Chapter 10 Splitting

This chapter describes *splitting*, a technique for transforming polymorphic code into multiple copies of monomorphic code amenable to further optimization such as inlining. Splitting thus resembles type casing (described in section 9.3) but introduces no additional run-time overhead for type tests. This chapter discusses splitting of straight-line code; splitting of loops will be discussed in Chapter 11.

10.1 A Motivating Example

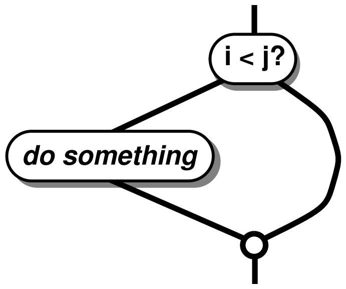
Splitting was originally motivated by attempting to generate good code for the following expression:

```
i < j ifTrue: [ do something ]
```

Expressions like this occur in virtually all programs, and so it is imperative to generate good code for this example. A reasonable C compiler, when faced with the similar C code:

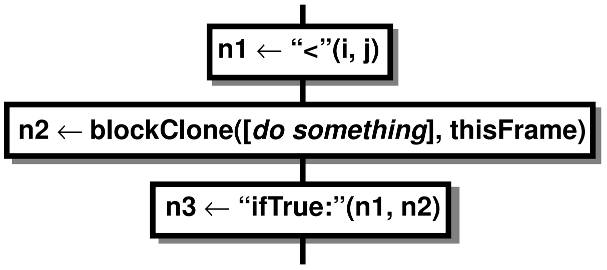
```
int i, j;
...
if (i < j) { do something; }
```

would produce the following control flow graph in some intermediate step:



The techniques presented so far can come close to this graph, but not quite.

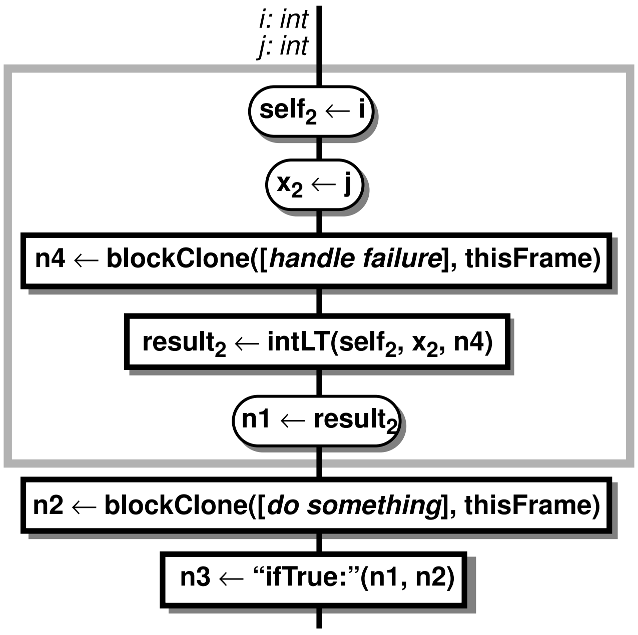
The SELF compiler would begin with the following control flow graph representation of the original conditional expression:



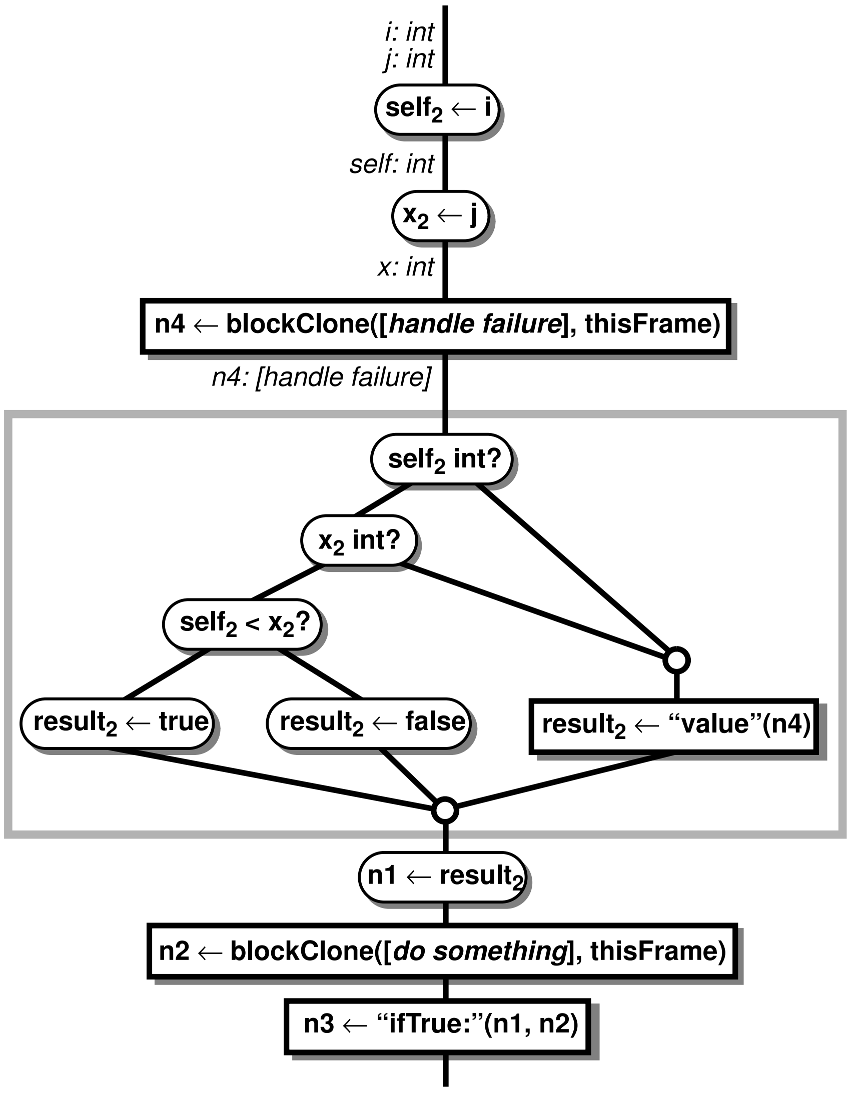
To simplify the example, assume that the SELF compiler is able to infer via type analysis that **i** and **j** are integers. Then the compiler can lookup the definition of **<** for an integer receiver to find the following method:

```
 < x = ( _IntLT: x IfFail: [ handle failure ] ).
```

The compiler can inline expand this method in place of the **<** message to produce the following control flow graph:

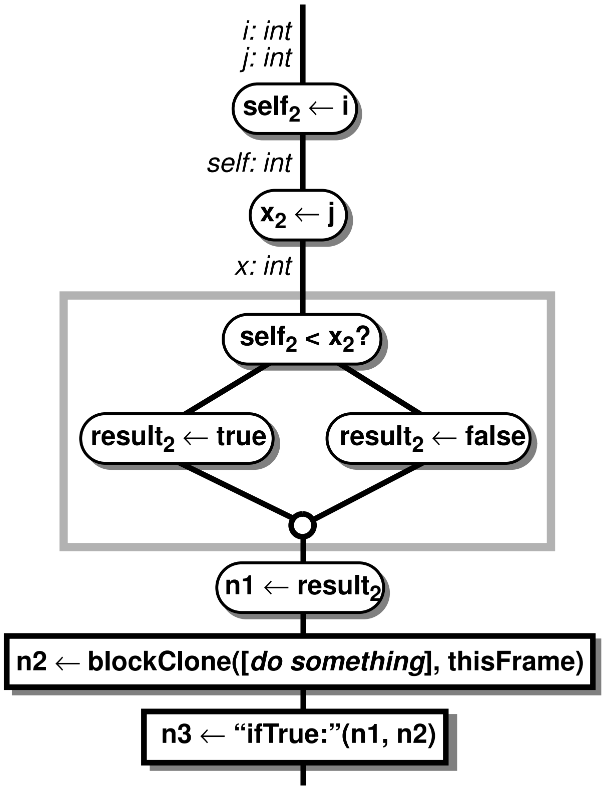

i: int
j: int
self₂ ← i
x₂ ← j
n4 ← blockClone([*handle failure*], thisFrame)
result₂ ← intLT(self₂, x₂, n4)
n1 ← result₂
n2 ← blockClone([*do something*], thisFrame)
n3 ← "ifTrue:"(n1, n2)

Type analysis proceeds, eventually reaching the call to the **`intLT`** primitive and expanding it in-line.

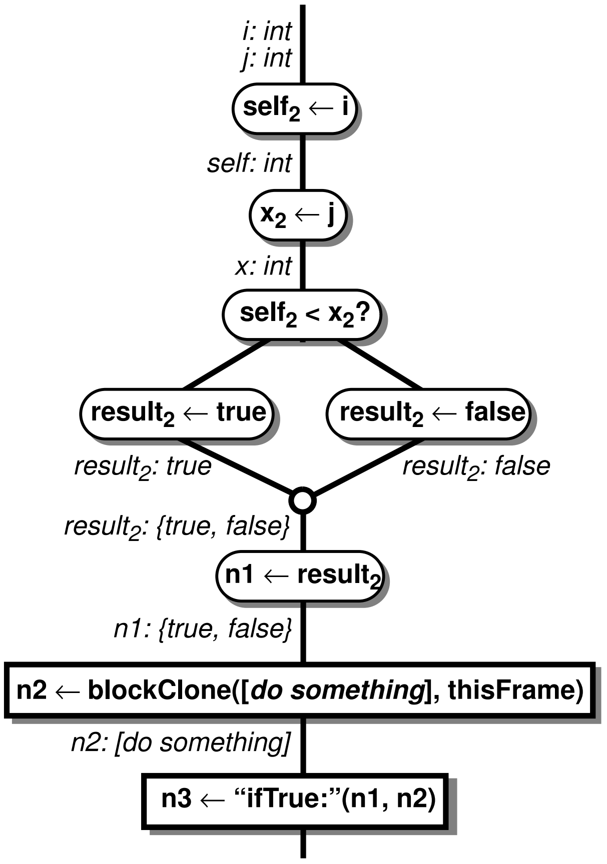


The compiler can eliminate the type checks on the arguments to the **`intLT`** primitive, since the compiler knows through type analysis that both **`self`** and **`x`** are integers. Subsequently, the compiler can eliminate the creation of the

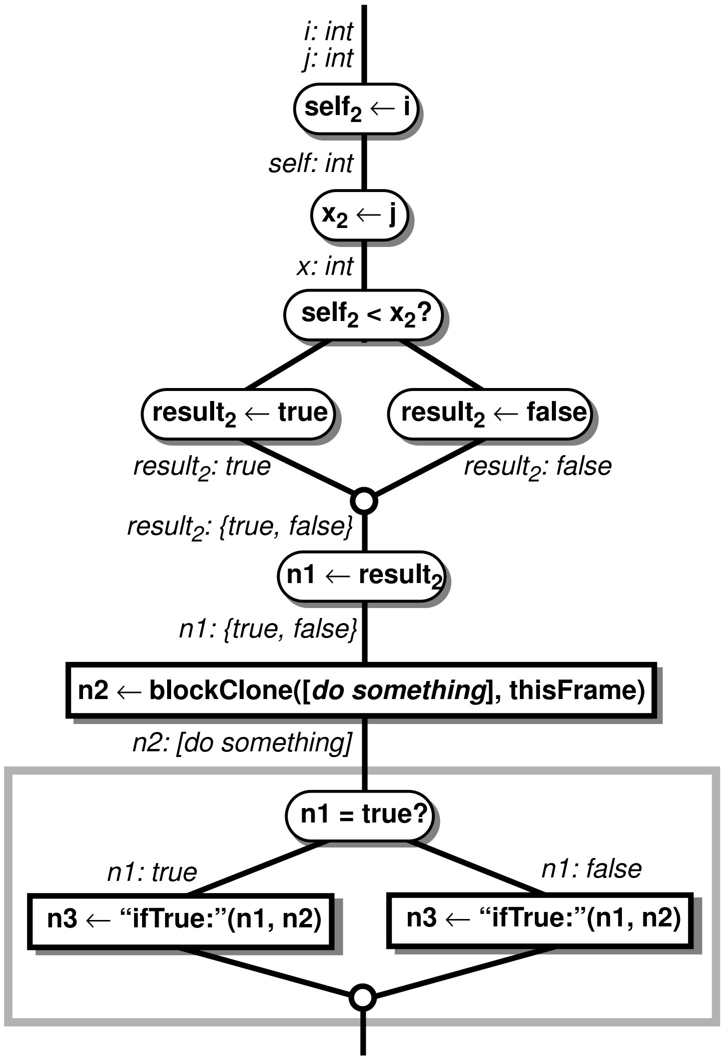
[*handle failure*] block, since it is no longer needed as a run-time value. These optimizations produce the following control flow graph:



