tau' \quad \forall i : \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \mathsf{Call}(e', e, e_1, \ldots, e_n).}$$

$$\frac{\Gamma \vdash e' : \mathsf{Pointer}_s(\tau) \quad \Gamma \vdash e : \mathsf{FunPtr}(\tau_1, \ldots, \tau_n \to \tau) \quad \vdash \tau < \tau' \quad \forall i : \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \mathsf{Call}(e', e, e_1, \ldots, e_n).}\ |\tau|_Y = 0$$

## 3.4 Safety rules

In this section we desctibe the *safety rules* that the binding-time annotated program must adhere to in addition to the binding-time type system defined above.

The first and second union rules (pages 80 and 81) can also be construed safety rules.

### 3.4.1 Pointer analysis

Most of the safety rules require that one is able to answer the question of where a given pointer-valued expression can point to. So as to make that possible,

C-Mix/II performs a **pointer analysis** immediately after the subject program has been translated to Core C.

The outcome[16] of the pointer analysis is that each of the expressions in the program is decorated with a **points-to set** that encompasses every object that the expression's value can possibly point to. (If the expression does not have pointer type, its points-to set is empty).

Like any other program analysis, the pointer analysis is only approximate. The points-to set for an expression may containn objects that its value cannot in fact point to when the subject program runs. It is never too small, however: unless the program does dirty things not blessed by the standard, pointers never point outside their points-to set.

For the purposes of the pointer analysis, a null pointer does not point to anything; thus there is no special indication in the points-to set that a pointer may be null.

Of course, the points-to sets cannot consist literally of objects, because objects only exist when the subject program runs and we are not running it. The real contents of a points-to set are *references* to objects.

More precisely, the points-to sets contain references to the Core C declarations that cause the pointed-to objects to exist.

### 3.4.2 Truly local variables

Most of the times we use the pointer analysis' results what we're really interested in knowing is not the precise identity of the objects that might be pointed to, but more general information such as

- May the pointer point to any global variable?
- May the pointer point to some unknown place that an external function gave the program a pointer to?
- May the pointer point to a local variable in the current stack frame (i.e., the function stack frame that is active when the expression is evaluated)?
- May the pointer point to a local variable in a *non*-current stack frame?

The discrimination between the latter two cases is not trivial. What the pointer analysis gives us is a set of declarations, and even if all of these declarations happen to be local variables in the function where we found the expression, we cannot be sure that the actual pointer at run-time will not point into non-current stack frames.

For an example, consider Figure 3.7. The pointer analysis annotates the expression "`pi`" in `foo` as being able to point only to `foo`'s own `i` variable. However, when the program runs, `pi` actually never points to the `i` in the current stack frame; only to the `i` in the *caller*'s stack frame.

We call a local variable **truly local** if no expression in the function where it is defined can point to other instances of that variable than the one in the current stack frame.

[16]It would take us too far to describe the inner workings of the pointer analysis here. The general approach is as described in Andersen (1994, Chapter 4), but the actual implementation in C-Mix/II is somewhat different because our Core C is different from Andersen's representation. The best reference for our actual pointer analysis at the time of this writing is the development notes that accompany the source code for C-Mix/II.

```
int foo(int *pi) {
  int i = 17 ;
  if( pi != NULL )
    *pi = 42 ;
  else
    foo(&i) ;
  return i ;
}
int pgm_P(void) {
  return foo(NULL);
}
```

*Figure 3.7: A program that illustrates the difficulty of deciding whether a pointer points into non-current stack frames.*

Obviously, in Figure 3.7, `i` is not a truly local variable. Most casually used temporary variables are truly local, however. The formal parameter `pi` is truly local[17]: no instances of `foo` have any way of learning the address of any other `pi` than its own.

After the pointer analysis, C-Mix/II performs a crude analysis[18] which is able to identify most of the truly local variables in the program. Again this analysis is only approximative: it may fail to flag a variable as truly local that in fact is.

### 3.4.3 The unique-return-state requirement

In Section 2.5.1 we defined the *unique-return-state requirement:*

- *The values of the static global variables upon exit from a function must be uniquely given as a function of their values before the call, and the static parameters to the function (and the result of the static external calls in the function).*

as an essential part of our techniques to handle function calls. Now we will discuss how to apply it to the full Core C language.

First, we note that whereas the rule talks about global variables, the reasoning in Section 2.5.1 actually applies to any variable that exists after the function has returned. Thus in Figure 3.8 the unique-return-state requirement also applies to what `bar` does to the `i` and `j` variables, and what `foo` (indirectly) does to `i`.

Intuitively, the unique-return-state requirement means that for the program in Figure 3.8 the `i` variable must be dynamic, because it depends on `d` what the execution of `foo` does to it. Well, really, the value of `i` happens to be `17` when `bar` returns irregardless of whether `foo` is called or not, but we should not expect C-Mix/II, a mere program, to be able to detect *that*.

[17]The *value* of `pi` may point to something that is not truly local, namely `i`. However, `pi` itself *is* truly local, because its *address* cannot be known to other `foo`s.

[18]Because of time constraints we shall not go into detail about how the truly-local analysis works. Alas, there is no better documentation for it than the development notes that accompany the C-Mix/II source

```
void bar(int *pi, int *pj) {
  *pi = 17 ;
  *pj = 42 ;
}
void foo(int *pi, int d) {
  int j = 2 ;
  ints q = 0 ;
  if ( d )
    q = 1 ;
  if ( q )
    bar(pi,&j);
}
void goal(intd d) {
  int i = 17 ;
  foo(&i,d);
}
```

*Figure 3.8: A program that we use as an example of how to enforce the unique-return-state requirement.*

However, we would expect `j` to remain static because `bar` has, in itself, predictable effects on it and `j` is not part of `foo`'s return state at all.

Because the unique-return-state requirement is not, as it stands, decidable, we reformulated it as:

- *Non-local side effects under dynamic control must not appear to static variables.*

This version is decidable, but it is also more conservative than the idealised unique-return-state requirement. For example, in Figure 3.8 the rule of non-local side effects will also force `j` to be dynamic, because the assignment to it in `bar` *is* indeed under dynamic control.

Our task now is to reinterpret—in the light of the binding-time type system we have presented—the key concepts of the rule of non-local side effects: (1) "non-local side effect", (2) "under dynamic control", and (3) "static variables".

### What is a non-local side effect?

In Section 2.5.1, a **non-local side effect** was a "an assignment to a global variable". That was an acceptable definition at that time because an assignment in Chapter 2 either affected a specific global variable or a specific local variable.

In Core C, an assignment is always through a pointer to the affected object, so we need a more refined definition. The property we intutively need is "an assignment through a pointer that might point to something that still exists after the current function returns".

In more specific terms we define

- *A non-local side effect is a statement whose destination expression's points-to set includes something that is not truly local in the current function.*

Note that for a Call statement we do not count the side effects possibly committed by the called function as happening in the Call statement itself. There are good reasons why we really should (see Box 3.8), but we discovered them too close to this report's deadline to integrate them into the rest of the report.

#### What is under dynamic control

The discussions of dynamic control in Sections 2.3.2 and 2.4.3 are still valid for Core C.

To summarize: a basic block is **under dynamic control** if

1. it is the target of a conditional jump with a $\mathsf{Primitive}_d$ condition, or
2. is is the target of a conditional or nonconditional jump from another basic block that is already under dynamic control, or
3. it is found in a function that is called from some basic block that is under dynamic control.

A basic block is under **local dynamic control** if it is under dynamic control even when not using (3).

Note here that Andersen (1994, section 3.5.2) also used the phrase "dynamic control" in his discussion about what we call the unique-return-state requirement. He got the definition wrong, though—he based it on a definiton of "control dependence" (Andersen 1994, Section 6.2.4) which was taken from a work on parallelizing compilers. That definition assumed that the influence of a conditional only reached as far as to the point where the two branches merge together in the flow graph. In the partial evaluation context, however, the speculative specialization of the branches need not end simply because the source code merges.

The program in Figure 3.8 is an example: the call to `bar` would not be under dynamic control by Andersen (1994)'s definition.

#### What are static variables

In Chapter 2, a variable was either static or dynamic. Due to the richer binding-time type system we use in C-Mix$_{/II}$, it needs to be clarified which variables are allowed to be changed by a non-local side effect under dynamic control.

The signature of the variable's binding-time type $\tau$ provides the answer: for a non-local side effect under dynamic control, $|\tau|_{\mathbf{0}}$ must be 0.

### 3.4.4 Residual scope issues

As mentioned on page 68, a pointer that is lifted from $\mathsf{Pointer}_s$ to $\mathsf{Pointer}_d$ must be known to point to something that is in scope at the residual place of the lift. The purpose of that restriction is to avoid cases like in Figure 3.9[19].

The task divides naturally into two subquestions:

- *Which objects may a particular pointer-valued expression point to?*

[19] Andersen (1994, Section 3.6.2) tried to avoid this problem by deciding that function arguments must not be static pointers to dynamic data. Apparently he overlooked the possibility that the offending pointer could have been communicated to the function through a global variable.

## Box 3.8—A LESS CONSERVATIVE ALTERNATIVE RULE OF NON-LOCAL SIDE EFFECTS

The trouble with the interpretation of the rule of non-local side effects is that it is too conservative. For example, in Figure 3.8 it requires `j` to be dynamic, thoug a dynamic `j` would not have violated the unique return state of any of the programs function's.

Basically the fault is that the rule says the assignment must be dynamic if the assigned object still exists when *the function that does the side effect* returns. It would be enough to restrict oneself to the case that the assigned object still exists when *the function that contains the dynamic conditional* returns.

One way to achieve that would be:

- Before the BTA, compute for each function a set of objects that the function may (directly or indirectly) commit side effects to, and that still exists after the function has returned.

  C-Mix/II already contains code to do this computation; currently we use the results for speeding up the memoization process at specialization time.
- Amend the definition of "non-local side effect" such that a call to a function whose side-effect set includes `x` is in itself a non-local side effect on `x`, unless `x` is truly local in the calling function.
- Non-local side effects are now only hazardous when they are under *local* dynamic control.

That approach would have the quite significant afvantage that it is able to take into account side effects done by external functions, which C-Mix/II does not do currently (that is a known bug which in certain cases causes incorrect residual programs to be generated).

There would also be some slight disadvantages. First, the time efficiency of the BTA might suffer. The arrow generation (see Chapter 4) for a call statement could no longer be expected to take constant time if it involved inspecting each member of a potentially large side-effect set.

Second, the user feedback from the BTA would lose quality. At the moment when the rule of non-local static side effects is invoked, the BTA trace presented to the user pinpoints the exact location of the assignment to the variable that is forced to be dynamic, and traces the dynamic control property up the call graph. That kind of feedback would not be easy to give with the alternative rule presented in this box.

The main reason, however, why this is not already C-Mix/II's default behavior is that we invented it too late to have time to integrate it in the rest of this report, let alone implement it.

```
void assign(int *ptr,int data) {
  *ptr = data ;
}
void pgm_Q(int_d d) {
  int i ;
  assign(&i,d) ;
}
```

```
void assign_i(int data) {
  i = data ;
}
void pgm_Q_res(int d) {
  int i ;
  assign_i(d);
}
```

*Figure 3.9: An example of what would happen if pointers were allowed to be lifted without being in scope. Then* `ptr` *can be a* $\mathsf{Pointer}_s(\mathsf{Primitive}_d)$ *and only lifted at the place of the actual assignment. The residual function will contain a reference to another function's local variable.*

- *Will all of those be in scope at the (potential) lift point in the residual program?*

The first question is answered by the pointer analysis (Section 3.4.1). To answer the second question: An object is in scope if it is either global or truly local (see Section 3.4.2) in the function where the potential lift site occurs.

One might note that this is a rather conservative estimate: if the function where we find the potential lift location is always inlined (Section 2.4.4), the caller's local variables will also be in scope at the residual location of the lift.

We speculate that it could lead to substantially improved specialization for some kinds of program if the binding-time analysis could take advantage of this scope widening in inlined functions. `pgm_Q` of Figure 3.9 is a prototypical example where the function `assign` manipulates a local variable in its caller through a pointer argument. If `assign` is inlined by C-Mix/II the residual code will still contain an explicit pointer variable. That might confuse compilers, leading to less efficient register allocation than if the name of `i` was simply mentioned in the residual code for `assign` (which would be safe when it is inlined).

It is still an open question how one might construct an analysis that identified cases like that, though.

# Chapter 4

# Synthesis of a BTA algorithm

In the preceding chapter we have defined conditions that define what an acceptable binding-time annotation of a program is. Now we are ready to develop an algorithm for **binding-time analysis** (BTA), which automatically decides on a "best possible" binding-time annotation, given a set of user requirements.

From the BTA's point of view, the "best possible" binding-time annotation means the one that lets as many as possible of the program's operations be done in $p_{gen}$.

In many cases this is not desirable because it results in

excessive *code bloat*: $p_{res}$ contains a lot of instances of the same subject program code. Even though not all of them is executed in a single run of the residual program, the sheer number of them may make $p_{res}$ so unwieldy and slow to load that the fact that it performs fewer actual operations than the subject program is not of much help.

*infinite specialization*: $p_{gen}$ never finishes a residual program but keeps adding new entries to a pending list or creating new residual functions.[1].

In these cases the user of C-Mix/II has to manually specify constraints on the BTA result sush that the "best possible" is something that works in practise.

## 4.1 Overall strategy

We start with a simplified example: how to binding-time analyze the program in Figure 2.1 which is repeated in Figure 4.1. The user has specified binding times for the program's input and output. We are to find the "best possible" (i.e., most static) binding-time annotation for it.

We can go through the program using binding-time typing rules like those from Chapter 3. That tells us which properties a binding-time annotation must have:

a. If `y` or `5` is dynamic, then so must the expression `y*5` be.

[1]The distinction between infinite specialization and severe code bloat—trying to create a finite $p_{res}$ that is simply too big to be created using the resources available to $p_{gen}$—is blurred and not very important.

```
int_d pgm_A(int x_d,int y_s) {
  int z ;
  y = y * 5 ;
  z = x + 1 ;
  return z - y ;
}
```

*Figure 4.1: Our first BTA example*

b. If `y*5` is dynamic then so must `y` be.
c. If `x` or `1` is dynamic then so must the expression `x+1` be.
d. If `x+1` is dynamic then so must `z` be.
e. If `z` or `y` is dynamic then so must the expression `z-y` be.
f. If `z-y` is dynamic then so must `pgm_A`'s return value be.
g. `pgm_A`'s return value must be dynamic.
h. `x` must be dynamic.
j. `y` must be static.

Conditions (a)–(f) come from blindly stating what binding-time consequences each operator and statement in the program has. Conditions (g)–(j) are what the user specified. Our task now is to find the least static binding-time annotation that fulfills these conditions.

The most popular way to accomplish this—though not the way followed by C-Mix/II—is to think of the conditions listed above as **constraints**. We will briefly summarize the ideas behind constraint-based BTA, so as to make the differences from what we do in C-Mix/II more explicit.

### 4.1.1 Constraint-based BTA

When we view the conditions as constraints, what they specify is that a certain relation between two or more *binding-time types* should hold. We can do binding-time analysis by doing **constraint normalization**, that is, by systematically deriving new constraints and throwing out redundant ones, until the constraint set we are left with has an appropriately simple structure.

In the example we could start by throwing out constraint (f), because any binding-time annotation that meets (g) also meets (f), so deleting (f) will not change the set of solutions. Then we can derive a new constraint from (h) and (c):

k. `x+1` must be dynamic.

After a series of such steps we end up with a set of constraints saying

a. If `y` or `5` is dynamic, then so must the expression `y*5` be.
b. If `y*5` is dynamic then so must `y` be.
g. `pgm_A`'s return value must be dynamic.
h. `x` must be dynamic.

j. y must be static.

k. x+1 must be dynamic.

l. z must be dynamic.

m. z-y must be dynamic.

from which it can be easily (?) seen that the most static solution is to annotate the return value, x, x+1, z, and z-y as dynamic.

With appropriate types of constraints and normalization rules, this technique can be extended to BTAs for more complex binding-time type systems. It can also handle BTA for untyped languages such as Lisp or Prolog, and it is quite fast when implemented intelligently.

Constraint-based BTA also has its drawbacks, however. As the binding-time type system gets more complex, the necessary constraint types also get complex. So does the constraint normalization rules, the proof that the constraint normalization actually terminates, and the algorithms that are needed to normalize constraints efficiently.

This means that a constraint-based BTA for a rich binding-time type system is hard to design and implement. It is also hard for the user of the partial evaluator to understand why some variable unexpectedly becomes dynamic. Constraint normalization algorithms are not designed to provide intuitively easy arguments about the why's of the analysis.

Until 1998 the BTA in C-Mix/II was based on a constraint normalization algorithm derived from Henglein (1991). Over the years we realized, however, that C-Mix/II did not actually need the chief strength of Henglein's algorithm, which is the ability to analyze weakly typed languages. This insight arised out of thought on how to give good user feedback from the BTA, and led to the development of the following, more simplistic, approach:

### 4.1.2 Logic-based BTA

The best way to understand C-Mix/II's BTA is to view conditions (a)–(j) as relations not between binding-time types but between *propositions about the binding-time annotations*. For example, condition (f) relates the assertations, "y is an $\mathtt{int}_\mathrm{d}$" and "y+z is an $\mathtt{int}_\mathrm{d}$" by requiring that if the first is true, then the second one must be, too.

One way to express this is that we are building a *formal theory* about what the solution should look like, and that the conditions provide the inference rules for that theory. Some of the conditions actually map to two inference rules—e.g., condition (c) corresponds to the two rules,

$$\frac{\mathtt{x}\text{ is an }\mathtt{int}_\mathrm{d}}{\mathtt{x+1}\text{ is an }\mathtt{int}_\mathrm{d}} \qquad\qquad \frac{\mathtt{1}\text{ is an }\mathtt{int}_\mathrm{d}}{\mathtt{x+1}\text{ is an }\mathtt{int}_\mathrm{d}}$$

Note that all of the propositions have the form, "*something* is an $\mathtt{int}_\mathrm{d}$". There are only finitely many of them, and it can be determined in advance of the analysis what they are. Similarly, there are only finitely many rules.

The entire theory about pgm_A is shown graphically in Figure 4.2. The vertices of the graph are the propositions (where the "is an $\mathtt{int}_\mathrm{d}$" part has been dropped for space reasons) and the edges are the rules.

One would expect conditions (g) and (h) to become axioms of the theory. However, for technical reasons it is convenient to introduce a single axiom "$\top$"

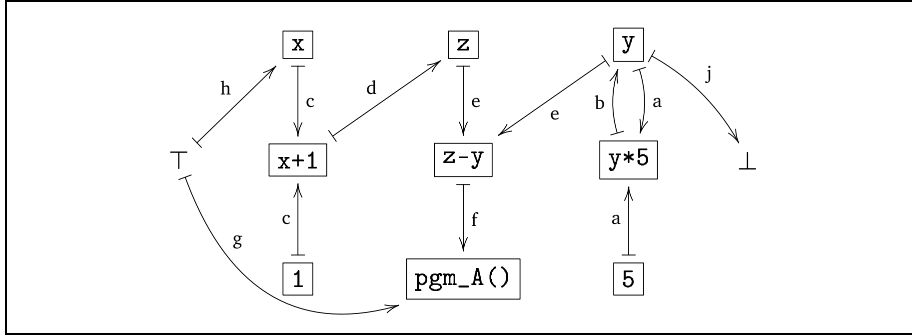



*Figure 4.2: A graphical representation of the logic-based BTA for* `pgm_A`.

($\top$ then formally becomes another proposition of the theory) and rephrase (h) to read, "if $\top$ then `x` must be an $\mathtt{int}_\mathrm{d}$".

Condition (j) is somewhat special. It turns out to create problems to add the proposision "`y` must be an $\mathtt{int}_\mathrm{s}$". In any case, (j) does not actually affect what the minimal solution *is*, because a binding time we do not know anything about explicitly might as well be static. What (j) really tells us is that if `y` needs to be dynamic there is *no* solution, minimal or not.

So we rephrase (j) as, "if `y` must be an $\mathtt{int}_\mathrm{d}$ then there is no solution" and introduce the proposition "$\perp$" to mean, "there is no solution".

Now, to finish the binding-time analysis, we can simply consider Figure 4.2 as a oriented graph and ask which vertices are reachable from $\top$. Exactly those vertices are provable in our theory; they name exactly the places where we need annotate something as $\mathtt{int}_\mathrm{d}$; the places not mentioned by provable propositions can be annotated $\mathtt{int}_\mathrm{s}$. If $\perp$ is a provable we give up and signal an error to the user.

**Proofs**

One nice feature of this BTA principle is that it naturally supports the notion of proof. The user of C-Mix/II may wonder why, say, `z` becomes dynamic. C-Mix/II can then produce a proof:

1. By (h), `x` must be dynamic.
2. Hence, by (c), `x+1` must be dynamic.
3. Hence, by (d), `z` must be dynamic. □

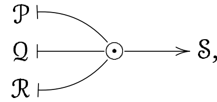
Finding a good proof is as simple as finding a shortest path from $\top$ to the queried vertex in the BTA graph.

Also, when the task is impossible (i.e., $\perp$ is reachable), C-Mix/II can produce a proof that, step by step, explains to the user why her request cannot be fulfilled. This is much more useful than a simple oracular "conflicting user annotations" message.

### 4.1.3 Application to the full binding-time type system

The rest of this chapter deals largely with how to construct theories for annotations with the full binding-time type system.