Type analysis starts again, analyzing the body of the **`intLT`** primitive, and eventually reaching the **`ifTrue:`** message with the knowledge that the type of the receiver of **`ifTrue:`** is *{true, false}*:



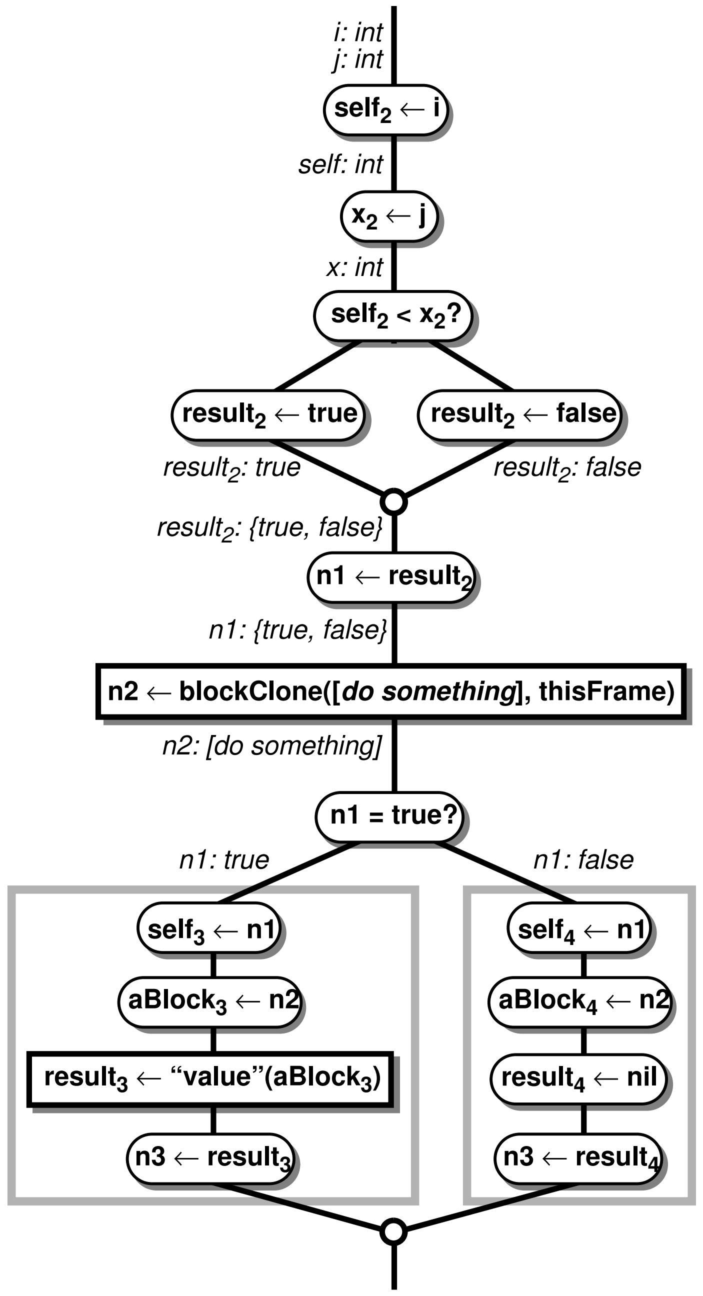
At this point, the compiler could use type-casing to insert a run-time type test to separate the **`true`** and **`false`** cases:



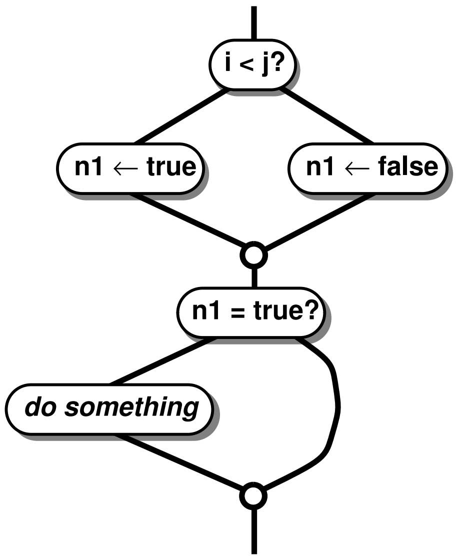
The compiler then can perform message lookup at compile-time for the two copies of the **`ifTrue:`** message, locating the following methods:

```
true = ( |
  ...
  ifTrue: aBlock = ( block value ).
  ...
| ).
false = ( |
  ...
  ifTrue: aBlock = ( nil ).
  ...
| ).
```

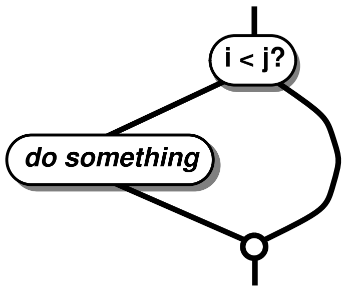
The compiler would inline these methods into the control flow graph:



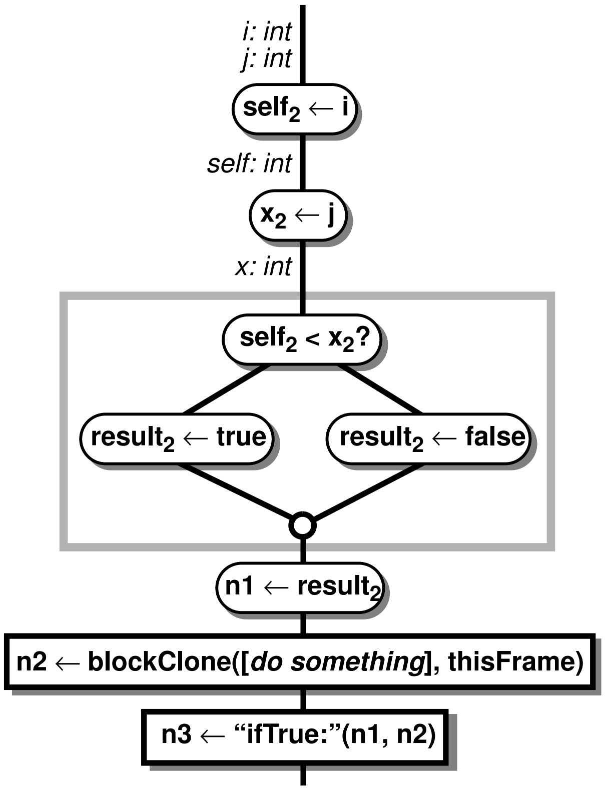
The compiler restarts type analysis, determines that the type of **`aBlock`**$_3$ is a particular cloned block literal, and inlines the block's **`value`** message down to just the body of the block; the block creation code can then be eliminated since there are no remaining uses of the block. This produces the following final control flow graph (after removing the assignment nodes which do not correspond to generated instructions):



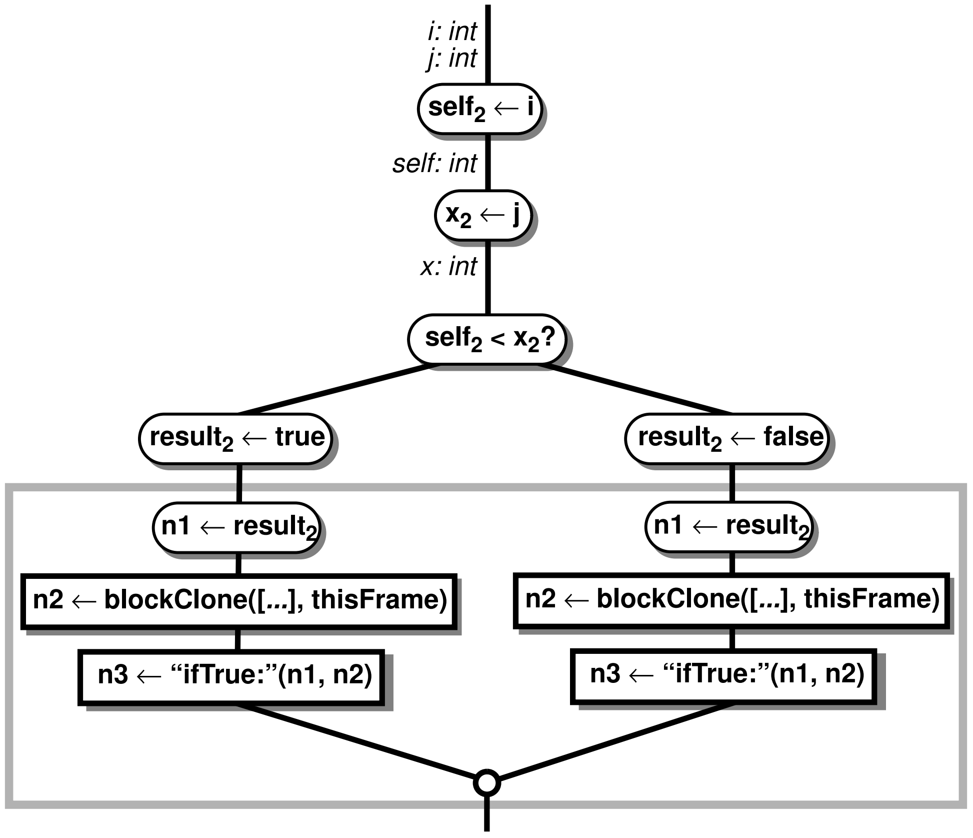
Unfortunately, this type casing approach produces code that is less efficient than the single compare-and-branch sequence generated by the C compiler:



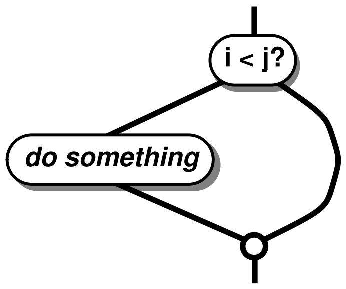
These inefficiencies stem from the control flow merging together after the **`intLT`** primitive only to be split apart again as part of type casing before the **`ifTrue:`** message. If the merge after the **`intLT`** primitive were simply postponed until after the two versions of the **`ifTrue:`** message, the inefficiencies would disappear, and the SELF compiler would generate the same code as the C compiler. For example, if after inlining the **`intLT`** primitive to reach the following graph:



the compiler had delayed the merge until after the **`ifTrue:`** message, copying all control flow graph nodes between the primitive result and the **`ifTrue:`** message to get the following graph:



then the compiler could directly inline the two **`ifTrue:`** messages without inserting any run-time type tests, since type analysis will infer that the type of the left-hand **`ifTrue:`** message is *true* and the type of the right-hand message is *false*. After inlining, the assignments to $\mathbf{result_2}$ can be optimized away, since they are no longer needed at run-time. This leads directly to the following control flow graph, after eliminating assignment nodes which do not generate machine code:



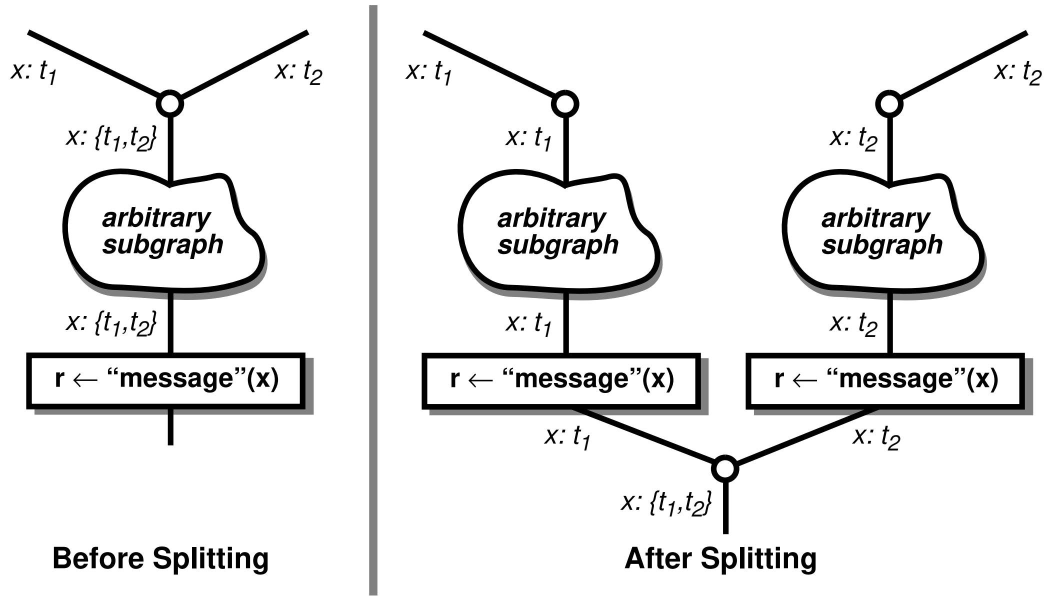
This graph is the same as that produced by the C compiler. To achieve this level of performance and be competitive with traditional languages, the SELF compiler needs some mechanism to postpone merges selectively, to avoid falling back on run-time type casing code. In other words, the SELF compiler needs some *splitting* mechanism.

We use "splitting" as a general term for techniques which lead to multiple versions being compiled of parts of the control flow graph, each version optimized for different situations such as for different type bindings. Several kinds of splitting have been implemented in the SELF compiler, each with a different trade-off between compilation speed and execution speed. The main discriminating characteristics of the various splitting strategies is when they decide to postpone a merge (and thus split nodes downstream of the postponed merge) and how they decide to stop postponing the merge. The next few sections describe these different approaches.

10.2 Reluctant Splitting

Reluctant splitting has been used in one form or another in all the implementations of the SELF compiler. In this splitting variant, the compiler initially merges control flow together but can later reverse this decision if desired, undoing the merge by copying parts of the control flow graph. This kind of splitting is called “reluctant” because the compiler splits merges only on demand; *lazy splitting* or *demand-driven splitting* would have been equally appropriate names.

For example, consider the following generalized example:



Under reluctant splitting, when reaching a potential merge point, the compiler merges control together. The compiler remembers the merge by forming union types for any names bound to different types before the merge. In this example, the **x** name is bound to the union type *$\{t_1, t_2\}$*. If later on some control flow node could be optimized if a name bound to a union type were instead bound to a component of the union type, the compiler will reverse its earlier decision to merge control together by duplicating all the control flow graph nodes between the premature merge and the node that demands the split. This duplication approximates the control flow graph that would have been generated had the original merge never taken place. In this example, the message send node could be optimized if the receiver **x** were bound to either component type rather than the joint union type, and so the message send node demands that the merge be postponed at least until after the message send. This demand is satisfied by duplicating all the nodes between the message send back to the original merge point. Subsequent splits may postpone the merge even farther. By copying parts of the control flow graph for each different type, the compiler has transformed what used to be a single polymorphic (and hence unoptimizable) message send into several independent monomorphic (and hence optimizable) message sends, without inserting any extra run-time overhead for type tests.

If the arbitrary subgraph copied as part of reluctant splitting is restricted to be empty (has no nodes that generate instructions), then we call the splitting algorithm *local reluctant splitting*, since the splitting takes place locally in the control flow graph. If arbitrary subgraphs are allowed, then we call the splitting algorithm *global reluctant splitting*.*

Splitting turns out to be a crucial technique for achieving good performance. As detailed in section 14.3, without any form of splitting SELF programs would run only half as fast.

* These two cases were termed *local message splitting* and *extended message splitting* in [CU90].

10.2.1 Splittable versus Unsplittable Union Types

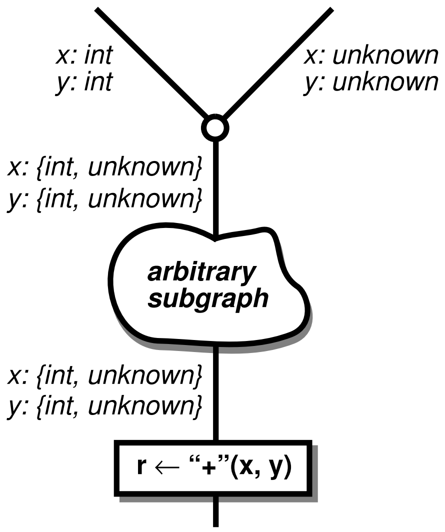
The compiler detects when reluctant splitting is possible by checking whether the type of some expression is a union type, since merges create union types of their component types and these component types may be split apart later. However, not all union types are created by merges: some are the results of primitives known to return one of a set of possible types. For example, the type of the result of a floating-point comparison primitive is the union of the **`true`** constant type and the **`false`** constant type. However, the compiler does not inline this primitive, instead invoking it by calling an external C++ function built into the SELF implementation, and so there is no merge node that can be split apart to separate the **`true`** result from the **`false`** result. Splitting is not applicable in this case, and the compiler should fall back on type casing (described in section 9.3) to optimize messages sent to the result of this primitive.

The compiler distinguishes between union types created as a result of merges in the control flow graph (and thus amenable to splitting) and those created externally (and hence not amenable to splitting). The former kind are called *splittable union types* while the latter are called *unsplittable union types*. Merge nodes create splittable union types; a floating-point comparison primitive returns an unsplittable union type. The compiler only attempts splitting for splittable union types.

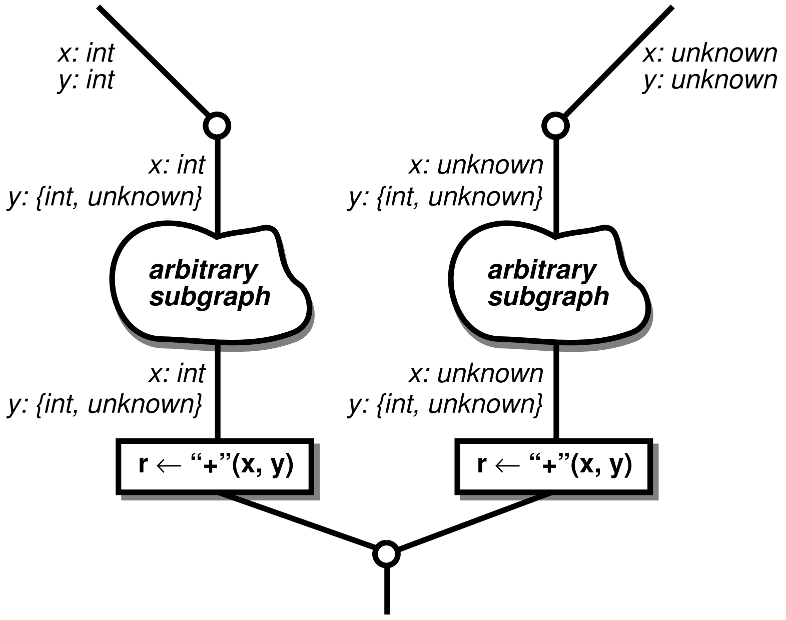
10.2.2 Splitting Multiple Union Types Simultaneously

The compiler uses splitting to transform a single piece of polymorphic code into several independent pieces of monomorphic code. Each splitting step typically operates on a single name and a single union type, such as the name and type of the receiver of some message. After the splitting operation, the compiler can update the type of the name along each of the split branches to the appropriate component type. These more precise types can then enable inlining along the split branches. Thus, an important function of splitting is to break apart union types into their component types which enables further optimizations.

This dividing of union types works fine for the single name being split upon, but does not improve the types of any other names that might also be more precise after the split. For example, consider the following control flow graph fragment:



After splitting the **+** message for the **`integer`** map type case of **`x`**, the compiler can alter its type bindings for **`x`** along the two split branches to get the following split control flow graph:

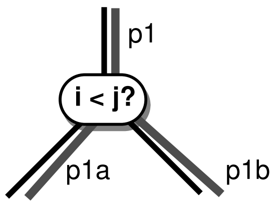


Unfortunately, using the techniques described so far the compiler would not notice that the type of **`y`** could also be narrowed along the two split branches. This would cause the compiler to insert an unnecessary integer type check for the **`y`** argument to the **`intAdd`** primitive that will eventually get inlined as part of the implementation of **+** for integers.

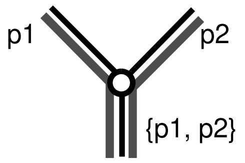
10.2.2.1 Paths

The SELF compiler solves this problem by augmenting type information with information about which possible paths through the control flow graph lead to various name/value and value/type bindings. These possible paths through the control flow graph are represented internally in the compiler by *path objects*. Path objects serve to link together pieces of type information that are guaranteed to occur together, such as components of different union types.

A path object represents a possible flow of control through the control flow graph. During type analysis the compiler keeps track of the set of path objects that represent the possible paths through the control flow graph that lead to the node being analyzed. The initial node in the control flow graph is associated with the initial path object. For straight-line code (i.e., basic blocks), the set of paths is propagated through from predecessor node to successor node unchanged. At branch nodes, for each incoming path the compiler creates a new path object for each outgoing branch, capturing the fact that two paths through the control flow graph are possible for each incoming path:

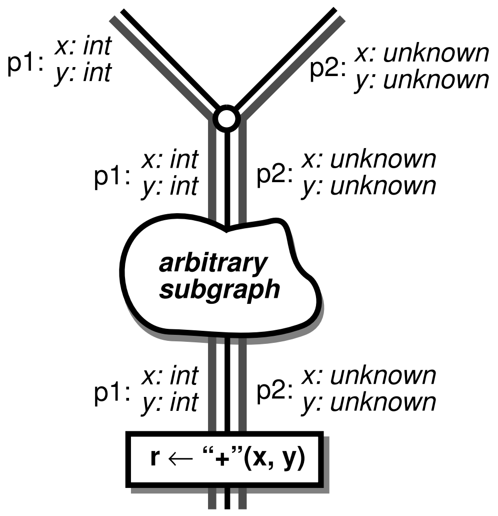


At merge nodes, the compiler forms the union of the set of paths along each incoming predecessor of the merge:

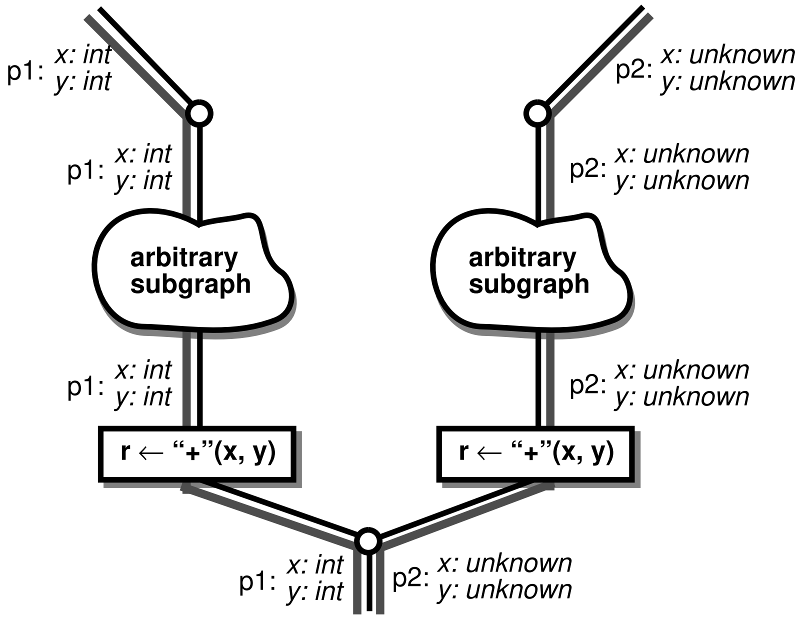


The compiler associates type information not with just a node in the control flow graph but with each path through the node. Conceptually, the compiler records name/value and value/type mappings (plus any other kinds of type information propagated such as cell/value mappings) for each path through the control flow graph. To compute the type of an expression at some point in the control flow graph, the compiler forms the union of the information about the expression associated with each path that reaches the node.

To illustrate, the following graph fragment shows how the compiler associates type information with paths rather than just nodes for the earlier example:



When the compiler reaches the **+** message, it elects to split off the integer case. To do this it must separate those paths where **x** is bound to *int*, namely p1, from the other paths, namely p2. This leads to the following graph:



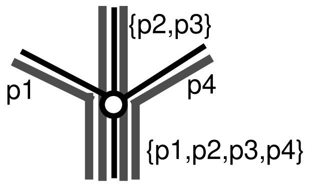
Paths elegantly solve the problem of narrowing the type of **y** appropriately after the split. When the compiler splits off one set of paths from another, in this case splitting off path p1 from path p2, all types associated with the split paths get narrowed implicitly. The type of **y** automatically is narrowed to *int* when p1 is split off from p2, since the type of **y** is associated directly with individual paths, not just with control flow graph nodes.

Splitting is couched in terms of separating one subset of paths from the remaining paths, instead of splitting some union type. The splitting subsystem of the compiler operates solely in terms of splitting paths apart and knows nothing about why the split is being performed or what affect the split should have on various kinds of type information. Other parts of the compiler are responsible for deciding when a split is in order, either to break apart a splittable union type, or to make some expression available for common sub-expression elimination, or even some combination of these. Once the compiler has decided to split, it calculates what path subset satisfies the desired criteria and invokes the splitting subsystem to actually perform the split.

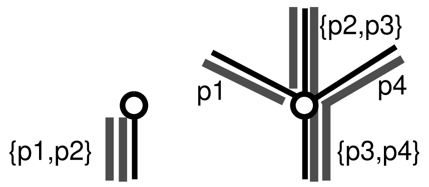
This separation of concerns simplifies the organization of the compiler by narrowing the interface between the splitting system and the rest of the compiler. It also enables certain kinds of splitting operations that otherwise would be costly or awkward. For example, sometimes the compiler needs to split off branches in which several expressions have particular types, instead of the normal case of splitting based on a single expression's type. This can occur when splitting off branches to a primitive in which all the primitive's arguments have the right types (i.e., where the primitive will not fail with a type error), or when connecting loop tails to loop heads as described later in section 11.4. The compiler can perform such splits simply by calculating which paths satisfy the right requirements and then calling the splitting subsystem with that path subset.