Most of the complexity will be handled by introducing more propositions and a greater number of inference rules. In some cases we need to introduce rules with more than one premise; we notate these as, e.g.,



meaning that $\mathcal{S}$ is provable if all of $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$ are provable. In Section 4.4 we discuss how to deal with such arrows in the solution phase.

Because the word "rule" has many other uses we shall call the inference rules of the annotation theory **arrows** in the rest of this chapter.

## 4.2 Design principles

### 4.2.1 Input to the BTA

We assume that the subject program has already been translated to Core C and typed; that is, every declaration, expression, and subexpression in the program has been decorated with a (yet not binding-time annotated) Core C type.

A Core C type is represented as a finite graph where the nodes are type constructors and the edges go from a constructor to its operand types. This graph is mostly tree-shaped, except where recursive types necessiate cycles. We use $\mathsf{T}$ variables to range over nodes in these graphs. For an expression $e$, $\mathsf{T}_e$ will mean the root of the type graph that represent $e$'s type.

In principle, the graphs that represent the types of different (sub)expressions or declarations are all disjoint (i.e., no $\mathsf{T}$ is a member of the decoration of two different Core C elements). In the implementation we reduce the memory and time usage of C-Mix/II somewhat by allowing type nodes to be shared between expressions in some cases; more about that in Section 4.4.1.

The "user annotations" that the C-Mix/II user supplied are converted to special decorations on declarations and $\mathsf{Call}$ statements before the binding-time analysis.

### 4.2.2 Catalogue of propositions

As described above, the entire analysis shares the meta-propositions $\top$ and $\bot$. The rest of the propositions are derived from parts of the subject program.

#### Propositions about types

Each type node $\mathsf{T}$ "owns" a number of propositions that say something about which specialization variant it should be annotated to be. It depends on the type constructor which propositions a given node has, as shown in the middle column of Table 4.1.

The general naming scheme for these propositions is "$\boxed{\mathsf{T}}_{\mathrm{x/y}}$", which means, "$\mathsf{T}$ is a $\mathsf{Someconstructor}_{\mathrm{x}}$ or a $\mathsf{Someconstructor}_{\mathrm{y}}$".

| T is | Owned propositions | $\boxed{\frac{\mathsf{T}}{\mathbf{Y}\neq 0}}$ is | $\boxed{\frac{\mathsf{T}}{\mathbf{Y}=1}}$ is | $\boxed{\frac{\mathsf{T}}{\mathbf{O}=0}}$ is |
|---|---|---|---|---|
| Primitive | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ |
| Enum | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ |
| Pointer(...) | $\boxed{\frac{\mathsf{T}}{\text{r/p/d}}}, \boxed{\frac{\mathsf{T}}{\text{p/d}}}, \boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{p/d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{p/d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ |
| Array(T′, $e$) | $\boxed{\frac{\mathsf{T}}{\text{p/d}}}$ | $\boxed{\frac{\mathsf{T}'}{\mathbf{Y}\neq 0}}$ | $\boxed{\frac{\mathsf{T}}{\text{p/d}}}$ | $\boxed{\frac{\mathsf{T}'}{\mathbf{O}=0}}$ |
| Struct(...) | $\boxed{\frac{\mathsf{T}}{\text{p/d}}}, \boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{p/d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{p/d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ |
| Union(...) | $\boxed{\frac{\mathsf{T}}{\text{p/d}}}, \boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{p/d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{p/d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ |
| FunPtr(...) | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ | $\boxed{\frac{\mathsf{T}}{\text{d}}}$ |

*Table 4.1: Propositions owned by type nodes for different constructors. Notice that the definition of the alias propositions is recursive when the type constructor is* Array.

The representation of specialization variants is slightly context-dependent: if $\boxed{\frac{\mathsf{Array}(\mathsf{T}', e)}{\text{p/d}}}$ is provable, the specialization variant actually used depends on whether T′ is annotated to have signature $(1, 0)$ or $(1, 1)$.

Note that a binding-time type is dynamically faithful if all of the associated propositions are provable and statically faithful if none of them are.

### Alias propositions

Table 4.1 also defines a **alias propositions** $\boxed{\frac{\mathsf{T}}{\mathbf{Y}\neq 0}}$, $\boxed{\frac{\mathsf{T}}{\mathbf{Y}=1}}$, and $\boxed{\frac{\mathsf{T}}{\mathbf{O}=0}}$, which are alternate names for some of T's owned propositions. The idea is that, for example, $\boxed{\frac{\mathsf{T}}{\mathbf{Y}=1}}$ is a short way of naming either $\boxed{\frac{\mathsf{T}}{\text{d}}}$ or $\boxed{\frac{\mathsf{T}}{\text{p/d}}}$, depending on which type constructor T has.

The intuitive meaning of the alias propositions are:

$\boxed{\frac{\mathsf{T}}{\mathbf{Y}\neq 0}}$ means $|\tau|_{\mathbf{Y}} \neq 0$.

$\boxed{\frac{\mathsf{T}}{\mathbf{Y}=1}}$ means $|\tau|_{\mathbf{Y}} = 1$.

$\boxed{\frac{\mathsf{T}}{\mathbf{O}=0}}$ means $|\tau|_{\mathbf{O}} = 0$.

for the binding-time type $\tau$ that the type rooted at T gets annotated to be.

By comparing with Table 3.6 (on page 86) one can see that this intuitive meaning actually agrees with the definitions in Table 4.1.

#### Expressions and lifts

We use the convention that the propositions for an expression's type describe the binding-time annotation of that type *before* any lifts of the expression's value.

The alternative choice—letting the expression's type annotation describe the binding-time type *after* lifting—would interact inconveniently with the fact that it is the *context* of an expression that decides whether a carry is allowed.

#### Propositions for Enum declarations

For each Enum declaration $\eta$ in the program there shall be a proposition $\boxed{\frac{\texttt{enum}\ \eta}{\mathrm{d}}}$ whose intuitive meaning is, "the Enum cannot be declared in $p_{gen}$ (because one enumeration constant expression is a $\mathsf{Primitive}_{\mathrm{d}}$)".

#### Propositions about dynamic control

For each basic block $\mathsf{b}$ in the program there shall be a proposition $\boxed{\frac{\mathsf{b}}{\mathrm{D.C.}}}$ whose intuitive meaning is, "$\mathsf{b}$ is under local dynamic control".

Also, for each function $\mathsf{f}$ in the program there shall be a proposition $\boxed{\frac{\mathsf{f}}{\mathrm{D.C.}}}$ whose intuitive meaning is, "there is a call to $\mathsf{f}$ under dynamic control".

Thus, a statement in a basic block $\mathsf{b}$ in function $\mathsf{f}$ is under (local or non-local) dynamic control if either $\boxed{\frac{\mathsf{b}}{\mathrm{D.C.}}}$ or $\boxed{\frac{\mathsf{f}}{\mathrm{D.C.}}}$ is provable.

## 4.3 Arrow generation

This section decribes how to generate the arrows in the BTA graph, given a subject program and propositions as described in the previous section.

Each type node, (sub)expression and statement in the subject program give rise to a number of arrows. It is in principle irrelevant in which order arrows are generated for different parts of the subject program.

### 4.3.1 Forcing identity of binding-time types

In several situations we have two type nodes, $T_1$ and $T_2$, and want that the two types rooted there should be annotated with exactly the same binding-time type. This only happens when the un-annotated types are known in advance to have identical structure.

The two types can be forced to be identically annotated by adding an appropriate set of arrows to the BTA graph. We call this set $[T_1 \equiv T_2]$.

Intuitively, $[T_1 \equiv T_2]$ shhoud consist of an arrow from each proposition owned by a node in the $T_1$ tree to its structural companion in the $T_2$ tree, and vice versa. This makes sure that the propositions that are provable about $T_1$ are the same as those that are provable about $T_2$.

In practise, the existence of recursive types means that we can not be sure that there is a one-to-one correspondence between nodes in the two type graphs.

So in the implementation we use a global union–find structure to gradually build an equivalence relation defining groups of type nodes that need to be annotated identically.

We can then use the following algorithm to compute an appropriate $[\mathsf{T}_1 \equiv \mathsf{T}_2]$ set:

1. If $\mathsf{T}_1$ and $\mathsf{T}_2$ are in the same partition of the union–find structure, then do nothing. They are already known to be identically annotated.
2. Otherwise, union the partitions in the union–find structure that contain $\mathsf{T}_1$ and $\mathsf{T}_2$. (Intuitive meaning: we promise to arrange for the arrows $[\mathsf{T}_1 \equiv \mathsf{T}_2]$ to be generated before the algorithm terminates.
3. Generate arrows between the propositions directly owned by the type nodes $\mathsf{T}_1$ and $\mathsf{T}_2$.
4. If the type nodes have children (i.e., if they are not Primitive, Abstract, or Enum nodes) then recursively compute $[\mathsf{T}_1' \equiv \mathsf{T}_2']$ for each matching pairs of children $\mathsf{T}_1'$ and $\mathsf{T}_2'$, and add it to the output.

Because each instance of the algorithm that reaches step (2) performs a nonredundant union operation, it is immediately clear that the total number of arrows added to any $[\mathsf{T}_1 \equiv \mathsf{T}_2]$ set during the BTA is linear in the number of type nodes in the entire subject program. This argument also shows that the algorithm terminates.

### 4.3.2 Ensuring that binding-time types are well-formed

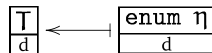
For each type node $\mathsf{T}$ in the program, a number of arrows may be necessary to ensure that the binding-time type it represents is well-formed and that natural relations among $\mathsf{T}$'s owned proposition hold.

It is a design invariant that the well-formedness arrows should make sure that $\frac{\mathsf{T}}{\mathbf{Y}\neq 0}$ is (perhaps indirectly) derivable from either of $\frac{\mathsf{T}}{\mathbf{Y}=1}$ or $\frac{\mathsf{T}}{\mathbf{O}=0}$.

Of course, which well-formedness arrows we need is determined from the type constructor of $\mathsf{T}$. When reading the following walkthrough it may be helpful to consult Tables 3.6 (page 86) and 4.1 (page 102).

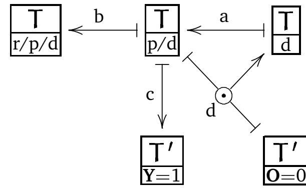
Primitive or Abstract: No well-formedness arrows are needed at all.

Enum($\eta$): We need a single arrow:

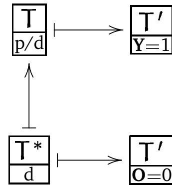



Pointer($\mathsf{T}'$): We need four arrows:

- a,b: to ensure a sane relationship between $\frac{\mathsf{T}}{\mathrm{r/p/d}}$, $\frac{\mathsf{T}}{\mathrm{p/d}}$, and $\frac{\mathsf{T}}{\mathrm{d}}$
- c: to make sure that $\mathsf{Pointer}_\mathrm{d}$ and $\mathsf{Pointer}_\mathrm{p}$ are only used when $|\tau'|_\mathbf{Y}$ of the pointed-to type is 1:
- d: to make sure that $\mathsf{Pointer}_\mathrm{p}$ is only used when the pointed-to type has a concrete model object.

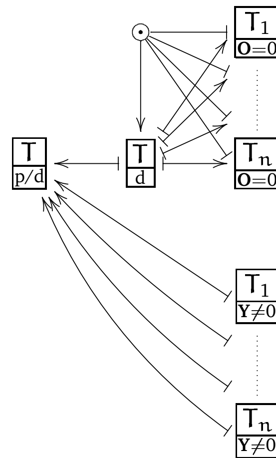



Array$(\mathsf{T}', e)$: A single arrow is needed to ensure that $\mathsf{Array_d}$ and $\mathsf{Array_p}$ is only used when $|\tau'|_{\mathbf{Y}}$ of the element type is 1. If the size expression is a $\mathsf{Primitive_d}$ the array must be an $\mathsf{Array_d}$. That needs two more arrows. Calling the type node of the size expression $\mathsf{T}^*$:




Struct$(\mathsf{T}_1, \ldots, \mathsf{T}_n)$: One set of arrows is needed to make sure that the type of the struct becomes $\mathsf{Struct_p}$ or $\mathsf{Struct_d}$ whenever one of the members have a dynamic component. Another set of arrows make sure that $\boxed{\frac{\mathsf{T}}{\mathrm{d}}}$ is provable exacly when none of the member types have static components.

Last, there should be an arrow from $\boxed{\frac{\mathsf{T}}{\mathrm{d}}}$ to $\boxed{\frac{\mathsf{T}}{\mathrm{p/d}}}$. It will be redundant if there are any members (because $\boxed{\frac{\mathsf{T}_i}{\mathbf{O}=0}} \vdash \boxed{\frac{\mathsf{T}_i}{\mathbf{Y}\neq 0}}$ for each $i$), but memberless Structs are used to represent `structs` that have no definitions in the subject program.




Union$(\mathsf{T}_1, \ldots, \mathsf{T}_n)$: We need the same arrows as for Struct, plus some more to help enforce the union rules.

The first union rule (page 80) can be enforced with an appropriate number of $[(\mathsf{T}_i)_j \equiv (\mathsf{T}_{i'})_j]$ arrow sets between relevant member type for those $\mathsf{T}_i$ union members that are themselves Struct nodes.

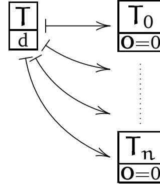
The second union rule (page 81) calls for a numer of arrows of the

form

$$T \longmapsto \boxed{T^*}_{o=0}$$

where $T^*$ range over type nodes that represent the types pointed to by the pointers affected by the second union rule.

$\mathsf{FunPtr}(T_1, ..., T_n \to T_0)$: This is simple: if $T$ is annotated as $\mathsf{FunPtr}_d$, all of the $T_i$s (including $T_0$) must be annotated to be dynamic:




In the following sections, the well-typedness arrows are sometimes shown as dotted arrows

$$\mathcal{P} \longmapsto \mathcal{Q}$$

where they are thought relevant.

### 4.3.3 Arrow generation for lifts

We use the notation $[T_1 \leqq T_2]$ to denote a set of arrows that make sure that the type rooted at $T_1$ is annotated to either be the same as or liftable into what the type rooted at $T_2$ is annotated to. As with $[T_1 \equiv T_2]$, it is presupposed that the unannotated types are the same.

Types Abstract, Enum, Array, Struct, Union, and FunPtr cannot be lifted, so for these types $[T_1 \leqq T_2]$ is simply $[T_1 \equiv T_2]$.

When the types are Primitive, $[T_1 \leqq T_2]$ will simply consist of the single arrow $\boxed{T_1}_{d} \longmapsto \boxed{T_2}_{d}$.

Matters are more complex when $T_1$ and $T_2$ are Pointer types. Figure 4.3 shows the derivation of the necessary arrows. The exact number of arrows depend on whether the pointer to be lifted is in scope and whether a carry is allowed.

Technically the lift arrows allow a $\mathsf{Pointer}_s$ to be carries into $\mathsf{Pointer}_r$. Strictly speaking that is not a valid carry, but we shall only use carries in situations where we explicitly disallow a $\mathsf{Pointer}_r$ result anyhow.

### 4.3.4 Forcing binding-time types to be faithful

The representation scheme defined Table 4.1 implies that when all the propositions owned by some type tree are provable, the corresponding binding-time type is dynamically faithful. Similarly, when none of them are provable the binding-time type is statically faithful.

Forcing a binding-time type to be faithful is therefore simply a question of letting all of the associated propositions depend on each other so that they are either all provable or all not provable. We denote the set of arrows that does this by $[T \in \mathbb{F}]$.

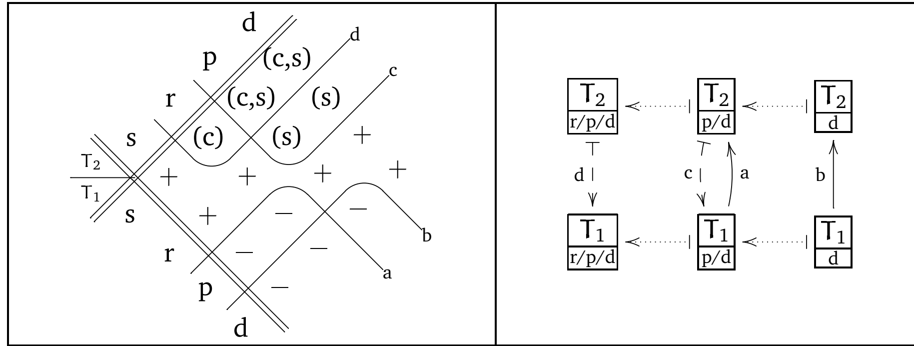



*Figure 4.3: Construction of arrows for lifting a pointer type. The diamond-shaped diagram to the left is a design sketch showing which combinations of* $\mathsf{T}_1$ *and* $\mathsf{T}_2$ *variant should be allowed by the network for* $[\mathsf{T}_1 \leqq \mathsf{T}_2]$*. "(c)" combinations are only allowed when for carries; "(s)" combinations are only allowed when the pointer to be lifted is in scope. Each of the arrows labeled* a *through* d *excludes a section of the diagram, as indicated by the correspondingly labeled lines on the sketch. Arrow* c *is not present when the pointed-to value is known to be in scope. Arrow* d *is not present for carries where the pointed-to value is known to be in scope.*

### 4.3.5 Arrow generation for expressions

Each operator in each expression of the program gives rise to a number of arrows, as defined in this section.

#### The Var operator

When $e = \mathsf{Var}(x)$, let $\mathsf{T}_x$ be the type node associated with $x$'s declaration and $\mathsf{T}_e = \mathsf{Pointer}(\mathsf{T}')$ be the type node associated with $e$.

We then naturally need $[\mathsf{T}' \equiv \mathsf{T}_x]$ but no other arrows except for well-formedness of the involved types: the typing rules on page 86 allow any Pointer variant for $\mathsf{T}_e$.

#### The FunAdr operator

When $e = \mathsf{FunAdr}(f)$, let $\mathsf{T}_x$ be the type node associated with f's declaration. This simply needs to have same binding-time type as the created pointer, so we simply generate $[\mathsf{T}_e \equiv \mathsf{T}_f]$.[2]. The result needs to be a $\mathsf{FunPtr}_s$, so we also generate

$$\boxed{\begin{array}{c}\mathsf{T}_e\\ \hline \mathrm{d}\end{array}} \longmapsto \bot$$

[2]In real life the situation is slightly more complex because what we write as FunPtr(*stuff*) is really represented in C-Mix$_{\mathrm{II}}$ as FunPtr(Fun(*stuff*)) and the FunPtr node is not part of the type information for f's declaration

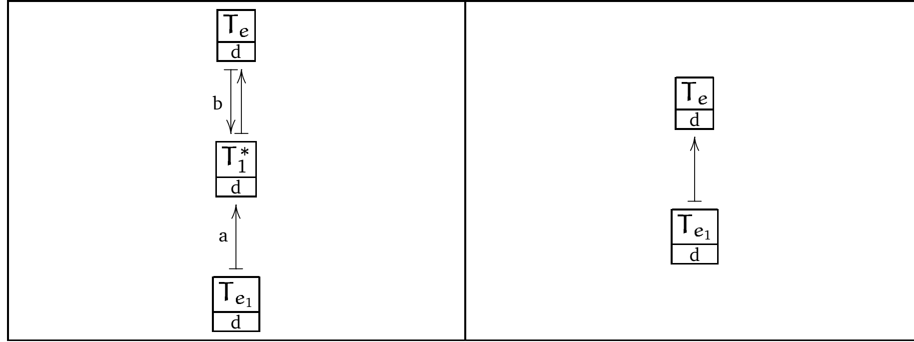
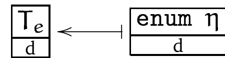



*Figure 4.4: Arrows for the expression* $e = \mathsf{Unary}(\diamond, e_1)$. *The rules on the left-hand side assume a hypothetical* $T_1^*$ *node denoting the lifted binding-time type of the operand. On the right-hand side the proposition associated with that node has been optimized out of the arrow network.*

### The EnumConst operator

For $e = \mathsf{EnumConst}(\eta, i)$ we generate




(remember that this is not implicit in the well-formedness of the type, because the type of $e$ is `int` rather than the `enum` type itself).

### The Const operator

For $e = \mathsf{Const}(c)$ we generate $[T_e \in \mathbb{F}]$.

### The Null operator

Null pointers can have any type, so we need no addition to the well-formedness arrows.

### The Unary operator

We need arrows as indicated on Figure 4.4:

a. $[T_{e_1} \leqq T_1^*]$: the operand's type may be lifted before the operation.
b. The lifted operand should have the same specialization variant as the result.

We first apply the necessary arrows to the hypothetical situation that the lifted operand is represented by a type node $T_{e_1}^*$ of its own. Then we can simplify the arrow network to the equivalent but simpler one shown on the right-hand side of Figure 4.4.