10.2.2.2 Splitting Merge Nodes

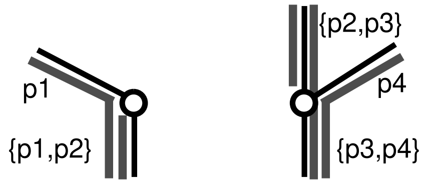
The splitting subsystem of the compiler performs the actual splitting operation by walking backwards through the control flow graph, copying each control flow graph node as it is traversed. At a merge node, the compiler examines each of the merge node's predecessors to decide how to split them, based on the paths that come in along that predecessor. For example, consider the following control flow graph fragment:



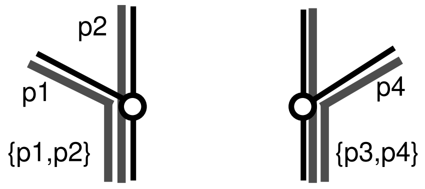
Say the compiler wants to split apart paths p1 and p2 from the other paths, and that the compiler has split nodes up to this merge node (the copied nodes are on the left):



The compiler first examines at the left-most predecessor. Since all of its paths (namely p1) are in the set of paths being split off (namely {p1, p2}), this predecessor is simply redirected to the copied merge node and not processed further:



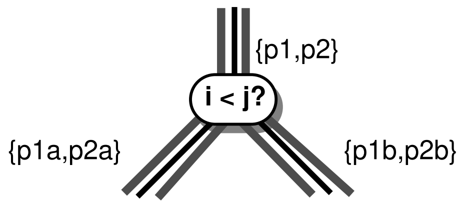
A subset of the middle predecessor's paths (i.e., p2) should be split off, while the rest (i.e., p3) should remain behind, unsplit. The compiler therefore continues to split the middle predecessor branch, eventually producing the following graph:



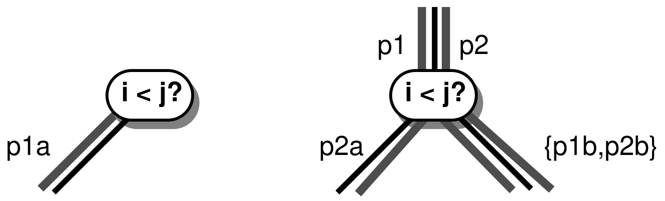
The compiler finally examines the right-most predecessor. Its paths (namely p4) are not in the split half, so this predecessor simply remains connected to the unsplit merge node.

10.2.2.3 Splitting Branch Nodes

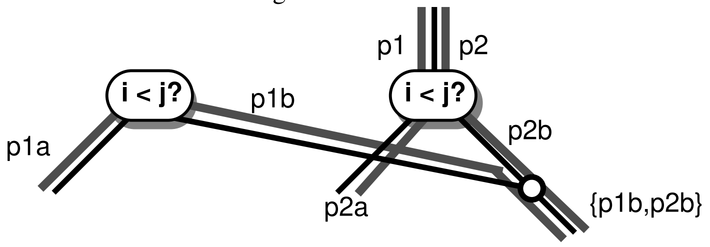
Branch nodes require special treatment to ensure that a branch node and its predecessor are split at most once, independently of whether one or both successors are split. Consider the following control flow graph fragment:



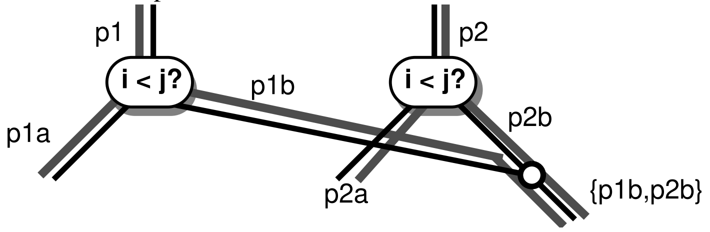
When the branch node is first split, say by splitting off the p1a path from the p2a path along the left-hand successor, the compiler first copies the branch node.



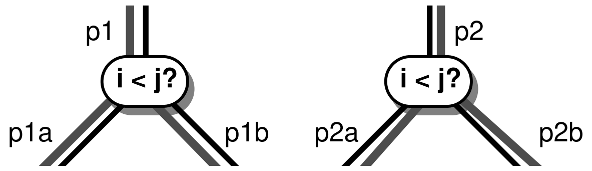
The compiler then inserts a new merge node after the other successor branch (the one that was not split) and connects the copied branch node's other successor arm to this merge node:



Finally, the compiler splits the branch node's predecessors:



If later on the branch's other successor is split, say by splitting off the p1b path from the p2b path, then normal splitting rules for merge nodes will break apart the freshly-inserted merge node, completely severing the link between the two branch nodes:



The current implementation of branch splitting actually optimizes this strategy by lazily creating and inserting the extra merge nodes. When a branch node is first split, instead of creating a merge node to join together the two unsplit successor branches, the compiler simply marks the branch node as "partially split" and links the two branch node copies together. If the other branch successor is split, then the appropriate control flow graph links are made, simulating the step of breaking apart the merge node that would have been inserted. To handle the case where the other successor to a branch node is not split, the compiler visits all "partially split" branches after the whole splitting operation is complete and then creates and inserts the necessary merge nodes. This optimization should speed splitting, especially if most branch nodes get split from both sides, but does introduce some complexity.

10.2.2.4 Implementation of Paths

A direct implementation of type information and paths as described could be quite inefficient if many paths have similar type information. Each path would have its own complete copy of the type information, with lots of duplication among paths. To avoid this potential problem, the current SELF compiler reintroduces splittable union types and associates each component of a splittable union type with the set of paths that generate that type. For example, the compiler represents the type binding where a variable **x** is bound to a type *t1* on paths p1 and p2 and a type *t2* on paths p3 and p4 as

```
x: {t1=[p1,p2], t2=[p3,p4]}
```

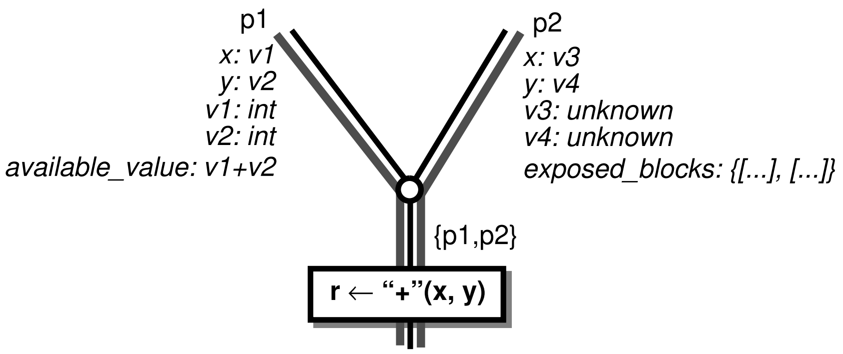
instead of

```
p1: x: t1; p2: x:t1; p3: x:t2; p4: x:t2.
```

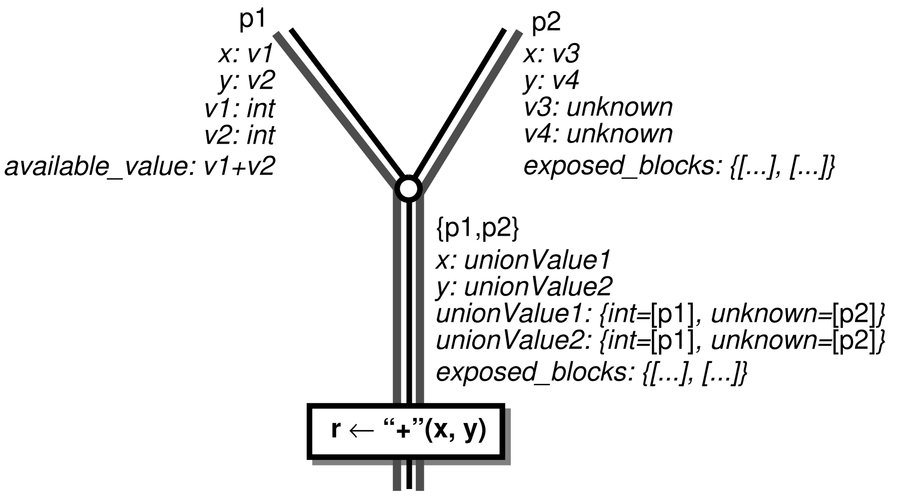
Values bound to types other than splittable union types are interpreted as being bound to the same type for all paths. Thus, information that does not vary from path to path is stored just as concisely as it was before paths were introduced.

This representation of type information is more compact than the straightforward representation using a complete copy of the type information for each path. It supports the same style of splitting including narrowing of all appropriate type bindings after a split. It also supports very efficient detection of when two paths lead to different types that might be split apart (by checking for values bound to splittable union types) and efficient calculation of which paths lead to desired types (by examining the set of paths associated with the desired component types of splittable union types).

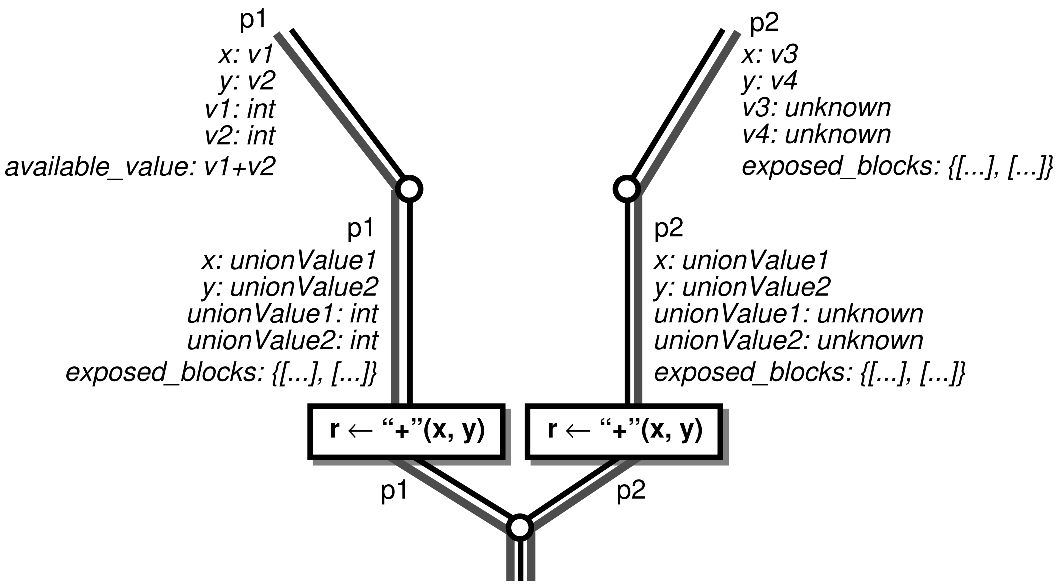
Unfortunately, this representation has some drawbacks over the straightforward representation. The chief drawback is that currently not all type information is connected with paths. Only types bound to values in the value/type mapping can depend on path information and be automatically narrowed, since only splittable union types relate to path information; all other information, such as name/value mappings, cell/value mappings, and exposed blocks lists, are assumed to apply to all paths. This lack of precision means that information other than the types associated with values can be lost if paths merge together and are later split apart. For example, in the following graph, the two paths merging together have different information:



When combining the information after the merge, the compiler creates new union values to represent the bindings for **x** and **y**. Paths are associated with the components of these splittable union types. The compiler must treat the *v1+v2* value as no longer available (since it is not available along path p2 and the compiler cannot associate available value information with particular paths) and must consider the exposed blocks from path p2 as exposed after the merge (since the compiler cannot associated exposed block information with individual paths):


p1
x: v1
y: v2
v1: int
v2: int
available_value: v1+v2
p2
x: v3
y: v4
v3: unknown
v4: unknown
exposed_blocks: {[...], [...]}
{p1,p2}
x: unionValue1
y: unionValue2
unionValue1: {int=[p1], unknown=[p2]}
unionValue2: {int=[p1], unknown=[p2]}
exposed_blocks: {[...], [...]}
r ← "+"(x, y)

When reaching the **+** message send, the compiler splits apart the type of **x** to optimize the integer receiver case, producing the following graph:



The compiler is able to narrow the type of **y** using the path information in the splittable union types of **x** and **y**. Unfortunately, the compiler cannot reclaim the lost information that *v1 + v2* is an available value along path p1 (which if the compiler could also restore the original value bindings of **x** and **y** would allow the compiler to eliminate as redundant the **x _IntAdd: y** calculation), nor can the compiler exclude the unnecessary blocks from the exposed blocks list along path p1. These differences do not lead to incorrect code: it is always legal to have fewer available values or more exposed blocks than is required. They do, however, sacrifice some information that could lead to better code.