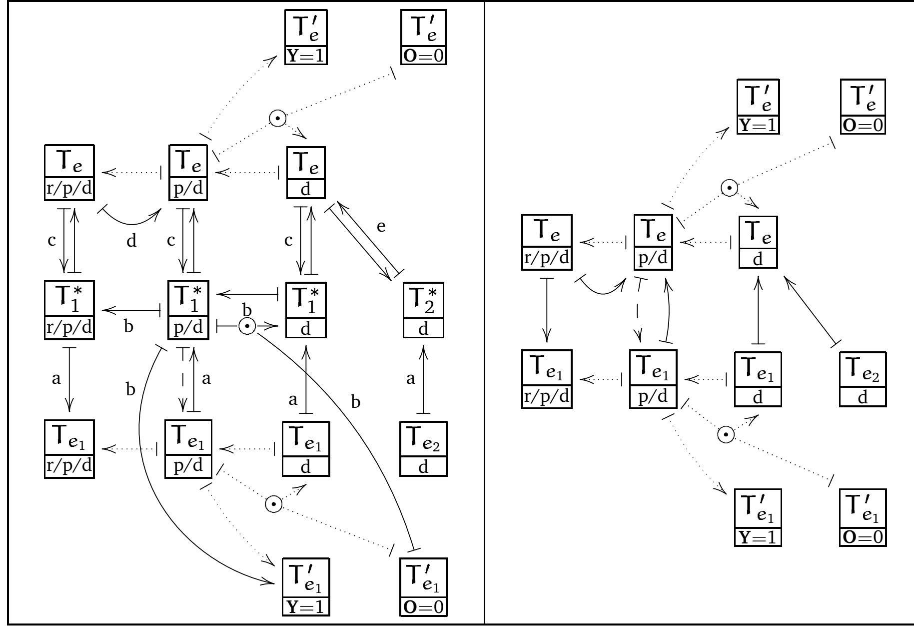



*Figure 4.5: Arrows for the expression* $e = \mathsf{PtrArith}(e_1, \circ, e_2)$*. The general organization of this figure, and of most of the following figures where arrow networks are constructed, is like the organization of Figure 4.4.*

### The PtrArith operator

Of course we need equality $[T'_e \equiv T'_{e_1}]$ between the types pointed to by the first argument and the result. The rest of the arrows, shown on Figure 4.5, are derived from the typing rules on page 87:

a. $[T_{e_i} \leqq T_i^*]$: the operands may be lifted. The dashed arrow is present if the first operand is not known to point to an object in scope.

b. The lifted operands should be well-formed. (In the simplified network to the right on Figure 4.5 all of these arrows have been optimized away because of (c) and the fact that $[T'_e \equiv T'_{e_1}]$ makes $\boxed{T'_e}$ $\boxed{Y{=}1}$ and $\boxed{T'_e}$ $\boxed{O{=}0}$ behave like $\boxed{T'_{e_1}}$ $\boxed{Y{=}1}$ and $\boxed{T'_{e_1}}$ $\boxed{O{=}0}$).

c. The lifted first operand always has the same type as the result.

d. $\mathsf{Pointer}_r$ is not a possible result type; the other variants are.

e. The lifted second operand is $\mathsf{Primitive}_d$ exactly when the result is $\mathsf{Pointer}_d$.

### The PtrCmp operator

The necessary arrows in addition to $[T'_{e_1} \equiv T'_{e_2}]$ are shown on Figure 4.6 and are derived from the typing rules on page 88 as follows:

a. $[T_{e_i} \leqq T_i^*]$: the operands may be lifted before the comparison.

b. The lifted operands must be well-formed.

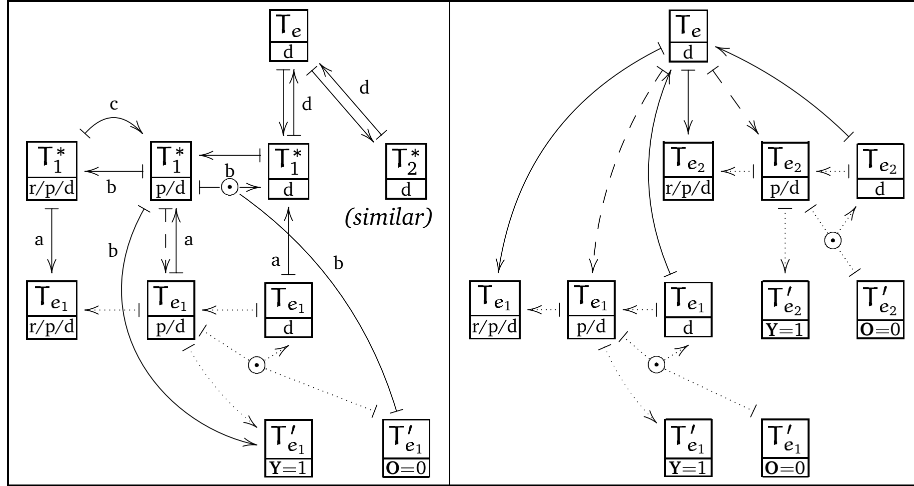



*Figure 4.6: Arrows for the expression* $\mathsf{e} = \mathsf{PtrCmp}(\mathsf{e}_1, \circ, \mathsf{e}_2)$*. The handling of* $\mathsf{e}_2$ *is identical to the handling of* $\mathsf{e}_1$ *so to the left it is omitted to save space.*

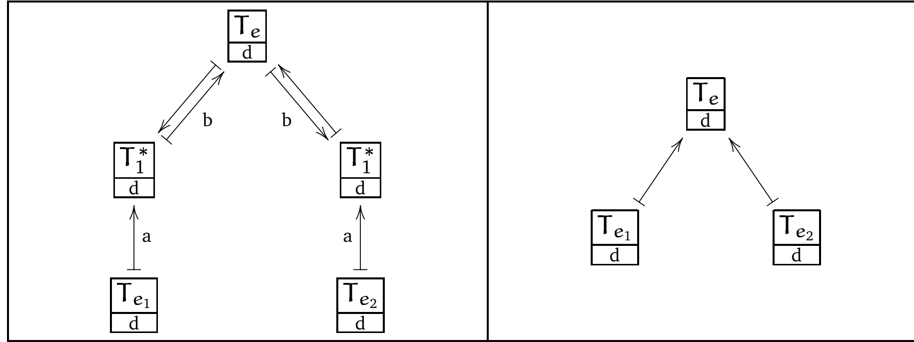



*Figure 4.7: Arrows for the expression* $\mathsf{e} = \mathsf{Binary}(\mathsf{e}_1, \circ, \mathsf{e}_2)$.

c. The lifted operands cannot be $\mathsf{Pointer}_\mathsf{r}$s.

d. The operand is $\mathsf{Pointer}_\mathrm{d}$ iff the result is $\mathsf{Primitive}_\mathrm{d}$.

In principle, there should be arrows from $\begin{array}{|c|}\hline \mathsf{T}_e \\ \hline \mathrm{d} \\ \hline \end{array}$ to either $\begin{array}{|c|}\hline \mathsf{T}'_{e_i} \\ \hline \mathrm{Y{=}1} \\ \hline \end{array}$. However, because the result may always be lifted we know that there will be no arrows leading to $\begin{array}{|c|}\hline \mathsf{T}_e \\ \hline \mathrm{d} \\ \hline \end{array}$ other than the ones shown in the figure. This means that $\begin{array}{|c|}\hline \mathsf{T}_e \\ \hline \mathrm{d} \\ \hline \end{array}$ will only be provable when one of the $\begin{array}{|c|}\hline \mathsf{T}_j \\ \hline \mathrm{d} \\ \hline \end{array}$s is, from which either $\begin{array}{|c|}\hline \mathsf{T}_i \\ \hline \mathrm{Y{=}1} \\ \hline \end{array}$ is already derivable.

### The Binary operator

The arrows for the Binary operator are shown on Figure 4.7. They are straightforward in comparison to the pointer operators.

a. $[\mathsf{T}_{e_i} \leqq \mathsf{T}^*_i]$.

*Figure 4.8: Arrows for the expression* $e = \mathsf{Member}(e_1, m)$*. In the diamond diagram to the left, the "$*$" combination is only allowed when* $|\tau'_e|_{\mathbf{Y}} = 0$.

b. The lifted operands have the same binding time as the result.

### The Member operator

The arrows we need for $e = \mathsf{Member}(e_1, m)$—in addition to equivalce $[(\mathsf{T}'_{e_1})_m \equiv \mathsf{T}'_e]$ between the type of the selected member and the type pointed to by the result—are shown on Figure 4.8. They are derived from the typing rules on page 88 as follows:

a. $[\mathsf{T}_{e_1} \leqq \mathsf{T}^*_1]$ in the variant that allows a carry. As usual, the dashed arrow is present whenever the operand is not known to point to objects in scope.

b. The lifted (or carried) operand type must be well-formed.

c. $\mathsf{Pointer}_\mathrm{r}$ is not a valid operand.

d. $\mathsf{Pointer}_\mathrm{s}$ is not a valid result.

e–h. The combination of operand and result variants must be allowed by the typing rules. The entry marked "$*$" in the diamond-shaped sketch corresponds to the rule,

$$\frac{\Gamma \vdash e \precsim \mathsf{Pointer}_\mathrm{p}(\mathsf{Struct}_\mathrm{p}(\tau_1, \ldots, \tau_n))}{\Gamma \vdash \mathsf{Member}(e, m) : \mathsf{Pointer}_\mathrm{r}(\tau_m)} \; |\tau_m|_\mathsf{Y} = 0$$

When $|\tau_m|_\mathsf{Y} \neq 0$ (i.e., when $\begin{array}{|c|}\hline (\mathsf{T}'_{e_1})_m \\ \hline \mathrm{Y}{\neq}0 \\ \hline\end{array}$ or the equivalent $\begin{array}{|c|}\hline \mathsf{T}'_e \\ \hline \mathrm{Y}{\neq}0 \\ \hline\end{array}$ is provable) there should be an arrow from $\begin{array}{|c|}\hline \mathsf{T}^*_1 \\ \hline \mathrm{p/d} \\ \hline\end{array}$ to $\begin{array}{|c|}\hline \mathsf{T}_e \\ \hline \mathrm{p/d} \\ \hline\end{array}$. Thus the two-premise arrow (g).

### The Array operator

The arrows, except for equivalence $[\mathsf{T}'_e \equiv \mathsf{T}''_{e_1}]$ between the element type of the array and the type pointed to by the result, are shown on Figure 4.9. They are derived from the typing rules on page 88 as follows:

a. $[\mathsf{T}_{e_1} \leqq \mathsf{T}^*_1]$ in the variant that allows a carry.

b. The lifted operand type must be well-formed.

c. The lifted operand type always has the same $\mathsf{Pointer}$ variant as the result. Remark that the interpretation of "$\begin{array}{|c|}\hline \mathsf{T} \\ \hline \mathrm{p/d} \\ \hline\end{array}$ but not $\begin{array}{|c|}\hline \mathsf{T} \\ \hline \mathrm{d} \\ \hline\end{array}$ provable" is the same for the two type nodes, because $|\mathsf{Array}(\tau, e)|_\mathsf{O} = |\tau|_\mathsf{O}$ for all $\mathsf{Array}$ variants.

d. $\mathsf{Pointer}_\mathrm{r}$ is neither allowed as (lifted) operand nor result.

e. The result is $\mathsf{Pointer}_\mathrm{d}$ or $\mathsf{Pointer}_\mathrm{p}$ precisely when the array is a $\mathsf{Array}_\mathrm{d}$ or $\mathsf{Array}_\mathrm{p}$.

### The DeRef operator

The arrows apart from identity $[\mathsf{T}_e \equiv \mathsf{T}'_{e_1}]$ between the pointed-to type and the result type are shown in Figure 4.10. They are derived in the following way:

a. $[\mathsf{T}_{e_1} \leqq \mathsf{T}^*_1]$ in the variant that allows a carrys.

b. The lifted operand type should be well-formed.

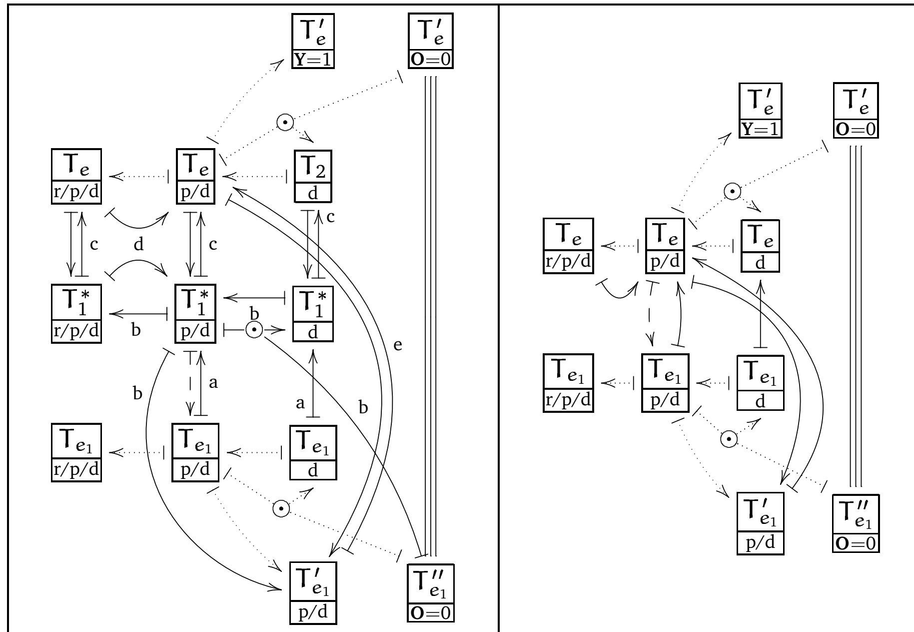



*Figure 4.9: Arrows for the expression* $\mathsf{e} = \mathsf{Array}(\mathsf{e}_1)$.

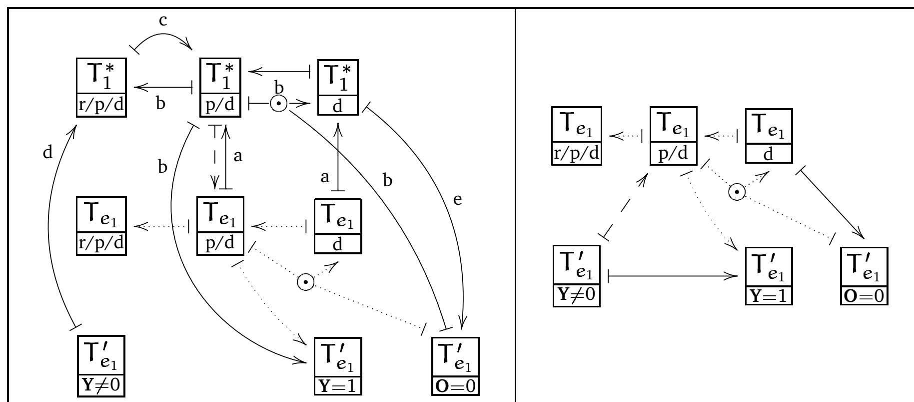



*Figure 4.10: Arrows for the expression* $\mathsf{e} = \mathsf{DeRef}(\mathsf{e}_1)$.

c. Access through a $\mathsf{Pointer}_\mathrm{r}$ is not allowed.

d. Access through $\mathsf{Pointer}_\mathrm{s}$ is not allowed if the pointed-to binding-time type has symbolic model objects.

e. Access through $\mathsf{Pointer}_\mathrm{d}$ is not allowed if the pointed-to binding-time type has concrete model objects. When $|\tau'|_{\mathbf{O}} \neq 0$ we *can* be sure that $\boxed{\genfrac{}{}{}{}{\mathsf{T}}{\mathrm{d}}}$ is provable whenever $\mathsf{Pointer}_\mathrm{d}$ is necessary.

One might consider the arrow from $\boxed{\genfrac{}{}{}{}{\mathsf{T}'_{e_1}}{\mathrm{Y}\neq 0}}$ to $\boxed{\genfrac{}{}{}{}{\mathsf{T}'_{e_1}}{\mathrm{Y}=1}}$ redundant, because in fact $\boxed{\genfrac{}{}{}{}{\mathsf{T}}{\mathrm{Y}\neq 0}} = \boxed{\genfrac{}{}{}{}{\mathsf{T}}{\mathrm{Y}=1}}$ for any $\mathsf{T}$ that can be the type of an expression. It does no harm, however, (because arrows with equal premise and conclusion can trivially be thrown away as soon as they are generated), so we might as well leave it in. It would be handy to have if someday we implement the $\mathsf{Struct}_\mathrm{sy}$ described on page 79.

#### The Cast operator

The arrows for $e = \mathsf{Cast}(e_1)$ depend on whether the cast is considered benign or malign.

*benign:* we simply use $[\mathsf{T}_e \equiv \mathsf{T}_{e_1}]$.

*malign:* we have to ensure that the operand and the result are both in dynamically faithful. For that we need

$$[\mathsf{T}_e \in \mathbb{F}] \quad \cup \quad [\mathsf{T}_{e_1} \in \mathbb{F}] \quad \cup \quad \left\{ \boxed{\genfrac{}{}{}{}{\mathsf{T}_e}{\mathrm{O}=0}} \longleftarrow \top \longrightarrow \boxed{\genfrac{}{}{}{}{\mathsf{T}_{e_1}}{\mathrm{O}=0}} \right\}$$

#### The Sizeof operator

The expression $e = \mathsf{Sizeof}(\mathsf{T}_1)$ is handled much like a malign cast:

$$[\mathsf{T}_1 \in \mathbb{F}] \quad \cup \quad \left\{ \boxed{\genfrac{}{}{}{}{\mathsf{T}_e}{\mathrm{d}}} \longleftarrow \top \longrightarrow \boxed{\genfrac{}{}{}{}{\mathsf{T}_1}{\mathrm{O}=0}} \right\}$$

### 4.3.6 Arrow generation for statements

This section defines which arrows are created for each statement in the program, apart from the ones generated for the expressions in the statement.

Generally we use $b_0$ to mean the basic block the statement belongs to and $f_0$ to mean the function it belongs to.

#### Arrows for Assign statements

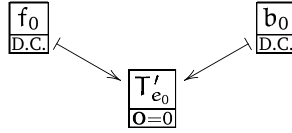
Let the statement be $s = \mathsf{Assign}(e_0, e)$ with $\mathsf{T}_{e_0} = \mathsf{Pointer}(\mathsf{T}'_{e_0})$. We then need

i. The same conditions on the type of $e_0$ as for the operand to a $\mathsf{DeRef}$ operator; see Figure 4.10.

ii. $[T_e \leqq T'_{e_0}]$: it must be possible to lift the value of the right-hand expression into the binding-time type of the location we're storing it in.

iii. If $e_0$ can point to anything else than $f_0$'s truly local variables (see Section 3.4.2), then there can be concrete model objects involved in the assignment only if the statement is not under dynamic control.

That is, we add the arrows




unless the pointer analysis shows that $e_0$ can point only to $f_0$'s truly local variables.

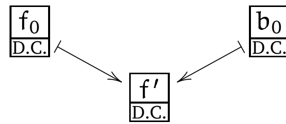
Note that the criterion for adding these arrows is *different* from the criterion for adding the dashed arrow in $[T \leqq T']$; a pointer that points to global variables may be lifted, but it cannot be used for static side effects under dynamic control.

**Arrows for $\mathsf{Call}$ statements**

Let the statement be $s = \mathsf{Call}(e_0, e_f, e_1...e_m)$ and $T_{e_f} = \mathsf{FunPtr}(T_1, ..., T_n \to T_0)$ be the type of the called function.

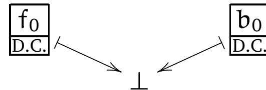
i-iii. If $e_0$ is present, we need the same conditions on it as for assignments, cases (i) and (iii) above. The lift in (ii) is of course replaced with $[T_0 \leqq T'_{e_0}]$.

iv. If the $\mathsf{Call}$ statement is under dynamic control, then so is every called function, so we generate

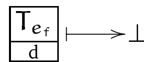



for each $f'$ in the points-to set for $e_f$.

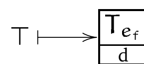
v. If the user has marked the call as a "sensitive" (see page 32 it must not happen under dynamic control at all, so we generate




If the user has marked the call to be "spectime", we generate




Otherwise, if an external function might be called or the call is explicitly marked as "residual", we generate



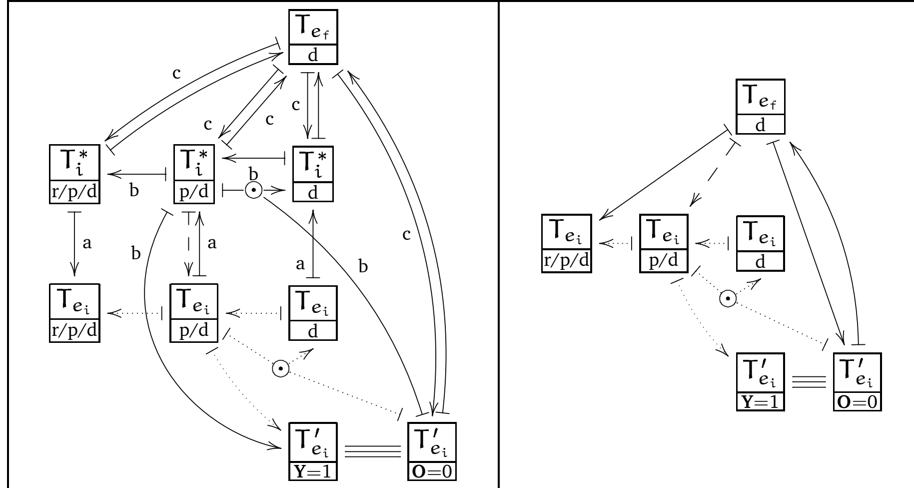



*Figure 4.11: Arrows for an "extra" argument* $e_i$ *to a call through* $e_f$ *when the type of* $e_i$ *is* $T_{e_i} = \mathsf{Pointer}(T'_{e_i})$.

vi. Generally each argument $e_1$ through $e_n$ should be liftable into the type of the corresponding formal parameter, so we generate arrows $[T_{e_i} \leqq T_i]$ for $1 \leq i \leq n$.

vii. Though it is not shown in the typing rules on page 90, the numer $m$ of actual arguments might be greater than the number $n$ of formal parameters in the called function's type. This can occur when calling an external function such as `printf` (whose type only specifies the first argument and that others—of diverse types—may follow). Because this situation only occurs when calling external functions[3] the "extra" arguments should always—after possible lifting—be statically or dynamically faithful, matching the binding time of $e_0$.

How to achieve that depends on the type of each $e_i$ $(n < i \leq m)$:

Pointer: In addition to $[T'_{e_i} \in \mathbb{F}]$ we need arrows as shown on Figure 4.11.

a. $[T_{e_i} \leqq T_i^*]$.

b. The lifted argument must be well-formed.

c. Each proposition owned by the type nodes of the lifted argument must be provable if and only if $\begin{array}{|c|}\hline T_{e_f} \\ \hline \mathrm{d} \\ \hline\end{array}$ is.

Primitive: Through a similar argument (too simple to waste a figure on) we find that what we need is

$$\begin{array}{|c|}\hline T_{e_i} \\ \hline \mathrm{d} \\ \hline\end{array} \longmapsto \begin{array}{|c|}\hline T_{e_f} \\ \hline \mathrm{d} \\ \hline\end{array}$$

is the arrow we need.

*anything else:* The argument cannot be lifted, so we simply generate

$$[T_{e_i} \in \mathbb{F}] \quad \cup \quad \begin{array}{|c|}\hline T_{e_f} \\ \hline \mathrm{d} \\ \hline\end{array} \begin{array}{c} \longleftarrow\!\!| \\ |\!\!\longrightarrow \end{array} \begin{array}{|c|}\hline T_{e_i} \\ \hline \mathrm{O}{=}0 \\ \hline\end{array}$$

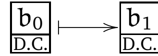
[3] C-Mix/II does not currently support specializing functions with a variable number of arguments.

### 4.3.7 Arrow generation for jumps

In this section we define which arrows to generate for Core C control structures.

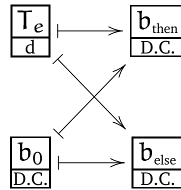
#### Unconditional jumps

Suppose the jump that ends basic block $b_0$ is $\mathsf{Goto}(b_1)$. Then generate




#### Conditional jumps

Suppose the jump that ends basic block $b_0$ is $\mathsf{If}(e, b_{then}, b_{else})$. The type of $e$ is Primitive[4], and each of the target blocks can either become under dynamic control because the condition is $\mathsf{Primitive}_d$ or because $b_0$ is itself under dynamic control:



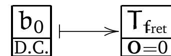

(note that lifting the value of the condition expression is pointless).

#### Return statements

Suppose the jump that ends basic block $b_0$ is $\mathsf{Return}(e)$, and let the current function's type be $\mathsf{FunPtr}(\ldots \to T_{fret})$.

When $e$ is present we need $[T_e \leqq T_{fret}]$.

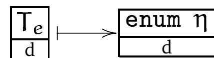
Also, when $e$ is present the return value must not have a static part if the return is under local dynamic control. Hence we also generate




### 4.3.8 Other arrows

#### Arrows for Enum declarations

If any of the constant declarations in the definition of an Enum type $\eta$ include a value expression $e$ (whose type must be `int`) we generate the arrow




[4]In C, an expression of pointer type can serve as a condition. However, in Core C an explicit comparison against a null pointer is generated in such cases.

### Arrows for user annotations

The user of C-Mix$_{\mathrm{II}}$ may, through specializer directives request that some variables in the subject program be annotated as “residual” or “spectime”. We translate these requests into the arrows $\top \longmapsto \begin{array}{|c|}\hline \mathsf{T} \\ \hline \mathbf{O}{=}0 \\ \hline \end{array}$ or $\begin{array}{|c|}\hline \mathsf{T} \\ \hline \mathbf{O}{=}0 \\ \hline \end{array} \longmapsto \bot$ , respectively, where $\mathsf{T}$ is the root of the affected variable’s type.

In default of an overriding specializer directive, an external variable (i.e., a variable that has a declaration but no definition in the subject program) is forced to be dynamic. Letting it be static is potentially hazardous, because without a proper definition it cannot be registered with the memoization code in Speclib.

### Arrows for externally visible data

The user may specify that certain variables should be “visible” to foreign code that is linked together with either $\mathfrak{p}_{\text{gen}}$ or $\mathfrak{p}_{\text{res}}$. For such a variable the arrows $[\mathsf{T} \in \mathbb{F}]$ are generated together with the appropriate binding-time forcing arrow, so that we can be sure the C-Mix$_{\mathrm{II}}$-generated code and the external code agree on its type.

The same mechanism is used to force the goal function’s parameters to have the specified binding time. The goal function’s return type (where non-`void`) is always forced to be dynamic by this mechanism. The reason for that is primarily that it makes the calling interface of $\mathfrak{p}_{\text{gen}}$ and $\mathfrak{p}_{\text{res}}$ predictable without bothering the user to explicitly specify a binding time for the return type.

### Arrows for initializers

A variable in Core C may have an initializer. Normally the Core C initializer has a structure that matches the structure of the variable’s type; in those cases we generate a number of $[\mathsf{T}''_e \leqq \mathsf{T}''_x]$ arrow sets to ensure that the intializer expressions can be lifted into the relevant parts of the variable’s binding-time type.

Sometimes it is not possible for the Core C translation to find out the structure of the initializer. For example in,

```
struct S{ int a[sizeof(int)]; int b[2]; } x = {1,2,3,4,5,6};
```

the structure of the initializer is not fully spelled out with braces. The only way we can see where the `a`s end and the `b`s begin is to know how long the two arrays are. Which we do not at analysis time.

In cases such as this, (a subtree of) the initializer is marked as “ambiguous”. For an ambiguous initializer we cannot reliably identify which initializer expressions match which parts of the variable’s type. Because we have to make sure that their binding-time types match nevertheless, we decide that either the (relevant part of the) variable together with all of the initializer expressions must all be dynamically faithful, or they must all be statically faithful.

We implement that rule by generating a number of $[\mathsf{T} \in \mathbb{F}]$ arrow sets, plus arrows letting all the, say, $\begin{array}{|c|}\hline \mathsf{T} \\ \hline \mathbf{O}{=}0 \\ \hline \end{array}$s of the relevant types depend on each other.

## 4.4 Finding the provable propositions

Now we will discuss how to compute the set of provable propositions once we have generated the arrows.

### Only single-premise arrows

Let us temporarily forget the existence of arrows with more than one premise. Then finding out which propositions are provable is as simple as doing a graph traversal in the graph of propositions and arrows, starting from $\top$.

Because we are also interested in generating user-readable proofs, we specify that strategy in a little more detail. Let the variable $v$ range over propositions (i.e., the **v**ertices of the BTA graph) and notate an arrow as the triple $(v_1, R, v_2)$ where R is a reference to the Core C construct (or user specification) that caused the arrow to be generated. With each $v$ we associate an initially empty "trace set" $S_v$. The algorithm to compute the set P of provable propositions is then

1. Set $W \leftarrow \{\top\}$ and $P \leftarrow \emptyset$.
2. Select a $v \in W$ (using a first-in-first-out discipline) and set $W \leftarrow W \setminus \{v\}$.
3. Set $P \leftarrow P \cup \{v\}$.
4. Do the following for each arrow $(v_1, R, v_2)$ with $v = v_1$ and $v_2 \notin P$:
   a. Set $S_{v_2} \leftarrow S_{v_2} \cup \{(v_1, R, v_2)\}$.
   b. Set $W \leftarrow W \cup \{v_2\}$.
5. If $W \neq \emptyset$ then go back to step (2).
6. If $\perp \in P$ then signal an error.

This is a standard breadth-first traversal of the graph with the added feature that the $S_v$ sets collect the arrows $(v_1, R, v)$ where $v_1$ was proved earlier than $v$. For any $v_0 \in P$ we can then find a chain C of arrows that proves $v$, thus:

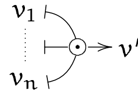
- If $v_0 = \top$, no further proof is needed.
- Choose any $(v, R, v_0) \in S_{v_0}$.
- Set $C \leftarrow C \cup \{(v, R, v_0)\}$.
- Set $v_0 \leftarrow v$ and start over.

If we favor the arrow that was first inserted in $S_v$, the chain C will be as short as possible.

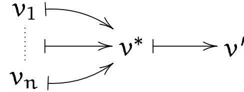
The reason why an entire set of arrows is maintained for each $S_v$ rather than, say, just the first one added, is that C-Mix/II supports an interactive BTA result browser that lets the user inspect $S_v$ for each provable $v$ and follow hyperlinks backwards in the BTA graph. $S_v$ is then the most easily computable approximation to "all the relevant reasons why $v$ is provable" that still guarantees that the user will eventually reach $\top$.

#### Adding multi-premise arrows

We implement an $n$-premise arrow




by adding a new vertex $v^*$ in the BTA graph with the special property that $v^*$ needs $n$ provable premises to be proved ifself (ordinary vertices only need 1 provable premise):




Set, for each node $v$, $N(v)$ is the number of provable premises $v$ needs to be proved itself. Then we can easily modify the search algorithm to handle $N(v) > 1$ vertices: Step 4(b)—inserting $v_2$ in $W$—should only be performed if $|S_{v_2}| = N(v_2)$.

Because multi-premise arrows complicate the structure of proofs, we want to avoid them in proofs as far as possible. We can do that by modifying the rule for selecting vertices from $W$ from straight first-in-first-out to: select $v \in W$ with minimal $N(v)$, using first-in-first-out when there are several candidates.

It should be clear that with these modifications, the search algorithm can still be implemented to run in time linear in the number of arrows (which is itself linear in the size of the Core C representation of the subject program).

### 4.4.1 Optimizations

The implementation of the binding-time analysis uses some optimizations relative to how it has been specified here.

First, many of the $[T_1 \equiv T_2]$ arrow sets are implemented by simply letting $T_1$ and $T_2$ be the same type node. A fair number of them are still reduced to real arrows, those.

The governing principle is that the type graphs associated with different variable declarations should be kept disjoint. This is to make proofs more readable: most type nodes (and, hence, their propositions) can then be named by their relation to a variable, and the variable's relation to the expression that cause arrows for the proposition should be clear from the source code.

Second, the majority of $[T \in \mathbb{F}]$ arrow sets are implemented by letting a single proposition share the roles of all propositions owned by $T$ and its descendants. The only point where a real $[T \in \mathbb{F}]$ constraint set is needed is for "extra" arguments to external function calls, as described on page 116.

## 4.5 Computing signatures

Most of the desired binding-time annotations can be read directly from the set of provable propositions. The only thing that still remains is the question of whether an array is $\mathsf{Array}_p$ or $\mathsf{Array}_d$. That amounts to finding the signature of the element type.

In general, it is convenient to have the signatures of all binding-time types explicitly represented.

Table 3.6 on page 86 shows how to compute the signature of a binding-time type given knowledge of the constituent binding-time types' signatures. We can use that as an algorithm, but we need to beware of infinite recursion due to the existense of recursive types.

We see from Table 3.6 turns out that Pointer and Array types are the only ones where the signature depends on other signatures. Because there can be no type recursion in C consisting entirely of Pointer and Array types (that is because the only types a program can *mention* before defining them are Struct and Union) this guarantees that signature computations terminate.

It is trivial to cache results of signature computations such that the total time used for finding signatures is linear in the size of the Core C program.

## 4.6 Efficiency of the BTA

For the BTA we have presented, efficiency should not be a problem. Our practical experience is that the BTA does not dominate the time usage of C-Mix/II. Most of the time is spent parsing the subject program and emitting feedback information and generating-extension source. Compiling $p_{gen}$ consistently takes even longer time than C-Mix/II itself uses.

It should be noted, however, that this practical experience is based on the currently available C-Mix/II, whose BTA is simpler because it does not treat partially static data. The analysis presented in this report uses more arrows than the current implementation—we estimate that on average the nubmer will roughly double when we implement partially static data (the actual growth rate ranges from no increase in the number of arrows for Primitive manipulation to a five-fold increase in the number of well-formedness arrows needed for a Pointer type node).

Still, we feel that even if the time spent on BTA is doubled or tripled, it will not be a major obstacle to using C-Mix/II.

The above argument rests on an assumption that the fraction of C-Mix/II's time usage that is spent on BTA does not increase when using C-Mix/II on bigger subject programs. To justify that, we will argue that the time usage of the BTA is almost-linear in the size of the Core C input.

The binding-time analysis consists of four phases:

1. Create the necessary propositions, as described in Section 4.2.2.
2. Generate the arrows implied by each Core C construct, as described in Section 4.3
3. Find the provable propositions, as described in Section 4.4.
4. Translate from propositions to explicit annotations and compute signatures as described in Section 4.5.

The running times of phases 1 and 4 are proportional to the size of the Core C input.

The running time of phase 2 is almost-linear in the size of the Core C input plus the number of generated arrows. The number of arrows can in turn be broken into arrows resulting from $[T_1 \equiv T_2]$ arrow sets and the rest. The size

of each group is linear in the size of the Core C input (see the discussion in Section 4.3.1).

The running time of phase 3 is (at most) proportional to the number of arrows, which is again proportional to the size of the Core C input.

# Chapter 5

# The splitter phase: eliminating partially static binding-time types

This section defines the **splitter** phase which takes as input a binding-time annotated program and transforms it into an equivalent program that uses only basic binding-time types. If you have not already done so, please read Section 2.6 before continuing. That introduces the key principles behind the splitter without the complexity of handling the full Core C language.

In this description we ignore the distinction between $\mathsf{Pointer}_s$ and $\mathsf{Pointer}_r$. Once it has been used in the BTA to regulate the positioning of lifts it is not relevant anymore, so we shall refer to both variants as simply $\mathsf{Pointer}_s$.

**Warning!** As of this writing, the splitter phase is not implemented in C-Mix/II; instead a crippled BTA that produces only basic binding-time types is used. We describe the splitter in the present tense, nevertheless, because we hope to be able to use a revised version of this report as documentation for its eventual implementation.

The hallmark of basic binding-time types is that their signatures are $(1,0)$, $(-1,0)$ or $(0,1)$. Thus the main task of the splitter phase is to *split* any binding-time type with signature $(1,1)$, $(-1,1)$, or $(0,2)$ into a pair of basic binding-time types whose signatures add up to the original type's signature.

## 5.1 How to split types

A binding-time type $\tau$ must be split exactly when $|\tau|^2_{\mathbf{Y}} + |\tau|_{\mathbf{O}} = 2$. We call such a type **splitable**[1].

The splitter transformation on types is defined in Figure 5.1. A splitable (binding-time) type $\tau$ splits into $\{\!|\tau|\!\}^{\mathbf{A}}$ and $\{\!|\tau|\!\}^{\mathbf{B}}$. The latter is always a static type, whereas the former can be either dynamic or static.

[1]Caveat: in some of the current C-Mix/II source documentation, the term "splitable" is used with a slightly different meaning.

$$
\begin{array}{rcl}
\{\!|\mathsf{Primitive}_d|\!\}^\circ & \Longrightarrow & \mathsf{Primitive}_d \\
\{\!|\mathsf{Primitive}_s|\!\}^\circ & \Longrightarrow & \mathsf{Primitive}_s \\
\{\!|\mathsf{Abstract}_d|\!\}^\circ & \Longrightarrow & \mathsf{Abstract}_d \\
\{\!|\mathsf{Abstract}_s|\!\}^\circ & \Longrightarrow & \mathsf{Abstract}_s \\
\{\!|\mathsf{Enum}_d|\!\}^\circ & \Longrightarrow & \mathsf{Enum}_d \\
\{\!|\mathsf{Enum}_s|\!\}^\circ & \Longrightarrow & \mathsf{Enum}_s \\
\{\!|\mathsf{Pointer}_d(\tau)|\!\}^\circ & \Longrightarrow & \mathsf{Pointer}_d(\{\!|\tau|\!\}^\circ \;\Box\; \{\!|\tau|\!\}^{\mathbf{A}}) \\
\{\!|\mathsf{Pointer}_s(\tau)|\!\}^\circ & \Longrightarrow & \mathsf{Pointer}_s(\{\!|\tau|\!\}^\circ) \\
\{\!|\mathsf{Array}_d(\tau, e)|\!\}^\circ & \Longrightarrow & \mathsf{Array}_d(\{\!|\tau|\!\}^\circ, \{\!|e|\!\}^\circ) \\
\{\!|\mathsf{Array}_s(\tau, e)|\!\}^\circ & \Longrightarrow & \mathsf{Array}_s(\{\!|\tau|\!\}^\circ, \{\!|e|\!\}^\circ) \\
\{\!|\mathsf{Struct}_d(\tau_1, \ldots, \tau_n)|\!\}^\circ & \Longrightarrow & \mathsf{Struct}_d(\{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}}) \\
\{\!|\mathsf{Struct}_s(\tau_1, \ldots, \tau_n)|\!\}^\circ & \Longrightarrow & \mathsf{Struct}_s(\{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}}) \\
\{\!|\mathsf{Union}_d(\tau_1, \ldots, \tau_n)|\!\}^\circ & \Longrightarrow & \mathsf{Union}_d(\{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}}) \\
\{\!|\mathsf{Union}_s(\tau_1, \ldots, \tau_n)|\!\}^\circ & \Longrightarrow & \mathsf{Union}_s(\{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}}) \\
\{\!|\mathsf{FunPtr}_s(\tau_1, \ldots, \tau_n \to \tau_0)|\!\}^\circ & \Longrightarrow & \mathsf{FunPtr}_s(\{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}} \to \{\!|\tau_0|\!\}^\circ \;\Box\; \{\!|\tau_0|\!\}^{\mathbf{B}}) \\
\{\!|\mathsf{FunPtr}_d(\tau_1, \ldots, \tau_n \to \tau_0)|\!\}^\circ & \Longrightarrow & \mathsf{FunPtr}_d(\{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}} \to \{\!|\tau_0|\!\}^\circ) \\
\\
\{\!|\mathsf{Pointer}_p(\tau)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{Pointer}_d(\{\!|\tau|\!\}^{\mathbf{A}}) \\
\{\!|\mathsf{Pointer}_p(\tau)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{Pointer}_s(\{\!|\tau|\!\}^{\mathbf{B}}) \\
\\
\{\!|\mathsf{Pointer}_s(\tau)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{Pointer}_s(\{\!|\tau|\!\}^{\mathbf{A}}) \\
\{\!|\mathsf{Pointer}_s(\tau)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{Pointer}_s(\{\!|\tau|\!\}^{\mathbf{B}}) \\
\\
\{\!|\mathsf{Array}_p(\tau, e)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{Array}_d(\{\!|\tau|\!\}^{\mathbf{A}}, \{\!|e|\!\}^\circ) \\
\{\!|\mathsf{Array}_p(\tau, e)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{Array}_s(\{\!|\tau|\!\}^{\mathbf{B}}, \{\!|e|\!\}^\circ) \\
\\
\{\!|\mathsf{Array}_s(\tau, e)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{Array}_s(\{\!|\tau|\!\}^{\mathbf{A}}, \{\!|e|\!\}^\circ) \\
\{\!|\mathsf{Array}_s(\tau, e)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{Array}_s(\{\!|\tau|\!\}^{\mathbf{B}}, \{\!|e|\!\}^\circ) \\
\\
\{\!|\mathsf{Struct}_p(\tau_1, \ldots, \tau_n)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{Struct}_d(\{\tau \in \{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}} \mid |\tau|_{\mathbf{Y}} \neq 0\}) \\
\{\!|\mathsf{Struct}_p(\tau_1, \ldots, \tau_n)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{Struct}_s(\{\tau \in \{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}} \mid |\tau|_{\mathbf{Y}} = 0\}) \\
\\
\{\!|\mathsf{Union}_p(\tau_1, \ldots, \tau_n)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{Union}_d(\{\tau \in \{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}} \mid |\tau|_{\mathbf{Y}} \neq 0\}) \\
\{\!|\mathsf{Union}_p(\tau_1, \ldots, \tau_n)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{Union}_s(\{\tau \in \{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}} \mid |\tau|_{\mathbf{Y}} = 0\}) \\
\\
\{\!|\tau_1, \ldots, \tau_n|\!\}^{\mathbf{L}} & \Longrightarrow & \{\{\!|\tau_i|\!\}^\circ \;\Box\; \{\!|\tau_i|\!\}^{\mathbf{A}}, \{\!|\tau_i|\!\}^{\mathbf{B}} \mid i \leq 1 \leq n\}
\end{array}
$$

*Figure 5.1: The splitter transformation on types. The mapping $\{\!|\tau|\!\}^\circ$ is defined for non-splitable $\tau$; the mappings $\{\!|\tau|\!\}^{\mathbf{A}}$ and $\{\!|\tau|\!\}^{\mathbf{B}}$ are for splitable $\tau$. The notation $\mathcal{P}(\tau) \;\Box\; \mathcal{Q}(\tau)$ denotes selecting between $\mathcal{P}(\tau)$ or $\mathcal{Q}(\tau)$ according to whether $\tau$ is splitable.*

Most of Figure 5.1 speaks for itself, but the handling of $\mathsf{Struct}$ and $\mathsf{Union}$ deserves some extra explanation.

The idea is that a nonsplitable member is transformed to "itself" and a splitable member is transformed to two members, one for the **A** component and one for the **B** component. The resulting member list is then filtered into a list of only dynamic members and a list of only static members. This filtering is only necessary for $\mathsf{Struct}_p$ because a well-formed $\mathsf{Struct}_d$ or $\mathsf{Struct}_s$ is known to have only dynamic or only static members.