A better implementation of type information and paths would enable the compiler to reclaim *all* type information after a split, as if no merge had ever occurred. The current SELF compiler only handles reclaiming type bindings; value bindings, available expressions, and exposed blocks lists are not reclaimed. The type binding case is by far the most important, since inlining a message is a much more important optimization in most cases than, for instance, common subexpression eliminating a single instruction. Therefore, we hope that the opportunities for optimization lost by the current imperfect implementation are relatively minor.

10.2.3 Heuristics for Reluctant Splitting

Since splitting can lead to an increase in compiled code space and compilation time, the compiler includes heuristics to determine when splitting should be performed and thus balance the expected improvement in run-time execution speed against the expected costs in increased code size and compile times. In the current SELF compiler, two factors influence the decision on whether to split part of the control flow graph: the *cost* of the split in terms of the number of copied control flow graph nodes and machine instructions, and the *weight* of the split paths in terms of their relative likelihood of execution. The compiler avoids splitting if the cost of the split is high or if the weight of the split paths is low, with stricter limits for splits that enable only less important optimizations.

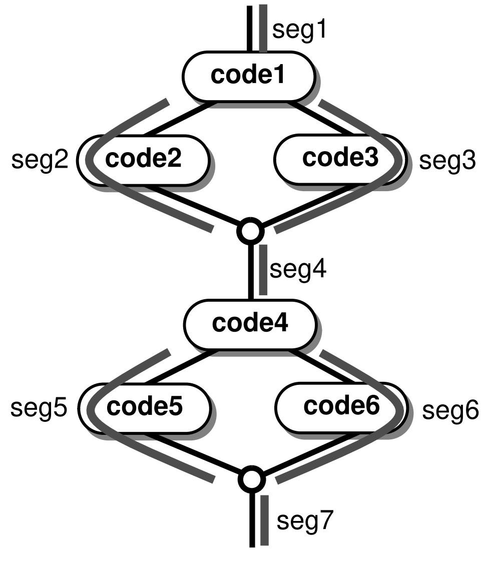
10.2.3.1 Costs

The cost of splitting one set of paths off from the remaining paths is calculated as the sum of the costs of the control flow graph nodes that would be copied as part of the split. Each node in the control flow graph has an associated cost, determined as follows:

- Many kinds of nodes have zero cost, since they do not generate machine instructions and can therefore be copied for free. For instance, name binding (assignment) nodes typically will generate no instructions, since the register allocator will arrange that the left- and right-hand-sides of the assignment end up in the same register.
- Other kinds of nodes will generate one or two machine instructions (such as arithmetic nodes and compare-and-branch nodes), and are given a cost of 1 or 2 as appropriate.
- Non-inlined message send nodes are given a cost of 5 to account for the extra space required for the send's in-line cache and for the extra instructions that are frequently needed to move the send's arguments into the locations defined by the calling convention.

It is difficult to determine efficiently which control flow graph nodes would be copied as part of a split. The compiler could determine this by simulating the split, traversing the graph in the same manner as the splitting operation would, but this would be expensive. The current SELF compiler represents paths as connected *path segments* to enable a more efficient computation of where paths overlap. Each straight-line sequence of code in the control flow graph (each basic block) is associated with a single path segment object. Path objects are represented as a list of the path segments that are traversed by the path. All paths that pass through a segment's corresponding basic block share the single path segment object.

For example, the following control flow graph sports 7 path segments which are linked together in 4 different paths (path segments listed from last to first, as path segments are "consed" onto the head of a path at each branch and merge point):



p1: (seg7 seg5 seg4 seg2 seg1)
p2: (seg7 seg6 seg4 seg2 seg1)
p3: (seg7 seg5 seg4 seg3 seg1)
p4: (seg7 seg6 seg4 seg3 seg1)

Path segments are concise abstractions of straight-line chunks of the control flow graph. Since straight-line chunks of code are split as a unit, the cost of splitting the nodes in the chunk can be computed once as the nodes are type analyzed and stored in the path segment object. Then to determine the cost of a total split, the compiler can first determine which path segments will get copied as part of a split, and then sum the costs of those path segments.

The compiler can compute which control flow graph segments will get copied as part of a split by finding those path segment objects that are shared between the set of paths being split off and the set of paths being left behind, up to the point at which the split paths and the unsplit paths diverge. In the above example, if paths p1 and p2 are to be split off from paths p3 and p4, the compiler calculates that segments seg7, seg6, seg5, and seg4 are shared between the split and unsplit paths. Segments seg3 and seg2 are not shared, and so will not be copied by the split, and even though seg1 is shared, it lies beyond the point at which the split and unsplit paths diverge, and so it will not be copied as part of the split.

This mechanism using path segments is more efficient than the straightforward approach of just simulating the splitting operation because it effectively caches the results of many of the operations that the simulation would perform, such as the costs of the nodes in each basic block. Also, the path graph is much smaller than the control flow graph, so traversing the path graph is faster than traversing the main control flow graph. Of course, such an optimization increases the complexity of the compiler.

10.2.3.2 Weights

The weight of a node represents the compiler's estimate of the likelihood of that particular node being executed. A weight is composed of two measures: a loop nesting depth and an "uncommonness" amount. The loop nesting depth component records the number of loops that have been entered but not exited since the beginning of the method. The initial loop nesting depth is zero.

The uncommonness component records how unlikely the compiler considers reaching that point in the control flow graph. For example, the compiler considers having a non-integer receiver for a message like **+**, in the absence of other information to the contrary, to be unlikely, and consequently increases the uncommonness component of the failure branch downstream of a run-time type test inserted as part of type-predicting the **+** message. (Type prediction was described in section 9.4.) The compiler also considers primitive failure to be unlikely, so if, for instance, the index argument to a **`_ByteAt:`** primitive either is not an integer or lies out of bounds, the compiler again increases the uncommonness component of the failure branch. The initial uncommonness value is zero, with higher uncommonnesses indicating less likely branches. A weight with an uncommonness value of zero is called *common case*, while weights with positive uncommonness values are called *uncommon cases*.

Weights do not attempt to record expected execution frequencies at a finer grain. For example, for branch nodes that correspond to normal comparisons in the source code, both successor branches are given equal weight, and this weight is the same as the weight of the predecessor of the branch. A more precise weighting system could mark the downstream branches each with half the weight of the branch predecessor. Unfortunately, this approach rapidly leads to trouble. Consider a series of conditional branches, corresponding to a series of **`if`**/**`elseif`** tests. By the time the leaves of this decision tree are reached, the weight of any individual leaf will be only $1/2^{\text{depthOfTree}}$ of the original weight before entering the decision tree. Since the compiler uses weight information to decide when splitting, inlining, and other optimizations are worthwhile, with these exponentially-reducing weight calculation rules the compiler might decide that none of the leaves of a decision tree are likely enough to merit optimization, even though for each execution of the decision tree some leaf is always executed. Clearly this behavior of weights is undesirable.

Another problem with fined-grain weights is how to combine weights together at merge nodes. A basic principle that the computation of weights should obey is *conservation of weight*: the sum of the weights leaving some arbitrary part of a control flow graph should equal the sum of the weights entering that part of the graph; otherwise, some execution frequency is being lost or gained. This implies that if weights are halved at branches, then weights should be summed at merges. Unfortunately, this fine-grained approach to computing weights is difficult to implement in the presence of loops. A loop head node merges together the loop entrance branch and any loop tail branches created by **`_Restart`** calls.[*] The weight of the loop head is therefore defined in terms of itself, since the loop tail weight depends on the loop head weight, and so a recurrence equation must be solved to calculate the weight of the loop. A simple approach would just increment a loop depth counter and ignore the weight of the loop tail branch. However, this approach violates

[*] The **`_Restart`** looping primitive was described in section 4.1.

conservation of weight, since the weight of the path through the loop that eventually connects back up to the loop head is unaccounted for and lost when summing the weights of the branches that exit the loop.

Since fine-grained weight propagation rules are difficult to support without weight loss, and since their extra precision is frequently unneeded and sometimes even counter-productive, the current SELF compiler uses coarser-grained weights that measure only loop depth and uncommonness. Successors to branch nodes are given equal weight with the predecessor to a branch node, and merge nodes take the maximum weight of their predecessors rather than the sum of their weights. These rules satisfy conservation of weight, since the weight of all loop exit branches is the same as the weight of the loop entrance branch (ignoring the effect of failed type predictions and primitives along loop exit branches). Also, since all leaves of a decision tree are the same weight as the entrance to the decision tree, all leaves will be optimized as if they were executed whenever the decision tree is executed, which is usually the desired effect.

Weights are associated with both paths and with nodes. The weight of individual paths is maintained as type analysis progresses and adjusted whenever a loop is entered, a loop is exited, or an uncommon branch is taken. Weights of paths remain unchanged when paths flow together at merge nodes or split in two at branch nodes. The weight of a node is the maximum of the weights of the paths reaching that node.

Since weights are associated with paths in addition to nodes, the splitting operation can easily compute the weight of a node that might be split off as the maximum of the weights of the split paths. After splitting, a new weight can be calculated in the same way for the branch left behind.

10.2.3.3 Cost and Weight Thresholds

Costs and weights are used to control which paths, if any, should be split when the compiler detects an opportunity for splitting. The compiler includes three threshold values, **`MaxSplitCost`**, **`MaxLowSplitCost`**, and **`MaxSplitUncommonAmount`**. **`MaxSplitCost`** and **`MaxLowSplitCost`** specify the largest cost of a splitting operation, above which the compiler will decide not to perform the split. **`MaxSplitUncommonAmount`** defines a threshold value of uncommonness that selects between **`MaxSplitCost`** and **`MaxLowSplitCost`**:

```
if splitWeight.uncommonAmount <= MaxSplitUncommonAmount then
  -- this is a relatively common path; use the more generous threshold
  if splitCost <= MaxSplitCost then
    -- this split is relatively inexpensive; go ahead!
    SPLIT
  endif
else
  -- this is an uncommon path; use the more stingy threshold
  if splitCost <= MaxLowSplitCost then
    -- this split is relatively inexpensive; go ahead!
    SPLIT
  endif
endif
```

Currently, for global reluctant splitting **`MaxSplitCost`** is set to 50, **`MaxLowSplitCost`** is 0, and **`MaxSplitUncommonAmount`** is also 0. This means that for common-case nodes (nodes with a zero uncommonness weight), the compiler is willing to duplicate up to 50 instructions as part of a split (this limit is rarely reached in practice). For uncommon nodes, however, the compiler is not willing to duplicate any instructions; only splits that do not increase the size of the compiled code are allowed along uncommon paths. Local reluctant splitting mode is enabled simply by changing **`MaxSplitCost`** to 0, thus preventing the compiler from ever duplicating instructions as part of a split.

10.2.4 Future Work

The current SELF compiler's heuristics for deciding when to perform reluctant splitting are fairly crude. They only weigh the expected cost of a split in terms of additional machine instructions generated and the likelihood of the split branch being executed and so benefitting from the split. Many other pieces of information could be used to improve the results of splitting. For example, the current heuristics take into account the expected increase in compilation time from a split only indirectly, through the dependence on the expected number of duplicated machine instructions. Also, these heuristics do not include any measures of the expected improvement in run-time performance caused by the split, other than the weight of the split paths. A better set of heuristics would differentiate various splitting opportunities with some characterization of the expected pay-off, for example by giving a high pay-off value for splitting that enables inlining of a message and a relatively low pay-off value for splitting that allows only constant folding or common subexpression elimination to be performed. Profile information gathered from previous executions of the program might also be employed to direct the compiler's attention to the most important parts of the program.

A more serious problem with the current heuristics is that they only examine local information. For example, when deciding whether or not to split a message, the compiler only looks at the costs and benefits for that single message. A better approach would take into account any future messages to or operations on the receiver. If many operations are going to be performed on some expression, then the compiler should be more willing to split the first operation, since the cost of the split can be amortized over all the subsequent uses of the split. Similarly, if the compiler has already duplicated many nodes as part of earlier splits, it should become more reluctant to split future messages, to avoid spending too much compile time and compiled code space on splitting. The current localized view influences other aspects of the compiler as well, such as deciding whether or not to type-predict or type-case a message. A more global perspective throughout the compiler could lead to significantly better trade-offs between compiled code space, compilation time, and run-time performance.