This is well and fine for $\mathsf{Struct}$s but care has to be taken for $\mathsf{Union}$s. The problem is that if a union member is split, it becomes two members, and union members are supposed to overlap in storage. This conflicts with our normal splitting principles which assume that the **A** and **B** components exist independently of each other. In the case that the **A** component is dynamic we get saved by the bell, because the two components end up in two different unions. But if the **A** and **B** components are both static—i.e., if the original member type $\tau$ had signature $(0, 2)$—we are in trouble. That must not be allowed to happen.

The solution we chose is to insert an extra phase between just before the splitter phase that wraps every $\mathsf{Union}$ member of signature $(0, 2)$ in a single-member $\mathsf{Struct}$ (which becomes two-member during the splitting phase). That is, $\mathsf{Union}_{p/s}(\ldots\tau_i\ldots)$ where $|\tau_i|_{\mathbf{O}} = 2$ is replaced by $\mathsf{Union}_{p/s}(\ldots\mathsf{Struct}_s(\tau_i)\ldots)$. At the same time, all $\mathsf{Member}(e, i)$ expressions referring to that union must be replaced by $\mathsf{Member}(\mathsf{Member}(e, i), 1)$.

In the implementation this preprocessing is done simultaneously with the actual splitting, but it is probably best to think of it conceptually as a separate phase.

## 5.2 How to split variables

Each variable declaration $d = \mathsf{Var}(id, \tau, \delta, i)$ is transformed into:

- If $\tau$ is *not* splitable: $d^\circ = \mathsf{Var}(id, \{\!|\tau|\!\}^\circ, \delta, \{\!|i|\!\}^\circ)$.
- If $\tau$ *is* splitable: Two variable declarations $d^{\mathbf{A}} = \mathsf{Var}(id, \{\!|\tau|\!\}^{\mathbf{A}}, \delta, \{\!|i|\!\}^{\mathbf{A}})$ and $d^{\mathbf{B}} = \mathsf{Var}(id, \{\!|\tau|\!\}^{\mathbf{A}}, \delta, \{\!|i|\!\}^{\mathbf{B}})$.

In both cases, Core C administrativa such as "subsidiary declarations" are of course rebuilt for the new declaration(s).

If the variable had an initializer $i$ that is transformed, too. The transformation of (non-ambigous) initializers is a simple extrapolation of the transformation of types and expressions. Writing it down would imply more boring details than it would give insight, so we won't.

## 5.3 How to split functions

A function is not "split" as such: it stays a single function during the splitter phase. Of course its type, body, parameters and local variables are translated; observe that when the parameters have been translated like any other variables, the resulting parameter list also matches the transformed type of the function.

The tricky thing about functions is what to do when their return type is splitable. We want to use the strategy of Section 2.6.2: let the function return

the **A** component normally and store the **B** component in a global variable to be picked up by the caller.

That is not as easy in Core C as in the simplified language we looked at in Chapter 2. When functions are called through function pointers it is not always clear at a call statement which function actually gets called. How should we choose the proper global variable to use as the **B** component of the return value?

The answer here is not to use a separate global variabe for each function but to let a global variable cover a *group* of functions. Two functions should be in the same group if they are both in the points-to set of the "function" expression of some statement in the program. If we use this condition (transitively) to partition the set of functions into groups we can be sure that any two functions in the same group have the same return type (because otherwise the type system would not have allowed one pointer to point to both).

Thus the function grouping can easily be computed using a union–find structure. We write the group that a given function $f$ belongs to as $\mathcal{G}(f)$, and generate a suitable global variable $d_{\mathcal{G}(f)}$ for each group of functions with splitable return type.

## 5.4 How to split expressions

The splitter transformations for expressions are defined in Figure 5.2 and Table 5.1. Like there is for types, there are actually three transformation mappings: $\{\!|e|\!\}^\circ$, which is used then the type of $e$ is not splitable; and $\{\!|e|\!\}^\mathbf{A}$ and $\{\!|e|\!\}^\mathbf{B}$, which are used when the type of $e$ is splitable.

There should be no great surprises in Figure 5.2. Some points that deserve some explanation are:

- The transformation of $\{\!|\mathsf{Var}(\tau, d)|\!\}^\circ$ is different according to whether $d$ is splitable (in which case $\tau$ can only be a $\mathsf{Pointer}_d$ and we're uninterested in $d$'s **A** component) or not.
- The operands to PtrCmp are transformed as $\{\!|e_i|\!\}^\circ$ if the result is $\mathsf{Primitive}_d$ (in which case the operands are $\mathsf{Pointer}_d$s) *or* if the pointed-to type is not splitable. Otherwise, the operands are splitable and transformed as $\{\!|e_i|\!\}^\mathbf{B}$.
- We do not need to define $\{\!|e|\!\}^\mathbf{A}$ and $\{\!|e|\!\}^\mathbf{B}$ for Const and Cast, because a faithful type is never splitable. For the same reason the operands to Cast and Sizeof can simply be transformed using $\{\!| \cdot |\!\}^\circ$. Notice that $\{\!|\tau|\!\}^\circ = \tau$ when $\tau$ is faithful.
- We do not need to define $\{\!|e|\!\}^\mathbf{A}$ and $\{\!|e|\!\}^\mathbf{B}$ for EnumConst, Unary, PtrCmp, Binary, and Sizeof, because Primitive is never splitable.

The more complex part of the transformation is the handling of Member operations, defined in Table 5.1.

We assume that during the transformation of Struct and Union the relation between member positions in the original and transformed types have been noted. If the "old" position is $m$, the "new" position is $m^\circ$ for a non-splitable member, or $m^\mathbf{A}$ and $m^\mathbf{B}$ for the two components of a splitable member.

The transformation is by case analysis on: (a) which variant of Struct (or Union) we select a member from, (b) the specialization variant of the *resulting* Pointer, and (c) the signature of the member we are selecting.

$$
\begin{array}{rcl}
\{\!|\mathsf{Var}(\tau, d)|\!\}^{\circ} & \Longrightarrow & \left\{ \begin{array}{l} \mathsf{Var}(\{\!|\tau|\!\}^{\circ}, d^{\mathbf{A}}) \\ \mathsf{Var}(\{\!|\tau|\!\}^{\circ}, d^{\circ}) \end{array} \right. \\
\{\!|\mathsf{Var}(\tau, d)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{Var}(\{\!|\tau|\!\}^{\mathbf{A}}, d^{\mathbf{A}}) \\
\{\!|\mathsf{Var}(\tau, d)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{Var}(\{\!|\tau|\!\}^{\mathbf{B}}, d^{\mathbf{B}}) \\
\{\!|\mathsf{FunAdr}(\tau, d)|\!\}^{\circ} & \Longrightarrow & \mathsf{FunAdr}(\{\!|\tau|\!\}^{\circ}, d) \\
\{\!|\mathsf{EnumConst}(\tau, \epsilon)|\!\}^{\circ} & \Longrightarrow & \mathsf{EnumConst}(\{\!|\tau|\!\}^{\circ}, \epsilon) \\
\{\!|\mathsf{Const}(\tau, c)|\!\}^{\circ} & \Longrightarrow & \mathsf{Const}(\{\!|\tau|\!\}^{\circ}, c) \\
\{\!|\mathsf{Null}(\tau)|\!\}^{\circ} & \Longrightarrow & \mathsf{Null}(\{\!|\tau|\!\}^{\circ}) \\
\{\!|\mathsf{Null}(\tau)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{Null}(\{\!|\tau|\!\}^{\mathbf{A}}) \\
\{\!|\mathsf{Null}(\tau)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{Null}(\{\!|\tau|\!\}^{\mathbf{B}}) \\
\{\!|\mathsf{Unary}(\tau, \diamond, e_1)|\!\}^{\circ} & \Longrightarrow & \mathsf{Unary}(\{\!|\tau|\!\}^{\circ}, \diamond, \{\!|e_1|\!\}^{\circ}) \\
\{\!|\mathsf{PtrArith}(\tau, e_1, \circ, e_2)|\!\}^{\circ} & \Longrightarrow & \mathsf{PtrArith}(\{\!|\tau|\!\}^{\circ}, \{\!|e_1|\!\}^{\circ}, \circ, \{\!|e_2|\!\}^{\circ}) \\
\{\!|\mathsf{PtrArith}(\tau, e_1, \circ, e_2)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{PtrArith}(\{\!|\tau|\!\}^{\mathbf{A}}, \{\!|e_1|\!\}^{\mathbf{A}}, \circ, \{\!|e_2|\!\}^{\circ}) \\
\{\!|\mathsf{PtrArith}(\tau, e_1, \circ, e_2)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{PtrArith}(\{\!|\tau|\!\}^{\mathbf{B}}, \{\!|e_1|\!\}^{\mathbf{B}}, \circ, \{\!|e_2|\!\}^{\circ}) \\
\{\!|\mathsf{PtrCmp}(\tau, e_1, \circ, e_2)|\!\}^{\circ} & \Longrightarrow & \left\{ \begin{array}{l} \mathsf{PtrCmp}(\{\!|\tau|\!\}^{\circ}, \{\!|e_1|\!\}^{\mathbf{B}}, \circ, \{\!|e_2|\!\}^{\mathbf{B}}) \\ \mathsf{PtrCmp}(\{\!|\tau|\!\}^{\circ}, \{\!|e_1|\!\}^{\circ}, \circ, \{\!|e_2|\!\}^{\circ}) \end{array} \right. \\
\{\!|\mathsf{Binary}(\tau, e_1, \circ, e_2)|\!\}^{\circ} & \Longrightarrow & \mathsf{Binary}(\{\!|\tau|\!\}^{\circ}, \{\!|e_1|\!\}^{\circ}, \circ, \{\!|e_2|\!\}^{\circ}) \\
\{\!|\mathsf{Member}(\tau, e_1, m)|\!\} & \Longrightarrow & \text{(see Table 5.1)} \\
\{\!|\mathsf{Array}(\tau, e_1)|\!\}^{\circ} & \Longrightarrow & \mathsf{Array}(\{\!|\tau|\!\}^{\circ}, \{\!|e_1|\!\}^{\circ}) \\
\{\!|\mathsf{Array}(\tau, e_1)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{Array}(\{\!|\tau|\!\}^{\mathbf{A}}, \{\!|e_1|\!\}^{\mathbf{A}}) \\
\{\!|\mathsf{Array}(\tau, e_1)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{Array}(\{\!|\tau|\!\}^{\mathbf{B}}, \{\!|e_1|\!\}^{\mathbf{B}}) \\
\{\!|\mathsf{DeRef}(\tau, e_1)|\!\}^{\circ} & \Longrightarrow & \mathsf{DeRef}(\{\!|\tau|\!\}^{\circ}, \{\!|e_1|\!\}^{\circ}) \\
\{\!|\mathsf{DeRef}(\tau, e_1)|\!\}^{\mathbf{A}} & \Longrightarrow & \mathsf{DeRef}(\{\!|\tau|\!\}^{\mathbf{A}}, \{\!|e_1|\!\}^{\mathbf{A}}) \\
\{\!|\mathsf{DeRef}(\tau, e_1)|\!\}^{\mathbf{B}} & \Longrightarrow & \mathsf{DeRef}(\{\!|\tau|\!\}^{\mathbf{B}}, \{\!|e_1|\!\}^{\mathbf{B}}) \\
\{\!|\mathsf{Cast}(\tau, e_1)|\!\}^{\circ} & \Longrightarrow & \mathsf{Cast}(\{\!|\tau|\!\}^{\circ}, \{\!|e_1|\!\}^{\circ}) \\
\{\!|\mathsf{Sizeof}(\tau, \tau')|\!\}^{\circ} & \Longrightarrow & \mathsf{Sizeof}(\{\!|\tau|\!\}^{\circ}, \{\!|\tau'|\!\}^{\circ})
\end{array}
$$

*Figure 5.2: The splitter transformation on expressions*

| Struct variant | Result type $\tau$ | Signature of member | Transformation rules |
|---|---|---|---|
| $\mathsf{Struct}_d$ | $\mathsf{Pointer}_d$ | $(1,0)$ | $\{\!\|e\|\!\}^\circ \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^\circ, \{\!\|e_1\|\!\}^\circ, m^\circ)$ |
| $\mathsf{Struct}_d$ | $\mathsf{Pointer}_s$ | $(\pm 1,0)$ | $\{\!\|e\|\!\}^\circ \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^\circ, \{\!\|e_1\|\!\}^\circ, m^\circ)$ |
| $\mathsf{Struct}_p$ | $\mathsf{Pointer}_d$ | $(1,0)$ | $\{\!\|e\|\!\}^\circ \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^\circ, \{\!\|e_1\|\!\}^\circ, m^\circ)$ |
| $\mathsf{Struct}_p$ | $\mathsf{Pointer}_d$ | $(1,1)$ | $\{\!\|e\|\!\}^\circ \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^\circ, \{\!\|e_1\|\!\}^\circ, m^{\mathbf{A}})$ |
| $\mathsf{Struct}_p$ | $\mathsf{Pointer}_s$ | $(\pm 1,0)$ | $\{\!\|e\|\!\}^\circ \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^\circ, \{\!\|e_1\|\!\}^{\mathbf{A}}, m^\circ)$ |
| $\mathsf{Struct}_p$ | $\mathsf{Pointer}_s$ | $(0,1)$ | $\{\!\|e\|\!\}^\circ \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^\circ, \{\!\|e_1\|\!\}^{\mathbf{B}}, m^\circ)$ |
| $\mathsf{Struct}_s$ | $\mathsf{Pointer}_s$ | $(0,1)$ | $\{\!\|e\|\!\}^\circ \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^\circ, \{\!\|e_1\|\!\}^\circ, m^\circ)$ |
| $\mathsf{Struct}_p$ | $\mathsf{Pointer}_p$ | $(1,1)$ | $\{\!\|e\|\!\}^{\mathbf{A}} \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^{\mathbf{A}}, \{\!\|e_1\|\!\}^{\mathbf{A}}, m^{\mathbf{A}})$ |
| | | | $\{\!\|e\|\!\}^{\mathbf{B}} \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^{\mathbf{B}}, \{\!\|e_1\|\!\}^{\mathbf{B}}, m^{\mathbf{B}})$ |
| $\mathsf{Struct}_p$ | $\mathsf{Pointer}_s$ | $(\pm 1,1)$ | $\{\!\|e\|\!\}^{\mathbf{A}} \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^{\mathbf{A}}, \{\!\|e_1\|\!\}^{\mathbf{A}}, m^{\mathbf{A}})$ |
| | | | $\{\!\|e\|\!\}^{\mathbf{B}} \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^{\mathbf{B}}, \{\!\|e_1\|\!\}^{\mathbf{B}}, m^{\mathbf{B}})$ |
| $\mathsf{Struct}_p$ | $\mathsf{Pointer}_s$ | $(0,2)$ | $\{\!\|e\|\!\}^{\mathbf{A}} \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^{\mathbf{A}}, \{\!\|e_1\|\!\}^{\mathbf{B}}, m^{\mathbf{A}})$ |
| | | | $\{\!\|e\|\!\}^{\mathbf{B}} \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^{\mathbf{B}}, \{\!\|e_1\|\!\}^{\mathbf{B}}, m^{\mathbf{B}})$ |
| $\mathsf{Struct}_s$ | $\mathsf{Pointer}_s$ | $(0,2)$ | $\{\!\|e\|\!\}^{\mathbf{A}} \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^{\mathbf{A}}, \{\!\|e_1\|\!\}^\circ, m^{\mathbf{A}})$ |
| | | | $\{\!\|e\|\!\}^{\mathbf{B}} \Longrightarrow \mathsf{Member}(\{\!\|\tau\|\!\}^{\mathbf{B}}, \{\!\|e_1\|\!\}^\circ, m^{\mathbf{B}})$ |

*Table 5.1: Splitter transformation for* $e = \mathsf{Member}(\tau, e_1, m)$

Table 5.1 was constructed by simply enumerating the possible cases and thinking about what the transformation should do in each of them. We give examples of the reasoning for two of the cases; the others are similar.

- To select a member from a $\mathsf{Struct}_p$, producing a $\mathsf{Pointer}_d$, when the member's binding-time type has signature $(1,1)$. The $\mathsf{Pointer}_d$ result is never splitable, so the transformation we must define is $\{\!\|e\|\!\}^\circ$. The results of the transformation should be a $\mathsf{Pointer}_d$ to the selected member's dynamic component, which is the $m^{\mathbf{A}}$th member of the **A** component of the $\mathsf{Struct}_p$.

  From the typing rules on page 88 we can see that the operand $e_1$ must be a $\mathsf{Pointer}_d$.

  Therefore, the transformed expression should start by evaluatin $\{\!\|e_1\|\!\}^\circ$ which produces a $\mathsf{Pointer}_d$ to the $\mathsf{Struct}_p$'s **A** component, and then extract member $m^{\mathbf{A}}$ from that.

- To select a member from a $\mathsf{Struct}_p$, producing a $\mathsf{Pointer}_s$, when the member's binding-time type has signature $(0,2)$. The $\mathsf{Pointer}_s$ result is splitable, so we must define the $\{\!\|e\|\!\}^{\mathbf{A}}$ and $\{\!\|e\|\!\}^{\mathbf{B}}$ transformations. The results of the transformation should be $\mathsf{Pointer}_s$s to either of the member's two components. Because both components are static, they are both part of the **B** component of the $\mathsf{Struct}_p$. Their positions are, of course, $m^{\mathbf{A}}$ and $m^{\mathbf{B}}$.

  From the typing rules on page 88 we can see that the operand $e_1$ can be a $\mathsf{Pointer}_s$ or a $\mathsf{Pointer}_p$. In either case the **B** part of $e_1$'s value is a $\mathsf{Pointer}_s$ to the **B** part of the $\mathsf{Struct}_p$.

  Therefore, the transformed expressions shoud first evaluate $\{\!\|e_1\|\!\}^{\mathbf{B}}$ and then extract either member $m^{\mathbf{A}}$ or member $m^{\mathbf{B}}$ from that.

$$\{\!|\mathsf{Primitive\text{-}lift}^{d}_{s}(e)|\!\}^{\circ} \implies \mathsf{Primitive\text{-}lift}^{d}_{s}(\{\!|e|\!\}^{\circ})$$
$$\{\!|\mathsf{Pointer\text{-}lift}^{d}_{1s}(e)|\!\}^{\circ} \implies \mathsf{Pointer\text{-}lift}^{d}_{1s}(\{\!|e|\!\}^{\circ})$$
$$\{\!|\mathsf{Pointer\text{-}lift}^{d}_{2s}(e)|\!\}^{\circ} \implies \mathsf{Pointer\text{-}lift}^{d}_{1s}(\{\!|e|\!\}^{\mathbf{A}})$$
$$\{\!|\mathsf{Pointer\text{-}lift}^{d}_{p}(e)|\!\}^{\circ} \implies \{\!|e|\!\}^{\mathbf{A}}$$
$$\{\!|\mathsf{Pointer\text{-}lift}^{p}_{2s}(e)|\!\}^{\mathbf{A}} \implies \mathsf{Pointer\text{-}lift}^{d}_{1s}(\{\!|e|\!\}^{\mathbf{A}})$$
$$\{\!|\mathsf{Pointer\text{-}lift}^{p}_{2s}(e)|\!\}^{\mathbf{B}} \implies \{\!|e|\!\}^{\mathbf{B}}$$

*Figure 5.3: Splitter transformation for lifts. We notate a lift of* $e$ *from* $\mathsf{Foobar}_{\alpha}$ *to* $\mathsf{Foobar}_{\beta}$ *as* $\mathsf{Foobar\text{-}lift}^{\beta}_{\alpha}$*. Also, for the purpose of the case analysis here we distinguish between splitable and non-splitable* $\mathsf{Pointer}_{s}$*'s by calling them* $\mathsf{Pointer}_{2s}$ *and* $\mathsf{Pointer}_{1s}$*, respectively.*

### How to split lifts

The transformation rules in Figure 5.2 and Table 5.1 do not take lifts into account.

If, for a moment, we assume that lifts were explicitly present in a binding-time annotated program, we could define lifts as in Figure 5.3.

Now, in reality lifts are implicit in the output from the BTA. We decide they should also be implicit in the output from the splitter. If we remove all of the lift tags from Figure 5.3, we end up with a number of no-op rules and two rules that both read,

$$\{\!|e|\!\}^{\circ} \implies \{\!|e|\!\}^{\mathbf{A}}$$

So we can implement lifts in the splitter simply by an universal fall-back rule:

- *If* $\{\!|e|\!\}^{\circ}$ *is asked for and the (pre-lift) type of* $e$ *is splitable, produce* $\{\!|e|\!\}^{\mathbf{A}}$ *instead.*

## 5.5 How to split statements

The splitter transformation does not change the overall structure of function bodies, but each statement and jump is replaced by a tranformed statement. For some statements and jumps the splitter transformation produces more than one statement.

### How to split assignments

If the type of the assigned value is non-splitable, it is easy. Because a pointer to a non-splitable type is never splitable itself, we can simply subject each of the expression to the $\{\!|\cdot|\!\}^{\circ}$ transformation.

$$\{\!|\mathsf{Assign}(e, e')|\!\} \implies \mathsf{Assign}(\{\!|e|\!\}^{\circ}, \{\!|e'|\!\}^{\circ})$$

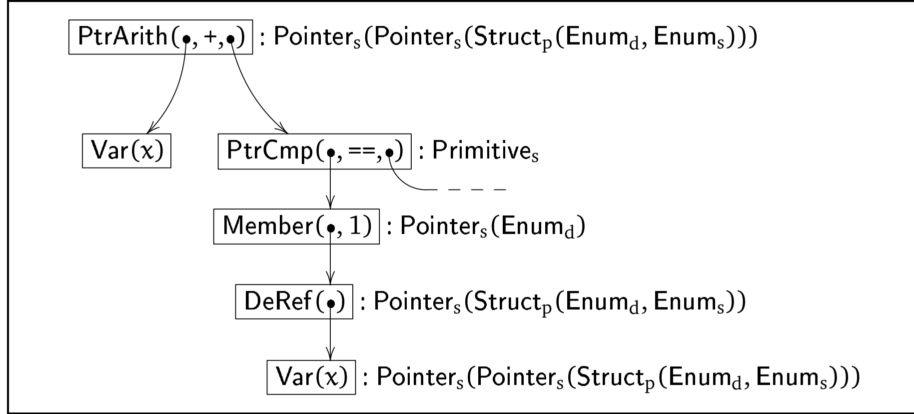



*Figure 5.4: An expression $e$ where evaluating $\{\!|e|\!\}^{\mathbf{B}}$ involves reading the value pointed to by $\{\!|e|\!\}^{\mathbf{A}}$.*

Matters are more complex if the type of the assigned value is splitable. In that case the typing rules makes sure that the target location pointer is itself splitable[2]. The basic idea is to transform the assignment into two assignments:

$$\{\!|\mathsf{Assign}(e, e')|\!\} \quad \Longrightarrow \quad \mathsf{Assign}(\{\!|e|\!\}^{\mathbf{A}}, \{\!|e'|\!\}^{\mathbf{A}})\,;\ \mathsf{Assign}(\{\!|e|\!\}^{\mathbf{B}}, \{\!|e'|\!\}^{\mathbf{B}})$$

We have to be careful, however: what if the evaluation of $\{\!|e|\!\}^{\mathbf{B}}$ or $\{\!|e'|\!\}^{\mathbf{B}}$ requires reading from an object that gets overwritten by the first assignment? Then the splitter might change the semantics of the program.

When we faced the same problem in Section 2.6.2 we were able to get around it by applying common sense: the component that was assigned "first" only existed in $p_{res}$, so there was no way it could possible influence the value of the $p_{gen}$ value that was assigned in the "second" assignment.

We are not so lucky this time if the type of the assigned value has signature $(0, 2)$. Then its $\mathbf{A}$ component is not dynamic. Indeed it is possible to construct an expression $e$ where evaluating $\{\!|e|\!\}^{\mathbf{B}}$ may involve reading the value pointed to by $\{\!|e|\!\}^{\mathbf{A}}$, as shown on Figure 5.4.

The cure in this case is to let the splitter create a new temporary variable $d_0$, and store in it the value of a subexpression $e_0$ of $\{\!|e|\!\}^{\mathbf{B}}$ or $\{\!|e'|\!\}^{\mathbf{B}}$ before the assignment through $\{\!|e|\!\}^{\mathbf{A}}$, such that the offending read is contained in $e_0$.

We do not want blindly to create temporary variables for each assignment to a splitable location in the program, because too many variables tend to confuse compilers. The questions are now: (1) is a given assignment dangerous? and (2) how should we choose the subexpression to precompute?

Define a subexpression $e_{00}$ of $e$ or $e'$ to be **dangerous** if the type of $e_{00}$ is a splitable $\mathsf{Pointer}_s$ and $\{\!|e|\!\}^{\mathbf{B}}$ or $\{\!|e'|\!\}^{\mathbf{B}}$ contains $\{\!|e_{00}|\!\}^{\mathbf{A}}$. We want to find a strategy that precomputes all dangerous subexpressions.

Define a $\mathsf{Member}$ expression to be **suspect** if it creates a $\mathsf{Pointer}_s$ to a dynamic member of a $\mathsf{Struct}_p$. It can then be seen by induction over the transformation

[2]A $\mathsf{Pointer}_d$ may point to a partially static (hence splitable) type and is not splitable itself. That combination is explicitly forbidden for assignments, however.

definitions that $\{\!|e_{00}|\!\}^{\mathbf{A}}$ can only be in any $\{\!|\cdot|\!\}^{\mathbf{B}}$ expression if $e_{00}$ it has a suspect expression among its ancestors.

A suspect subexpression is a **candidate** if it has no suspect ancestors and has a subexpression with splitable $\mathsf{Pointer}_{s/r}$ type. Then any dangerous subexpression has precisely one candidate ancestor.

Now we replace each candidate $e_c$ with a temporary variable $d_c$ and prepend the transformed assignments with $\mathsf{Assign}(\mathsf{Var}(d_c), \{\!|e_c|\!\}^{\circ})$. That eliminates the danger of the first of the transformed assignments affecting the meaning of the second one.

**How to split function calls**

Given the statement $\mathsf{Call}(e, e_f, e_1, ..., e_n)$.

If $e$ is absent (i.e., the return value from the called function is `void` or ignored) or a pointer to a non-splitable type, the transformation is easy. We can then transform the statement into

$$\mathsf{Call}(\{\!|e|\!\}^{\circ}, \{\!|e_f|\!\}^{\circ}, \left\{\{\!|e_i|\!\}^{\circ} \,\square\, \{\!|e_i|\!\}^{\mathbf{A}}, \{\!|e_i|\!\}^{\mathbf{B}} \mid 1 \leq i \leq n\right\})$$

where $\{\!|e_i|\!\}^{\circ} \,\square\, \{\!|e_i|\!\}^{\mathbf{A}}, \{\!|e_i|\!\}^{\mathbf{B}}$ means a choice based on whether the parameter is splitable according to the type of $e_f$.

The binding-time type system allows the return value of functions to be lifted at the call statement. One would expect that lifts from a splitable to a non-splitable type (i.e., lifts from $\mathsf{Pointer}_p$ to $\mathsf{Pointer}_d$) could create problems here. It turns out, however, that the above transformation works fine when calling a function that returns $\mathsf{Pointer}_p$ though $e$ is a pointer to $\mathsf{Pointer}_d$. This is because the transformed version of a function returning a splitable actually returns the $\mathbf{A}$ component of the return value, which in the case of $\mathsf{Pointer}_p$ is excactly the $\mathsf{Pointer}_d$ that the return value should be lifted to.

Now consider the case where $e$ is a pointer to a splitable type. Then, the called function does actually return a splitable type, too. We then transform the call to

$$\begin{array}{l}\mathsf{Call}(\{\!|e|\!\}^{\mathbf{A}}, \{\!|e_f|\!\}^{\circ}, \left\{\{\!|e_i|\!\}^{\circ} \,\square\, \{\!|e_i|\!\}^{\mathbf{A}}, \{\!|e_i|\!\}^{\mathbf{B}} \mid 1 \leq i \leq n\right\}); \\ \mathsf{Assign}(\{\!|e|\!\}^{\mathbf{B}}, \mathsf{DeRef}(\mathsf{Var}(d_{\mathcal{G}(f)})))\end{array}$$

where $d_{\mathcal{G}(f)}$ is the return variable shared by the functions in $e_f$'s points-to set.

If the signature of the return type is $(0, 2)$, there is again a risk that the assignment through $\{\!|e|\!\}^{\mathbf{A}}$ may have invalidated the value of $\{\!|e|\!\}^{\mathbf{B}}$ (any side effects from the called function itself are not allowed by the C standard to affect the destination expression). Therefore we perform the same candidate computation and temporary variable creation on $e$ as we do for assignments.

**How to split jumps**

The transformation of jumps is defined in Figure 5.5.

$$
\begin{array}{rcl}
\{\!|\mathsf{Goto}(b)|\!\} & \Longrightarrow & \mathsf{Goto}(b) \\
\{\!|\mathsf{If}(e, b_{\text{then}}, b_{\text{else}})|\!\} & \Longrightarrow & \mathsf{If}(\{\!|e|\!\}^{\circ}, b_{\text{then}}, b_{\text{else}}) \\
\{\!|\mathsf{Return}()|\!\} & \Longrightarrow & \mathsf{Return}() \\
\{\!|\mathsf{Return}(e)|\!\} & \Longrightarrow & \begin{cases} \mathsf{Assign}(\mathsf{Var}(d_{\mathcal{G}(f)}), \{\!|e|\!\}^{\mathbf{B}});\ \mathsf{Return}(\{\!|e|\!\}^{\mathbf{A}}) \\ \mathsf{Return}(\{\!|e|\!\}^{\circ}) \end{cases}
\end{array}
$$

*Figure 5.5: Splitter transformation for jumps. For* Return *jumps, the branch taken is selected by whether the function's return type is splitable or not.*

# Chapter 6

# Conclusion

In this chapter I put this report in relation to earlier work on C-Mix and suggest directions for further development and research.

## 6.1 Contributions relative to Andersen (1994)

The following is a list of the major differences between the C-Mix descibed by Andersen (1994) and C-Mix$_{II}$ as described by this report.

The list does not exhaustively specify the current system's relation to the old prototype; there has been numerous improvements in areas not covered by this report which are not mentioned here.

- **Core C** (as described in Appendix A) is a lot more idealized and orthogonal than the "abstract C" that Andersen (1994, Section 2.2) uses. In particular, the fact that expressions never have side effects and always have rvalue semantics make many of the analyses, including the binding-time analysis, easier to specify.

  The current Core C was designed jointly by Jens Peter Secher and me.

- The support for **partially static data** is totally new compared to Andersen (1994). The term "partially static" has been used about C-Mix features for a long time, but its meaning has not always been clearly defined. In some passages of Andersen (1994, e.g., Section 3.6.1), "partially static" seems to merely mean what in my terminology would be "not faithful".

  The partial staticness we talk about in this report was apparently first introduced in a C-Mix context by Peter Holst Andersen. Since then the idea has been refined and mutated by Arne Glenstrup, Jens Peter Secher, and me.

  Original contributions[1] of this reports are the $\mathsf{Pointer}_\mathrm{p}$ and $\mathsf{Array}_\mathrm{p}$ ideas, as well as the idea of a $\mathsf{Pointer}_\mathrm{d}$ to partially static data.

  There is some prior art (in the form of C-Mix$_{II}$ design documents) on specifications for the splitter transformation, but to the best of my knowledge this report contains (Section 5.5) the first explicit analysis of the

[1] In the sense that if any of the other mentioned persons have had similar ideas, they at least did not tell me.

semantic soundness of splitting a static assignment into two static assignments and of how to treat splitable return values.

The didactic trick of using "backed-up ints" (Section 2.6) for introducing basic splitter ideas is new with this report. I am not the one to judge wheter it actually helps more than it confuses.

- The overall description of the **binding-time type system** , with concepts such as specialization variant, signature, and faithful, is an original contribution of this report.
- Andersen (1994) did not describe **lifts of pointer values**—he treated access through a static pointer to dynamic data as a special construction.

  We supersede the awkward and erroneous rules in Andersen (1994, Section 3.6.2) with a general concept of pointer lifts, subject to the in-scope rule of Section 3.4.4. It evades my memory who first came up with that idea.

  The strategy for separating pointer lifts from pointer arithmetic that I describe in Box 3.2 on page 67, including the "carry" concept, is an original contribution of this report. In fact, it is not implemented in C-Mix/II yet. The current C-Mix/II instead allows $\mathsf{Pointer}_s$ to be lifted to $\mathsf{Pointer}_d$ unless the pointer may (by the pointer analysis) point to an array element. That strategy does not correctly handle programs that treat single variables as one-element arrays.
- The exact conventions for calling **generating functions** are new, and an original contribution of this report. The have the dual advantage of (1) allowing pointers to external functions to be mixed with pointers to generating functions, and (2) allowing specialization-time inlining (Section 2.4.4) of functions without needing special cooperation from the caller.

  Andersen (1994, Section 3.5.3) did describe the general principles of specialization-time inlining, but did not go into detail about changes to the calling conventions for generating functions (ordinarily, he expected generating functions to return the name of a residual function, which clearly is meaningless in the case of inlining). Inlining was never implemented in the prototype C-Mix.
- The ability of the current C-Mix/II to **inline** functions replaces a hack (apparently never described in writing) in the old C-Mix which consisted of dividing the subject functions into a "static program" and a "dynamic program".
- Andersen (1994, Sections 3.5.2 and 6.2)'s definition of "**under dynamic control**" was found to be erroneous (which did not hurt much, because it was never implemented in the C-Mix prototype). We—Arne Glenstrup, Jens Peter Secher and I—corrected the definition (Section 3.4.3) and integrated its enforcement in the binding-time analysis as described in this report.
- The "**logic based BTA**" paradigm I describe in Section 4.1.2 is—to the extent that such a simple idea *can* be original—an original contribution of this report.
- Our handling of **external function calls** in Section 2.3 is more systematic an flexible than in Andersen (1994, Section 3.8)

- Andersen (1994, Section 3.4.5) never really defined which **transition compression** strategy C-Mix should use. We do in Section 2.2.6.

## 6.2 Problems not yet satisfactorily solved

The following is a list of known problems which may cause C-Mix/II to fail or give incorrect results, for which no good solution has yet been proposed.

- How can we mecanically avoid **infinite specialization** without sacrificing unacceptably much of the potential for specialization?

  Some results do exist in the area of *termination analysis* for partial evaluation, but far as I know they are still a long way from being applicable to “difficult” languages such as C.

- How can we avoid **commiting static run-time errors speculatively** (Section 2.2.5)?

  This risk seems to be an intrinsic property of the generating extension paradigm. We consider it a non-solution simply to conclude that generating extensions are a bad idea; but does that leave us any other options than accepting that things sometimes go wrong?

- How does one treat **functions with an external success state** such as `log` or `sqrt` safely? (footnote on page 32).

  These functions behave mostly like pure mathematical functions, and most real programs treat them as such. But in some rare cases they signal an error by performing a side effect on a global variable `errno`.

  Is there any way for C-Mix/II to acknowledge this use of `errno` *without* essentially forcing all call to any `errno`-using function to have the same binding time?

- How does one specialize **functions with a variable number of arguments**? Currently C-Mix/II simply refuses to specialize a program that includes the `<stdarg.h>` header.[2]

- How does one allow **dynamic pointers to specializable functions**? A proposal is given in Box 3.7 on page 84, but it remains to be seen whether it can be integrated in the binding-time analysis.

## 6.3 Directions for further extension

The following is a list of proposals for future enhancements of the C-Mix/II system in the areas covered by this report. These proposals add strengh to the specialization process rather than fix actual malfunctions of the current implementation. They are roughly sorted in order of intellectual difficulty, ranging from “can be done now” to “needs extensive new theory”.

- Implement **partially static data** as described in this report. The algorithms and transformation rules have been developed; all they need is to be coded into the system.

[2] This and the following question are not merely classified as extension proposals, because the lack of an answer make it impossible for C-Mix/II to treat certain portable C programs at all, even by making everything dynamic.

- Implement lifts for Enum typles, leading to the proper symbolic constants appearing in the residual program. This is described in Box 3.1 on page 65, but we judge the practical utility to be small.
- Implement $\mathsf{Struct}_{sy}$ as described in Box 3.5 on page 79: a struct whose members become separate variables in $p_{gen}$.
- Relax the rule of **non-local side effects**, as described in Box 3.8 on page 95.
- Allow some form of **function pointer lifting**, for example one of the ones discussed in Box 3.7 on page 84.
- Implement $\mathsf{Array}_{dy}$ as described in Box 3.4 on page 73: an array of dynamic data that can be indexed statically as well as dynamically. This involves designing support for it into the binding-time analysis and the strategy for controlling pointer lifts.
- Relax the **second union rule** to allow $\mathsf{Pointer}_{d}$s to partially static data to occupy the affected portions of a Union.

  In Box 3.6 on page 81 it is argued the such relaxation would ruin the uniqueness of most static binding-time annotations, so the key part of an implementation of this would be a mechanism to allow the user to select which of several versions of the second union rule she wants used for a particular Union (or a particular pointer inside a particular Union).
- Allow pointer lifts in inlined functions even if the pointer may point to one of the caller's variables. This was suggested on page 96. The pointer analysis will need to be made substantially more precise if it must be able to decide "always a local variable in the caller" except in trivial cases. Another problem is to develop a strategy for selecting calls to be unfolded *before* the binding-time analysis is completed.
- Change the binding-time analysis to be **polyvariant**, meaning that the binding-time annotations on a function can depend on where it is called from.

  Andersen (1994, Section 2.3.2) describes a solution method based on conceptually duplicating each function as many time as there are calls to it. The problem here is that even with the cautious recursion elimination strategy that section defines, exponentially many copies of a function might be needed in realistic programs.
- Relax the **unique-return-state requirement**, implementing instead the `statecode` solution hinted at in Figure 2.29. This would reqiure finding acceptable answers to the objections given in Box 2.1 on page 45.
- Develop a theory of specializing object-oriented code and write C++Mix.

# Appendix A

# Core C[1]

Input to C-Mix$_{II}$ is any strictly confoming ISO C program ANSI (1990). In most analyses[2] we represent it internally in the C-Mix$_{II}$ system in a reduced representation that we call Core C. The reason for this is to make it easier both to implement and to ensure correctness of various analyses and transformations.

Core C is sufficiently close to C that one can view it as a restricted subset of ISO C. However, several details are treated more orthogonally than the real language. The most important differences are:

- Core C has no structured control constructs apart from function calls. The control statements of ISO C are translated into a "flat" flow graph where only `goto` statements and "`if (`$e$`) goto` $l_1$`; else goto` $l_2$`;`" appear.
- Core C *expressions* have no side effects. Side effects are produced by primitive *statements*. (Expressions can still have the "side effect" of trying to derefence a null pointer and killing the program).
- Core C has no lvalue expressions. Thus there is no generic "address-of" operator, but the constructs that in ISO C produce lvalues (such as naming a variable or selecting a member of a struct) have Core C counterparts that evaluate to pointers to the location the lvalue references.
- Core C has no implicit conversions or coercions. Any such thing—ranging from promoting an `int` to `long` so one can add it to another `long`, to the "pointer decay rule" for arrays—is represented as an explicit construct.
- Core C has only two lifetimes for variables: they are either global or local to a function. `static` variables are made global, and `auto` or `register` variables declared in compound statements are collected in a single list of the function's local variables.

Core C also has an element of "non-ortogonality": where a construction can easily be replaced with an equivalent representation in a certain context, we generally disallow it in this description.

Examples of this strategy are the decisions that pointer-valued expressions cannot be direct operands to the logical operators `!`, `||`, and `&&`, or the rule that

[1] This appendix is a reproduction of a section of the source documentation for C-Mix$_{II}$.

[2] The only exceptions are an initial conventional type check and the resolution of identifiers in user annotation directives.

a suitable expression *must* be present in the return statements in a nonvoid function.

This is intended to simplify the analyses: e.g., one could imagine an analysis that has to do something special when a pointer value is "consumed"; such an analysis would benefit from not having to check the operand types of the logical operators for pointerness.

Internally Core C is represented as heavily interlinked heap allocated C++ objects. However, it is instructive to think of a Core C program as a tree structure where cross-links occasionally appear as payload data in the nodes—even if the cross-links are physically represented as the same types of pointers that parent nodes use to link to their children. The grammar for this tree appears in figure A.1.

The grammar uses the following conventions:

- c is any (non-string) constant.
- id is a name from the original ISO C program.
- m is the ordinal number of a struct or union member.
- x? is an optional x.
- $\vec{x}$ is a list of x. Lists are represented by the Plist<T> template for a list of pointers to T, defined in Plist.h. These lists support STL-like iterators.
- $\langle\!|x|\!\rangle$ is a cross-link to an element somewhere else in the program. This notation is not used for types (t) because the structural sharing rules are not as simple as for the other classes. A detailed description appears in section A.6.1.

The definitions of the C++ classes that make up the actual Core C representation are defined in the source file corec.h. The auxilliary types whose names do not start with C_ are defined in tags.h.

The Core C structures use tags and unions; not inheritance and virtual functions. The reason for this is that the core language is not expected to change, whereas the various kinds of analyses are very likely to change. In a traditional OO design, every construct should contain a virtual method that carried out the part of some analysis that applies for that particular construct. This would result in each analysis being distributed into several parts of the program, which would make C-Mix$_{II}$ hard to understand and maintain.