10.2.5 Related Work

Reluctant splitting is similar to redirecting predecessors in the TS compiler for Typed Smalltalk (described in section 3.1.3). Both approaches can duplicate a node after a merge, postponing the merge until after the duplicated node, to take advantage of extra information available prior to the merge. There are several differences, however. Reluctant splitting is performed using type information as part of type analysis, while redirecting predecessors is performed later in the compilation process using lower-level conditional branch information. By being performed as part of type analysis and inlining, splitting considerations can influence how the control flow graph gets constructed, and information other than conditional expressions can be split upon. Also, global reluctant splitting in the SELF compiler can split arbitrary amounts of code, while redirecting predecessors only splits nodes immediately after a merge.

Reluctant splitting has many similarities to Wegman's node distinction (described in section 3.4.5). Both approaches copy control flow graph nodes when they lead to different properties that some later code wants to optimize independently. However, node distinction is primarily a theoretical framework for explaining a variety of code motion and other optimizations, both traditional ones and novel ones, but not a practical mechanism for type-based splitting. The main disadvantage of node distinction as currently formulated is that the distinguishing criteria along which to duplicate nodes must be known in advance, before any duplication takes place. In other words, the compiler must know if it will be splitting before it merges any information together. In practice, however, the compiler does not know which merges will be split apart later and so should be postponed, and which merges should merge normally. This information only becomes available to the compiler when it reaches a message send node that wants the merge to be split apart, well after it has already made the decision of whether or not to split the merge. Reluctant splitting has the practical advantage that it operates on demand, first merging branches together and later splitting only if the information needs to be split apart and it is cost-effective to do so. One potential avenue for future work would combine the two approaches to produce a more theoretical framework for demand-driven splitting algorithms.

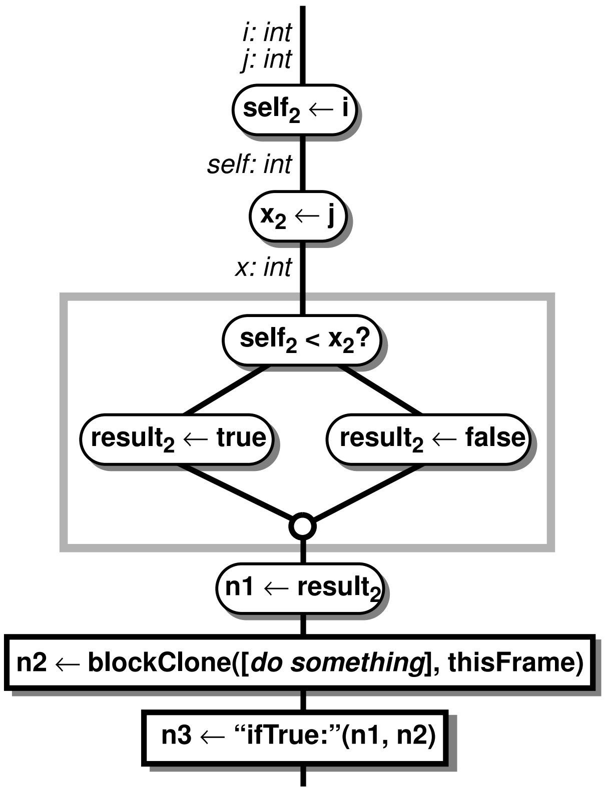
10.3 Eager Splitting

The SELF compiler supports an alternate splitting strategy that is almost diametrically opposed in philosophy to reluctant splitting. Where reluctant splitting initially merges branches together until they demand to be separated, *eager splitting* initially separates branches that would merge together in the source code. Each possible path through the control flow graph leads to its own sequence of control flow graph nodes; the control flow graph becomes a tree with no merge points.

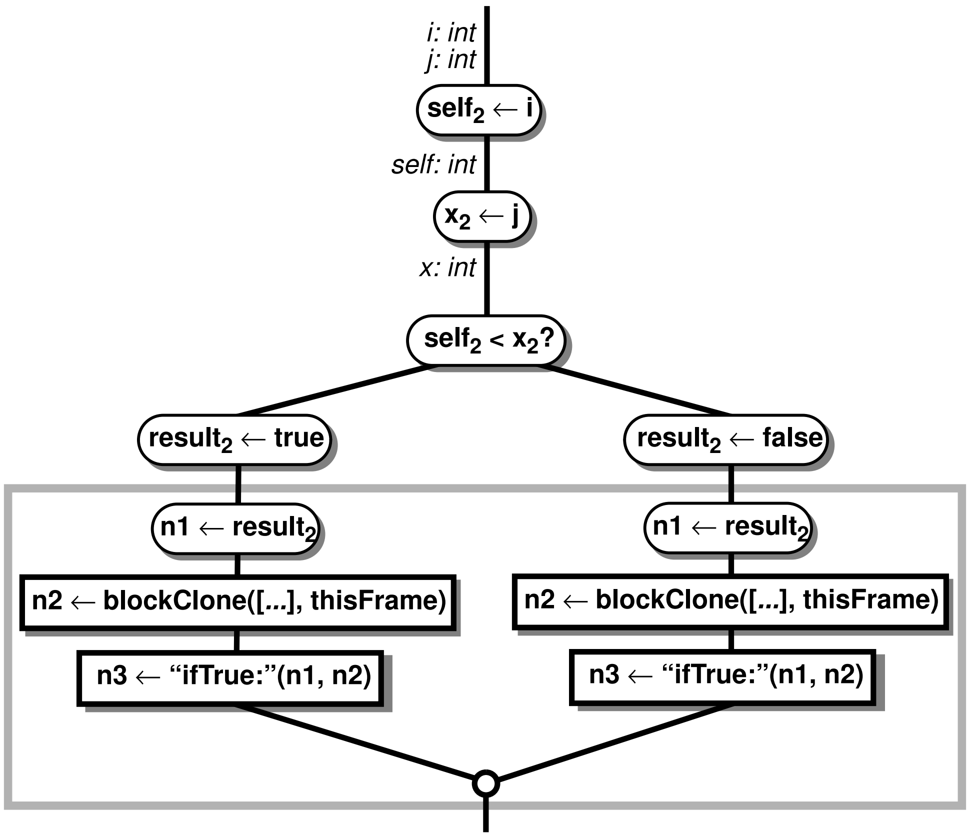
For example, when compiling the

```
  i < j ifTrue: [ "do something" ]
```

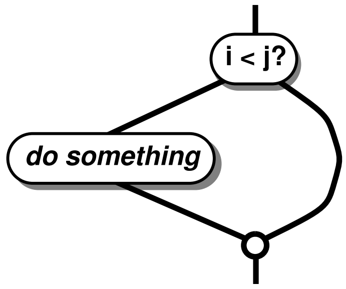
example, the compiler reaches the following control flow graph after inlining the **`intLT`** primitive:



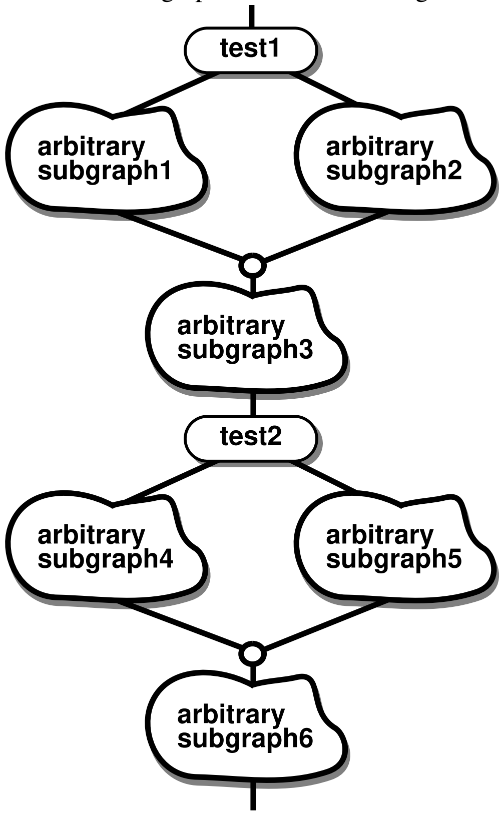
When the compiler reaches the merge node, under eager splitting the compiler does not merge control together but instead pursues each branch independently by duplicating the rest of the control flow graph for each predecessor of the merge:



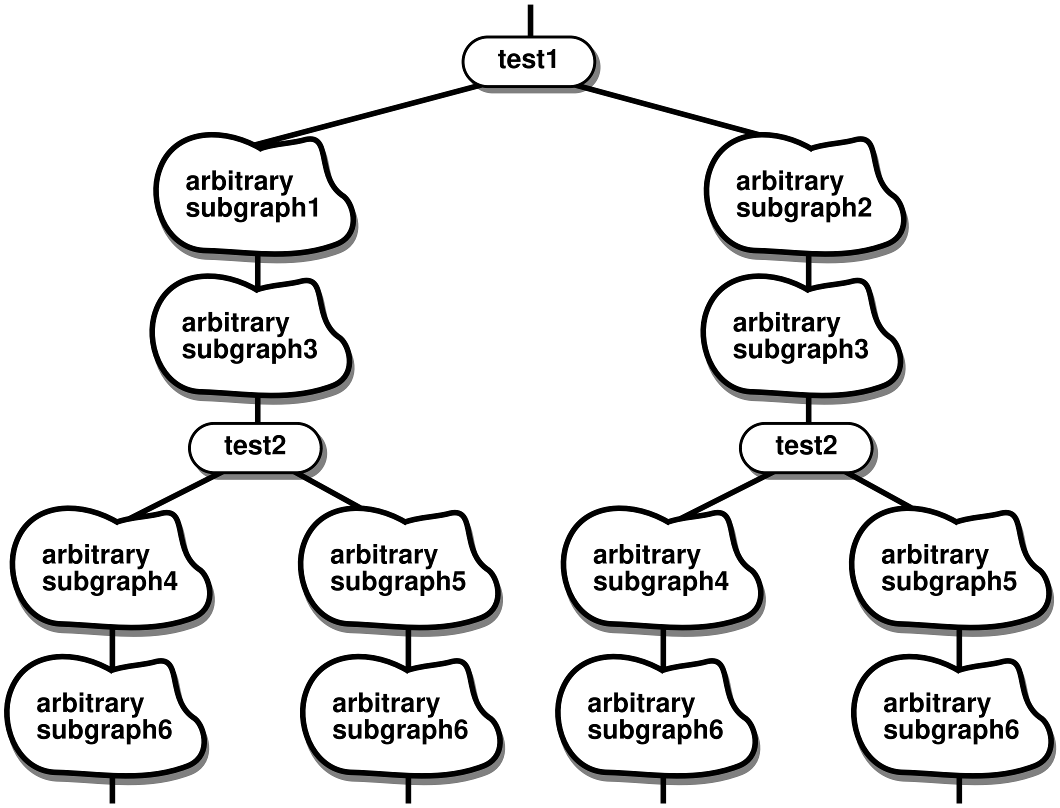
After processing these two branches independently, the compiler reaches the following final control flow graph:



In general, eager splitting transforms control flow graphs like the following:



into tree-shaped control flow graphs like the following:



Eager splitting has a number of advantages. Since there is no backtracking in the compilation and type analysis process (the compiler never changes its mind about whether two paths should be split apart), the compiler can generate the best possible code with a relatively simple type analysis system. No path data structures are required, since the control flow graph itself becomes isomorphic to the path data structure. Since the graph never merges, paths and splittable union types are not needed. Consequently, no inaccuracies or approximations induced by the representation of path-based type information can occur.

Also, the parts of the compiler that make use of the type information, such as the message inliner and the primitive operation inliner, need not constantly concern themselves with path-dependent type information and whether or not to split to get more specific information. This makes the rest of the compiler much simpler. In the reluctant splitting world, the complexity of testing for splittable union types and electing to perform a split is spread out through the parts of the compiler that exploit the type information. In eager splitting, the rest of the compiler is isolated from splitting issues, since a centralized system is making the splitting decisions (namely, to split always).

However, eager splitting has a number of serious drawbacks. The most obvious is the potential exponential code explosion as each path of control flow in the source code in turned into a separate physical branch through the control flow graph. As a first cut at reducing this space problem, the SELF compiler only performs eager splitting among common-case control flow graph branches; uncommon branches use an alternate splitting technique such as reluctant splitting. This heuristic attempts to balance better the benefits and costs of splitting by paying a high code space price only where it is likely to pay off (the common-case paths), and paying a much-reduced price where it is less likely to pay off (the uncommon case paths).

A problem still remains for control flow graphs containing loops: if the graph can never merge, then loop tails cannot merge with their loop heads, and programs containing loops suddenly have infinite tree-shaped expanded control flow graphs after eager splitting. The solution in the current SELF compiler's implementation of eager splitting is to treat loop heads differently from other merge nodes and allow loop tails to be connected to loop heads to form merges.

Even with these two modifications to the pure eager splitting model, however, the compiler still is likely to duplicate far too much code. For example, if two common-case branches merge together in the original unsplit graph, eager splitting will compile two completely independent copies of the rest of the control flow graph for those two branches. If those two branches have the same type information before the merge, then the two copies of the rest of the control flow graph will be *completely identical*! Performance measurements reported in section B.4.4 of Appendix B show that the SELF compiler takes about an order of magnitude more compiled code space and compile time with this version of eager splitting than it does using reluctant splitting.