## A.1 Programs

A program is a collection of

- struct and union definitions.
- enum definitions.
- global variables (which include static and extern variables declared inside the ISO C program's functions).
- function definitions.
- "goal" function definitions.

| C++ class | grammar | comments |
|---|---|---|
| `C_Pgm` | $p ::= (\vec{\sigma}, \vec{\eta}, \vec{d}_v, \vec{d}_f, \vec{d}_{gf})$ | |
| `C_UserDef` | $\sigma ::= \mathsf{Struct}(id?, \vec{\epsilon}, \langle\vec{t}\rangle)$ | |
| (struct/union defn) | $\mid \mathsf{Union}(id?, \vec{\epsilon}, \langle\vec{t}\rangle)$ | |
| `C_EnumDef` | $\eta ::= \mathsf{Enum}(id?, \vec{\epsilon})$ | |
| (enum definition) | $\epsilon ::= (id, q, e?)$ | enum constant with possible value |
| `C_Decl` | $d ::= \mathsf{Var}(id?, q, t, \delta, i?, \vec{d})$ | |
| (declaration) | $\mid \mathsf{Fun}(id, t, \vec{d}_p, \vec{d}_v, \vec{b})$ | empty $\vec{b}$ means prototype |
| `C_Init` | $i ::= \mathsf{Simple}(e)$ | |
| (initializer) | $\mid \mathsf{FullyBraced}(\vec{\imath})$ | |
| | $\mid \mathsf{SloppyBraced}(\vec{\imath})$ | |
| | $\mid \mathsf{StringInit}(\mathit{string})$ | |
| `VariableMode` | $\delta ::=$ (choices omitted to save space) | encodes user annotations |
| `C_BasicBlock` | $b ::= (\vec{s}, j)$ | |
| `C_Jump` | $j ::= \mathsf{If}(e, \langle b_{then}\rangle, \langle b_{else}\rangle)$ | `if (`$e$`) goto` $b_{then}$`; else goto` $b_{else}$`;` |
| (control stmt) | $\mid \mathsf{Goto}(\langle b\rangle)$ | |
| | $\mid \mathsf{Return}(e?)$ | |
| `C_Stmt` | $s ::= \mathsf{Assign}(e, e')$ | `*`$e$`=`$e'$ |
| (statement) | $\mid \mathsf{Call}(e?, \gamma, e_f, \vec{e})$ | `*`$e$`=(*`$e_f$`)(`$\vec{e}$`)` |
| | $\mid \mathsf{Alloc}(e, d)$ | `*`$e$`=calloc(`$e'$`,sizeof (`$t_d$`))` |
| | $\mid \mathsf{Free}(, e')$ | `free(`$e'$`)` |
| `CallMode` | $\gamma ::=$ (choices omitted to save space) | encodes user annotations |
| `C_Expr` | $e ::= \mathsf{Var}(t, \langle d\rangle)$ | the address of the variable |
| (expression) | $\mid \mathsf{EnumConst}(t, \langle\epsilon\rangle)$ | an enum constant (always an `int`) |
| | $\mid \mathsf{Const}(t, c)$ | (constants can have any type) |
| | $\mid \mathsf{Null}(t)$ | null pointer constant |
| | $\mid \mathsf{Unary}(t, \diamond, e)$ | |
| | $\mid \mathsf{PtrArith}(t, e_1, \circ, e_2)$ | ptr $\circ$ num = ptr |
| | $\mid \mathsf{PtrCmp}(t, e_1, \circ, e_2)$ | ptr $\circ$ ptr = num |
| | $\mid \mathsf{Binary}(t, e_1, \circ, e_2)$ | num $\circ$ num = num |
| | $\mid \mathsf{Member}(t, e, m)$ | ptr to struct $\longrightarrow$ ptr to member |
| | $\mid \mathsf{Array}(t, e)$ | ptr to array $\longrightarrow$ ptr to first element |
| | $\mid \mathsf{DeRef}(t, e)$ | contents of pointed-to address |
| | $\mid \mathsf{Cast}(t, e)$ | |
| | $\mid \mathsf{SizeofType}(t, t')$ | |
| | $\mid \mathsf{SizeofExpr}(t, e)$ | |
| `UnOp` | $\diamond ::= \texttt{-} \mid \texttt{~} \mid \texttt{!}$ | |
| `BinOp` | $\circ ::= \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{+} \mid \texttt{-} \mid \texttt{«} \mid \texttt{»} \mid \texttt{<} \mid \texttt{>}$ | |
| | $\mid \texttt{<=} \mid \texttt{=>} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{\&} \mid \texttt{\vert} \mid \texttt{\^{}} \mid \texttt{\&\&} \mid \texttt{\vert\vert}$ | |
| `C_Type` | $t ::= \mathsf{Abstract}(id)$ | Non-arithmetic primitive type |
| (type) | $\mid \mathsf{Primitive}(id)$ | Arithmetic primitive type |
| | $\mid \mathsf{Enum}(\langle\eta\rangle)$ | Enumerated type |
| | $\mid \mathsf{Array}(t, e?, q)$ | Array of $q$ $t$, length $e$ |
| | $\mid \mathsf{Pointer}(t, q)$ | Pointer to $q$ $t$ |
| | $\mid \mathsf{FunPtr}(\mathsf{Fun}(t, \vec{t}))$ | Pointer to function |
| | $\mid \mathsf{Fun}(t, \vec{t})$ | Function of $\vec{t}$ returning $t$ |
| | $\mid \mathsf{StructUnion}(q, \langle\sigma\rangle, \vec{t})$ | Struct or union with member types $\vec{t}$ |
| `constvol` | $q ::= (\mathsf{Const}?, \mathsf{Volatile}?)$ | |

Figure A.1: The Core C grammar.

A struct and or definition consist of an optional tag and a list of member descriptions. They also have a list of all the (StructUnion) type nodes that reference them.

The member descriptions (denoted with $\epsilon$'s in the grammar) only include the member's name, type qualifiers, and and optional expression that denote a possible bitfield width. If a definition for the aggregate needs to be written, one needs to go via one of the referencing StructUnion nodes to find the type of each member. Aggregates with no referencing StructUnion should and need not exist in Core C at all.

## A.2 Declarations

Despite its name, the `C_Decl` class represents not only variable declaration but anything in the program the pointer analysis needs to say a pointer may point to. This is the reason why variables and functions are represented by the same construction.

The source file `ALoc.h` defines types and functions that manage unordered sets of declaration references, used by several of the analyses.

### A.2.1 Function declarations

$\mathsf{Fun}(\mathtt{id}, \mathtt{t}, \vec{\mathtt{d}}_p, \vec{\mathtt{d}}_e, \vec{\mathtt{b}})$ declarations are simple. They have a name and a type (always a Fun type), lists of formal parameter and local variable declarations, and a nonempty list of basic blocks. Execution of the function body starts at the first basic block in the list, and always ends with an explicit Return statement. Functions cannot return simply by "falling through the body".

External functions are not separately represented in the Core C program; references to them are Const expressions.

### A.2.2 Variable declarations

Var declarations are more complex. They always have a type but may have no name: sometimes the c2core phase needs to introduce new temporary variables. Each variable has a *varmode* that defines its linkage and any user annotations on it. A variable may have an *initializer* (section A.3).

Additionally, if the variable consists of sub-objects that can be individually pointed to (*i.e.*, if it has Array or StructUnion type), its declaration also includes one or more *subsidiary declarations* which represent the sub-objects in the pointer analysis and other analyses that depend on that. If the type is an array there is a single subsidiary declaration which collectively represents the elements of the array. If the type is a struct or union there is a subsidiary declaration for each of its members.

A subsidiary declaration per convention have no name or initializer; its varmodes specify internal linkage. However, it may contain its own subsidiaries, according to its type.

An example on the use of subsidiary declaration appears on figure A.4, page 146.

Declarations for formal parameters behave like variable declarations but never have initializers. So does the name-less declaration that is part of an Alloc

| Declaration | Initializer |
|---|---|
| `int a1[sizeof(int)][] = {1,2,3,4,5,6};` | SloppyBraced(1, 2, 3, 4, 5, 6) |
| `int a2[4][] = {1,2,3,4,5,6};` | FullyBraced(FullyBraced(1, 2, 3, 4), FullyBraced(5, 6)) |
| `int a3[sizeof(int)][] = {{1,2,3,4},{5,6}};` | FullyBraced(FullyBraced(1, 2, 3, 4), FullyBraced(5, 6)) |

Figure A.2: Examples of the initializer representation

statement; it collectively represents every block of memory that gets allocated by the statement.

## A.3 Initializers

An initializer is an optional part of a local or global variable declaration. Its structure resembles the C read syntax for initializers.

The FullyBraced($\vec{\iota}$) construction is used where it can be guaranteed that each of the elements of $\vec{\iota}$ initialize exactly one of the immediate subobjects of the aggregate initialized by the initializer.

Sometimes, however, that is not possible, and the structure of the read ISO C initializer is simply copied using SloppyBraced nodes. This reduces the precision of the analyses and prevents splitting of the initialized object into more than a single spectime or residual part.

A FullyBraced initializer can have a SloppyBraced node as one of its subinitializers.

## A.4 Statements

All statements have an optional left-side expression that, if present, points to the location where the result of the statement is stored. If the left side is not present, the result is ignored. Actually the left-side is only genuinely optional in the Call statement, but it is treated uniformly across all statement cases.

The Assign statement is a simple assignment; its only side effect is the assignment to the left-side. An Assign with an empty left-side would be a no-op, so is optimized away in the c2core phase.

The Call statement performs a call through a function pointer expression. The statement has a *callmode* attribute that tells when (at spectime or at residual time) the call should be made if it is a call to an external function. Specializable functions (ones for which we know a body) are called through static pointers, so for the purposes of callmode they are considered spectime.

Calls to the `malloc()` and `calloc()` standard library functions are intercepted in the c2core phase and translated into Alloc($e$, d) statements. The declaration d is used in the analyses to represent collectively all the pieces of memory allocated by that particular Alloc statement. The type of d is always an Array type; its size expression specifies the number of objects to allocate on the heap. Contrary to other array size expressions, this one does not need to be a constant expression; it is evaluated in the context of the Alloc statement. If the Array

type has no size expression, a single object is allocated. The left-hand side of the Alloc statement is never empty: an allocation that turns out to be ignored gets optimized away in the c2core phase.

The Free statement is the translation of a call to the `free()` standard library function. This function has a return type of `void`, so the left side of the statement must be empty.

### A.4.1 Control statements

The definitions of control statements and basic blocks is fairly straight-forward.

The Return statements are normalized so that the expression is present precisely if the function does not return `void`. If necessary, an anonymous global variable of the proper type is generated, and its (never assigned-to) value is returned.

## A.5 Expressions

A Core C expression has no side effects and always specifies a value, never a location. An ISO C expression that is used as an lvalue is translated to a Core C expression whose value is a pointer to the location of the lvalue.

Every expression has a type as its first attribute; this is the type of the expression's value. This attribute is treated uniformly across the possible expression forms, so that analyses can act on the type of an expression without doing a case analysis on its form. See section A.6.1 for a description of structural sharing between these types.

A $\mathsf{Var}(\mathtt{t}, \langle\!|\mathtt{d}|\!\rangle)$ expression evaluates to the *address* of the referenced variable or function. d must be a function, a global variable, or a formal parameter or local variable in the same function the expression appears in. It cannot be a subsidiary object declaration—pointers to structure members and array elements are produced by the Member and Array expressions.

EnumConst expressions model enumeration constants. They are different from Var expression because enumeration constants don't have addresses and can have different binding times each time they are mentioned. They are different from Const in that they need to be name managed.

The $\mathsf{Const}(\mathtt{t}, \mathtt{c})$ expression is used for numeric constants, *and* for identifiers declared as `well-known constant:` by user annotations. These constants can have arbitrarily complex types. This feature is used by the shadow headers—*e.g.*, by the `stdio.h` shadow which defines `stderr` to be a well-known constant of type "pointer to `FILE`". This differs from using a `const` variable in that different occurences of the well-known constant can have different binding times. This situation is similar to what should happen when external functions are mentioned. Thus, Const is also used for representing the expressions consisting of the name of an external function.

A special case $\mathsf{Null}(\mathtt{t})$ exist for expressing null pointer constants. This is different from $\mathsf{Const}(\mathtt{t}, \mathtt{NULL})$ in that a pointer Const points to some unknown external location, while Null never points to anything.

The only Unary operators in Core C are the various senses of negation. The operand is never a pointer—when the `!` operator is applied to a pointer expression c2core substitutes an explicit test against a null pointer constant.

The binary operators are translated to PtrArith, PtrCmp, or Binary expressions, depending on the types of the operands and result. If pointer expressions appear as operands to the && and || operators, explicit tests against null pointer constants are inserted. The short-circuiting behavior of the && and || operators in ISO C is only partly relevant in Core C because expressions do not have side effects.

The operand ordering to PtrCmp is normalized so the left operand is always the pointer and the right always the integer.

In the Member$(t, e, m)$ expression, $e$ must evaluate to a pointer to a struct or union. The Member expression produces a pointer to the $m$th of its members. This is sufficient to represent the dot operator of ISO C [3].

The Array$(t, e)$ expression produces a pointer to the first element of the array pointed to by the pointer that $e$ evaluates to. The ISO C counterpart is the implicit "pointer decay" that happens to expressions with array type.

Type casts, Cast$(t, e)$, can appear between any pair of types $t_e$ and $t$. That is theory at least; in practise programs with badly-behaved casts will not survive the type checker to tell the tale.

### A.5.1 Restrictions on expressions

- No expressions have Fun type.
- Expressions with Array type only appear as operands to SizeofExpr.
- Expressions with StructUnion type only appear as operands to SizeofExpr, as right-hand sides of Assign statements, or in argument lists of Call statements.
- The expression in a Simple initializer and the length of an Array type are constant expressions. They may not contain any DeRef expressions (except in operands to SizeofExpr), and Var expressions may not reference formal parameters or local variables.

### A.5.2 Displaying expressions

The left-hand side of figure A.3 defines the closest one gets to a fully compositional write syntax for Core C expression that could be parsed by an ISO C parser.

That mapping produces correct yet horribly unreadable expressions with a lot of spurious address-of and dereference operators. Therefore, by default the modules that unparse Core C (outcore and gegen) use the "optimizing" rendering defined at the right side of figure A.3.

The downside of the "optimizing" format is that not all Core C constructs are individually visible. This means that the annotated Core C output in this notation cannot include the annotation on every subexpression. When debugging C-Mix/II it is sometimes convenient to switch to the canonical representation in

[3] One might think that this assertion ignores the fact that there can be non-lvalue expression with struct or union type in ISO C. However, this only happens with function calls, simple assignment, and the ?: operator—neither of which are expressions in Core C. In each of these cases the c2core translation already uses a translation strategy that automatically provides an addres to a struct or union object with the same contents as the non-lvalue struct or union.

| Canonically | Optimizing |
|---|---|
| $(\!(\mathsf{Var}(t, \langle d \rangle))\!) = \texttt{\&id}_d$ | $(\!| e = \mathsf{Var}(\ldots) |\!) = \texttt{\&}(\!| e |\!)^*$ |
| $(\!(\mathsf{EnumConst}(t, \langle \epsilon \rangle))\!) = \texttt{id}_\epsilon$ | $(\!| \mathsf{EnumConst}(t, \langle \epsilon \rangle) |\!) = \texttt{id}_\epsilon$ |
| $(\!(\mathsf{Const}(t, c))\!) = c$ | $(\!| \mathsf{Const}(t, c) |\!) = c$ |
| $(\!(\mathsf{Null}(t))\!) = \texttt{NULL}$ | $(\!| \mathsf{Null}(t) |\!) = \texttt{NULL}$ |
| $(\!(\mathsf{Unary}(t, \diamond, e))\!) = \diamond(\!(e)\!)$ | $(\!| \mathsf{Unary}(t, \diamond, e) |\!) = \diamond(\!| e |\!)$ |
| $(\!(\mathsf{PtrArith}(t, e_1, \circ, e_2))\!) = (\!(e_1)\!) \circ (\!(e_2)\!)$ | $(\!| \mathsf{PtrArith}(t, e_1, \circ, e_2) |\!) = (\!| e_1 |\!) \circ (\!| e_2 |\!)$ |
| $(\!(\mathsf{PtrCmp}(t, e_1, \circ, e_2))\!) = (\!(e_1)\!) \circ (\!(e_2)\!)$ | $(\!| \mathsf{PtrCmp}(t, e_1, \circ, e_2) |\!) = (\!| e_1 |\!) \circ (\!| e_2 |\!)$ |
| $(\!(\mathsf{Binary}(t, e_1, \circ, e_2))\!) = (\!(e_1)\!) \circ (\!(e_2)\!)$ | $(\!| \mathsf{Binary}(t, e_1, \circ, e_2) |\!) = (\!| e_1 |\!) \circ (\!| e_2 |\!)$ |
| $(\!(\mathsf{Member}(t, e, m))\!) = \texttt{\&}(\!(e)\!)\texttt{->id}^m_{\sigma_t}$ | $(\!| e = \mathsf{Member}(\ldots) |\!) = \texttt{\&}(\!| e |\!)^*$ |
| $(\!(\mathsf{Array}(t, e))\!) = \texttt{\&(*}(\!(e)\!)\texttt{)[0]}$ | $(\!| \mathsf{Array}(t, e) |\!) = (\!| e |\!)^*$ |
| $(\!(\mathsf{DeRef}(t, e))\!) = \texttt{*}(\!(e)\!)$ | $(\!| \mathsf{DeRef}(t, e) |\!) = (\!| e |\!)^*$ |
| $(\!(\mathsf{Cast}(t, e))\!) = \texttt{(t)}(\!(e)\!)$ | $(\!| \mathsf{Cast}(t, e) |\!) = \texttt{(t)}(\!| e |\!)$ |
| $(\!(\mathsf{SizeofType}(t, t'))\!) = \texttt{sizeof(t)}$ | $(\!| \mathsf{SizeofType}(t, t') |\!) = \texttt{sizeof(t)}$ |
| $(\!(\mathsf{SizeofExpr}(t, e))\!) = \texttt{sizeof}(\!(e)\!)$ | $(\!| \mathsf{SizeofExpr}(t, e) |\!) = \texttt{sizeof}(\!| e |\!)$ |
| | $(\!| \mathsf{Var}(t, \langle d \rangle) |\!)^* = \texttt{id}_d$ |
| | $(\!| \mathsf{PtrArith}(t, e_1, +, e_2) |\!)^* = (\!| e_1 |\!)\texttt{[}(\!| e_2 |\!)\texttt{]}$ |
| parentheses as required by precedence are implicit for both mappings | $(\!| \mathsf{Member}(t, e, m) |\!)^* = \begin{cases} (\!| e |\!)\texttt{->id}^m_{\sigma_t}, & \text{for } e = \mathsf{DeRef}(\ldots) \\ (\!| e |\!)^*\texttt{.id}^m_{\sigma_t}, & \text{otherwise} \end{cases}$ |
| | $(\!| \text{any other } e |\!)^* = \texttt{*}(\!| e |\!)$ |

Figure A.3: Translating Core C expressions back to C.

outcore; we provide a `-S` switch that does that (almost. Array(t, e) is rendered as a unary operator spelled `<DECAY>`).

## A.6 Types

The Abstract(id, q), Arithmetic(id, q), and Primitive(id, q) types are primitive types, at least as far as C-Mix/II is concerned.

The ISO C base types such as `signed char` and `float` are all Primitive types, with the `id` of the type record containing the type's full name. The exception is `void` which is represented as Abstract(`void`).

The user (or, more likely, shadow header files) can define additional primitive types with the special syntax

```
typedef __CMIX(bar) baz ;
```

which declares `baz` as an *abstract type* with the `b`, `a`, and `r` properties (the only one of these that actually does something at the moment is `a`). The type maps to Primitive(`baz`) or Abstract(`baz`) in Core C, depending on whether it is numeric or not. The type checker differentiates between integral and other arithmetic abstract types, and also maintains a guess on whether the type is signed or not; those attributes are stored in a "hidden" (which only means: not shown in the Core C grammar) field of the Primitive type node.

Enum types represent enumerated types. Except for the output routines in `outcore` and `gegen`, they are treated mostly like primitive types.

Pointer and Array are straight-forward—the optional expression in the Array type nodes is the length of the array. Each of them contains the type qualifiers of the *pointed-to* type. That is, in Core C, "pointer to const int" parses as "(pointer

to const) int" rather than ISO C's "pointer to (const int)". As far as partial evaluation is concerned, this difference is not important, but the c2core translation works best with having the qualifiers as part of the pointer.

Fun is a function type with result and parameter types. FunPtr is a pointer to function; its single subitem is always a Fun type. This is also the only way Fun types can be used to form other types.

StructUnion deserves some further explanation. The master definition for the struct or union concerned is referenced from the type node, so that the type tag and member names can be accessed. However, the type has its own private list of member *types*, allowing us to control the degree of structural sharing between types. This is important because we attach binding-time information to type nodes.

### A.6.1 Structural sharing of type representation

For the purpose of defining the rules for sharing of type nodes, we divide the references to type nodes in a program into "owning" and "borrowing" references, and say that a declaration or expression "owns" the type nodes that are reachable from its type field, following only "owning" references. Then:

- A local or global variable declaration owns its entire type. The only borrowing references that can be reached from the type node is the back edges to StructUnion nodes necessary to keep the representation finite.
- A function declaration similarly owns its entire type.
- A formal parameter declation borrows its type from the function's type's parameter type list.
- The declaration in an Alloc statement borrows its type from the relevant part of the statement's left-hand expression[4].
- A subsidiary declaration borrows its type from the appropritate part of its parent's type.
- A Var, Member, Array, or PtrArith expression owns the topmost Pointer node of its type. The pointed-to type is borrowed from the referenced declaraion or the relevant part of the operand's type, repectively.
- A Const, Unary, PtrCmp, Binary, Cast, SizeofType, or SizeofExpr expression owns its entire type, and, in the case of SizeofType, its entire operand type.
- A DeRef expression borrows its type from the relevant part of its operand's type.