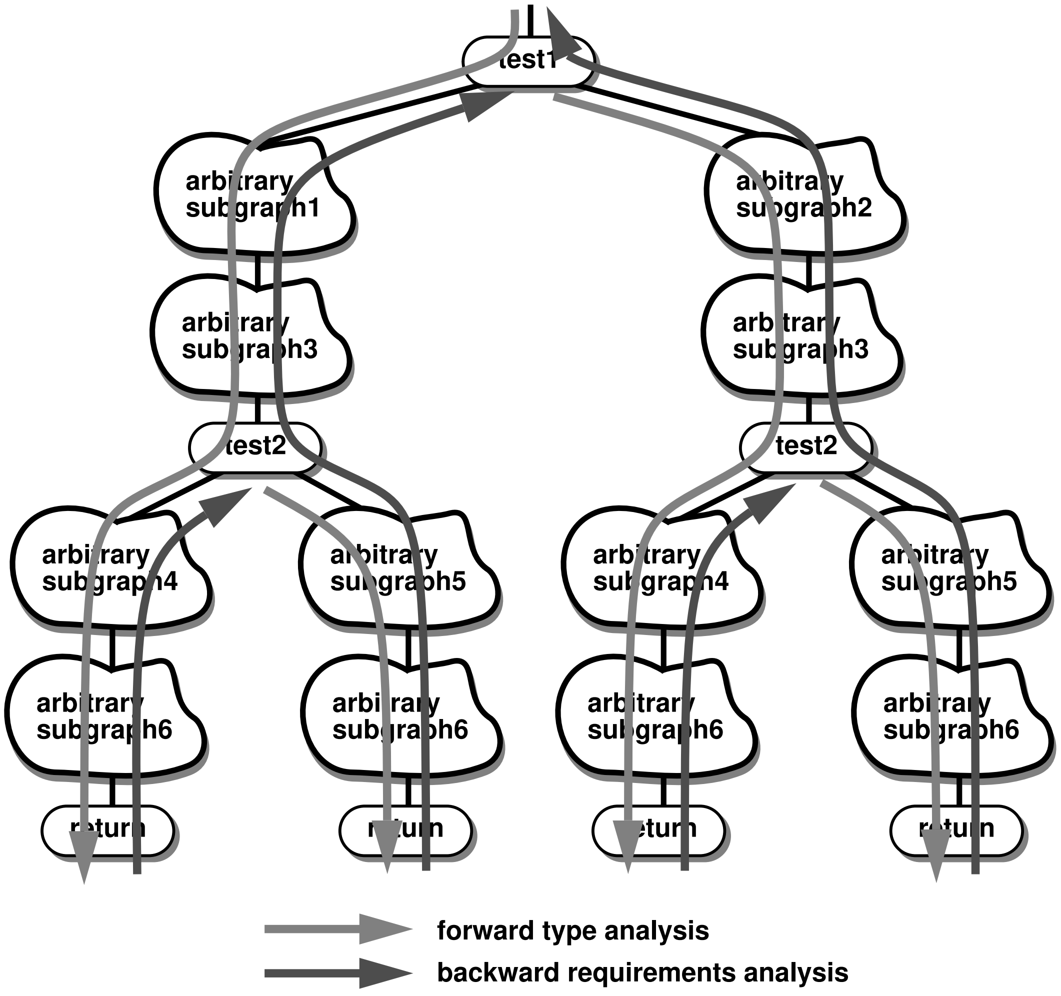
This observation suggests an approach for reducing the code explosion further: *tail merging*. In general, if two terminal branches in the control flow graph are the same, then they can be combined to save code by introducing a merge to connect the two branches together. Of course, to save compilation time as well the compiler would like to be able to detect when two terminal branches are going to be identical without actually compiling both branches and then comparing them. This approach is called tail merging after the similar traditional optimization that combines the ends of the two arms of a conditional statement if they are the same.

An easy way of detecting that two branches will generate the same code is to check whether the type information is the same at the beginning of the two branches. If they are the same, then the two branches must generate the same code and can be merged together. This prevents unnecessary duplication for "diamonds" in the control flow graph (i.e., conditional branches following by corresponding merges) where neither of the two arms of the conditional ("sides of the diamond") affect the type information entering the diamond, and so the type information exiting the diamond at the merge is the same along both branches.

Unfortunately, this simple approach is overly conservative and consequently does not save as much compiled code space or compilation time as would be desired: compilation speed and compiled code density improve by a factor of two over eager splitting without tail merging, according to measurements reported in section B.4.4 of Appendix B, but still fall well short of the performance of reluctant splitting. This form of tail merging misses cases in which two branches start out with different type information but still end up producing identical control flow graphs because the rest of the control flow graph does not depend on all of the available type information.

To enable more merging in the control flow graph than is enabled by the simple form of tail merging, the SELF compiler includes a technique we call *reverse requirements analysis* that identifies the subset of the available type information that actually is used when compiling a terminal branch of the control flow graph. The compiler allows tail merging whenever two branches have the same *required* type information, ignoring any unused type information.

Integrating forward type analysis, splitting, and tail merging decisions with backwards requirements analysis is somewhat tricky. The compiler type-analyzes the control flow graph in depth-first manner. Once a tip of the control flow graph is reached (i.e., some sort of return node in the graph with no successors), the compiler scans backwards through the graph to a branch point with a successor that has not been forward-analyzed yet.



As part of this backwards traversal, the compiler accumulates the assumptions (the requirements) that generated nodes make on the type information. In this way, the compiler determines the subset of type information that was used during the forwards compilation of a branch of the control flow graph. Later, when determining whether a to-be-generated branch can be merged with a previously-compiled branch, the compiler checks to see if the type information available at the to-be-generated branch is compatible with that required by the previously-compiled branch, as computed by reverse requirements analysis. If compatible, then the compiler merges the two branches together. Otherwise, the compiler continues generating the to-be-compiled branch separately, checking for potential merges at later points in the compilation process.

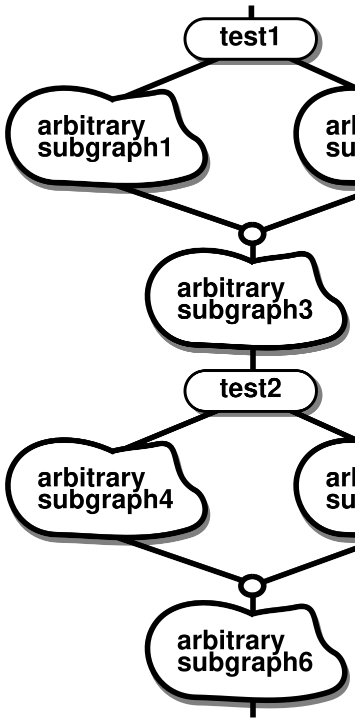
Loops complicate reverse requirements analysis much as they complicate forward data flow analysis in traditional compilers. The requirements of a loop head are computed from the requirements imposed by the loop body (and all branches downstream of the loop body), which in turn depend on the requirements of the loop head, since the requirements imposed by the loop tail depends on the requirements imposed by the loop head. Tail merging within loop bodies is even more complicated, since merging depends on requirements analysis that can only be performed after the loop has been completely constructed. The current implementation of requirements analysis side-steps these problems by assuming conservatively that the body of the loop depends on *all* the type information available at the loop head. This eliminates the problem of iteratively computing requirements but sacrifices some opportunities for tail merging. We do not know how important this sacrifice is, however.

Unfortunately, even with tail merging based on requirements analysis, eager splitting still takes too much compile time: according to the results reported in section B.4.4 of Appendix B, this most sophisticated form of eager splitting consumes twice as much compile time and compiled code space as local reluctant splitting. Ultimately, eager splitting may be deemed unusable because it is inherently inflexible relative to other strategies such as reluctant splitting. With reluctant splitting, the compiler writer can trade-off compilation speed and execution speed by varying the cost and weight splitting threshold values. With eager splitting, however, it is very difficult to "reign in" the compiler to generate less optimized code and so speed compilation. Once the eager-splitting compiler has assumed a certain piece of type information when generating code for a control flow graph node, this decision cannot be easily changed if the cost of such a decision later is deemed too great.

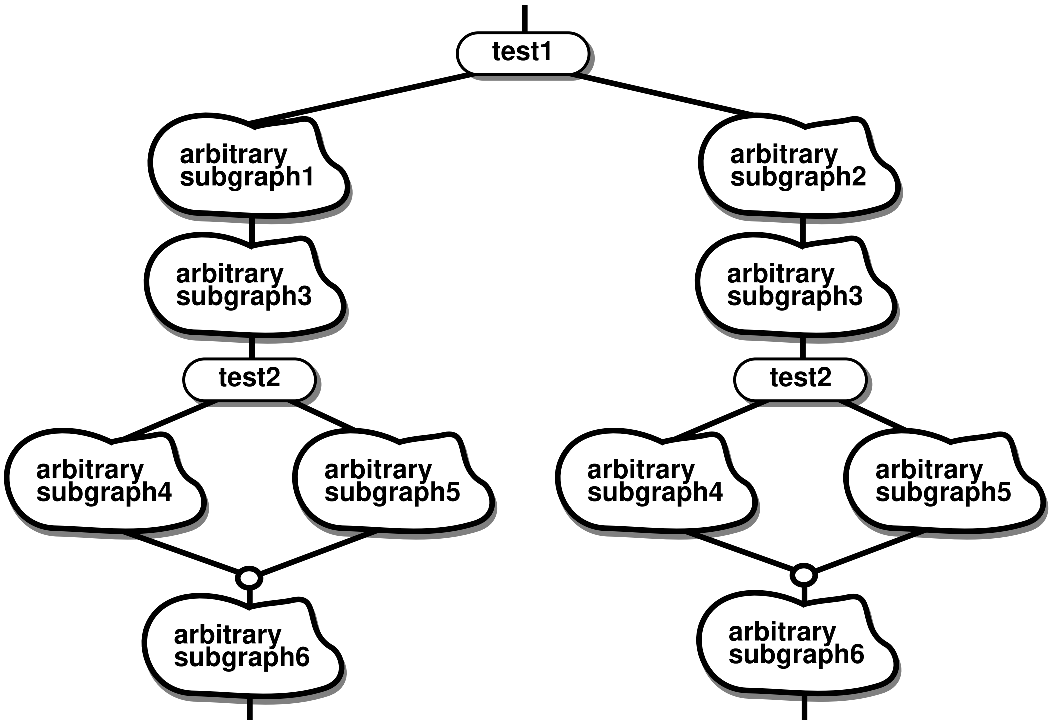
Consider an example where the compiler pursues a branch of the control flow graph, and in the last node assumes that the value of some variable is the constant `0`. Then a second branch is generated, and it is identical to the first branch except that the last node assumes that the value of the variable is the constant `1`. The cost of depending on the value of this constant may be very high, if the two branches are long and would otherwise be identical. But the compiler would have a very hard time detecting that the cost of this particular use of this particular constant type will be high without actually generating the two branches to completion. Reluctant splitting does not have this problem, since the compiler can decide at the point it is compiling the last node whether or not to split earlier branches apart. Eager splitting errs on the side of splitting branches apart, while reluctant splitting errs on the side of keeping branches merged together. The behavior of reluctant splitting probably is preferable to that of eager splitting.

10.4 Divided Splitting

Divided splitting is a practical compromise between reluctant splitting and eager splitting. Its description is simple: the compiler eagerly separates the common-case paths from the uncommon-case paths, but only reluctantly separates individual common-case paths from each other. In general, divided splitting transforms control flow graphs like the following:



into control flow graphs like the following, with the common-case paths completely separated from the uncommon-case paths, but with each half split apart only when desirable:



Divided splitting should at most double the size of the control flow graph over straight reluctant splitting instead of exponentially growing the size of the control flow graph as is characteristic of pure eager splitting.

Divided splitting is an engineering compromise motivated by practical concerns and empirical observations of programs. In most cases, the compiler will want to split common-case paths apart from uncommon-case paths. The common-case paths usually have many more sends inlined away, leading to more precise type information, more values available for common subexpression elimination, and fewer exposed blocks. Divided splitting therefore always splits apart the common-case and the uncommon-case paths. Since splitting apart various common-case paths one from another is not nearly as cut-and-dried, the more conservative reluctant splitting algorithm is used in those cases.