Examples of the type sharing can be seen at figure A.4

[4]This does not impose any risk of "circular borrowing", because only the alloc declaration's possible subsidiaries borrow from its type, and nothing outside the subsidiary tree borrows from it

```
struct node {
  int ID: 24;
  struct node *link;
  char data[20];
} a_node, *a_ptr;
/* ... */
a_ptr->data[5]
```

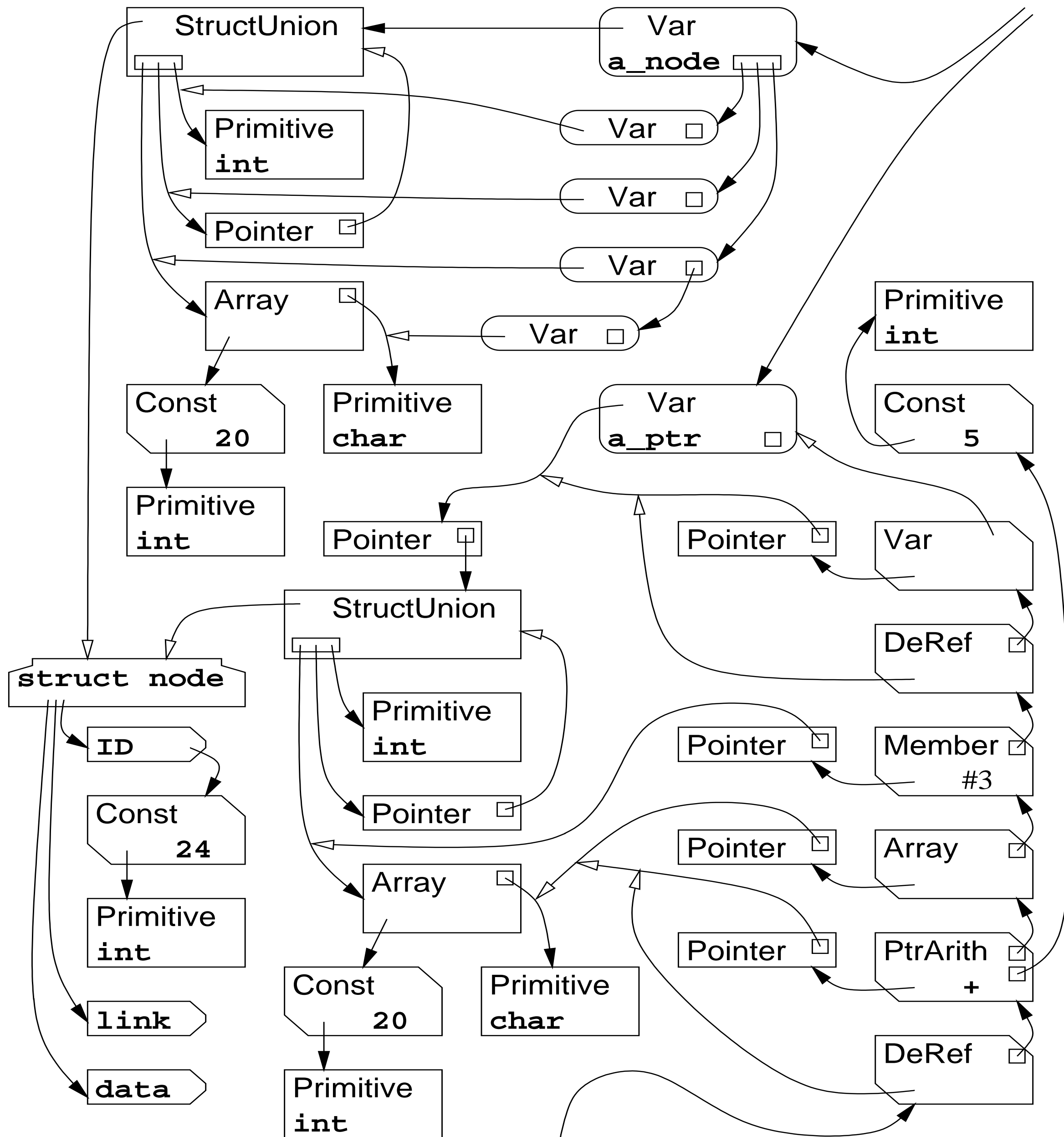



Figure A.4: Type sharing example

# Appendix B

# Why not to specialize arrays into structures

In this appendix we provide arguments against the idea (put forth in Box 72 on page 72) of implementing an $\mathsf{Array}_s$ with a dynamic element type by collecting the elements' residues in a struct type in $p_{res}$. We could call this variant $\mathsf{Array}_{struct}$.

## B.1 The pragmatic value is doubtful

$\mathsf{Array}_{struct}$ is an advantage over $\mathsf{Array}_d$ only if the C compiler that compiles $p_{res}$ generates better code for a `struct` than for an array that is only accessed with constant indices.

It is arguably easier for a compiler to generate good code in the struct case. But our practical experience (based on some semisystematic testing) is that with the compilers we use at DIKU there is generally no measureable differences in the running time of code when intermediates are stored in structs rather than constant-indexed arrays.

## B.2 Inventing new struct types should be avoided

The $\mathsf{Array}_{struct}$ idea involves declaring user-defined types where no user-defined type was before. How many of these do we need? Clearly there should at least be enough that we do not attempt to share a container struct between two Array types that are in fact different.

On the other hand, it is also wrong to declare too *many* container structs: if `foo` and `bar` both have the same $\mathsf{Array}_{struct}$ type, the container structs we use for each of them in $p_{res}$ ought to be the same lest a type error would result from trying to point the same pointer to `foo` and `bar` in succession. This is because in C two separately declared structs are *not the same type*, even if their members have identical types[1].

[1] Unless they are declared in different source files (ANSI 1990, clause 6.1.2.6), which types we declare in $p_{res}$ are not. Splitting $p_{res}$ into several source files would not be a possible solution, e.g. if `foo` and `bar` were local variables in the same function.

The basic problem that now creates a conflict is the fact that *type equivalence can not be reliably decided at Gegen time*. The reason for this is that we want to create generating extensions that are reasonably portable across platforms and compilers, though the residual programs are not[2]. Thus if the subject program includes, e.g., the standard header `<float.h>` which makes characteristics of the floating-point representation available as preprocessor macro constants, we do not want to make the generating extension depend on the precise values of those constant. This means that we have no way to tell whether, e.g., `char[FLT_DIG+4]` and `char[12]` are the same or different types.

One might be able to produce a safe (for this particular purpose) approximation to type equivalence by using the assumption that the subject program as read by C-Mix/II is type correct on the implementations where the user wants to run the generating extension.

## B.3 Necessary pointer types are hard to construct

Access to array elements always involves doing pointer arithmetic on a pointer-to-element derived from a pointer to the array itself. In order to use an $\mathsf{Array}_{\mathrm{struct}}$ meaningfully we would need a pointer binding-time type with the strange property that it is not known in $\mathrm{p_{gen}}$ *which* array it points to, but it *should* be known in $\mathrm{p_{gen}}$ which of the array's *elements* it points to.

We could implement such a pointer with two model objects: one symbolic `Code` model object which names a residual pointer to the struct that is the array's residual, and one concrete `unsigned` object which keeps the index into the array. The variant could be named $\mathsf{Pointer}_{\mathrm{so}}$ ("so" stands for "Static Offset"). This would involve problems of its own, though:

- The exact type of the residual of $\mathsf{Pointer}_{\mathrm{so}}(\tau)$ should be a pointer to the struct that represents $\mathsf{Array}_{\mathrm{struct}}(\tau, e)$. But the $\mathsf{Pointer}_{\mathrm{so}}(\tau)$ type itself does not reveal whether it came from $\mathsf{Array}_{\mathrm{struct}}(\tau, 2)$ or $\mathsf{Array}_{\mathrm{struct}}(\tau, 3)$.

  C has no problem with mixing pointers to elements of arrays of different length. There *would* be a problem with mixing $\mathsf{Pointer}_{\mathrm{so}}$s into arrays of different length, however.

  To extend the binding-time type system to take that into account would be very complicated. We would not be able to keep the fundamental idea that binding-time annotation consists of choosing among a fixed finite number of possible variants of each type constructor.

- One would like to support $\mathsf{Array}_{\mathrm{struct}}$ with a partially static element binding-time type.

  However,a $\mathsf{Pointer}_{\mathrm{so}}$ to a partially static type would need *three* model objects, which breaks the usual maximum of two model objects per type. In fact, using $\mathsf{Pointer}_{\mathrm{so}}$ allows one to construct binding-time types which need arbitraily many model objects. This would vastly complicate the design of the splitter phase.

[2]The rather loose specification of the semantics of C programs makes it close to impossibe to devise *any* nontrivial source-to-source transformation of C programs that guarantees semantic correctness no matter what implementation is used to run the output program—even when the input program itself is fully portable. That is outside the scope of this report, however.

# Appendix C

# References

Ammarguellat, Z. (1992). A control-flow normalization algorithm and it's complexity. *IEEE Transactions on Software Engineering*, 18(3):237–250.

Andersen, L. (1992). C program specialization. Technical Report 92/14, DIKU, University of Copenhagen, Denmark.

Andersen, L. (1994). *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark. DIKU Research Report 94/19.

Andersen, P. H. (1996). Static memory management in C-Mix. Technical report, Department of Computer Science, University of Copenhagen (DIKU).

ANSI (1990). *American National Standard for Programming Languages — C*. New York, USA. ANSI/ISO 9899-1990.

Consel, C. and Danvy, O. (1993). Tutorial notes on partial evaluation. In *Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993*, pages 493–501. ACM, New York: ACM.

Consel, C. et al. (1996). A uniform approach for compile-time and run-time specialization. In Danvy et al. (1996), pages 54–72.

Danvy, O., Glück, R., and Thiemann, P., editors (1996). *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*. Berlin: Springer-Verlag.

Erosa, A. and Hendren, L. J. (1994). Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of IEEE 1994 International Conference on Computer Languages*.

Futamura, Y. (1971). Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50.

Henglein, F. (1991). Efficient type inference for higher-order binding-time analysis. In Hughes, J., editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Berlin: Springer-Verlag.

Jones, N., Gomard, C., and Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall.

Jones, N., Sestoft, P., and Søndergaard, H. (1989). Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50.

Jones, N. D. (1996). An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–504.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 2 edition.

# Index


$\bot$, **100**, 107, 115, 118, 119
$\top$, **100**, 111, 114, 115, 118, 119
$+_s$, $+_d$, etc., 12
->, 68, 76
$\boxed{\frac{\cdot}{Y\neq0}}$, $\boxed{\frac{\cdot}{Y=1}}$, $\boxed{\frac{\cdot}{O=0}}$, *see* alias propositions
$\boxed{\frac{\cdot}{x/y}}$, 101
$\preceq$, 85, 90
$<$, 85, 90
$[T \in \mathbb{F}]$, **106**, 120
$[T_1 \equiv T_2]$, **103**, 120
$[T_1 \leqq T_2]$, **106**

Abstract, **63**
    $\text{Abstract}_d$, **63**
    $\text{Abstract}_s$, **63**
abstract C, 133
alias propositions, **102**
aliasing, 72
Array, **71**, 75, 105
    $\text{Array}_d$, **71**, 75
    $\text{Array}_{dy}$, **73**
    $\text{Array}_p$, **75**
    $\text{Array}_s$, **72**, 74
    $\text{Array}_{struct}$, **147**
Array Core C operator, 70, **74**, 75, 88, 112
array
    dynamically indexed, *see* $\text{Array}_d$
    statically indexed, *see* $\text{Array}_s$
arrow, **101**
    dashed, 109, 115
    dotted, **106**
    multi-premise, **101**, 105, 112, 120
Assign, 90, 114, 129–131, *also see* pointer, access through
assignment, 9, 50, 60, 69, *also see* Assign
audience, 5
axiom, 99, *also see* $\top$

back-up, 22, 49
backed-up ints, **48**, 64
basic, *see* binding-time type
basic block, **16**, 103
benign, *see* external call *or* Cast
bidsing-time annotations
    user supplied, 40
Binary, **63**, 88, 110
binding-time analysis, **7**, 10, 12, 23, 28, 32, 83, 84, 97–122
    constraint based, 98–99
    feedback from, 99, 100, 119
binding-time annotations, **10**, 56, 58, 120, 148
    impossible, 12, 23, 28, 33
    on expressions, 12, 61
    optimal, 12
    user supplied, 23, 32, 33, 101, 115, 118
binding-time type, **58**
binding-time type
    basic, **61**, 62, 123
    compound, **61**, 62
    dynamic, **60**, 81
    erasure, **58**, 61, 85
    faithful, **61**, 62, 80, 82, 83, 85, 87, 89, 102, 106, 116, 118, 134
    of expression, 61
    partially static, **60**, 78
    signature, **60**, 94, 102, 121, 123, 134
    splitable, **123**
    static, **60**
    well-formed, **58**, **86**, 104–106

binding-time type system, **58**, 62–85, **85–90**
branch, *see* conditional
BTA, *see* binding-time analysis

C++, 75, 136
C-Mix$_{II}$, **4–5**, 7, 15, 23–25, 28, 32, 34, 40, 44, 56–145
Call, 34–37, 85, 90, 115, 131
  and side effects, 94
call graph, 38
carry, **68**, 69, 70, 74, 76, 85, 90, 103, 106, 107, 112
Cast, 89, 114
  benign, **89**
  malign, **89**
`Code`, **13–14**
code bloat, 24, 38, 97
code sharing, 19–22, 29, 46
  and external functions, 28, 30, 40
  of basic blocks, 19–22
  of functions, 38–40, 46
Cogen, *see* Gegen
common initial sequence, 80, 81
concrete, *see* model object
conditional, 15
  dynamic, 17–18, 25, 31, 47, 94
  static, 16–17
Const, 83, **87**, 108
constraints, **98**
control dependence, 5, 94
control flow
  specialization and, 16–26
  structured, 15
conversion, *see* lift, Cast
Core C, 5, 15, 56, 133, **137–145**

$\frac{\cdot}{\text{D.C.}}$, 103
database query, 32
DeRef, **69**, 70, 89, 112, *also see* pointer, access through
dirty tricks, 64, 80, 89
division by zero, 24
domain-specific languages, 15
double representation, 34, 41
drop, 12

dynamic, 5, **10**, *also see* binding-time type
dynamic control, **31**, 32, **40**, 44, 45, 47, 53, **94**, 103, 115
  local, **47**, 53, **94**, 103

`Emit`, 13, 41
entry state, 38, 47
Enum, **64**, 103, 104, 117
  Enum$_d$, **64**
  Enum$_s$, **64**
$\frac{\texttt{enum}\ \eta}{\text{d}}$, 103
EnumConst, **65**, 87, 108
erasure, *see* binding-time type
example language, 9, 15, 26, 33, 43
execution path, 17, 18
exit state, 47
external call
  benign, **32**
  extra arguments in, 116
  sensitive, **32**, 40, 115
external functions, 24, 26–32, 37, 61
  and code sharing, 28, 30
  and speculative specialization, 28, 30

$\mathbb{F}$, *see* faithful
faithful, *see* binding-time type
Fibonacci sequence, 28
`FILE`, 63
FunAdr, **83**, 87, 107
function call, *also see* Call
function inlining, 37, 40–43, 48, 96
function pointers, *see* FunPtr
FunPtr, 37, **82**, 106
  FunPtr$_c$, **84**
  FunPtr$_d$, **82**
  FunPtr$_{dn}$, **84**
  FunPtr$_s$, **82**, 85
Futamura projections, **7**

$\Gamma$, 85
garbage, 29, 39, 80
GCC, 18
Gegen, **7**, 9–48
  hand-written, 7
generating extension, **6**, 10
  generator, *see* Gegen

hand-written, 6
generating function, **34**, 41, 82, 134
$\mathcal{G}(f)$, **126**, 131, 132
global variable, 43–47, 96
goal function, **34**, 37, 118
`goal` specializer directive, 34
Gödel numbers, 11
goto, 15, 24, 117
assigned, 18

heap allocation, 5

If, 117
infinite specialization, **19**, 23, 40, 135
initializer, 118, 125
ambiguous, 118
inlining, *see* function inlining
input division, 11
$\texttt{int}_b$, **49**
elimination, 51–52
$\texttt{int}_d$, 10
$\texttt{int}_s$, 10
functions returning, 47

jump, 15, 131

labels, 18
reuse of, *see* code sharing
lattice theory, 11
Lift, **50**
lift, **11**, 12, 49, 63, 65–67, 84, 85, 90, 94, 103, 106, 129, 134
`log`, 32
loop, 20
perpetual, 22, 23
lvalue, 68

magic words, 87
`MakeName`, 14
master version, **49**
Member, **76**, 78, 81, 88, 112, 125–128
memoization, **18**, 22, 38–40, 59, 80, 118
and global variables, 46
mental trap, 55
mix, 4, 7
model object, **59**, 60, 61, 71
concrete, **59**, 60
number of, 60
symbolic, **59**, 60, 75

name management, **14**, 15, 43
`NULL`, 66, 117
Null, **68**, 87, 108

$|\cdot|_O$, **60**
object, **59**, 67, *also see* model object
object-oriented programming, 59, 136
operators, Core C
Array, 70, **74**, 75, 88, 112
Binary, **63**, 88, 110
Cast, 89, 114
benign, **89**
malign, **89**
Const, 83, **87**, 108
DeRef, **69**, 70, 89, 112, *also see* pointer, access through
EnumConst, **65**, 87, 108
FunAdr, **83**, 87, 107
Member, **76**, 78, 81, 88, 112, 125–128
Null, **68**, 87, 108
PtrArith, **69**, 70, 74, 87, 109
PtrCmp, **69**, 70, 88, 109
Sizeof, 64, **89**, 114
Unary, **63**, 87, 108
Var, **68**, 83, 86, 107

partial evaluation, **4**, 41
partial evaluator, **4**
partially static, *see* binding-time type
partially static data, 48
pending list, **17**, 18, 22, 30, 41, 46, 59, 97
and `goto`s, 24
and loops, 20
old entries, 22
risks, 19–24
static values in, 18
pending loop, **18**, *also see* pending list
$p_{gen}$, *see* generating extension
Pointer, **65**, 71, 104
$\text{Pointer}_d$, **65**, **70**, 74, 75
$\text{Pointer}_p$, **70**

Pointer$_r$, **66**, 67, **70**, 123
Pointer$_s$, **65**, **70**, 74, 75, 123
Pointer$_{so}$, **148**
pointer
abuse, 24
access through, **69**, 70, 71
arithmetic, *see* PtrArith, PtrCmp
dynamic, *see* Pointer$_d$
static, *see* Pointer$_s$, Pointer$_r$
to function, *see* FunPtr
pointer analysis, 57, 82, **91**, 96, 115
points-to set, **91**, 93, 115
$p_{res}$, *see* residual progam
Primitive, **63**
Primitive$_b$, **64**
Primitive$_d$, **63**
Primitive$_s$, **63**
`printf`, 11, 13, 116
program specializer, *see* specializer
proof, 100, 120
propositions, 99, 101
provable, 119, 120
PtrArith, 67, **69**, 70, 74, 87, 109
PtrCmp, **69**, 70, 88, 109
`PutFunction`, 39
`PutPending`, 18, 20

`read`, 26–30
recursion, 38–40, 43
register allocation, 72, 96
residual, 6, 115
residual function
current, 13, 37, 41
empty, 43
non-returning, 47
recursive, 39, 47
stack of, 37
residual program, **4**
intermediate representation, 13
restore
from pending list, 18
side effects of Calls, 46, 47
Return, 117
return type
dynamic, 37
splitable, 52, 54, 125
static, 37, 47
`void`, 37
reuse, *see* code sharing
reuse pool, 38, 47
run-time code generation, 15

safety rules, **57**
scope, 68, 70, 86, 94
sensitive, *see* external call
shadow header, 87
sharing, *see* code sharing
side effect, 46
non-local, 5, **44**, 53, 54, **93**, 95
of Call, 94
signature, *see* binding-time type
`sin`, 32
Sizeof, 64, **89**, 114
slowdown, 45, 72, 76, 97
specialization
infinite, 97
specialization library, *see* Speclib
specialization variant, **58**, 86, 101, 134
basic, 62, *also see* binding-time type
specialized program, *see* residual program
specializer, **4**
monolithic, 6
self-application of, 6, 7
specializer directives, 118
Speclib, **13**, 14, 18, 37, 39, 41
spectime, 6, 115
speculative execution, 18
speculative specialization, **18**, 24, 28, 30, 31, 44, 94
run-time errors and, 24
splitter, 50–54, 61, 62, 85, **123–131**, 148
stack frame, 44, 91
staging of computations, 6
statecode, 44, 136
static, 5, **10**, *also see* binding-time type
`static`, 6, 39
`stderr`, 87
Struct, **75**, 79, 105, 125
Struct$_d$, **75**
Struct$_p$, **78**
Struct$_s$, **76**
Struct$_{sy}$, **79**, 114

StructUnion, *see* Struct *and* Union
structured programming, 15
subject program, **4**
suspect, **130**
`switch`, 18
symbolic, *see* model object

$\mathsf{T}^*_{e_i}$, **108**
Tempo, 5, 15
`this`, 75
trace set, 119
transition compression, 24–26
truly local, 86, **91**, 93, 96, 115
truth
  absolute, 48
$\tau_\mathtt{T}$, 62, 63, **77**
type
  checking, 12
  constructor, 58
  conversion, *see* lift, Cast
  equivalence, 73, 148
  notation, 58
  recursive, 58, 101, 103, 121
typing rules, **56**

Unary, **63**, 87, 108
Union, **80**, 105, 125
  $\mathsf{Union}_\mathrm{d}$, **80**
  $\mathsf{Union}_\mathrm{p}$, **81**
  $\mathsf{Union}_\mathrm{s}$, **80**
union rule
  first, **80**, 90, 105
  second, **81**, 90, 105, 136
union–find, 104, 126
unique-return-state requirement, **44**, 45–47, 53, 92, 94
unresolved problems, 23, 24, 32, 40, 135
unshareable function, **40**, 41, 43, 47
user annotations, *see* binding-time annotations, user-supplied

Var, **68**, 83, 86, 107
variadic arguments, 116
variant, *see* specialization variant
`void`, 9

`write`, 26

$|\cdot|_\mathtt{Y}$, **60**