Divided splitting improves over the current path-based reluctant splitting algorithm by effectively placing a *firewall* between the type information along common-case paths and the type information along uncommon paths; the two sets of type information never mix across this firewall. This prevents the uncommon-case type information from diluting the common-case type information. If reluctant splitting were perfect, in that there was no loss or degradation of type information if a decision to merge branches together were later changed and no opportunities for optimizations were missed in the process, then divided splitting would not be very interesting. In reality, however, reluctant splitting is *not* perfect, especially as currently implemented with path objects used only for the value/type bindings. If common-case and uncommon-case paths continually merged together only to be split apart later, as is typically the case with pure reluctant splitting, then any loss of type information caused by imprecision in the implementation of reluctant splitting will take its toll over and over again. This typically would mean that the type information would not contain any available values or heap cell information, since the uncommon-case type information would not normally have them and the current reluctant splitting implementation does not parameterize this information by path. In contrast, with divided splitting, the common-case type information is never mixed with the uncommon-case information, and so uncommon-case paths do not cause type information along common-case paths to be lost. Since in many cases there is only one common case path, divided reluctant splitting should approach the code quality of pure eager splitting with only the compile time costs of pure reluctant splitting. According to results reported in section B.4.5 of Appendix B, divided splitting speeds SELF programs by up to 20% over the base local reluctant splitting algorithm, with no additional compile time or compiled code space costs. Divided splitting allows the SELF implementation to use an imperfect, faster implementation of reluctant splitting without sacrificing too much code quality.

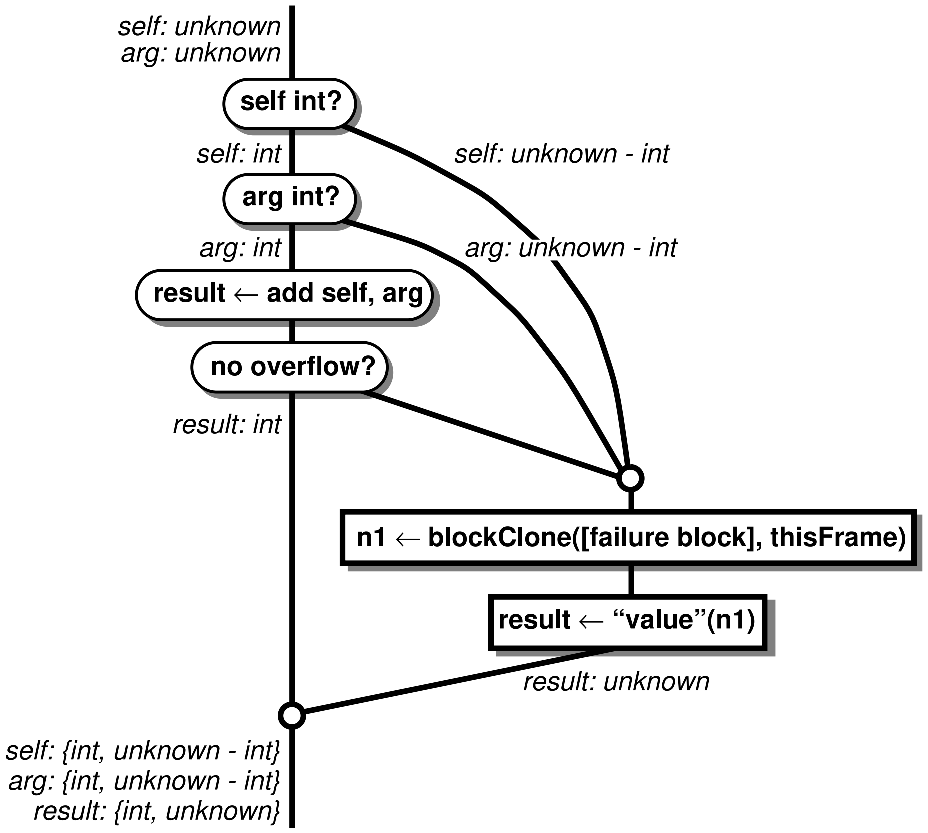
10.5 Lazy Compilation of Uncommon Branches

Many of the techniques in the SELF compiler distinguish between the common-case parts of the control flow graph and the uncommon-case parts. Some techniques such as divided splitting are designed specifically to separate the common-case paths from the uncommon-case paths. The burden of compiling the uncommon-case paths is fairly high: these parts of the control flow graph can be quite large, handling all the unusual events that might happen. Furthermore, uncommon branches are frequent in the compiled code. For example, most primitive operations such as arithmetic can fail in one way or another, leading to an uncommon-case branch that needs to be compiled. To add insult to injury, uncommon cases are expected to be uncommon, occurring rarely; most possible uncommon events never occur in practice. Thus, much of the compiler's effort is wasted.

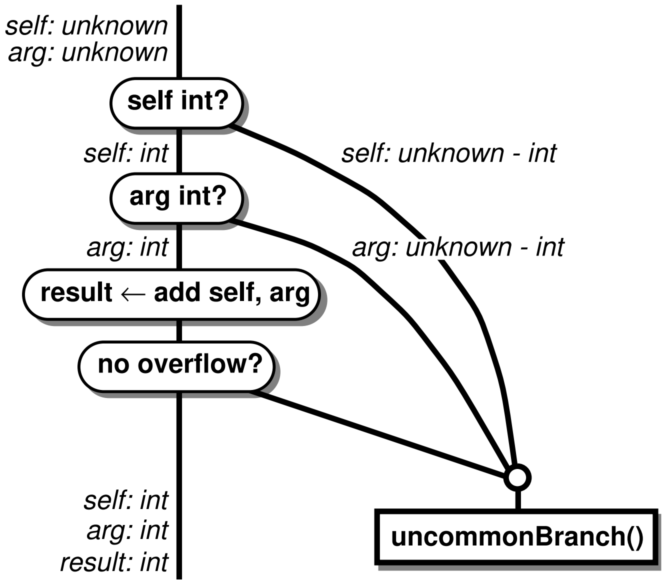
The SELF compiler takes advantage of this skewed distribution of execution frequency for different parts of the control flow graph by only compiling the uncommon parts of methods when the uncommon events actually occur; we call this technique *lazy compilation of uncommon branches*.* When the compiler is compiling a method for the first time, it only pursues the common-case paths of the control flow graph. At the entrances to all uncommon-case paths (for instance after a failed type prediction conditional branch or after a branch-on-overflow conditional branch) the

* Lazy compilation of uncommon branches was first suggested to us in 1989 by John Maloney, then a graduate student at the University of Washington.

compiler generates code to call a routine in the run-time system. Thus for a simple integer addition primitive, instead of generating the following graph:



the compiler will generate this graph:



Lazy compilation has a number of advantages. Foremost, it dramatically reduces both compile time and compiled code space by not compiling uncommon branches unless actually used in practice. The uncommon branches are typically much larger than the normal common-case code since the uncommon branch code has far more run-time type tests and space-consuming message sends than the common case branches. These expectations are borne out in practice: measurements reported in section 14.3 show that lazy compilation of uncommon branches speeds compilation by between a factor of 2 and a factor of 5; compiled code space costs fall by the same amount.

Lazy compilation has several other more subtle advantages. For one, lazy compilation automatically provides the advantages of divided splitting. Uncommon branches effectively are split off eagerly from the common-case paths, just as in divided splitting, by not compiling them at all. By providing the advantages of divided splitting as a fringe benefit, lazy compilation presents the rest of the type analysis and splitting system with a much simpler control flow graph, frequently one with only *one* remaining path. This allows the compiler to rely on a simpler, less accurate type analysis and splitting implementation without sacrificing much in performance and without actually needing to implement divided splitting.

Another subtle advantage of lazy compilation is that it eases the problem of good register allocation. As described in section 12.1, the SELF compiler performs global register allocation that allocates each name to a single location for its entire lifetime. Names that must survive across calls cannot be allocated to certain locations, such as caller-save registers. Since uncommon-case branches contain far more calls than common-case branches, eliminating the uncommon-case branches eliminates many register allocation restrictions and allows the SELF compiler's simple allocation strategy to produce a reasonable allocation. Together with improved type analysis, lazy compilation improves the execution speed of SELF programs by up to 30%, according to the results in section 14.3.

If the **`uncommonBranch`** procedure is called, the compiler starts up and compiles an additional chunk of code, called an *uncommon branch extension method*, to handle the control flow graph from the point of failure that led to the uncommon branch to the end of the original method. In the current implementation, an uncommon branch extension method takes over the original common-case method's run-time stack frame and returns to the original common-case method's caller, thus completing the original method's charter.* Since the uncommon branch extension method does *not* return to the middle of the common-case version, the common-case version is freed from handling uncommon cases merging back into the main common-case flow of control. This allows the compiler to generate excellent code for the common cases, at the cost of somewhat larger compiled code space usage. Analogously to in-line caching (described in section 8.5.1), the compiler backpatches the call to the **`uncommonBranch`** routine, replacing it with a call to the newly-compiled uncommon branch extension code. Since the compiler's beliefs about what is uncommon have already proved incorrect, the compiler does not distinguish common from uncommon branches in an uncommon branch extension. The compiler applies a conservative splitting strategy (currently local reluctant splitting) when compiling an uncommon branch extension, and the compiler does not perform lazy compilation within an uncommon branch extension method. This strategy limits the amount of compile time and compiled code space spent on uncommon branches.

The current implementation for compiling uncommon branches treats the extensions as new methods that are in some ways subroutines called by the common-case version of the method. An alternative strategy would recompile the original method when an uncommon branch was taken, this time with all uncommon branches compiled in-line (i.e., without any stub routines), replace the old common-case-only code with the more general all-cases code, modify the existing stack frame accordingly, and then restart the new method at the point of the entrance to the uncommon branch. The current strategy is significantly simpler to implement than this alternative strategy but may take up more compiled code space and compile time in cases where several different uncommon-branch entrances are taken on different invocations of the original method. With the current strategy each separate uncommon branch entry point will lead to a separate uncommon-branch extension method, while with the alternative strategy the original common-case-only method will be generalized to handle all future uncommon branches in one step. Fortunately, we have not observed many multiple uncommon branch extensions for the same common-case method, and so we are content to remain with the simpler strategy.

* The first operations performed by the uncommon branch extension adjust the stack frame size and execute any register moves required to shift from the common-case method's register allocation to the uncommon extension's register allocation.

10.6 Summary

The SELF compiler uses splitting to transform a single polymorphic piece of code into multiple monomorphic versions that can be further optimized independently, thus trading away some compiled code space for significantly improved execution speed. Reluctant splitting records enough information at merge points so that the merge can be reversed later if the more specific type information available before the merge would be helpful; the merge is simply postponed until after the point which desires the more specific information. Path objects enable the compiler to narrow multiple union types after a split, thus keeping type information fairly accurate in the face of repeated merging and splitting. The compiler includes heuristics based on the estimated space cost of the split and the estimated execution frequency of the split nodes to decide when the benefits of a potential split outweigh the costs. The demand-driven nature of reluctant splitting works well in practice, balancing compilation time against the quality of the generated code by exploiting most splitting opportunities with only a single forward pass over the control flow graph.

Eager splitting is an alternative splitting strategy that initially always splits merges apart, assuming that most such merges will be split apart anyway. Eager splitting offers a simpler, possibly faster implementation of splitting with no backtracking and no loss of type information from merges that are later split apart. Unfortunately, pure eager splitting leads to an exponential increase in the size of the control flow graph and hence in compilation time. To avoid unnecessary code duplication, eager splitting can be augmented with tail merging, a technique that merges two branches together if the second branch would generate the same control flow graph as was generated for the first branch. The compiler implements two forms of tail merging, one based on forward-computed available type information and the other based on more precise reverse-computed required type information. These tail merging extensions reduce compilation time for eager splitting somewhat, but in the end compilation time is still too long for eager splitting as currently implemented to be a practical splitting alternative.

Divided splitting is a hybrid approach that strives to provide the precision of eager splitting with the costs of reluctant splitting. With divided splitting, the compiler eagerly splits apart common-case paths from uncommon-case paths, but uses reluctant splitting to split apart common-case paths from one another. Divided splitting overcomes some of the weaknesses in the current implementation of reluctant splitting by placing a firewall between the relatively precise type information available for the common-case paths and the relatively imprecise type information for the uncommon-case paths. The compilation speed of divided splitting is roughly the same as the compilation speed of pure reluctant splitting, with significantly better code quality.

The SELF compiler saves a great deal of compilation time and compiled code space and even improves execution performance by compiling uncommon branches lazily. When a method is first compiled, only the common-case paths are compiled; stubs are generated for all branches into uncommon-case paths that simply call a routine in the run-time system. Only if one of these uncommon branches is taken does the compiler actually generate code for the uncommon branches. This division automatically provides the effect of divided splitting with no additional effort over that for lazy compilation.

Lazy compilation enables the SELF implementation to "have its cake and eat it, too." Since the common-case paths typically correspond to those paths that are the only ones supported by the semantics of traditional languages such as C, the compile-time and run-time speed of SELF is close to that for traditional languages (both systems end up generating much the same code), but SELF also supports the extra power of, for example, generic arithmetic; the programmer pays for these advanced features only when they are used.